FBOSS: Building Switch Software at Scale

Sean Choi*
Stanford University

Boris Burkov
Facebook, Inc.

Alex Eckert
Facebook, Inc.

Tian Fang
Facebook, Inc.

Saman Kazemkhani
Facebook, Inc.

Rob Sherwood
Facebook, Inc.

Ying Zhang
Facebook, Inc.

Hongyi Zeng
Facebook, Inc.

ABSTRACT

The conventional software running on network devices, such as switches and routers, is typically vendor-supplied, proprietary and closed-source; as a result, it tends to contain extraneous features that a single operator will not most likely fully utilize. Furthermore, cloud-scale data center networks often times have software and operational requirements that may not be well addressed by the switch vendors.

In this paper, we present our ongoing experiences on overcoming the complexity and scaling issues that we face when designing, developing, deploying and operating an in-house software built to manage and support a set of features required for data center switches of a large scale Internet content provider. We present FBOSS, our own data center switch software, that is designed with the basis on our *switch-as-a-server* and *deploy-early-and-iterate* principles. We treat software running on data center switches as any other software services that run on a commodity server. We also build and deploy only a minimal number of features and iterate on it. These principles allow us to rapidly iterate, test, deploy and manage FBOSS at scale. Over the last five years, our experiences show that FBOSS's design principles allow us to quickly build a stable and scalable network. As evidence, we have successfully grown the number of FBOSS instances running in our data center by over 30x over a two year period.

CCS CONCEPTS

• **Networks → Data center networks**; *Programming interfaces*; Routers;

KEYWORDS

FBOSS, Facebook, Switch Software Design, Data Center Networks, Network Management, Network Monitoring

ACM Reference Format:
Sean Choi, Boris Burkov, Alex Eckert, Tian Fang, Saman Kazemkhani, Rob Sherwood, Ying Zhang, and Hongyi Zeng. 2018. FBOSS: Building Switch Software at Scale. In *SIGCOMM '18: SIGCOMM 2018, August 20–25, 2018, Budapest, Hungary.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3230543.3230546

1 INTRODUCTION

The world's desire to produce, consume, and distribute online content is increasing at an unprecedented rate. Commensurate with this growth are equally unprecedented technical challenges in scaling the underlying networks. Large Internet content providers are forced to innovate upon all aspects of their technology stack, including hardware, kernel, compiler, and various distributed systems building blocks. A driving factor is that, at scale even a relatively modest efficiency improvement can have large effects. For us, our data center networks power a cloud-scale Internet content provider with billions of users, interconnecting hundreds of thousands of servers. Thus, it is natural and necessary to innovate on the software that runs on switches. [1]

Conventional switches typically come with software written by vendors. The software includes drivers for managing dedicated packet forwarding hardware (e.g., ASICs, FPGAs, or NPUs), routing protocols (e.g., BGP, OSPF, STP, MLAG), monitoring and debugging features (e.g., LLDP, BFD, OAM), configuration interfaces (e.g., conventional CLI, SNMP, NetConf, OpenConfig), and a long tail of other features needed to run a modern switch. Implicit in the vendor model is the assumption that networking requirements are *correlated* between customers. In other words, vendors are successful because they can create a small number of products and reuse them across many customers. However our network size and the rate of growth of the network (Figure 1) are unlike most other data center networks. Thus, they imply that our requirements are quite different from most customers.

*Work done while at Facebook, Inc.

[1] We use "switch" for general packet switching devices such as switches and routers. Our data center networks are fully Layer 3 routed similar to what is described in [36].

SIGCOMM '18, August 20–25, 2018, Budapest, Hungary

ACM ISBN 978-1-4503-5567-4/18/08…$15.00
https://doi.org/10.1145/3230543.3230546

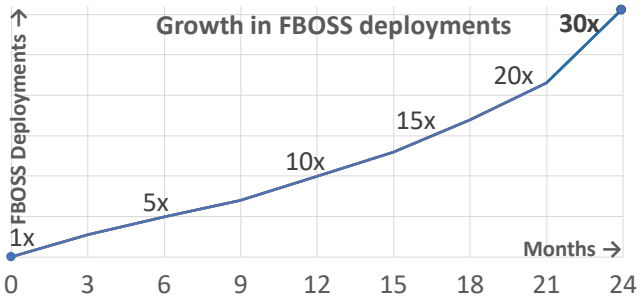

Figure 1: Growth in the number of switches in our data center over two year period as measured by number of total FBOSS deployments.

One of the main technical challenges in running large networks is managing complexity of excess networking features. Vendors supply common software intended for their entire customer base, thus their software includes the union of all features requested by all customers over the lifetime of the product. However, more features lead to more code and more code interactions, which ultimately lead to increased bugs, security holes, operational complexity, and downtime. To mitigate these issues, many data centers are designed for simplicity and only use a carefully selected subset of networking features. For example, Microsoft's SONiC focuses on building a "lean stack" in switches [33].

Another of our network scaling challenges is enabling a high-rate of innovation while maintaining network stability. It is important to be able to test and deploy new ideas at scale in a timely manner. However, inherent in the vendor-supplied software model is that changes and features are prioritized by how well they correlate across all of their customers. A common example we cite is IPv6 forwarding, which was implemented by one of our vendors very quickly due to widespread customer demand. However, an important feature to our operational workflow was fine-grained monitoring of IPv6, which we quickly implemented for our own operational needs. Had we left this feature to the demands of the customer market and to be implemented by the vendors, we would not have had this feature until over four years later, which was when the feature actually arrived to the market.

In recent years, the practice of building network switch components has become more open. First, network vendors emerged that do not build their own packet forwarding chips. Instead they rely on third-party silicon vendors, commonly known as "merchant silicons". Then, merchant silicon vendors along with box/chassis manufacturers have emerged that create a new, disaggregated ecosystem where networking hardware can be purchased without any software. As a result, it is now possible for end-customers to build a complete custom switch software stack from scratch.

Thanks to this trend, we started an experiment of building our in-house designed switch software five years ago. Our server fleet already runs thousands of different software services. We wanted to see if *we can run switch software in a similar way we run software services.* This model is quite different from how conventional networking software is managed. Table 1 summarizes the differences between the two high-level approaches using a popular analogy [17].

The result is Facebook Open Switching System (*FBOSS*), which is now powering a significant portion of our data center infrastructure. In this paper, we report on five years of experiences on building, deploying and managing FBOSS. The main goals of this paper are:

(1) Provide context about the internal workings of the software running on switches, including challenges, design trade-offs, and opportunities for improvement, both in the abstract for all network switch software and our specific pragmatic design decisions.

(2) Describe the design, automated tooling for deployment monitoring, and remediation methods of FBOSS.

(3) Provide experiences and illustrative problems encountered on managing a cloud-scale data center switch software.

(4) Encourage new research in the more accessible/open field of switch software and provide a vehicle, an open source version of FBOSS [4], for existing network research to be evaluated on real hardware.

The rest of the paper closely follows the structure of Table 1 and is structured as follows: We first provide a couple of our design principles that guide FBOSS's development and deployment (Section 2). Then, we briefly describe major hardware components that most data center switch software needs to manage (Section 3) and summarize the specific design decisions made in our system (Section 4). We then describe the corresponding deployment and management goals and lessons (Section 5, Section 6). We describe three operational challenges (Section 7) and then discuss how we have successfully overcome them. We further discuss various topics that led to our final design (Section 8) and provide a road map for future work (Section 9).

2 DESIGN PRINCIPLES

We designed FBOSS with two high-level design principles: (1) Deploy and evolve the software on our switches the same as we do our servers (*Switch-as-a-Server*). (2) Use early deployment and fast iteration to force ourselves to have a minimally complex network that only uses features that are strictly needed (*Deploy-Early-and-Iterate*). These principles have been echoed in the industry - a few other customized switch software efforts like Microsoft ACS [8]/SONiC [33] is based on similar motivation. However, one thing to note is

	Switch Software	General Software
Hardware (S3)	Closed, custom embedded systems. Limited CPU/Mem resources.	Open, general-purpose servers. Spare and fungible CPU/Mem resources
Release Cycle (S5)	Planned releases every 6-12 months.	Continuous deployment.
Testing (S5)	Manual testing, slow production roll out.	Continuous integration in production.
Resiliency Goals (S5)	High device-level availability with target of 99.999% device-level uptime.	High system-level availability with redundant service instances.
Upgrades (S6)	Scheduled downtime, manual update process.	Uninterrupted service, automated deployment.
Configuration (S6)	Decentralized and manually managed. Custom backup solutions.	Centrally controlled, automatically generated and distributed. Version controlled backups.
Monitoring (S6)	Custom scripting on top of SNMP counters.	Rich ecosystem of data collection, analytics and monitoring software libraries and tools.

