Submitted for publication. Author Copy - do not redistribute.

Parallel Simulation of Software Defined Networks

Dong Jin
Department of Electrical and Computer Engineering
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801
dongjin2@illinois.edu

David M. Nicol
Department of Electrical and Computer Engineering
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801
dmnicol@illinois.edu

ABSTRACT

Existing network architectures fall short when handling networking trends, e.g., mobility, server virtualization, and cloud computing, as well as market requirements with rapid changes. Software-defined networking (SDN) is designed to transform network architectures by decoupling the control plane from the data plane. Intelligence is shifted to the logically centralized controller with direct programmability, and the underlying infrastructures are abstracted from applications. The wide adoption of SDN in network industries has motivated development of large-scale, high-fidelity testbeds for evaluation of systems that incorporate SDN. We leverage our prior work on a hybrid network testbed with a parallel network simulator and a virtual-machine-based emulation system. In this paper, we extend the testbed to support OpenFlow-based SDN simulation and emulation; show how to exploit typical SDN controller behavior to deal with potential performance issues caused by the centralized controller in parallel discrete-event simulation; and investigate methods for improving the model scalability, including an asynchronous synchronization algorithm for passive controllers and a two-level architecture for active controllers. The techniques not only improve the simulation performance, but also are valuable for designing scalable SDN controllers.

1. INTRODUCTION

Complex networks with large numbers of vendor-dependent devices and inconsistent policies greatly limit network designers' options and ability to innovate. For example, to deploy a network-wide policy in a cloud platform, the network operators must (re)configure thousands of physical and virtual machines, including access control lists, VLANs, and other protocol-based mechanisms. Trends in networking, such as cloud services, mobile computing, and server virtualization, also impose requirements that are extremely difficult to meet in traditional network architectures. Software-defined networking (SDN) is designed to transform network architectures. In an SDN, the control plane is decoupled from the data plane in the network devices, such as switches, routers, access points, and gateways, and the underlying infrastructure is abstracted from the applications that use it. Therefore, SDN enables centralized control of a network with flexibility and direct programability. SDNs have been widely accepted and deployed in enterprise networks, campus networks, carriers, and data centers. For example, Google has deployed SDN in its largest production network: its data-center to data-center WAN [9].

OpenFlow [15] is the first standard communications interface defined between the control and forwarding layers of an SDN architecture. Existing OpenFlow-based SDN testbeds include MiniNet [13], MiniNet-HiFi [10], OFTest [3], OFlops [20], and NS-3 [2]. MiniNet probably is the most widely used SDN emulation testbed at present. It uses an OS-level virtualization technique called Linux container, and is able to emulate scenarios with 1000+ hosts and Open vSwitches [4]. However, MiniNet has not yet achieved performance fidelity, especially with limited resources, since resources are time-multiplexed by the kernel, and the overall bandwidth is limited by CPU and memory constraints. Execution of real programs in virtual machines exhibits high functional fidelity, and creation of multiple virtual machines on a single physical machine provides scalability and flexibility for running networking experiments, but low temporal fidelity is a major issue for virtual-machine-based network emulation systems. By default, all virtual machines use the same system clock of the physical machine. As a result, when a virtual machine is idle, its clock still advances. That raises the temporal fidelity issue when applications running on a virtual machine are expected to behave as if they were being executed on a real machine. All existing SDN emulation testbeds lack temporal fidelity. We have developed an emulation virtual time system in our prior work [22]. Freeing the emulation from real time enables us to run experiments slower or faster than real time. When resources are limited, we can always slow down experiments to ensure accuracy. On the other hand, experiments can be accelerated if system resources are sufficient. In this work, we extend the system to support OpenFlow-based SDN emulation with high functional fidelity and temporal fidelity.

While emulation testbeds offer fidelity, they are not suitable for large-scale network experiments. Simulation testbeds can provide scalability, but the accuracy degrades because of models' simplification and abstraction. Therefore, in prior work, we integrated a parallel network simulator with a virtual-machined-based emulation system based on virtual time [12]. When conducting network experiments, we can

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGSIM-PADS'13, May 19–22, 2013, MontrŐal, QuŐbec, Canada.
Copyright 2013 ACM 978-1-4503-1920-1/13/05 ...$15.00.

execute critical components in emulation and use simulation to provide a large-scale networking environment with background traffic. In this work, we also develop a framework to support OpenFlow-based SDN simulation, including models of an OpenFlow switch, controller, and protocol. The ns-3 network simulator also has an OpenFlow module [2] supporting both simulation and emulation, but, unlike our system, it does integrate virtual-time-based emulation and parallelized simulation. In addition, we have conducted extensive studies of application-level behaviors in our emulation system [21], and discovered that the errors introduced by the emulation system are bounded by one time slice, which is the system execution unit (e.g., 100 μs). However, the processing delay within a Gigabit switch is on the scale of one microsecond. If such a delay is critical to the SDN applications being tested, we can use simulation models instead, in which the time granularity is defined by the users to achieve the desired timing accuracy.

In this paper, we extend our network testbed with the capability to emulate and simulate OpenFlow-based SDNs. The extension is based on close analysis of how SDN controllers typically behave, which led to organizational and synchronization optimizations that deal with problems that might otherwise greatly limit scalability and performance. The testbed provides the following benefits and is a useful tool for SDN-based research in general:

- Temporal fidelity through a virtual time system, especially with limited system and networking resources, e.g., it can emulate 10 hosts, each with a gigabyte Ethernet interface, on a single physical machine with only one one-gigabyte Ethernet network card,

- More scalable network scenarios with simulated nodes,

- Background traffic at different levels of detail among the simulated nodes, and

- Sophisticated underlying network environments (e.g., wireless CSMA/CA).

An OpenFlow controller typically connects with a large number of OpenFlow switches. Not only is the centralized controller a potential communication bottleneck in the real networks, but also it gives rise to performance drawbacks in simulating such networks in the context of parallel discrete-event simulation. The large number of switch-controller links could potentially reduce the length of the synchronization window in barrier-based global synchronization, and could also negatively impact the channel scanning type of local synchronization. We investigate techniques to improve the scalability of the simulated OpenFlow controllers through analysis of a list of basic controller applications in POX, which is a popular OpenFlow controller written in Python [6]. We classify the controller applications into active controllers and passive controllers based on whether the controllers proactively insert rules into the switches, or the rule insertions are triggered by the switches. We design an asynchronous synchronization algorithm for the passive controllers, and a two-level architecture for the active controllers for use in building scalable OpenFlow controller models.

The main contributions of this paper are as follows: (1) We develop a network testbed for simulating and emulating OpenFlow-based SDNs. The testbed integrates a virtual-time embedded emulation system and a parallel network simulator to achieve both fidelity and scalability. (2) We explore and evaluate means to improve the performance of simulation of OpenFlow controllers in parallel discrete-event simulation, including an asynchronous synchronization algorithm for controllers that are "passive" (according to a definition given in Section 4), and a two-level architecture for controllers we classify as "active." (3) The two-level controller architecture is also a valuable reference for designing real scalable SDN controllers.

