accepted to appear in the Proceedings of 28th International Conference on Computer Communications and Networks (ICCCN 2019), July 29 - August 1, 2019, Valencia, Spain - invited paper

A Distributed SDN Application for Cross-Institution Data Access

Shafaq Chaudhry
Electrical & Computer Engineering
University of Central Florida
Orlando, FL, USA
Shafaq.Chaudhry@ucf.edu

Eyuphan Bulut
Department of Computer Science
Virginia Commonwealth University
Richmond, VA, USA
ebulut@vcu.edu

Murat Yuksel
Electrical & Computer Engineering
University of Central Florida
Orlando, FL, USA
Murat.Yuksel@ucf.edu

***Abstract*—SDN is based on the idea of a centralized controller with a global view of the network topology. However, in large networks, multiple controllers work together to perform global network functions. This presents challenges in load balancing, consistent network view, and controller placement for scalability and reliability. When multiple controllers reside in different domains, their communication challenges are increased as each may have its own security and access policies. We present a reactive distributed SDN application built as a custom module in Floodlight that allows multiple controllers to make joint decisions where the controllers reside in different domains by communicating with an external server for information about participating organizations. Such a framework would be useful, for example, for providing access to patient information distributed among different hospital networks, big data sets between research institutions, or public safety data sets during disaster or emergency. The resulting approach will allow the SDN application layer to handle inter-domain traffic and enable data access between different organizations/agencies while respecting their different respective policies.**

***Index Terms*—Multi-domain SDN, Floodlight, distributed reactive SDN application, cross-institution data access.**

I. Introduction

Software-Defined Networking (SDN) virtualizes the network layer by defining a logically centralized control plane that manages and programs the data plane behavior. This provides flexible flow management and opportunity for innovation in network management and control services that run on top of the SDN controller. For scalability in large networks, this logically centralized control plane comprises of multiple controllers that need to coordinate and communicate with each other to manage the network through a global network view.

In a typical SDN setup (Figure 1), each controller manages multiple SDN-capable switches and communicates with the switches through a Southbound Interface (SBI), currently standardized as OpenFlow [1], developed by Open Networking Foundation (ONF). Network and traffic management services are programmed in the SDN application layer that interfaces with the controllers through the Northbound Interface (NBI).

The SDN landscape has focused on localized solutions where enterprise networking administrators are interested in virtualizing the network layer and gaining more agility in order to meet the traffic demands of their enterprise. However, demand for Software-Defined Wide Area Networking (SD-WAN) is growing. SD-WAN will allow organizations with branch offices spanning the globe to connect their geographically dispersed data centers more efficiently through increased automation and intelligent service-delivery across the WAN. According to a study by International Data Corp (ID), it is expected that by the year 2020, SD-WAN revenues will be over six billion [2]. However, scaling SD-WAN solutions to multiple branches/sites requires better management of controller-controller communication through an East/West interface. The East/West interface can be used to connect a traditional IP network with an SDN network. It can also be used to facilitate authentication and authorization across different administrative domains. No clear standards exist for the East/West interface.

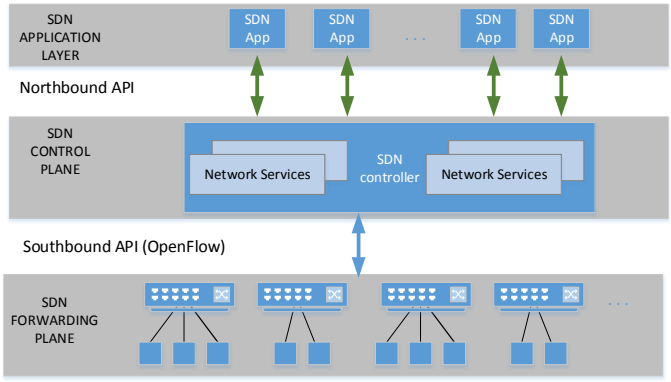


Fig. 1: The SDN application layer runs programs like firewall and load-balancing, and the controller runs network services.