Table 1: Comparison of development and operation patterns between conventional switch software and general software services, based on popular analogy [17].

that our design principles are specific to our own infrastructure. Data center network at Facebook has multiple internal components, such as Robotron [48],FbNet [46], Scuba [15] and Gorilla [42], that are meticulously built to work with one another, and FBOSS is no different. Thus, our design has our specific goal of easing the integration of FBOSS into our existing infrastructure, which ultimately means that it may not be generalized for any data center. Given this, we specifically focus on describing the effects of these design principle in terms of our software architecture, deployment, monitoring, and management.

2.1 Switch-as-a-Server

A motivation behind this principle comes from our experiences in building large scale software services. Even though many of the same technical and scaling challenges apply equally to switch software as to general distributed software systems, historically, they have have been addressed quite differently. For us, the general software model has been more successful in terms of reliability, agility, and operational simplicity. We deploy thousands of software services that are not feature-complete or bug-free. However, we carefully monitor our services and once any abnormality is found, we quickly make a fix, deploy the change. We found this practice to be highly successful in building and scaling our services.

For example, database software is an important part of our business. Rather than using a closed, proprietary vendor-provided solution that includes unnecessary features, we started an open source distributed database project and modified it heavily for internal use. Given that we have full access to the code, we can precisely customize the software for the desired feature set and thereby reduce complexity. Also, we make daily modifications to the code and, using the industry practices of continuous integration and staged deployment, are able to rapidly test and evaluate the changes in production. In addition, we run our databases on commodity servers, rather than running them on custom hardware, so that both the software and the hardware can be easily controlled and debugged. Lastly, since the code is open source, we make our changes available back to the world and benefit from discussions and bug fixes produced by external contributors.

Our experiences with general software services showed that this principle is largely successful in terms of scalability, code reuse, and deployment. Therefore, we designed FBOSS based on the same principle. However, since data center networks have different operational requirements than a general software service, there are a few caveats to naively adopting this principle that are mentioned in Section 8.

2.2 Deploy-Early-and-Iterate

Our initial production deployments were intentionally lacking in features. Bucking conventional network engineering wisdom, we went into production without implementing a long list of "must have" features, including control plane policing, ARP/NDP expiration, IP fragmentation/reassembly, or Spanning Tree Protocol (STP). Instead of implementing these features, we prioritized on building the infrastructure and tooling to efficiently and frequently update the switch software, e.g., the warm boot feature (Section 7.1).

Keeping with our motivation to evolve the network quickly and reduce complexity, we hypothesized that we could dynamically *derive the actual minimal network requirements* by iteratively deploying switch software into production, observing what breaks, and quickly rolling out the fixes. By starting small and relying on application-level fault tolerance, a small initial team of developers were able to go from nothing to code running in production in an order of magnitude fewer person-years than in typical switch software development.

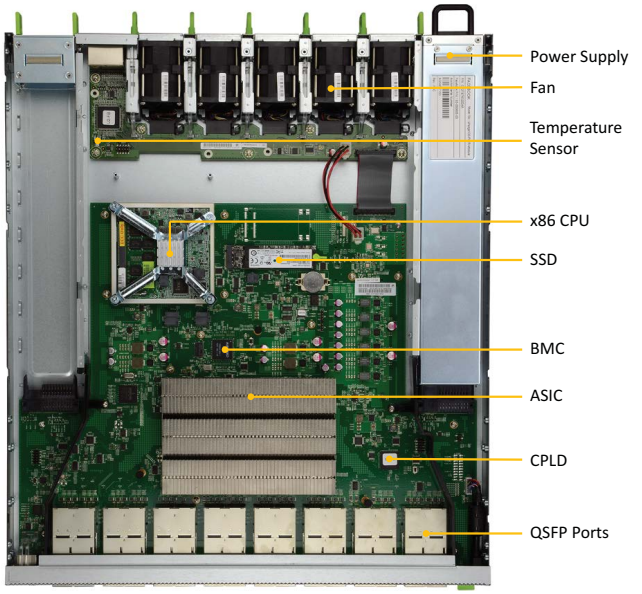

Figure 2: A typical data center switch architecture.

Perhaps more importantly, using this principle, we were able to derive and build the simplest possible network for our environment and have a positive impact on the production network sooner. For example, when we discovered that lack of control plane policing was causing BGP session time-outs, we quickly implemented and deployed it to fix the problem. By having positive impact to the production network early, we were able to make a convincing case for additional engineers and with more help. To date, we still do not implement IP fragmentation/reassembly, STP, or a long list of widely believed "must have" features.

3 HARDWARE PLATFORM

To provide FBOSS's design context, we first review what typical switch hardware contains. Some examples are a switch application-specific integrated circuit (ASIC), a port subsystem, a Physical Layer subsystem (PHY), a CPU board, complex programmable logic devices, and event handlers. The internals of a typical data center switch are shown in Figure 2 [24].

3.1 Components

Switch ASIC. Switch ASIC is the important hardware component on a switch. It is a specialized integrated circuit for fast packet processing, capable of switching packets up to 12.8 terabits per second [49]. Switches can augment the switch ASIC with other processing units, such as FPGAs [53] or x86 CPUs, at a far lower performance [52]. A switch ASIC has multiple components: memory, typically either CAM, TCAM or SRAM [19], that stores information that needs to

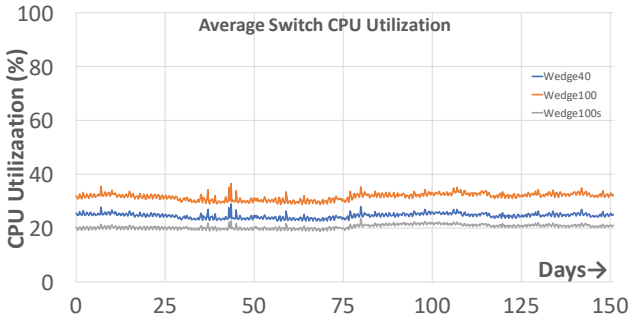

Figure 3: Average CPU utilization of FBOSS on across various type of switches in one of Facebook's data centers.

be quickly accessed by the ASIC; a parse pipeline, consisting of a parser and a deparser, which locates, extracts, saves the interesting data from the packet, and rebuilds the packet before egressing it [19]; and match-action units, which specify how the ASIC should process the packets based on the data inside the packet, configured packet processing logic and the data inside the ASIC memory.

PHY. The PHY is responsible for connecting the link-layer device, such as the ASIC, to the physical medium, such as an optical fiber, and translating analog signals from the link to digital Ethernet frames. In certain switch designs, PHY can be built within the ASIC. At high-speeds, electrical signal interference is so significant that it causes packet corruption inside a switch. Therefore, complex noise reduction techniques, such as PHY tuning [43], are needed. PHY tuning controls various parameters such as preemphasis, variable power settings, or the type of Forward Error Correction algorithm to use.

Port Subsystem. The port subsystem is responsible for reading port configurations, detecting the type of ports installed, initializing the ports, and providing interfaces for the ports to interact with the PHY. Data center switches house multiple Quad Small Form-factor Pluggable (QSFP) ports. A QSFP port is a compact, hot-pluggable transceiver used to interface switch hardware to a cable, enabling data rates up to 100Gb/s. The type and the number of QSFP ports are determined by the switch specifications and the ASIC.

FBOSS interacts with the port subsystem by assigning dynamic lane mapping and adapting to port change events. Dynamic lane mapping refers to mapping multiple lanes in each of the QSFPs to appropriate port virtual IDs. This allows changing of port configurations without having to restart the switch. FBOSS monitors the health of the ports and once any abnormality is detected, FBOSS performs remediation steps, such as reviving the port or rerouting the traffic to a live port.