The remainder of the paper is organized as follows. Section 2 introduces background on software-defined networks. Section 3 describes the system design with emulation and simulation support for running of OpenFlow-based SDN experiments. Section 4 presents the performance issues caused by the simulated centralized controller and investigates various techniques to improve the scalability. Section 5 evaluates the performance of the proposed controller models. Section 6 concludes the paper with plans for future work.

2. OPENFLOW-BASED SOFTWARE-DEFINED NETWORKS

In a traditional network architecture, the control plane and the data plane cooperate within a device via internal protocols. By contrast, in an SDN, the control plane is separated from the data plane, and the control logic is moved to an external controller. The controller monitors and manages all of the states in the network from a central vantage point. The controller talks to the data plane using the OpenFlow protocol [1], which defines the communication between the controller and the data planes of all the forwarding elements. The controller can set rules about the data-forwarding behaviors of each forwarding device through the OpenFlow protocol, including rules such as drop, forward, modify, or enqueue.

Each OpenFlow switch has a chain of flow tables, and each table stores a collection of flow entries. A *flow* is defined as the set of packets that match the given properties, e.g., a particular pair of source and destination MAC addresses. A flow entry defines the forwarding/routing rules. It consists of a bit pattern that indicates the flow properties, a list of actions, and a set of counters. Each flow entry states "execute this set of actions on all packets in this flow," e.g., forward this packet out of port A. Figure 1 shows the main components of an OpenFlow-based SDN and the procedures by which an OpenFlow switch handles an incoming packet. When a packet arrives at a switch, the switch searches for matched flow entries in the flow tables and executes the corresponding lists of actions. If no match is found for the packet, the packet is queued, and an inquiry event is sent to the OpenFlow controller. The controller responds with a new flow entry for handling that queued packet. Subsequent packets in the same flow will be handled by the switch without contacting the controller, and will be forwarded at the switch's full line rate.

Some benefits of applying SDN in large-scale and complex networks include the following:

- The need to configure network devices individually is eliminated.

- Policies are enforced consistently across the network

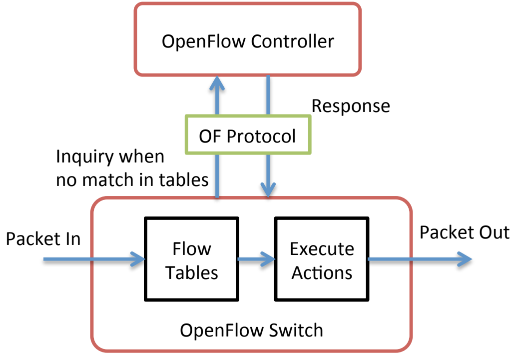

Figure 1: How an OpenFlow Switch Handles Incoming Packets

infrastructures, including policies for access control, traffic engineering, quality of service, and security.

- Functionality of the network can be defined and modified after the network has been deployed.
- Addition of new features does not require change of the software on every switch, whose APIs are generally not publicly available.

3. SYSTEM DESIGN

We have developed a large-scale, high-fidelity network testbed by integrating the OpenVZ network emulation into an S3F-based network simulation framework, and extended the testbed to support OpenFlow-based SDN simulation/emulation. Figure 2 shows the overall system architecture on a single physical Linux box. The system consists of three major components: OpenVZ, the network emulation system; S3F, the system kernel; and S3FNet, the parallel network simulator.

3.1 System Components

3.1.1 S3F – Parallel System Kernel

SSF (Scalable Simulation Framework) defines an API that supports modular construction of simulation models, with automated exploitation of parallelism [17]. Building on ten years of experience, we developed a second-generation API named *S3F* [19]. In both SSF and S3F, a simulation is composed of interactions among a number of *entity* objects. Entities interact by passing *events* through *channel* endpoints they own. Channel endpoints are described by *InChannels* and *OutChannels* depending on the message direction. Each entity is aligned to a *timeline*, which hosts an event list and is responsible for advancing all entities aligned to it. Interactions between co-aligned entities need no synchronization other than this event-list. Multiple timelines may run simultaneously to exploit parallelism, but they have to be carefully synchronized to guarantee global causality. The synchronization mechanism is built around explicitly expressed delays across channels whose end-points reside on entities that are not aligned. We call those *cross-timeline channels*. The concepts of entity, channel, timeline, and (non-)cross-timeline channel are depicted in the lower portion of Figure 2. The global synchronization algorithm in S3F creates synchronization windows, within which all timelines can safely advance without being affected by other timelines. More details about S3F are in [19].

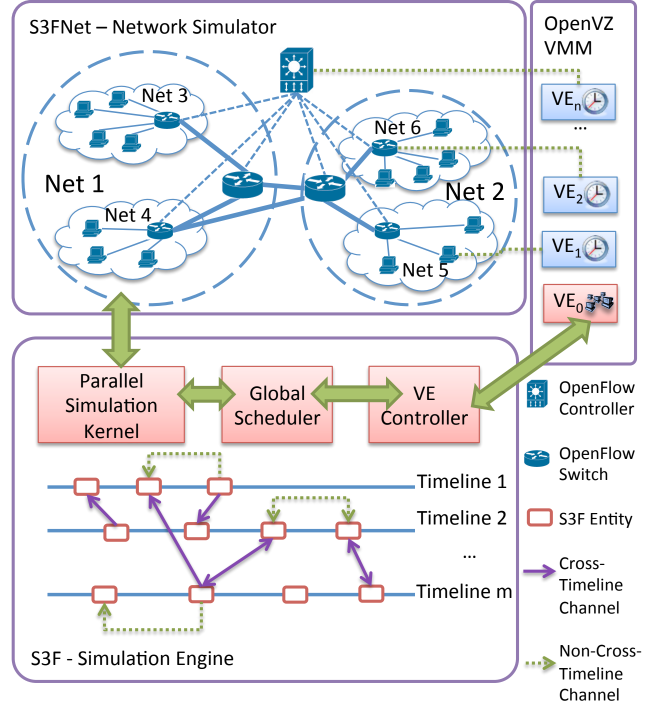

Figure 2: S3F System Architecture Design with OpenFlow Extension

One important feature of S3F is its support of OpenVZ-based emulation experiments based on the notion of virtual clocks. The VE controller and the global scheduler, shown in Figure 2, are responsible for managing the interaction between the simulation and the emulation systems.

The *VE controller* is designed to control all the emulation hosts according to S3F's command, and to provide necessary communications between VEs and their corresponding simulated nodes. The VE controller is a special API that offers three services: an advance emulation clock, the ability to transfer packets bidirectionally between simulation and emulation, and emulation lookahead. More details on the VE controller are in [12].

The *Global scheduler* coordinates safe and efficient advancement of the two systems. It ensures that the emulation runs ahead of the simulation, generating the network's offered load, but not so far ahead as to miss delivery of traffic. It also schedules the sequence and length of each emulation and simulation execution cycle to preserve causal relationships. Each VE has its own virtual clock, which is synchronized with the simulation clock in S3F. The complete synchronization algorithm used by the global scheduler is described in [12]. In addition, the global scheduler makes the emulation integration nearly transparent to the upper-layer network simulator.