Furthermore, SD-WAN applications needing access to datasets distributed over large areas and potentially owned by multiple institutions are still limited to traditional pre-SDN technologies. Enabling SDN for these types of SD-WAN-and-beyond applications requires SDN frameworks capable of handling multi-institution engagements and controller-to-controller setups. What we propose is addressing this need of facilitating access to data sets spread across different institutions. In particular, we focus on real-time applications such as multi-agency public safety response operations, and hence, explore the delay involved in access provisioning over multi-institution distributed SDN setups. Key contributions of

this work are:

- A distributed, reactive SDN application for real-time access to another institution's data store through an external verification server.
- Detailed protocol design to use this reactive SDN application for cross-institution data access.
- Overhead delay formulation of the reactive SDN application; a simplified analytical model to capture the effect of flow table size and flow rule expiration timer on this delay; and a simplified model that expresses average flow table size with respect to the expiration timer.
- Comparison of the reactive SDN application to a proactive design both in terms of protocol specifics and performance.
- Proof-of-concept prototype of the reactive SDN application in Floodlight v1.0, Mininet and OpenFlow v1.0.

The rest of the paper is structured as follows. We discuss the related work in Section II. In Section III, we describe the details of the proposed approach and provide a delay analysis in Section IV. In Section V, we provide our initial simulation results in Mininet. Finally, we outline the future work in Section VI and provide concluding remarks in Section VII.

II. Related Work

This section presents related work and challenges in distributed SDN applications, distributed control plane architectures, and controller communication across SDN domains.

A. Distributed SDN Applications

Distributed SDN application designs have mostly focused on network traffic engineering for the purpose of higher Quality of Service (QoS) provisioning. MonSamp [3] is a QoS-monitoring distributed SDN application, that samples flows for QoS analysis by installing flows in the flow tables to send packets of the flows being monitored to an external Collector and Analyzer monitoring application, as a reaction to changes in link utilization.

Additionally, balancing the load on distributed SDN controllers has also attracted attention, again, to provide better SDN performance. In [4], the authors present a load-balancing SDN application and highlight the importance of a consistent network view by showing that load balancing decisions suffer when made without considering inconsistencies observed in the global view. Similarly, in [4], the authors present a load-balancing SDN application and highlight the importance of a consistent network view by showing that load balancing decisions suffer when made without considering inconsistencies observed in the global view. In [5], the authors tackle the load balancing problem of multiple distributed controllers, where decisions are either made centrally by a coordinating controller that collects load information from all the controllers, figures out the best course of action and sends back commands to the distributed controllers; or, locally, at each controller by using the load information received from other controllers. Both these cases incur delay for reaching a decision.

Although these studies provide ways of addressing load-balancing issues in distributed SDN applications (and hence controllers), they do not address the case of distributed SDN applications over different SDN domains.

B. Logically Distributed Control Plane

In a logically centralized control plane, the controllers are physically distributed but the decisions are made centrally by a designated root controller that manages devices, all part of the same domain, so the network administrator has full control of the SDN controllers. In a logically distributed control plane, we have multiple domains, each managed by its own controller as shown in Figure 2.

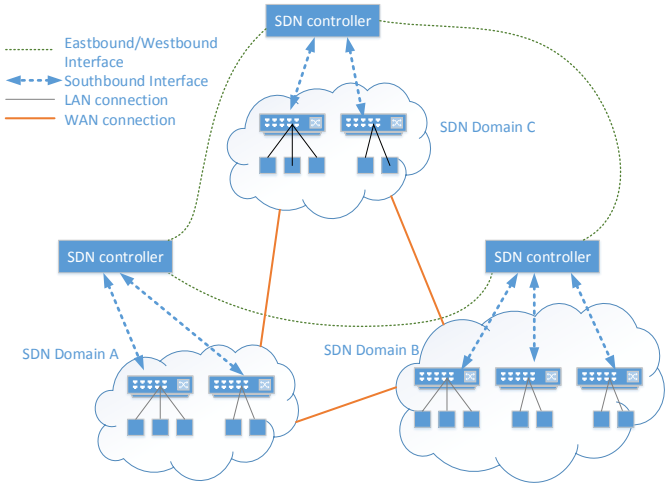


Fig. 2: Multiple controllers in different SDN domains form a control plane that is both physically and logically distributed.