CPU Board. There exists a CPU board within a switch that runs a microserver [39]. A CPU board closely resembles a commodity server, containing a commodity x86 CPU, RAM

and a storage medium. In addition to these standard parts, a CPU board has a PCI-E interconnect to the switch ASIC that enables quick driver calls to the ASIC. The presence of a x86 CPU enables installation of commodity Linux to provide general OS functionalities. CPUs within switches are conventionally underpowered compared to a server-grade CPUs. However, FBOSS is designed under the assumption that the CPUs in the switches are as powerful as server-grade CPUs, so that the switch can run as much required server services as possible. Fortunately, we designed and built our data center switches in-house, giving us flexibility to choose our own CPUs that fits within our design constraints. For example, our Wedge 100 switch houses an Quad Core Intel E3800 CPU. We over-provision the CPU, so that the switch CPU runs under 40% utilization to account for any bursty events from shutting down the switch. Such design choice can be seen in various types of switches that we deploy, as seen in Figure 3. The size allocated for the CPU board limited us from including an even powerful CPU [24].

Miscellaneous Board Managers. A switch offloads miscellaneous functions from the CPU and the ASIC to various components to improve overall system performance. Two examples of such components are Complex Programmable Logic Device (CPLD) and the Baseboard Management Controller (BMC). The CPLDs are responsible for status monitoring, LED control, fan control and managing front panel ports. The BMC is a specialized system-on-chip that has its own CPU, memory, storage, and interfaces to connect to sensors and CPLDs. BMC manages power supplies and fans. It also provides system management functions such as remote power control, serial over LAN, out-of-band monitoring and error logging, and a pre-OS environment for users to install an OS onto the microsever. The BMC is controlled by custom software such as OpenBMC [25].

The miscellaneous board managers introduce additional complexities for FBOSS. For example, FBOSS retrieves QSFP control signals from the CPLDs, a process that requires complex interactions with the CPLD drivers.

3.2 Event Handlers

Event handlers enable the switch to notify any external entities of its internal state changes. The mechanics of a switch event handler are very similar to any other hardware-based event handlers, thus the handlers can be handled in both synchronous or asynchronous fashion. We discuss two switch specific event handlers: the link event handler, and the slow path packet handler.

Link Event Handler. The link event handler notifies the ASIC and FBOSS of any events that occur in the QSFP ports or the port subsystem. Such events include link on and off events and change in link configurations. The link status handler is usually implemented with a busy polling method where the switch software has an active thread that constantly monitors the PHY for link status and then calls the user-supplied callbacks when changes are detected. FBOSS provides a callback to the link event handler, and syncs its local view of the link states when the callback is activated.

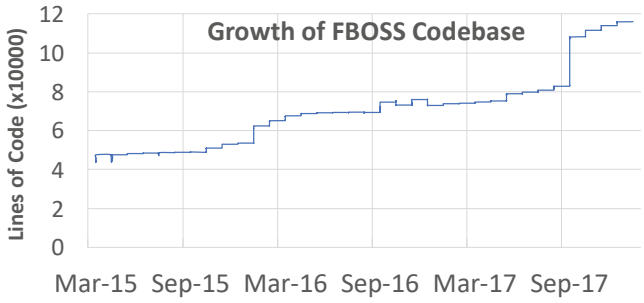

Figure 4: Growth of FBOSS open source project.

Slow Path Packet Handler. Most switches allow packets to egress to a designated CPU port, the *slow path*. Similar to the link status handler, the slow packet handler constantly polls the CPU port. Once a packet is received at a CPU port, the slow path packet handler notifies the switch software of the captured packet and activates the supplied callback. The callback is supplied with various information, which may include the actual packet that is captured. This allows the slow path packet handler to greatly extend a switch's feature set, as it enables custom packet processing without having to change the data plane's functionality. For example, one can sample a subset of the packets for in-band monitoring or modify the packets to include custom information. However, as indicated by its name, the slow path packet handler is too slow to perform custom packet processing at line rate. Thus it is only suitable for use cases that involve using only a small sample of the packets that the switch receives.

4 FBOSS

To manage the switches as described in Section 3, we developed FBOSS, vendor-agnostic switch software that can run on a standard Linux distribution. FBOSS is currently deployed to both ToR and aggregation switches in our production data centers. FBOSS's code base is publicly available as an open source project and it is supported by a growing community. As of January 2018, a total of 91 authors have contributed to the project and the codebase now spans 609 files and 115,898 lines of code. To give a scope of how much lines of code a feature may take to implement, implementing link aggregation in FBOSS required 5,932 lines of newly added code. Note that this can be highly variable depending on the feature of interest, and some features may not be easily divisible from one another. Figure 4 shows the growth of the open source

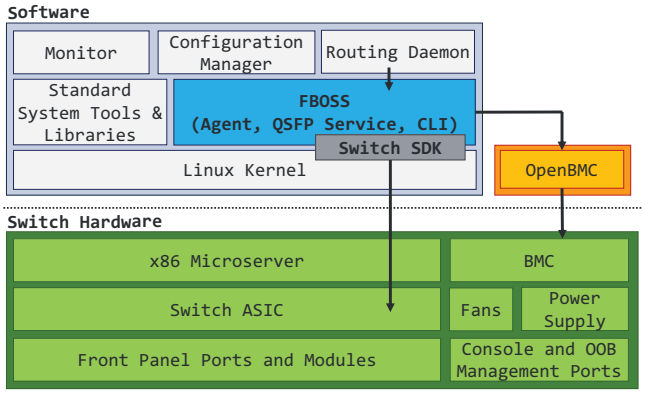

Figure 5: Switch software and hardware components.

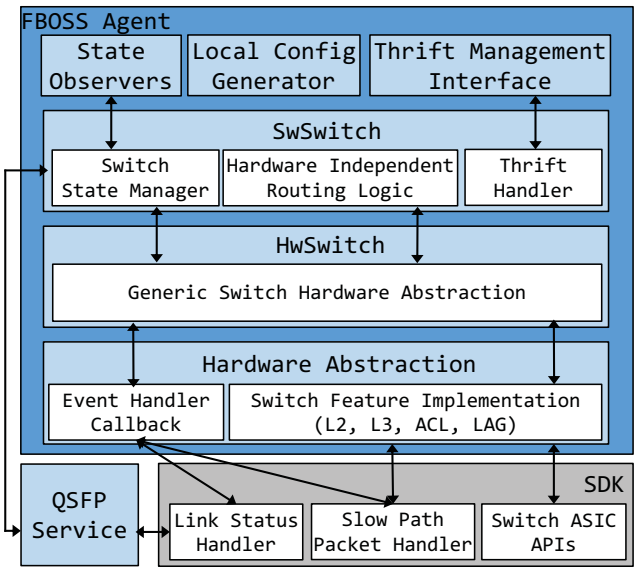

Figure 6: Architecture overview of FBOSS.

project since its inception. The big jump in the size of the codebase that occurred in September of 2017 is a result of adding a large number of hardcoded parameters for FBOSS to support a particular vendor NIC.

FBOSS is responsible for managing the switch ASIC and providing a higher level remote API that translates down to specific ASIC SDK methods. The external processes include management, control, routing, configuration, and monitoring processes. Figure 5 illustrates FBOSS, other software processes and hardware components in a switch. Note that in our production deployment, FBOSS share the same Linux environment (e.g., OS version, packaging system) as our server fleet, so that we can utilize the same system tools and libraries on both servers and switches.

4.1 Architecture

FBOSS consists of multiple interconnected components that we categorize as follows: Switch Software Development Kit (SDK), HwSwitch, Hardware abstraction layer, SwSwitch, State observers, local config generator, a Thrift [2] management interface and QSFP service. FBOSS agent is the main process that runs most of FBOSS's functionalities. The switch SDK is bundled and compiled with the FBOSS agent, but is provided externally by the switch ASIC vendor. All of the other components besides the QSFP service, which runs as its own independent process, reside inside the FBOSS agent. We discuss each component in detail, except the local config generator, which we will discuss in Section 6.

Switch SDK. A switch SDK is ASIC vendor-provided software that exposes APIs for interacting with low-level ASIC functions. These APIs include ASIC initialization, installing forwarding table rules, and listening to event handlers.

HwSwitch. The HwSwitch represents an abstraction of the switch hardware. The interfaces of HwSwitch provide generic abstractions for configuring switch ports, sending and receiving packets to these ports, and registering callbacks for state changes on the ports and packet input/output events that occur on these ports. Aside from the generic abstractions, ASIC specific implementations are pushed to the hardware abstraction layer, allowing switch-agnostic interaction with the switch hardware. While not a perfect abstraction, FBOSS has been ported to two ASIC families and more ports are in progress. An example of a HwSwitch implementation can be found here [14].