3.1.2 S3FNet – Network Simulator

S3FNet is a network simulator built on top of the S3F kernel. While it is capable of creating models of network devices (e.g., host, switch, router) with layered protocols (e.g., IP, TCP, UDP) like other network simulators, the primary

purposes of S3FNet for the entire system are to simulate a sophisticated underlying network environment (e.g., detailed CSMA/CD for traditional Ethernet, and/or CSMA/CA for wireless communication) and to efficiently model the extensive communication and computation in large-scale experiment settings (e.g., background traffic simulation models [18] [11]).

S3FNet has a global view of the network topology. Every VE in the OpenVZ emulation system is represented in the S3FNet as a host model within the modeled network, together with other simulated nodes. Within S3FNet, traffic that is generated by a VE emerges from its proxy host inside S3FNet, and, when directed to another VE, is delivered to the recipient's proxy host as shown in Figure 2. Only the global scheduler in S3F knows the distinction between an emulated host (VE-host) and a virtual host (non-VE host).

3.1.3 OpenVZ-based Network Emulation

OpenVZ is an OS-level virtualization technology that creates and manages multiple isolated Linux containers, also called *Virtual Environments (VEs)*, on a single physical server. Each VE performs exactly like a stand-alone host with its own root access, users and groups, IP and MAC address, memory, process tree, file system, system libraries, and configuration files. All VEs share a single instance of the Linux OS kernel for services such as TCP/IP. Compared with other virtualization technologies such as Xen (para-virtualization) and QEMU (full-virtualization), OpenVZ is lightweight (e.g., we are able to run more than 300 VEs on a commodity server with eight 2GHz cores and 16 GB memory), at the cost of diversity in the underlying operating system.

OpenVZ emulation allows users to execute real software in the VEs to achieve high functional fidelity. It also frees a modeler from developing complicated simulation models for network applications and the corresponding protocol stacks. In addition to functional fidelity, our version of OpenVZ also provides temporal fidelity through a virtual clock implementation [22], allowing integration with the simulation system.

In a real system, an application perceives time elapses when doing computation or when waiting for I/Os. Similarly, our system advances a VE's virtual clock between the time when the VE starts to execute and when the VE stops, and also between the time when the VE starts a blocking I/O, and when the VE finishes.

We used a timeslice-based mechanism to implement our virtual time system. A VE can run only after the VE controller has given it a timeslice and released it. Without being given a timeslice, a VE will remain suspended, and its clock will not advance. When an event occurs to unsuspend it (e.g., arrival of a message that had been blocked), the clock is advanced to the time of the event, leaping over an idle epoch. While a VE is suspended, the simulator can perform arbitrarily long simulation computation, as this will not affect the VE's state. That frees our emulation from having to run in real time; emulation can now run either faster or slower than real time, depending on the system load and the simulation model that is being used.

Another advantage is that the VE controller is able to control VE executions to prevent some VEs from running too far ahead (in terms of virtual time), potentially leading to causal violations. In regards to VE synchronization, we currently use a barrier-based mechanism. In particular, S3F computes a barrier (a virtual time value) to which all VEs can safely advance, and lets all VEs run to this barrier. Within two consecutive barriers, VEs are considered independent, and any VE interactions are postponed until arrival at the barrier. Details of the synchronization algorithm are provided in [12].

Our virtual time system introduces some timing errors via the timeslice-based mechanism. Once a VE has received a timeslice, the VE controller cannot interact or stop the VE until the timeslice has expired up on the hardware interrupt. In particular, the VE controller cannot deliver an event (i.e., a network packet arrival) to the VE at the exact same virtual time the packet leaves the simulation. Events occur at timeslice boundaries, so any error is bounded in magnitude by the length of the timeslice. We may reduce the error bound by setting a smaller hardware interrupt interval [22]. However, that interval cannot be arbitrarily small, because of efficiency concerns and hardware limits.

3.2 OpenFlow Integration

The design of SDN separates the control plane from the data plane in network forwarding devices (switches, routers, gateways, or access points). Users can design their own logically centralized controllers to define the behaviors of those forwarding elements via a standardized API, such as OpenFlow, which provides flexibility to define and modify the functionalities of a network after the network has been physically deployed. Many large-scale networks, such as data centers, carriers, and campus networks, have been deployed with SDNs. Network simulation/emulation is often used to facilitate the testing and evaluation of large-scale network designs and applications running on them. Therefore, we extend our existing network testbed to support OpenFlow, in particular, simulation and/or emulation of network experiments that contains OpenFlow switches and controllers, which communicate via the OpenFlow protocol.

To emulate OpenFlow-based networks, we can run unmodified OpenFlow switch and controller programs in the OpenVZ containers, and the network environment (such as wireless or wireline media) is simulated by S3FNet. Since the executables are run on the real network stacks within the containers, the prototype behaviors are close to the behaviors in real SDNs. Once an idea works on the testbed, it can be easily deployed to production networks. Other OpenFlow emulation testbeds, like MiniNet [13], have good functional fidelity too, but lack performance fidelity, especially with heavy loads. Imagine that one OpenFlow switch with ten fully loaded gigabit links is emulated on a commodity physical machine with only one physical gigabit network card. There is no guarantee that a switch ready to forward a packet will be scheduled promptly by the Linux scheduler in real time. Our system does not have such limitations, since the emulation system is virtual-time embedded. The experiments can run faster or slower depending on the workload. When load is high, performance fidelity is ensured by running the experiment slower than real time. On the other hand, when the load is low, we can reduce the execution time by quickly advancing the experiment.

However, experimental results show that errors introduced by the virtual time system at the application level are bounded by the size of one emulation timeslice. We may reduce the error bound by setting a smaller hardware interrupt interval [22]. Nevertheless, the interval cannot be arbitrarily small because of efficiency concerns and hardware limits. We typ-

ically use 100 μs as the smallest timeslice. If we want to simulate a gigabit switch whose processing delay is on the scale of a micro-second, a 100 μs error bound is simply too large. That motivated us to develop a simulated OpenFlow switch and an OpenFlow controller model; the simulation virtual time unit is defined by the users and can be arbitrarily small, typically a micro-second or nano-second.

Our OpenFlow simulation model has two components: the OpenFlow switch, and the OpenFlow controller. The switches and the controller communicate via the OpenFlow protocol [1], and the protocol library we use is the OpenFlow reference implementation at Stanford University [5], which has been used in many hardware-based and software-based SDN applications. The initial version of the switch and the controller models have been developed with reference to the ns-3 OpenFlow models [2]. Figure 3 depicts the model implementation details.

Our OpenFlow switch model can handle both simulated traffic and real traffic generated by the applications running in the containers. The switch model has multiple ports, and each port consists of a physical layer and a data link layer. Different physical and data link layer models allow us to simulate different types of network, such as wireless and wireline networks. A switch layer containing a chain of flow tables is located on top of the ports, as shown in Figure 3. It is responsible for matching flow entries in the tables and performing the corresponding predefined actions or sending an inquiry to the controller when no match is found in the tables. The controller model consists of a group of applications (e.g., learning switch, link discovery) as well as a list of connected OpenFlow switches. It is responsible for generating and modifying flow entries and sending them back to the switches.