Logically distributed controllers coordinate the control functions of the control plane with each other by maintaining a global view of the network. This is accomplished through controller-controller communication and synchronizing the networking view by sharing network state information among all the controllers. For example, in HyperFlow [6], inter-controller communication occurs through a publish/subscribe model with three network channels: a data channel for publishing local network events, a control channel for discovery and its own channel for receiving OpenFlow commands. Another method of sharing information among controllers is by means of distributed constructs like distributed databases or distributed hash tables and then using distributed locking and consensus algorithms for achieving consistency of state information. For example, Onix [7] proposed an SQL database for infrequently changing topology and a Distributed Hash Table (DHT) for frequently changing information like link utilization. Scalability is achieved by partitioning the Network Information Base (NIB) which stores the network state among controllers, and aggregating the nodes under a controller to present as a single node to the global network.

Maintaining global state suffers from delay in the time it takes to sync network states across all controller nodes for

a consistent network view. Methods have been proposed to reduce this delay by caching flow rules and distributing them throughout the controllers. Rules can be updated periodically when link failures occur. Using this strategy, Distributed Rule Store (DRS) [8] is another architecture for multiple controllers based on Floodlight [9] that demonstrates a reduced time to setup flows as compared with ONOS [10] and Floodlight.

HyperFlow and Onix suffer from synchronization delay to achieve network state information consistency, while our approach does not require a global network state to be maintained as the controller in one institution does not need to know the network topology inside the other institution. Additionally, the DRS architecture presents performance improvements for reactive applications but it does not deal with controllers that belong to different domains or institutions.

C. Challenges with Multiple Controllers

The survey paper in [11] describes four design choices in distributed controllers: a) whether a switch connects to the controller statically or dynamically; b) whether all controllers have global network state information or only the root layer has the global view; c) how controllers work to resolve any conflicting or competing rules through consensus algorithms; and d) whether or not dedicated links are used for managing traffic between controller-controller and switch-controller. The authors then classify existing distributed SDN architectures like DISCO [12], HyperFlow [6], D-SDN [13], Onix [7], Kandoo [14], and FlowBroker [15] according to these design choices and discuss their impact on scalability, privacy, robustness and consistency of the network.

In [16], the authors survey existing work on multi-domain SDNs and describe challenges of distributed controllers as: a) having a consistent global view of the network state [6], [7]; b) defining the optimal number and placement of these distributed controllers [17]; and c) synchronization and coordination of events local to the controller and events occurring globally in the distributed control plane. Logically centralized but physically distributed controllers like Onix [7] and HyperFlow [6] must share network state with each other and usually employ distributed data stores, like a distributed file system or distributed hash table. With every change of network state at a local controller, it needs to be synchronized with other controllers. With fully distributed controllers, such as DISCO [12], ONOS [10], Kandoo [14], and others, a complete global network state does not need to be maintained across all controllers and is thus suited for a multi-domain SDN.

In addition to the architecture of the distributed control plane, there is a need to work on the Eastbound/Westbound interface. One such work is the East-West Bridge [18] that connected global research and education networks, but this remains an open research area.

In this paper, we propose a distributed SDN application that works in SDN domains owned by two different organizations to facilitate access to data distributed and managed by different organizations that cannot leave the boundary of the institution due to its institutional policies.