Hardware Abstraction Layer. FBOSS allows users to easily add implementation that supports a specific ASIC by extending the HwSwitch interface. This also allows easy support for multiple ASICs without making changes to the main FBOSS code base. The custom implementation must support the minimal set of functionalities that are specified in HwSwitch interface. However, given that HwSwitch only specifies a small number of features, FBOSS allows custom implementation to include additional features. For example, open-source version of FBOSS implements custom features such as specifying link aggregation, adding ASIC status monitor, and configuring ECMP.

SwSwitch. The SwSwitch provides the hardware-independent logic for switching and routing packets, and interfaces with the HwSwitch to transfer the commands down to the switch ASIC. Some example of the features that SwSwitch provides are, interfaces for L2 and L3 tables, ACL entries, and state management.

State Observers. SwSwitch make it possible to implement low-level control protocols such as ARP, NDP, LACP, and

```
1 struct L2EntryThrift {
2   1: string mac,
3   2: i32 port,
4   3: i32 vlanID,
5 }
6 list<L2EntryThrift> getL2Table()
7   throws (1: error.FBossBaseError error)
```

Figure 7: Example of Thrift interface definition to retrieve L2 entries from the switch.

LLDP, by keeping protocols apprised of state changes[2]. The protocols are notified of state changes via a mechanism called *state observation*. Specifically, any object at the time of its initialization may register itself as a State Observer. By doing so, every future state change invokes a callback provided by the object. The callback provides the state change in question, allowing the object to react accordingly. For example, NDP registers itself as a State Observer so that it may react to port change events. In this way, the state observation mechanism allows protocol implementations to be decoupled from issues pertaining to state management.

Thrift Management Interface. We run our networks in a split control configuration. Each FBOSS instance contains a local control plane, running protocols such as BGP or OpenR [13], on a microserver that communicates with a centralized network management system through a Thrift management interface. The types of messages that are sent between them are as in the form seen in Figure 7. The full open-source specification of the FBOSS Thrift interface is also available [5]. Given that the interfaces can be modified to fit our needs, Thrift provides us with a simple and flexible way to manage and operate the network, leading to increased stability and high availability. We discuss the details of the interactions between the Thrift management interface and the centralized network management system in Section 6.

QSFP Service. The QSFP service manages a set of QSFP ports. This service detects QSFP insertion or removal, reads QSFP product information (e.g., manufacturer), controls QSFP hardware function (i.e., change power configuration), and monitors the QSFPs. FBOSS initially had the QSFP service within the FBOSS agent. However, as the service continues to evolve, we must restart the FBOSS agent and the switch to apply the changes. Thus, we separated the QSFP service into a separate process to improve FBOSS's modularity and reliability. As the result, FBOSS agent is more reliable as any restarts or bugs in QSFP service do not affect the agent directly. However, since QSFP service is a separate process, it needs separate tools for packaging, deployment, and monitoring. Also, careful process synchronization between QSFP service and FBOSS agent is now required.

[2]The other functions include control packets transmission and reception and programming of switch ASIC and hardware.

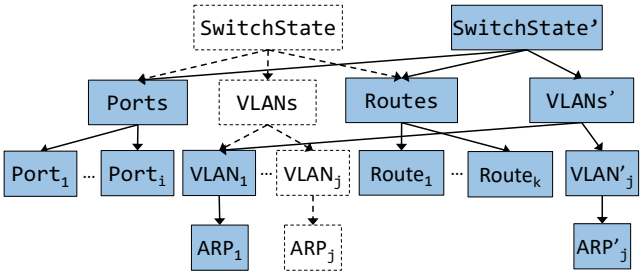

Figure 8: Illustration of FBOSS's switch state update through copy-on-write tree mechanism.

4.2 State Management

FBOSS's software state management mechanism is designed for high concurrency, fast reads, and easy and safe updates. The state is modeled as a versioned copy-on-write tree [37]. The root of the tree is the main switch state class, and each child of the root represents a different category of the switch state, such as ports or VLAN entries. When an update happens to one branch of the tree, every node in the branch all the way to the root is copied and updated if necessary. Figure 8 illustrates a switch state update process invoked by an update on an VLAN ARP table entry. We can see that only the nodes and the links starting from the modified ARP table up to the root are recreated. While the creation of the new tree occurs, the FBOSS agent still interacts with the prior states without needing to capture any locks on the state. Once the copy-on-write process completes for the entire tree, FBOSS reads from the new switch state.

There are multiple benefits to this model. First, it allows for easy concurrency, as there are no read locks. Reads can still continue to happen while a new state is created, and the states are only created or destroyed and never modified. Secondly, versioning of states is much simpler. This allows easier debugging, logging, and validation of each state and its transitions. Lastly, since we log all the state transitions, it is possible to perform a restart and then restore the state to its pre-restart form. There also are some disadvantages to this model. Since every state change results in a new switch state object, the update process requires more processing. Secondly, implementation of switch states is more complex than simply obtaining locks and updating a single object.

Hardware Specific State. The hardware states are the states that are kept inside the ASIC itself. Whenever a hardware state needs to be updated in software, the software must call the switch SDK to retrieve the new states. The FBOSS HwSwitch obtains both read and write locks on the corresponding parts of the hardware state until the update completes. The choice of lock based state updates may differ based on the SDK implementation.

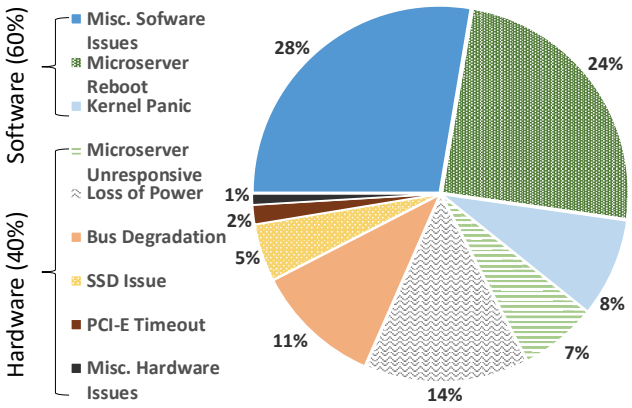

Figure 9: Culprits of switch outages over a month.

5 TESTING AND DEPLOYMENT

Switch software is conventionally developed and released by switch vendors and is closed and proprietary. Therefore, a new release to the switch software can take months, in lengthy development and manual QA test cycles. In addition, given that software update cycles are infrequent, an update usually contains a large number of changes that can introduce new bugs that did not exist previously. In contrast, typical large scale software deployment processes are automated, fast, and contain a smaller set of changes per update. Furthermore, feature deployments are coupled with automated and incremental testing mechanisms to quickly check and fix bugs. Our outage records (Figure 9) show that about 60% of the switch outages are caused by faulty software. This is similar to the known rate of software failures in data center devices, which is around 51% [27]. To minimize the occurrences and impact of these outages, FBOSS adopts agile, reliable and scalable large scale software development and testing schemes.

Instead of using existing automatic software deployment framework like Chef [3] or Jenkins [6], FBOSS employs its own deployment software called ***fbossdeploy***. One of the main reason for developing our own deployment software is to allow for a tighter feedback loop with existing external monitors. We have several existing external monitors that continuously check the health of the network. These monitors check for attributes such as link failures, slow BGP convergence times, network reachability and more. While existing deployment frameworks that are built for deploying generic software are good at preventing propagation of software related bugs, such as deadlocks or memory leaks, they are not built to detect and prevent network-wide failures, as these failures may be hard to detect from a single node. Therefore, *fbossdeploy* is built to react quickly to the network-wide failures, such as reachability failures, that may occur during deployment.

The FBOSS deployment process is very similar to other continuous deployment processes [22] and is split into three distinct parts: continuous canary, daily canary and staged deployment. Each of these parts serves a specific purpose to ensure a reliable deployment. We currently operate roughly at a monthly deployment cycle, which includes both canaries and staged deployment, to ensure high operational stability.

Continuous Canary. The continuous canary is a process that automatically deploys all newly committed code in the FBOSS repository to a small number of switches that are running in production, around 1-2 switches per each type of switch, and monitors the health of the switch and the adjacent switches for any failures. Once a failure is detected, continuous canary will immediately revert the latest deployment and restore the last stable version of the code. Continuous canary is able to quickly catch errors related to switch initialization, such as issues with warm boot, configuration errors and unpredictable race conditions.