Needless to say, running executables of OpenFlow components in the emulation mode has better functional fidelity than the simulation models do. We attempt to keep the high fidelity in the simulation models by using the original unmodified OpenFlow library, which has been used to design many real SDN applications. Also, the simulation models are not constrained by the 100 μs timeslice error bound in emulation. In addition, we can run experiments in the simulation mode with much larger network sizes. Finally, as a bonus effect of the OpenFlow design, we no longer have to preload a forwarding table at every network device, since where and how to forward the packets are decided on demand by the controller. Simulating a network with millions or even more network devices at the packet level is affordable in our system.

Figure 3 also shows the journey of a packet in our system. A packet is generated at the source end-host, either from the simulated application layer or from the real network application in a container. The packet is pushed down through the simulated network stacks of the host and then is popped up to the OpenFlow switch via the connected in-port. Depending on the emulation or simulation mode of the switch (the OpenVZEmu module in Figure 3 is used to handle the case where the entity is associated with a virtual-machine container), the packet is searched within the simulated flow tables or the real flow tables in the container, and a set of actions are executed when matches are found. Otherwise, a new flow event is directed to the controller, meaning either the simulated controller model or the real controller program in the container. The controller generates new flow entries

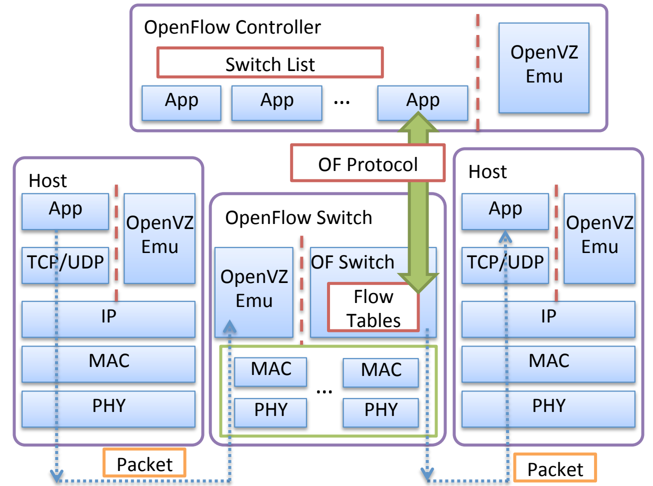

Figure 3: OpenFlow Implementation in S3F

and installs the flow entries onto the switches via the OpenFlow protocol. Afterwards, the switch knows how to process the incoming packet (and subsequent additional packets of this type), and transmit it via the correct out-port. Eventually the packet is received by the application running on the destination end-host.

4. SCALABILITY OF SIMULATED OPENFLOW CONTROLLERS

SDN-based network designs have multiple OpenFlow switches communicating with a single OpenFlow controller. However, many-to-one network topologies not only create communication bottlenecks at the controllers in real networks, but also negatively impact the performance of conservative synchronization of parallel discrete-event simulations. The conservative synchronization approaches in parallel discrete-event simulation generally fall into two categories: synchronous approaches based on barriers [7] [14] [16], and asynchronous approaches, in which a submodel's advance is a function of the advances of other submodels that might affect it [8]. The single-controller-to-many-switch architecture can be bad for both types of synchronization.

We can view a network model as a direct graph: nodes are entities like hosts, switches, and routers; edges are the communication links among entities; and each link is weighted by a link delay. From S3F, the parallel simulation engine's viewpoint, the graph is further aggregated: a node represents a group of entities on the same timeline, and the simulation activity of all entities on a timeline is serialized; multiple links between timelines t_i and t_j are simplified into one link whose weight is the minimum (cross-timeline) delay between t_i and t_j.

A barrier-based synchronous approach is sensitive to the minimum incoming edge weight in the entire graph. If one of the OpenFlow switch-controller links has a very small link delay (e.g., a controller and a switch could be installed on the same physical machine in reality), even if there are few activities running on the link, the overall performance will be poor because of the small synchronization window. On the other hand, an asynchronous approach focuses on timeline interactions indicated by the topology, but is subject to

Table 1: **Basic Network Applications in the POX OpenFlow Controller**

Controller Application	Description	Passive/ Active	State	Other Comments
openflow. keep_alive	The application sends periodic echo requests to connected switches. Some switches will assume that an idle connection indicates a loss of connectivity to the controller and will disconnect after some period of silence, and this application is used to prevent that.	Active	Distributed	The request-sending period is a good source of lookahead.
forwarding. l2_learning	The application makes an OpenFlow switch act as a type of L2 learning switch. While the component learns L2 addresses, the flows it installs are exact matches on as many fields as possible. For example, different TCP connections will result in installation of different flows.	Passive	Distributed	
forwarding. l2_pairs	The application also makes an OpenFlow switch act like a type of L2 learning switch. However, it installs rules based purely on MAC addresses, probably the simplest way to do a learning switch correctly.	Passive	Distributed	
forwarding. l2_learning	The application is not quite a router. It uses L3 information to perform data switching similar to a L2 switch.	Passive	Distributed	
openflow. link_discovery	The application sends specially crafted messages out of OpenFlow switches to discover link status, e.g., to discover the network topology. Switches raise events to the controller when links go up or down.	Active	Shared among the connected switches	The link status query period is a good source of lookahead.
forwarding. l2_multi	This application can still be seen as a learning switch, but it learns where a MAC address is by looking up the topology of the entire network.	Passive	Network-wide	
openflow. spanning_tree	This application constructs a spanning tree based on the network topology, and then disables flooding on switch ports that are not on the tree.	Active	Network-wide	

significant overhead costs on timelines that are highly connected. The SDN-based architectures unfortunately always have a centralized controller with a large degree, which is not a desired property for asynchronous approaches either.

To improve the simulation performance with SDN architectures, we explore the properties of an OpenFlow controller with reference to a list of basic applications in POX, which is a widely used SDN controller written in Python [6]. The applications are summarized in Table 1. We have two key observations.

First, controllers can be classified as either passive or active. A controller is *passive* if the applications running on it never initiate communication to switches, but only passively respond to inquires from switches when no matches are found in switches' flow tables. The forwarding.l2_learning application is a good example of an application that runs on a passive controller. An *active* controller initiates communication to switches, e.g., detecting whether a switch or a link is working or broken. The openflow.link_discovery application is an example of an application that runs on an active controller.

Second, a controller is not simply a large single entity shared by all the connected switches. A controller actually has states that are shared by switches at different levels. Suppose there are N switches in a network, and m switches ($1 \leq m \leq N$) share a state.

- When $m = N$, the state is network-wide, i.e., the state is shared by all switches. For example, the openflow.spanning_tree application has a network-wide state, which is the global network topology.

- When $m = 1$, the state is distributed, i.e., no other switch shares the state with this switch. For example, the forwarding.l2_learning application has a distributed state, which is the individual learning table for each switch.

- When $1 < m < N$, the state is shared by a subset of switches of size m. For example, the openflow.link_discovery application has such a state, which is the link status shared among all the switches connected to that link.

Based on the above two observations, we revisit the controller design and investigate techniques to improve the performance of the simulation of OpenFlow-based SDNs with parallel discrete-event simulation. In particular, we design an efficient asynchronous algorithm for passive controllers; we also propose a two-level controller architecture for active controllers, and analyze performance improvement for three applications with different types of states. The proposed architecture is not only helpful with respect to simulation

performance, but also a useful reference for designing scalable OpenFlow controller applications.

4.1 Passive Controller Design

A passive controller indicates that applications running on the controller do not proactively talk to switches, a feature we can use in designing a controller whose functionality is known to be passive. A new design is also motivated by another observation: when a switch receives an incoming packet, the switch can handle the packet without consulting the controller if a matched rule is found in the switch's flow table; and, for some applications, the number of times the controller must be consulted (e.g., first time the flow is seen, or when the flow expires) is far lower than the number of packets being processed locally. All the learning switch applications in the POX controller have this property.

Therefore, our idea is that for a passive controller, switches are free to advance their model states without constraint, until the switches have to communicate with the controller. If multiple switches share a state, then the controller needs to refrain from answering the inquiring switch until all its cross-timeline dependent switches have advanced to the time of the inquiring switch. The algorithm is presented as follows:

Let A be the set of applications running in the OpenFlow controller, and R be the set of OpenFlow switches in a network model. We define $TL(r)$ to be the timeline to which switch r is aligned. For each $a \in A$ and $r \in R$, we define $f(r, a)$ to be the subset of OpenFlow switches that share at least one state with switch r for application a. For example, $f(r_1, a_1) = \{r_2, r_3\}$ means that switches r_1, r_2, r_3 are dependent on (sharing states with) application a_1. Causality is ensured only if the controller responds to the inquiry from switch r_1 with timestamp t_1, after all the dependent cross-timeline switches r_i, i.e., $r_i \in f(r_1, a)$ and $TL(r_i) \neq TL(r_1)$, have advanced their times to at least t_1. For an application with network-wide states, $f(r, a) = R - \{r\}$; for a fully distributed application, $f(r, a) = \phi$.

Algorithm 1 is designed for efficient synchronization among switches and fast simulation advancement with correct causality in the case of a passive controller. The algorithm is divided into two parts: one at the controller side and another at the switch side. Since the controller cannot actively affect a switch's state, it is safe for a switch to advance independent of the controller until a switch-controller event happens (e.g., a packet is received that has no match in the switch's flow tables). The delays between the controller and the switches thus do not affect the global synchronization. The causality check is performed at the controller, since it has the global dependency information of all the connected switches. Upon receiving an OpenFlow inquiry, the controller is responsible for making sure no response will be sent back to the switch (thus, the switch will not advance its clock further) until all its dependent switches have caught up with it in simulation time. In addition, this design does **not** require that the controller be modeled as an S3F entity, which means that the controller does not have to align with any timeline. All the interactions can be done through function calls instead of via message passing through S3F channels. This design works only for a passive controller and can greatly reduce the communication overhead between the controller and switches. As a result, a passive controller is not a bottleneck in conservatively synchronized parallel simulation, as low latency to switches and high fan-out might otherwise cause it to be.

Algorithm 1 Synchronization Algorithm with Passive Controller

```
Controller Side
/* Upon receiving an OpenFlow inquiry from switch r_i with timestamp t_i */
for each application a_j related to the inquiry do
  for each switch r_k ∈ f(r_i, a_j) AND TL(r_k) ≠ TL(r_i) do
    get the current simulation time, t_k, of r_k
    if t_k < t_i then
      schedule a timing report event at time t_i on the timeline of r_k
      increase dcts[i] by 1
      /* dcts[i] is the counter of unresolved dependent cross-timeline switches for switch r_i */
    end if
  end for
end for

pthread_mutex_lock()
while dcts[i] > 0 do
  pthread_cond_wait()
end while
pthread_mutex_unlock()

process the inquiry (i.e., generate rules to handle packets)
send an OpenFlow response to switch r_i

Switch Side
/* Upon receiving a packet at an ingress port */
check flow tables
if found matched rule(s) then
  process the packet accordingly
else
  send an OpenFlow inquiry to the controller
end if

/* On reception of an OpenFLOW response */
store the rule(s) in the local flow table(s)
process the packet accordingly

/* Scheduled timing report event for switch r_i fires */
pthread_mutex_lock()
decrease dcts[i] by 1
if dcts[i] = 0 then
  pthread_cond_signal()
end if
pthread_mutex_unlock()
```

Figure 4 illustrates our synchronization algorithm with an example. The steps are as follows.

1. Switch r_1 sends an inquiry to the controller with $t_1(now)$
2. The controller gets the current times of all dependent cross-timeline switches, e.g., $t_2(now)$, $t_3(now)$
3. Because $t_2(now) < t_1(now)$, the controller schedules a timing report event at $t_1(now)$ on timeline 2; no event is scheduled on timeline 3 since $t_3(now) > t_1(now)$

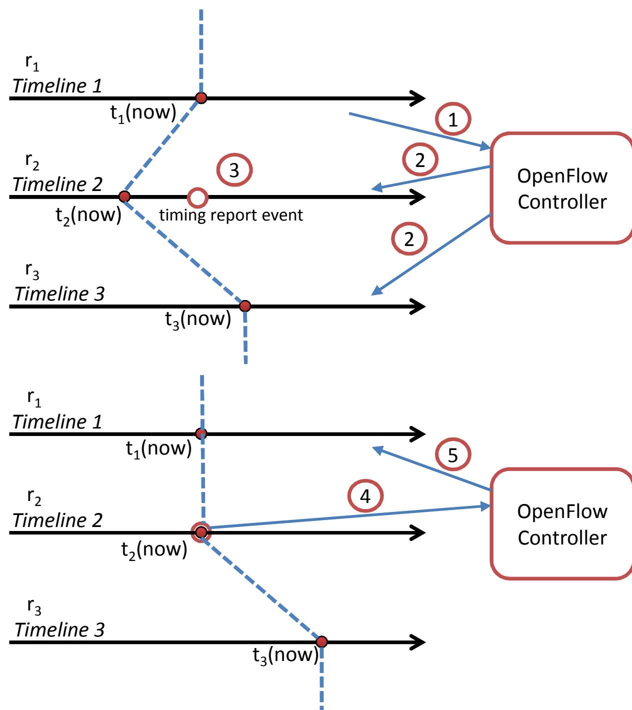

Figure 4: Sample Communication Patterns between the OpenFlow Switch and the Passive OpenFlow Controller Using Our Synchronization Algorithm

4. At $t_1(now)$ on Timeline 2, r_2 signals the controller
5. The controller generates rule(s) based on the up-to-date states and sends a response back to r_1

The algorithm works for all passive controllers, whether the state is distributed (e.g., forwarding.l2_learning) or network-wide (e.g., forwarding.l2_multi). The state property plays an important role in the active controller design, which will be discussed in Section 4.2. However, the performance of passive controllers does benefit from use of distributed states as well as the smaller number of cross-timeline dependent switches.

4.2 Active Controller Design

Active OpenFlow controllers proactively send events to the connected OpenFlow switches, and those events can potentially affect the states of switches. Therefore, the switches do not have the freedom to advance the model states like those switches that connect to passive controllers, but are subject to the minimum link latency between the controller and the switches. However, the question we have is: are the assumptions about connectivity in SDNs overly pessimistic? For example, can any timeline generate an event at any instant that might affect every other timeline?