Daily Canary. The daily canary is a process that follows continuous canary to test the new commit at a longer timescale with more switches. Daily canary runs once a day and deploys the latest commit that has passed the continuous canary. Daily canary deploys the commit to around 10 to 20 switches per each type of the switch. Daily canary runs throughout the day to capture bugs that slowly surface over time, such as memory leaks or performance regressions in critical threads. This is the final phase before a network-wide deployment.

Staged Deployment. Once daily canary completes, a human operator intervenes to push the latest code to all of the switches in production. This is the only step of the entire deployment process that involves an human operator and roughly takes about a day to complete entirely. The operator runs a deployment script with the appropriate parameters to slowly push the latest code into the subset of the switches at a time. Once the number of failed switches exceed a preset threshold, usually around 0.5% of the entire switch fleet, the deployment script stops and asks the operator to investigate the issues and take appropriate actions. The reasons for keeping the final step manual are as follows: First, a single server is fast enough to deploy the code to all of the switches in the data center, meaning that the deployment process is not bottlenecked by one machine deploying the code. Secondly, it gives fine grained monitoring over the unpredicted bugs that may not be caught by the existing monitors. For example, we fixed unpredicted and persistent reachability losses, such as inadvertently changing interface IP or port speed configurations and transient outages like as port flaps, that we found during staged deployment. Lastly, we are still improving our testing, monitoring and deployment system. Thus, once the test coverage and automated remediation is within a comfortable range, we plan automate the last step as well.

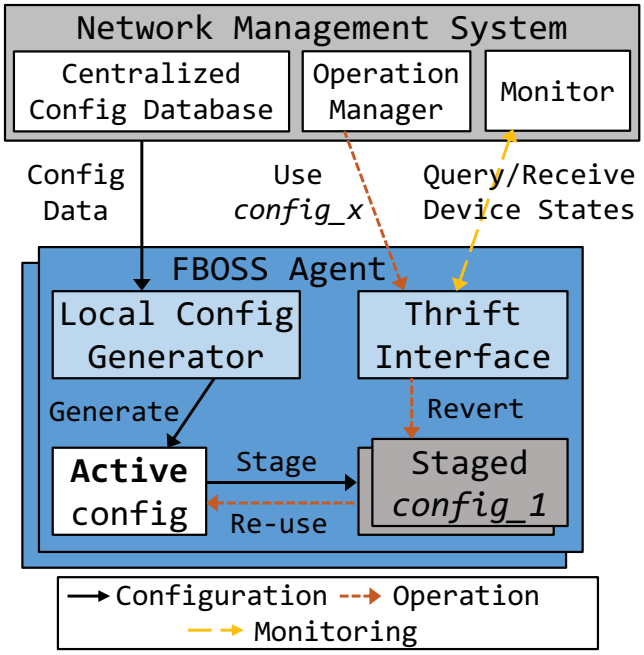

Figure 10: FBOSS interacts with a central network management system via the Thrift management interface.

6 MANAGEMENT

In this section, we present how FBOSS interacts with management system and discuss the advantages of FBOSS's design from a network management perspective. Figure 10 shows a high-level overview of the interactions.

6.1 Configurations

FBOSS is designed to be used in a highly controlled data center network with a central network manager. This greatly simplifies the process of generation and deployment of network configurations across large number of switches.

Configuration Design. The configuration of network devices is highly standardized in data center environments. Given a specific topology, each device is automatically configured by using templates and auto-generated configuration data. For example, the IP address configurations for a switch is determined by the type of the switch (e.g., ToR or aggregation), and its upstream/downstream neighbors in the cluster.

Configuration Generation and Deployment. The configuration data is generated by our network management system called Robotron [48] and is distributed to each switch. The local config generator in FBOSS agent then consumes the configuration data and creates an active config file. If any modification is made to the data file, a new active config file is generated and the old configuration is stored as a staged config file. There are multiple advantages to this configuration process. First, it disallows multiple entities from modifying the configuration concurrently, which limits inconsistencies in the configuration. Secondly, it makes the configuration reproducible and deterministic, since the configurations are versioned and FBOSS agent always reads the latest configuration upon restarts. And lastly, it avoids manual configuration errors. On the other hand, there are also disadvantages to our fully automated configuration system - it lacks a complex human interactive CLI, which makes manual debugging difficult; also, there is no support for incremental configuration changes, which makes each configuration change require a restart of the FBOSS agent.

6.2 Draining

Draining is the process safely removing an aggregation switch from its service. ToR switches are generally not drained, unless all of the services under the ToR switch are drained as well. Similarly, undraining is the process of restoring the switch's previous configuration and bringing it back into service. Due to frequent feature updates and deployments performed on a switch, draining and undraining a switch is one of the major operational tasks that is performed frequently. However, draining is conventionally a difficult operational task, due to tight timing requirements and simultaneous configuration changes across multiple software components on the switch [47]. In comparison, FBOSS's draining/undraining operation is made much simpler thanks to the automation and the version control mechanism in the configuration management design. Our method of draining a switch is as follows: **(1)** FBOSS agent retrieves the drained BGP configuration data from a central configuration database. **(2)** The central management system triggers the draining process via the Thrift management interface. **(3)** The FBOSS agent activates the drained config and restarts the BGP daemon with the drained config. As for the undraining process, we repeat the above steps, but with an undrained configuration. Then, as a final added step, the management system pings the FBOSS agent and queries the switch statistics to ensure that the undraining process is successful. Draining is an example where FBOSS's Thrift management interface and the centrally managed configuration snapshots significantly simplify an operational task.

6.3 Monitoring and Failure Handling

Traditionally, data center operators use standardized network management protocols, such as SNMP [21], to collect switch statistics, such as CPU/memory utilization, link load, packet loss, and miscellaneous system health, from the vendor network devices. In contrast, FBOSS allows external systems to collect switch statistics through two different interfaces: a Thrift management interface and Linux system logs. The Thrift management interface serves the queries in the form specified in the Thrift model. This interface is mainly used to

monitor high-level switch usage and link statistics. Given that FBOSS runs as a Linux process, we can also directly access the system logs of the switch microserver. These logs are specifically formatted to log the category events and failures. This allows the management system to monitor low-level system health and hardware failures. Given the statistics that it collects, our monitoring system, called FbFlow [46], stores the data to a database, either Scuba [15] or Gorilla [42], based on the type of the data. Once the data is stored, it enables our engineers to query and analyze the data at a high level over a long time period. Monitoring data, and graphs such as Figure 3, can easily be obtained by the monitoring system.

To go with the monitoring system, we also implemented an automated failure remediation system. The main purpose of the remediation system is to automatically detect and recover from software or hardware failures. It also provides deeper insights for human operators to ease the debugging process. The remediation process is as follows. Once a failure is detected, the remediation system automatically categorizes each failure to a set of known root causes, applies remediations if needed, and logs the details of the outage to a datastore. The automatic categorization and remediation of failures allows us to focus our debugging efforts on undiagnosed errors rather than repeatedly debugging the same known issues. Also, the extensive log helps us drive insights like isolating a rare failure to a particular hardware revision or kernel version.

In summary, our approach has the following advantages:

Flexible Data Model. Traditionally, supporting a new type of data to collect or modifying an existing data model requires modifications and standardization of the network management protocols and then time for vendors to implement the standards. In contrast, since we control the device, monitoring data dissemination via FBOSS and the data collection mechanism through the management system, we can easily define and modify the collection specification. We explicitly define the fine-grained counters we need and instrument the devices to report those counters.

Improved Performance. Compared to conventional monitoring approaches, FBOSS has better performance as the data transfer protocol can be customized to reduce both collection time and network load.

Remediation with Detailed Error Logs. Our system allows the engineers to focus on building remediation mechanisms for unseen bugs, which consequently improves network stability and debugging efficiency.

7 EXPERIENCES

While the experiences of operating a data center network with custom switch software and hardware has been mostly satisfactory, we faced outages that are previously unseen and are unique to our development and deployment model.

7.1 Side Effect of Infrastructure Reuse

For improved efficiency, our data centers deploy a network topology with a single ToR switch, which implies that the ToR switches are a single point of failure for the hosts in the rack. As a result, frequent FBOSS releases made on the ToR switches need to be non-disruptive to ensure availability of the services running on those hosts. To accomplish this, we use an ASIC feature called "warm boot". Warm boot allows FBOSS to restart without affecting the forwarding tables within the ASIC, effectively allowing the data plane to continue to forward traffic while the control plane is being restarted. Although this feature is highly attractive and has allowed us to achieve our desired release velocity, it also greatly complicates the state management between FBOSS, routing daemons, switch SDK and the ASIC. Thus, we share a case where warm boot and our code reuse practices have resulted in a major outage.

Despite the fact that we have a series of testing and monitoring process for new code deployments, it is inevitable for bugs to leak into data center-wide deployments. The most difficult type of bugs to debug are the ones that appear rarely and inconsistently. For example, our BGP daemon has a graceful restart feature to prevent warm boots from affecting the neighbor devices when BGP sessions are torn down by FBOSS restarts or failures [38]. The graceful restart has a timeout before declaring BGP sessions are broken, which effectively puts a time constraint on the total time a warm boot operation can take. In one of our deployments, we found that the Kerberos [7] library, which FBOSS and many other software services, use to secure communication between servers, caused outages for a small fraction of switches in our data center. We realized that the reason for the outages is that the library often took a long time to join the FBOSS agent thread. Since the timing and availability constraints for other software services are more lenient than FBOSS's warm boot requirements, existing monitors were not built detect such rare performance regressions.

Takeaway: Simply reusing widely-used code, libraries or infrastructure that are tuned for generic software services may not work out of the box with switch software.

7.2 Side Effect of Rapid Deployment

During the first few months of our initial FBOSS deployment, we occasionally encountered unknown cascading outages of multiple switches. The outage would start with a single device and would spread to nearby devices, resulting in very high packet loss within a cluster. Sometimes the network would recover on its own, sometimes not. We realized that the outages were more likely to occur if a deployment would go awry, yet they were quite difficult to debug because we

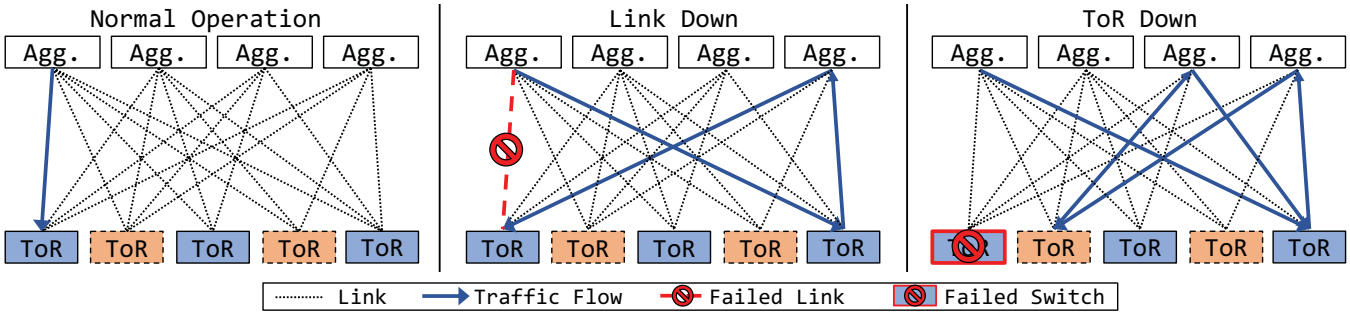

Figure 11: Overview of cascading outages seen by a failed ToR switch within a backup group.

had deployed a number of new changes simultaneously as it was our initial FBOSS deployment.

We eventually noticed that the loss was usually limited to a multiple of 16 devices. This pointed towards a configuration in our data center called *backup groups*. Prior to deploying FBOSS, the most common type of failure within our data center was a failure of a single link leading to a black-holing of traffic [36]. In order to handle such failures, a group (illustrated on the left side of Figure 11) of ToR switches are designated to provide backup routes if the most direct route to a destination becomes unavailable. The backup routes are pre-computed and statically configured for faster failover.

We experienced an outage where a failure of a ToR resulted in a period where packets ping pong between the backup ToRs and the aggregation switches, incorrectly assuming that the backup routes are available. This resulted in a loop in the backup routes. The right side of Figure 11 illustrates the creation of path loops. The loop eventually resulted in huge CPU spikes on all the backup switches. The main reason for the CPU spikes was because FBOSS was not correctly removing the failed routes from the forwarding table and was also generating TTL expired ICMP packets for all packets that had ping-ponged back and forth 255 times. Given that we had not seen this behavior before, we had no control plane policing in place and sent all packets with TTL of 0 to the FBOSS agent. The rate the FBOSS agent could process these packets was far lower than the rate we were receiving the frames, so we would fall further and further behind and starve out the BGP keep-alive and withdraw messages we need for the network to converge. Eventually BGP peerings would expire, but since we were already in the looping state, it often made the matters worse and caused the starvation to last indefinitely. We added a set of control plane fixes and the network became stable even through multiple ToR failures.

Takeaway: A feature that works well for conventional networks may not work well for networks deploying FBOSS. This is a side effect of rapid deployment, as entire switch outages are more frequently than in conventional networks. Thus, one must be careful in adopting features that are known to be stable in conventional networks.

7.3 Resolving Interoperability Issues

Although we developed and deployed switches that are built in-house, we still need the switches and FBOSS to interoperate with different types of network devices for various reasons. We share our experiences where the design of FBOSS allowed an interoperability issue to be quickly resolved.

When configuring link aggregation between FBOSS and a particular line of vendor devices, we discovered that flapping the logical aggregate interface on the vendor device could disable all IP operations on that interface. A cursory inspection revealed that, while the device had expectantly engaged in Duplicate Address Detection (DAD) [50] for the aggregate interface's address, it had unexpectedly detected a duplicate address in the corresponding subnet. This behavior was isolated to a race condition between LACP and DAD's probe, wherein an artifact of the hardware support for link aggregation could cause DAD's *Neighbor Solicitation* packet to be looped back to the vendor switch. In accordance with the DAD specification, the vendor device had interpreted the looped back *Neighbor Solicitation* packet as another node engaging in DAD for the same address, which the DAD specification mandates should cause the switch to disable IP operation on the interface on which DAD has been invoked. We also found that interconnecting the same vendor device with a different vendor's switch would exhibit the same symptom.

Flapping of interfaces is a step performed by our network operators during routine network maintenance. To ensure that the maintenance could still be performed in a non-disruptive manner, we modified the FBOSS agent to avoid the scenario described above. In contrast, in response to our report of this bug to the vendor, whose switch exhibited the same behavior as ours, the vendor recommended the other vendors to implement an extension to DAD. By having entire control over our switch software, we were able to quickly provide what's necessary for our network.

Takeaway: Interoperability issues are common in networks with various network devices. FBOSS allows us to quickly diagnose and fix the problem directly, instead of waiting for vendor updates or resorting to half-baked solutions.

8 DISCUSSION

Existing Switch Programming Standards. Over time, many software standards have been proposed to open up various aspects of the software on the switch. On the academic side, there are decades of approaches to open various aspects of switches, including active networking [35], FORCES [32], PCE [26], and OpenFlow [40]. On the industry side, upstart vendors have tried to compete with incumbents on being *more open* (e.g., JunOS's SDK access program, Arista's SDK program) and the incumbents have responded with their own open initiatives (e.g., I2RS, Cisco's OnePK). On both the academic and industry sides, there also are numerous control plane and management plane protocols that similarly try to make the switch software more programmable/configurable. Each of these attempts have their own set of trade-offs and subset of supported hardware. Thus, one could argue that some synthesis of these standards could be "the one perfect API" that gives us the functionalities we want. So, why didn't we just use/improve upon one of these existing standards?

The problem is that these existing standards are all "top down": they are all additional software/protocols layered on top of the existing vendor software rather than entirely replacing it. That means that if ever we wanted to change the underlying unexposed software, we would still be limited by what our vendors would be willing to support and on their timelines. By controlling the entire software stack "bottom up", we can control all the possible states and code on the switch and can expose any API anyway we want at our own schedule. Even more importantly, we can experiment with the APIs we expose and evolve them over time for our specific needs, allowing us to quickly meet our production needs.

FBOSS as a Building Block for Larger Switches. While originally developed for ToR, single-ASIC style switches, we have adapted FBOSS as a building block to run larger, multi-ASIC chassis switches as well. We have designed and deployed our own chassis-based switch with removable line cards that supports 128x100Gbps links with full bisection connectivity. Internally, this switch is composed of eight line cards each with their own CPU and ASIC, connected in a logic CLOS topology to four fabric cards also with their own CPU and ASIC.