We make the following observations about the active controller applications. First, not all controllers have only network-wide states. Some have fully distributed states, e.g., a switch's on/off state (openflow.keep_alive application); some have states that are shared among a number of switches, e.g., a link's on/off state is shared among switches connected to the same link (openflow.link_discovery application). Second, not all events will result in global state changes, and quite often a large number of events are handled locally, and only influence switches' local states. For instance, only when a communication link fails, or when a link is recovered from failure or when some new switches join or leave the network, a link status change event is generated for the openflow.link_discovery application, so that it updates its global view; during the remaining (most of the) time, the network-wide state, i.e., the global topology, remains unchanged. Therefore, for such applications, instead of having a centralized controller frequently send messages to all the connected switches and wait for responses, we can first determine which events affect network-wide states and which do not, and then off-load those events, which only cause local state changes, towards the switch side. Thus, we relieve the pressure at the controller.

Based on our observations of active controller applications, we propose a two-level active controller architecture as depicted in Figure 5. Local states are separated from network-wide states in a controller, and the controller is divided into a top-controller and a number of local controllers. The top controller communicates only with the local controllers, not with the switches, to handle events that potentially affect the network-wide states. The local controllers run applications that can function using the local states in switches, e.g., the local policy enforcer, or link discovery. There are no links among local controllers. With the two-level controller design, we aim to improve the overall scalability, especially at the top controller, as follows:

- The top controller has a smaller degree, which is good for local synchronization approaches.
- Fewer messages are expected to occur among local controllers and the top controller for many applications, since the heavy communication is kept between the switches and local controllers. That is good for local synchronization approaches as well.
- If we align the switches that share the same local controller to the same timeline, the local controllers actually do not have to be modeled as an entity. Message passing through channels is not needed, as function calls are sufficient.

Conversion of a controller into a two-level architecture requires that modelers carefully analyze the states in the controller applications. That process is not only useful in creating a scalable simulation model, but also helpful in designing a high-performance real SDN controller, because it offloads local-events processing to local resources. In the remainder of this section, we will inspect three active controller applications with our two-level controller design: openflow.keep_alive with fully distributed states, openflow.link_discovery with shared states, and openflow.spanning_tree with network-wide states.

The **openflow.keep_alive** application is designed to send periodic echo requests to all connected switches to prevent loss of connections, since some switch firmware by default treats an idle link as a link failure. Meanwhile, the controller also maintains a complete list of the running switches in the network. Since the on/off state of a switch is independent of other switches, the openflow.keep_alive is an example of applications with fully distributed states. In the two-level controller design, one local controller is created for each switch. Let us now compute the total number of simulation events

(e.g., message-sending events, message-receiving events, and time-out events) within one echo request period with the single controller and the two-level controller.

Assume that the network has n switches, and let p be the probability that a switch fails during one request period. In the single-controller case, the controller needs one timer event to enable the echo request sending process, and then sends one request to each switch. If a switch is working, the request is received at it, and it replies with a response; finally, the controller receives the response. If a switch fails, a timeout event occurs at the controller. Therefore, for the single-controller case, the total number of events within one period is $M_1 = 1 + n + (1 - p) * n * (1 + 1 + 1) + p * n = 1 + 4n - 2pn$. In the two-level controller case, we have n local controllers, and each local controller fires one timer event to start the request-sending process. Since the communication with the local controllers and the switches is modeled by function calls instead of by message passing, no event occurs as long as a switch is working. If a switch fails, the local controller reports the state change to the global controller, and the global controller receives the report. Therefore, for the two-level controller case, the total number of events within one period is $M_2 = n + p * n * (1 + 1) = n + 2pn$.

Assume $n \gg 1$; when the fail rate of a switch, p, is 0.75, the single-controller and the two-level controller have approximately the same total number of events. If $p < 0.75$, the two-level controller has fewer events to simulate. If p is close to 0, which means that switches rarely fail during experiments, $M_2 \approx n$ events, and $M_1 \approx 4n$ events. Also, let us count the total number of events processed at the top controller within one period. The single controller has the total number of events $C_1 = 1+n+n*(1-p)+n*p = 1+2n$, and the two-level controller has the total number of events $C_2 = np$. The latter is always smaller than the former, and thus the two-level controller for the openflow.keep_alive application results in a more scalable controller. Actually, it is possible to remove the top controller as long as all the applications running in the controller are fully distributed, like openflow.keep_alive. If so, we can prevent a single global controller from becoming a potential system bottleneck.

The **openflow.link_discovery** application periodically requests messages from switches in order to discover the link status, and switches raise events to the controller when links go up or down. Potentially, this application could be used to discover the network topology. The states of this application are the links' on/off states shared among all the switches connecting to the link. With the two-level controller design, we assign one local controller to each switch, and let the top controller manage the shared states.

Let us calculate the total number of simulation events within one link discovery period for both cases; the switch-switch and switch-host interactions are not counted, since they are the same for both cases. Assume that the network has n switches, and that the probability that a switch has at least one link state change is q. For the single-controller case, the controller fires one timer event to start the link discovery process, and then sends each switch a link discovery request. Upon receipt of the request at every switch, the switch sends back a response if at least one of its links has a state change. Finally, the response is received at the controller to update the global topology. Therefore, the total number of events within one period is $M_1 = 1+n+n+q*(n+n) = 1+2n+2qn$. For the two-level controller case, n local controllers fire one timer event each to start the link discovery process locally. Since the interaction between the local controller and the switch is modeled by function calls instead of by message passing, no event is generated if no link state change is detected. If a link change is discovered by the switch, the local controller then reports the change to the global controller, and the global controller receives the report. Therefore, the total number of events within one period for the two-level controller is $M_2 = n + q * n(1 + 1) = n + 2qn$. We can see that there are $2n$ fewer events than in the single-controller scenario. Let us count the number of events processed at the top controller within one period. The single controller has $1 + n + qn$ events, while the two-level controller has qn events. Therefore, the two-level controller always has better scalability, especially when the network has a relatively steady topology. The two-level controller is also a good reference design for the real link discovery application in the SDN controller. The number of packets being processed at the top controller is greatly reduced to achieve good scalability.

Good lookahead can improve performance in simulating networks with either the single controller or the two-level controller. We notice that the link discovery request is sent to the switch periodically. The period is often quite long compared with the time to transmit a packet, and the period can serve as a good source of lookahead. In addition, another optimization we could do is to model the top controller as a passive controller since the requests are now initiated at the local controllers.

The **openflow.spanning_tree** application establishes a spanning tree based on the network topology, and then it disables flooding on switch ports that are not on the tree. The application has only one state, the global network topology. The state is a network-wide state. Therefore, it is hard to convert this application to a two-level controller. However, the openflow.spanning_tree application still has invariants that we could utilize to improve the system performance.

S3F synchronizes its timelines at two levels. The inner level uses the barrier-synchronization and the channel-scanning-synchronization for parallel discrete-event simulation. At the outer level, timelines are left to run during an epoch, which could terminate either after a specified length of simulation time, or when the global state meets some specified conditions. Between epochs S3F allows a modeler to do computations that affect the global simulation state, without concern for interference by timelines. A spanning tree algorithm is often running at the beginning of an experiment and then is triggered by topology changes or a global timer. In the scenarios in which a network topology does not change frequently, it is reasonable to assume that the topology is invariant for a certain minimum length of time, and we can use that time length as the length of an epoch. Between two epochs, a spanning tree is recomputed. States (particular spanning trees) created by those computations are otherwise taken to be constant when the simulation is running.