We run an instance of FBOSS on each of the twelve (eight line cards plus four fabric cards) CPUs and have them peer via BGP internally to the switch, logically creating a single high-capacity switch that runs the aggregation layers of our data centers. While appearing to be a new hardware design, the data plane of our switches follows closely conventional vendor-sourced chassis architectures. The main difference is that we do not deploy additional servers to act as supervisor cards and instead leverage our larger data center automation tooling and monitoring. While this design does not provide the same single logical switch abstraction that is provided by conventional vendor switches, it allows us to jump to larger switch form factors with no software architectural changes.

Implicit and Circular Dependency. One subtle but important problem we discovered when trying to run our switches like a server was hidden and implicit circular dependencies on the network. Specifically, all servers on our fleet run a standard set of binaries and libraries for logging, monitoring, and etc. By design, we wanted to run these existing software on our switches. Unfortunately, in some cases, the software built for the servers implicitly depended on the network and when the FBOSS code depended on them, we created a circular dependency that prevented our network from initializing. Worse yet, these situations would only arise during other error conditions (e.g., when a daemon crash) and were hard to debug. In one specific case, we initially deployed the FBOSS onto switches using the same task scheduling and monitoring software used by other software services in our fleet, but we found that this software required access to the production network before it would run. As a result, we had to decouple our code from it and write our own custom task scheduling software to specifically manage FBOSS deployments. While this was an easier case to debug, as each software package evolves and is maintained independently, there is a constant threat of well-meaning but server focused developers adding a subtle implicit dependency on the network. Our current solution is to continue to fortify our testing and deployment procedures.

9 FUTURE WORK

Partitioning FBOSS Agent. FBOSS agent currently is a single monolithic binary consisting of multiple features. Similar to how QSFP service was separated to improve switch reliability, we plan to further partition FBOSS agent into smaller binaries that runs independently. For example, if state observers exist as external processes that communicates with FBOSS agent, any events that can overwhelms the state observers no long brings FBOSS agent down with it.

Novel Experiments. One of our main goals for FBOSS is to allow more and faster experimentation. We are currently experimenting with custom routing protocols, stronger slow path isolation (e.g., to deal with buggy experiments), micro-burst detection, macro-scale traffic monitoring, big data analytic of low-level hardware statistics to infer failure detection, and a host of other design elements. By making FBOSS open source and our research more public, we hope to aid researchers with

tools and ideas to directly implement novel research ideas on production ready software and hardware.

Programmable ASIC Support. FBOSS is designed to easily support multiple types of ASICs simultaneously. In fact, FBOSS successfully iterated through different versions of ASICs without any huge design changes. With the recent advent of programmable ASICs, we believe that it will be useful for FBOSS to support programmable ASICs [19] and the language to program these ASICs, such as P4 [18].

10 RELATED WORK

Existing Switch Software. There are various proprietary switch software implementations, often referred to as "Network OS", such as Cisco NX-OS [12] or Juniper JunOS [41], yet FBOSS is quite different from them. For example, FBOSS allows full access to the switch Linux, giving users flexibility to run custom processes for management or configuration. In comparison, conventional switch software are generally accessed through their own proprietary interfaces.

There is also various open-source switch software that runs on Linux, such as Open Network Linux (ONL) [30], OpenSwitch [11], Cumulus Linux [20] and Microsoft SONiC [33]. FBOSS is probably most comparable to SONiC: both as results of running switch software at scale to serve ever increasing data center network needs, and with similar architecture (hardware abstraction layer, state management module, etc.). One major difference between SONiC and FBOSS is that FBOSS is not a separate Linux distribution, but using the same Linux OS and libraries in our large server fleet. This allows us to truly reusing many best practices of monitoring, configuring, and deploying for server software. In general, open source communities around switch software are starting grow, which is promising for FBOSS.

Finally, there are recent proposals to completely eliminate switch software [31, 51] from a switch. They provide new insights for the role of switch software and the future of data center switch design.

Centralized Network Control. In the recent Software-Defined Network (SDN) movement, many systems (e.g., [28, 34]), sometimes also referred to as "Network OS", are built to realize centralized network control. While we rely on centralized configuration management and distributed BGP daemons, FBOSS is largely orthogonal to these efforts. By functionality, FBOSS's is more comparable to software switches such as Open vSwitch [44], even if the implementation and performance characteristics are quite different. In fact, similar to how Open vSwitch uses OpenFlow, FBOSS's Thrift API, in theory, can interface with a central controller to provide a more SDN-like functionality.

Large-scale Software Deployment. `fbossdeploy` is influenced by other cloud scale [16] continuous integration frameworks that support continuous canary [45]. Some notable examples are Chef [3], Jenkins [6], Travis CI [10] and Ansible [1]. Contrary to other frameworks, `fbossdeploy` is designed specifically for deploying switch software. It is capable of monitoring the network to perform network specific remediations during the deployment process. In addition, `fbossdeploy` can deploy the switch software in a manner that considers the global network topology.

Network Management Systems. There are many network management systems built to interact with vendor specific devices. For example, HP OpenView [23] has interfaces to control various vendors' switches. IBM Tivoli Netcool [29] handles various network events in real-time for efficient troubleshooting and diagnosis. OpenConfig [9] recently proposed a unified vendor-agnostic configuration interface. Instead of using a standardized management interface, FBOSS provides *programmable* APIs that can be integrated with other network management systems that are vendor-agnostic.

11 CONCLUSION

This paper presents a retrospective on five years of developing, deploying, operating, and open sourcing switch software built for large-scale production data centers. When building and deploying our switch software, we departed from conventional methods and adopted techniques widely used to ensure scalability and resiliency for building and deploying general purpose software. We built a set of modular abstractions that allows the software to be not tied down to a specific set of features or hardware. We built a continuous deployment system that allows the software to be changed incrementally and rapidly, tested automatically, and deployed incrementally and safely. We built a custom management system that allows for simpler configuration management, monitoring and operations. Our approach has provided significant benefits that enabled us to quickly and incrementally grow our network size and features, while reducing software complexity.

ACKNOWLEDGMENT

Many people in the Network Systems team at Facebook have contributed to FBOSS over the years and toward this paper. In particular, we would like to acknowledge Sonja Keserovic, Srikanth Sundaresan and Petr Lapukhov for the extensive help with the paper. We also would like to thank Robert Soulé and Nick McKeown for providing ideas to initiate the paper. We would like to acknowledge Facebook for the resource it provided for us. And finally, we are also indebted to Omar Baldonado, our shepherd, Hitesh Ballani, as well as the anonymous SIGCOMM reviewers for their comments and suggestions on earlier drafts.

REFERENCES

[1] [n. d.]. Ansible is Simple IT Automation. https://www.ansible.com/.

[2] [n. d.]. Apache Thrift. https://thrift.apache.org/.

[3] [n. d.]. Chef. https://www.chef.io/chef/.

[4] [n. d.]. FBOSS Open Source. https://github.com/facebook/fboss.

[5] [n. d.]. FBOSS Thrift Management Interface. https://github.com/facebook/fboss/blob/master/fboss/agent/if/ctrl.thrift.

[6] [n. d.]. Jenkins. https://jenkins-ci.org/.

[7] [n. d.]. Kerberos: The Network Authentication Protocol. http://web.mit.edu/kerberos/.

[8] [n. d.]. Microsoft showcases the Azure Cloud Switch. https://azure.microsoft.com/en-us/blog/microsoft-showcases-the-azure-cloud-switch-acs/.

[9] [n. d.]. OpenConfig. https://github.com/openconfig/public.

[10] [n. d.]. Travis CI. https://travis-ci.org/.

[11] 2016. OpenSwitch. http://www.openswitch.net/.

[12] 2017. Cisco NX-OS Software. http://www.cisco.com/c/en/us/products/ios-nx-os-software/nx-os-software/index.html.

[13] 2018. Facebook Open Routing Group. https://www.facebook.com/groups/openr/about/.

[14] 2018. HwSwitch implementation for Mellanox Switch. https://github.com/facebook/fboss/pull/67.

[15] Lior Abraham, John Allen, Oleksandr Barykin, Vinayak Borkar, Bhuwan Chopra, Ciprian Gerea, Daniel Merl, Josh Metzler, David Reiss, Subbu Subramanian, Janet L. Wiener, and Okay Zed. 2013. Scuba: Diving into Data at Facebook. *Proc. VLDB Endow.* 6, 11 (Aug. 2013), 1057–1067. https://doi.org/10.14778/2536222.2536231

[16] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. 2010. A View of Cloud Computing. *Commun. ACM* 53, 4 (April 2010), 50–58. https://doi.org/10.1145/1721654.1721672

[17] Randy Bias. 2016. The History of Pets vs Cattle and How to Use the Analogy Properly. http://cloudscaling.com/blog/cloud-computing/the-history-of-pets-vs-cattle/.

[18] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. 2014. P4: Programming Protocol-independent Packet Processors. *SIGCOMM Comput. Commun. Rev.* 44, 3 (July 2014), 87–95. https://doi.org/10.1145/2656877.2656890

[19] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. 2013. Forwarding Metamorphosis: Fast Programmable Match-Action Processing in Hardware for SDN. In *SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*. 99–110. http://doi.acm.org/10.1145/2486001.2486011

[20] Cumulus. [n. d.]. Cumulus Linux. https://cumulusnetworks.com/products/cumulus-linux/.

[21] Harrington D., R. Presuhn, and Wijnen B. 2002. An Architecture for Describing Simple Network Management Protocol (SNMP) Management Frameworks. https://tools.ietf.org/html/rfc4862.

[22] Sebastian Elbaum, Gregg Rothermel, and John Penix. 2014. Techniques for Improving Regression Testing in Continuous Integration Development Environments. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)*. ACM, New York, NY, USA, 235–245. https://doi.org/10.1145/2635868.2635910

[23] HP Enterprise. [n. d.]. HP Openview. https://software.microfocus.com/en-us/products/application-lifecycle-management/overview.

[24] Facebook. 2017. Wedge 100S 32x100G Specification. http://www.opencompute.org/products/facebook-wedge-100s-32x100g/.

[25] Tian Fang. 2015. Introducing OpenBMC: an open software framework for next-generation system management. https://code.facebook.com/posts/1601610310055392.

[26] A. Farrel, J.-P. Vasseur, and J. Ash. 2006. *A Path Computation Element (PCE)-Based Architecture*. Technical Report. Internet Engineering Task Force.

[27] Phillipa Gill, Navendu Jain, and Nachiappan Nagappan. 2011. Understanding Network Failures in Data Centers: Measurement, Analysis, and Implications. In *Proceedings of the ACM SIGCOMM 2011 Conference (SIGCOMM '11)*. ACM, New York, NY, USA, 350–361. https://doi.org/10.1145/2018436.2018477

[28] Natasha Gude, Teemu Koponen, Justin Pettit, Ben Pfaff, Martín Casado, Nick McKeown, and Scott Shenker. 2008. NOX: Towards an Operating System for Networks. *SIGCOMM Comput. Commun. Rev.* 38, 3 (July 2008), 105–110. https://doi.org/10.1145/1384609.1384625

[29] IBM. [n. d.]. Tivoli Netcool/OMNIbus. https://www-03.ibm.com/software/products/en/ibmtivolinetcoolomnibus.

[30] Big Switch Networks Inc. 2013. Open Network Linux. https://opennetlinux.org/.

[31] Xin Jin, Nathan Farrington, and Jennifer Rexford. 2016. Your Data Center Switch is Trying Too Hard. In *Proceedings of the Symposium on SDN Research (SOSR '16)*. ACM, New York, NY, USA, Article 12, 6 pages. https://doi.org/10.1145/2890955.2890967

[32] D Joachimpillai and JH Salim. 2004. Forwarding and Control Element Separation (forces). https://datatracker.ietf.org/wg/forces/about/.

[33] Yousef Khalidi. 2017. SONiC: The networking switch software that powers the Microsoft Global Cloud. https://azure.github.io/SONiC/.

[34] Teemu Koponen, Martin Casado, Natasha Gude, Jeremy Stribling, Leon Poutievski, Min Zhu, Rajiv Ramanathan, Yuichiro Iwata, Hiroaki Inoue, Takayuki Hama, and Scott Shenker. 2010. Onix: A Distributed Control Platform for Large-scale Production Networks. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI'10)*. USENIX Association, Berkeley, CA, USA, 351–364. http://dl.acm.org/citation.cfm?id=1924943.1924968

[35] David L. Tennenhouse and David J. Wetherall. 2000. Towards an Active Network Architecture. 26 (07 2000), 14.

[36] P. Lapukhov, A. Premji, and Mitchell J. 2016. Use of BGP for Routing in Large-Scale Data Centers. https://tools.ietf.org/html/rfc7938.

[37] Ville Lauriokari. 2009. Copy-On-Write 101. https://hackerboss.com/copy-on-write-101-part-1-what-is-it/.

[38] K. Lougheed, Cisco Systems, and Y. Rkhter. 1989. A Border Gateway Protocol (BGP). https://tools.ietf.org/html/rfc1105.

[39] R. P. Luijten, A. Doering, and S. Paredes. 2014. Dual function heat-spreading and performance of the IBM/ASTRON DOME 64-bit microserver demonstrator. In *2014 IEEE International Conference on IC Design Technology*. 1–4. https://doi.org/10.1109/ICICDT.2014.6838613

[40] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. 2008. OpenFlow: Enabling Innovation in Campus Networks. *SIGCOMM Comput. Commun. Rev.* 38, 2 (March 2008), 69–74. https://doi.org/10.1145/1355734.1355746

[41] Juniper Networks. 2017. Junos OS. https://www.juniper.net/us/en/products-services/nos/junos/.

[42] Tuomas Pelkonen, Scott Franklin, Justin Teller, Paul Cavallaro, Qi Huang, Justin Meza, and Kaushik Veeraraghavan. 2015. Gorilla: A Fast, Scalable, In-memory Time Series Database. *Proc. VLDB Endow.* 8, 12 (Aug. 2015), 1816–1827. https://doi.org/10.14778/2824032.2824078

[43] A.D. Persson, C.A.C. Marcondes, and D.P. Johnson. 2013. Method and system for network stack tuning. https://www.google.ch/patents/US8467390 US Patent 8,467,390.

[44] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Joe Stringer, Pravin Shelar, Keith Amidon, and Martin Casado. 2015. The Design and Implementation of Open vSwitch. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. USENIX Association, Oakland, CA, 117–130. https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/pfaff

[45] Danilo Sato. 2014. Canary Release. https://martinfowler.com/bliki/CanaryRelease.html.

[46] Brandon Schlinker, Hyojeong Kim, Timothy Cui, Ethan Katz-Bassett, Harsha V. Madhyastha, Italo Cunha, James Quinn, Saif Hasan, Petr Lapukhov, and Hongyi Zeng. 2017. Engineering Egress with Edge Fabric: Steering Oceans of Content to the World. In *Proceedings of the ACM SIGCOMM 2017 Conference (SIGCOMM '17)*. ACM, New York, NY, USA, 418–431. https://doi.org/10.1145/3098822.3098853

[47] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, Anand Kanagala, Jeff Provost, Jason Simmons, Eiichi Tanda, Jim Wanderer, Urs Hölzle, Stephen Stuart, and Amin Vahdat. 2015. Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network. *SIGCOMM Comput. Commun. Rev.* 45, 4 (Aug. 2015), 183–197. https://doi.org/10.1145/2829988.2787508

[48] Yu-Wei Eric Sung, Xiaozheng Tie, Starsky H.Y. Wong, and Hongyi Zeng. 2016. Robotron: Top-down Network Management at Facebook Scale. In *Proceedings of the ACM SIGCOMM 2016 Conference (SIGCOMM '16)*. ACM, New York, NY, USA, 426–439. https://doi.org/10.1145/2934872.2934874

[49] David Szabados. 2017. Broadcom Ships Tomahawk 3, Industry's Highest Bandwidth Ethernet Switch Chip at 12.8 Terabits per Second. http://investors.broadcom.com/phoenix.zhtml?c=203541&p=irol-newsArticle&ID=2323373.

[50] S Thomson, Narten T., and Jinmei T. 2007. IPv6 Stateless Address Autoconfiguration. https://tools.ietf.org/html/rfc4862.

[51] F. Wang, L. Gao, S. Xiaozhe, H. Harai, and K. Fujikawa. 2017. Towards reliable and lightweight source switching for datacenter networks. In *IEEE INFOCOM 2017 - IEEE Conference on Computer Communications*. 1–9. https://doi.org/10.1109/INFOCOM.2017.8057152

[52] Jun Xiao. 2017. New Approach to OVS Datapath Performance. http://openvswitch.org/support/boston2017/1530-jun-xiao.pdf.

[53] Xilinx. [n. d.]. Lightweight Ethernet Switch. https://www.xilinx.com/applications/wireless-communications/wireless-connectivity/lightweight-ethernet-switch.html.