We have studied three active applications in the POX controller. Through careful application state analysis, we can often convert applications with local states into two-level controller architectures for simulation performance gain. The analysis not only helps modelers to create scalable SDN network models, but also helps in the design of scalable real SDN controllers. Many SDN applications can be far more

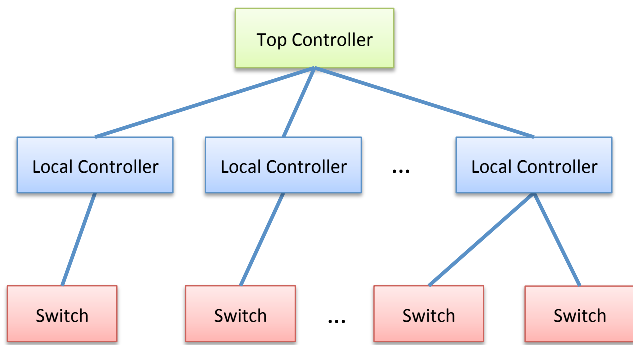

Figure 5: Two-level Controller Architecture

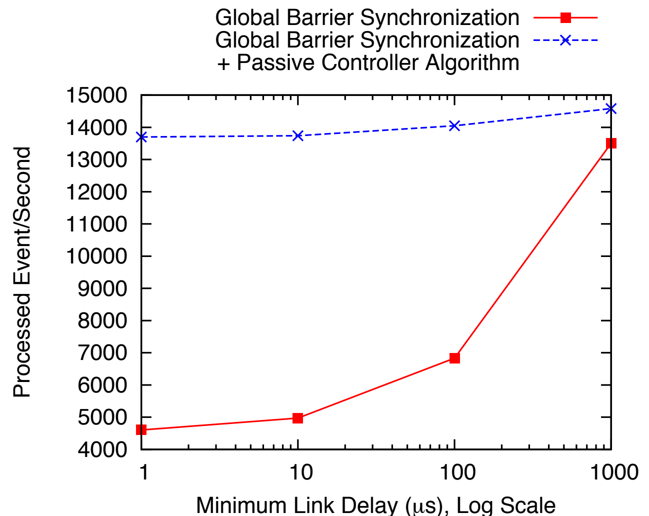

Figure 6: Performance Improvement with the Passive Controller Asynchronous Synchronization Algorithm: Minimal Controller-Switch Link Latency

complicated than the basic applications in POX. They could be a combination of passive and active applications, with both distributed and network-wide states, and the states may change dynamically with time and network conditions. Even so, it is still useful to divide a complex controller into small building-block applications, classify them according to the state type as well as passiveness/activeness, and then apply the corresponding performance optimization techniques.

5. EVALUATION

We first study the performance improvement with the asynchronous synchronization algorithm for the passive controllers. We created a network model with 16 timelines. The backbone of the network model consisted of 32 OpenFlow switches, and each switch connected 10 hosts. Half of the hosts ran client applications, and the other half ran server applications. File transfers between clients and servers were on UDP. The minimum communication link delay among switches and hosts was set to be 1 ms. During the experiments, each client randomly chose a server from across the entire network, and started to download a 10 KB file. Once the file transfer was complete, the client picked another server and started the downloading process again. All the OpenFlow switches connected to an OpenFlow controller, and the controller ran the openflow.learning_multi application, which is essentially a learning switch, but learns where a MAC address is by looking up the topology of the entire network. In the first case, we modeled the controller as a normal S3F entity, and it connected to switches through channels. In the second case, we used the passive controller design described in Section 4.1 . Since all switches shared the global network topology, they were dependent on one another. We ran the experiments on a machine with 16 2GHz-processors and 64 GB memory.

In the first set of experiments, we varied the link delay between the switches and the controller with values of 1 μs, 10 μs, 100 μs and 1 ms. Each experiment was executed for 100 seconds in simulation time for both cases. We observed 12,810 completed file transfers for both cases. Figure 6 shows the event-processing rates for both cases. In the first case, the traditional barrier-type global synchronization was used. Since the minimal controller-switch link delay was set to be no larger than other link delays within the network, that delay dominated the synchronization window calculation with the global synchronization algorithm in S3F. Therefore, the performance degraded as the minimum link latency decreased because of more frequent synchronization overheads. In the second case, synchronization was actually two-level: the global synchronization was used to simulate activities among switches and hosts, and within a synchronization window, the asynchronous synchronization algorithm was used to simulate interactions between the switches and the passive controller. Therefore, the controller-switch delay did not affect the global synchronization window size. For the passive controller case, the event-processing rate remained almost unchanged, and the performance was 2 to 3 times better than in the first case, when the controller-switch delay was small. The performance was still better in the second case even when the controller-switch link delay was set to 1 ms (equal to the minimal delay of the other links). The reason is that the interactions between the passive controller and the switches were through function calls instead of event-passings with S3F channels, which we explored further in the next set of experiments.

In the second set of experiments, we increased the number of OpenFlow switches connected to the controller, and the size of the network was thus increased proportionally. Every client application performed the same actions as in the first experiment set. The model still used 16 timelines, and each experiment was simulated for 100 seconds with the two types of controller. Let λ be the event-processing rate of a model using the passive controller algorithm over the event-processing rate of the same model with the S3F entity controller. λ indicates the performance improvement of asynchronous synchronization designed for passive controllers. As shown in Figure 7, λ grows as the number of switches increases. Since each switch needs a larger flow table as the size of the network increases, switches have more interactions with the controller. As a result, the performance gain of replacing the message-passing mechanism with function calls for the controller-switch interactions is more prominent. In addition, the modeled controller application has a network-wide state, resulting in a fully dependent switch list in the passive controller. More performance improve-

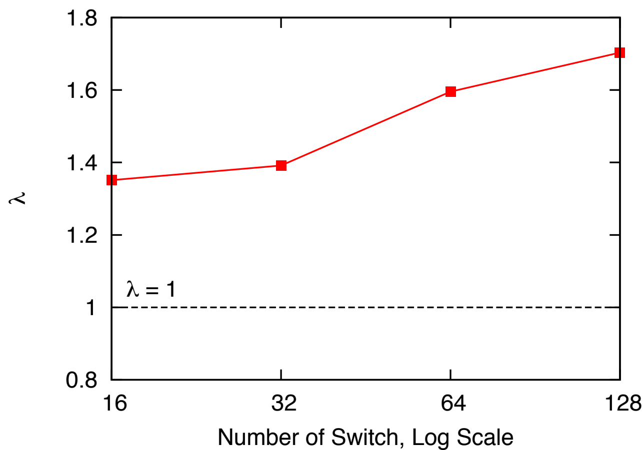

Figure 7: Performance Improvement with the Passive Controller Asynchronous Synchronization Algorithm: Number of OpenFlow Switches in the Network

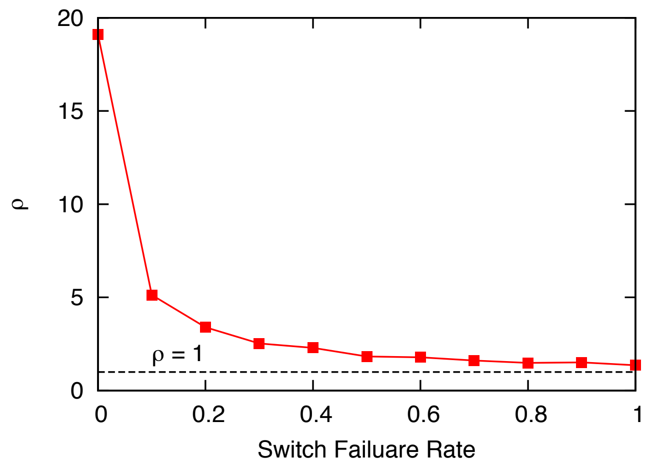

Figure 8: Performance Improvement for the openflow.keep_alive Application with a Two-level Controller Architecture

ment is expected for modeling controller applications with distributed states.

We also investigated the performance improvement with the two-level architecture for active controllers. The openflow. keep_alive application was developed with the two designs as discussed in Section 4.2: (1) a single centralized controller, and (2) a two-level controller in which the local controllers are responsible for querying switch states and reporting to the top controller when switch failures are discovered. The network model had 80 OpenFlow switches with a minimal link delay of 1 ms, running with 16 timelines. The switch state inquiry period was set to be 1 second. Let us define the performance indicator ρ to be the event-processing rate of a model using the two-level controller architecture over the event-processing rate of the same model using the single controller. We varied the switch failure rate, and the results are shown in Figure 8.

The model with the two-level controller performed better than the model with the single controller under all switch failure rates. The reasons are that (1) switches and their local controller were in the same timeline, and the switch state inquiry activities were well-parallelized; and (2) the switch state responses were sent back to the top controller only when switch failures were detected, resulting in less data communication with the top controller. Simulating a network with smaller switch failure rates had better performance, e.g., 19 times faster with a zero failure rate, and 5 times faster with a 10% failure rate, as shown in Figure 8.

6. CONCLUSION AND FUTURE WORK

In this paper, we described how we extended our network testbed, consisting of virtual-machine-based emulation and parallel simulation, to support OpenFlow-based SDNs. Users can conduct SDN-based network experiments with real OpenFlow switch and controller programs in virtual-time-embedded emulations for high functional and temporal fidelity, or with OpenFlow switch and controller simulation models for scalability, or with both. We also explore ways to improve the scalability of the centralized simulated controllers, including an asynchronous synchronization algorithm for passive controllers and a two-level architecture design for active controllers, which also serves as a useful guideline for real SDN controller design.

We plan to evaluate the network-level and application-level behaviors for different SDN applications under various network scenarios (e.g., long delay, or lossy link). We would also like to conduct a performance comparison with ground truth data collected from a real physical SDN testbed as well as data collected from other testbeds, like MiniNet. In addition, we plan to utilize the testbed to design and evaluate SDN applications, especially in the context of the smart grid; an example would be design of efficient quality-of-service mechanisms in substation routers, where all types of smart grid traffic aggregate.

7. ACKNOWLEDGMENTS

This material is based upon work supported in part by the Department of Energy under Award Number DE-OE0000097, and by the Boeing Corporation [1].

8. REFERENCES

[1] Openflow switch specification version 1.1.0. `http://www.openflow.org/documents/openflow-spec-v1.1.0.pdf`, 2011.

[1]Disclaimer: This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

[2] ns-3 OpenFlow switch support. http://www.nsnam.org/docs/release/3.13/models/html/openflow-switch.html, Accessed 2012.

[3] OFTest, a Python based OpenFlow switch test framework. http://www.openflow.org/wk/index.php/OFTestTutorial, Accessed 2012.

[4] Open vSwitch. http://openvswitch.org/, Accessed 2012.

[5] OpenFlow Switching Reference System. http://www.openflow.org/wp/downloads/, Accessed 2012.

[6] POX. http://www.noxrepo.org/pox/about-pox/, Accessed 2012.

[7] R. Ayani. *A parallel simulation scheme based on distances between objects*. Royal Institute of Technology, Department of Telecommunication Systems-Computer Systems, 1988.

[8] K. Chandy and J. Misra. Distributed simulation: A case study in design and verification of distributed programs. *IEEE Transactions on Software Engineering*, (5):440–452, 1979.

[9] Google. Inter-Datacenter WAN with centralized TE using SDN and OpenFlow. https://www.opennetworking.org/images/stories/downloads/misc/googlesdn.pdf, Accessed 2012.

[10] N. Handigol, B. Heller, V. Jeyakumar, B. Lantz, and N. McKeown. Reproducible network experiments using container-based emulation. In *Proceedings of the 8th international conference on Emerging networking experiments and technologies*, pages 253–264. ACM, 2012.

[11] D. Jin and D. Nicol. Fast simulation of background traffic through fair queueing networks. In *Proceedings of the 2010 Winter Simulation Conference (WSC)*, pages 2935–2946, Baltimore, MD, December 2010.

[12] D. Jin, Y. Zheng, H. Zhu, D. Nicol, and L. Winterrowd. Virtual time integration of emulation and parallel simulation. In *Proceedings of the 2012 Workshop on Principles of Advanced and Distributed Simulation (PADS)*, pages 120–130, Zhangjiajie, China, July 2012.

[13] B. Lantz, B. Heller, and N. McKeown. A network in a laptop: rapid prototyping for software-defined networks. In *Proceedings of the Ninth ACM SIGCOMM Workshop on Hot Topics in Networks*, page 19. ACM, 2010.

[14] B. Lubachevsky. Efficient distributed event-driven simulations of multiple-loop networks. *Communications of the ACM*, 32(1):111–123, 1989.

[15] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: Enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, 2008.

[16] D. Nicol. The cost of conservative synchronization in parallel discrete event simulations. *Journal of the ACM (JACM)*, 40(2):304–333, 1993.

[17] D. Nicol, J. Liu, M. Liljenstam, and G. Yan. Simulation of large scale networks using SSF. In *Proceedings of the 2003 IEEE Winter Simulation Conference*, volume 1, pages 650–657, 2003.

[18] D. Nicol and G. Yan. High-performance simulation of low-resolution network flows. *Journal of Simulation*, 82(1):21–42, 2006.

[19] D. M. Nicol, D. Jin, and Y. Zheng. S3F: The Scalable Simulation Framework revisited. In *Proceedings of the 2011 Winter Simulation Conference (WSC)*, pages 3283–3294, Phoenix, AZ, December 2011.

[20] R.Sherwood. OFlops. http://www.openflow.org/wk/index.php/Oflops, Accessed 2012.

[21] Y. Zheng, D. Jin, and D. M. Nicol. Validation of application behavior on a virtual time integrated network emulation testbed. In *Proceedings of the Winter Simulation Conference*, page 246, 2012.

[22] Y. Zheng, D. M. Nicol, D. Jin, and N. Tanaka. A virtual time system for virtualization-based network emulations and simulations. *Journal of Simulation*, 6(3):205–213, August 2012.