III. Approach

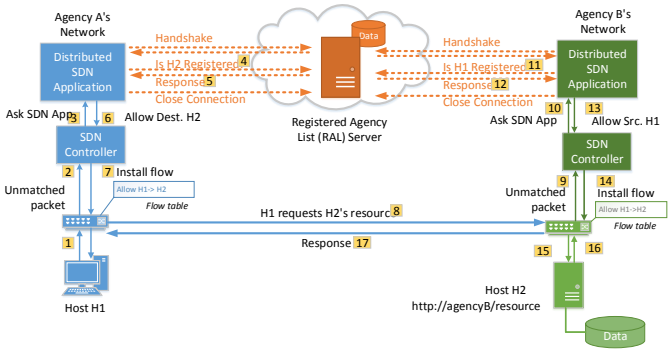


Fig. 3: Proposed multi-agency data access architecture with a reactive SDN application setup and RAL server.

Our proposed approach falls under logically distributed control plane frameworks where the domains have different administrators and each domain has its own controller. We consider Floodlight [9], a Java-based controller developed by BigSwitch Networks. Aside from the core functions of the controller, the Floodlight controller offers functions implemented as modules that interact with the core controller through Java APIs. We have developed a custom module that can promptly respond to network events due to its interface with the core controller. Such an application is also called a reactive SDN application as it can insert flow rules into the switch flow tables as a response (reaction) to these events.

The controller manages traffic by managing the flow tables of the switches. Flow tables consist of header fields for packet matching, action fields for forwarding instructions, and counters for statistics. When a packet arrives at a switch, the switch matches the packet with the header fields in the flow entries and if a match is found, the switch performs the specific actions in the flow entry. When several matches are found, the rule with the highest priority applies. If a match is not found, the switch asks the controller for instruction by sending it a *Packet In* message. A reactive SDN application can register an event listener with the controller to listen in for these *Packet In* messages and can inspect the headers of the packet to determine how to handle the packet and what flow rules to install to handle subsequent such packets.

A. Proposed Architecture

The proposed architecture for data access between two SDN domains A and B is presented in Figure 3. In our approach, we will create a Registered Access List (RAL) server to maintain the list of organizations participating in this multi-institution data access framework. Our reactive SDN application running on the controller will intercept packets sent to the controller and talk to the RAL server to determine if access to the remote domain's resource should be granted or denied. Flow rules will be jointly installed into the edge switches controlled by the controllers at both domains to handle this inter-domain traffic. Assume there is a server in domain B that has a resource that

a client in domain A wants to access. The sequence of steps that occur to allow traffic from A to B is as follows:

1) Client at host h1 requests the resource at server h2.
2) The switch in A's network detects a new traffic flow and sends this unmatched packet to the SDN controller.
3) The reactive SDN module running on the controller intercepts the *Packet In* message to handle this packet.
4) The SDN application extracts source and destination from header and queries the RAL server to determine if the source and destination are registered.
5) Assume that the source and destination are participants in the data access framework, so RAL server responds that both source and destination are registered.
6) If so, the SDN application composes *FlowMod* messages to allow traffic from the host h1 in domain A to h2 in domain B and sends to the controller.
7) The controller then instructs the switch to install the respective flows in its flow table.
8) The packet is then forwarded as normal and reaches the switch at domain B.
9) When the traffic is received by the switch in domain B, it does not find a match in its flow table and forwards the packet to the controller in domain B.
10) Domain B's controller invokes the packet handler of the reactive SDN application listening for message events.
11) The SDN application extracts the source and destination from the packet header, connects with the RAL server to check if source and destination are registered.
12) Assume that the RAL server responds positively.
13) The SDN application then follows its packet handling logic to allow this flow.
14) The SDN controller instructs the switch to install flow rules to allow traffic from source to destination.
15) Request is forwarded as normal to the outport of the switch that is linked to host h2.
16) Host h2 responds to the request, and sends the response packets to the switch at domain B.
17) The resulting packets are sent as a response to domain A's switch which forwards the response to host h1.

B. SDN Components

Key SDN Components of the proposed framework are:

- Multiple physically distributed Floodlight controllers managing different SDN domains, each governed by its own institutional policies for data egress.
- A remote Registered Access List (RAL) server that maintains the list of registered agencies intending to participate in the multi-institution data access framework. Agencies can register themselves or unregister themselves at any time. The RAL server is implemented as a multi-threaded Java server application to handle queries from multiple SDN controllers.
- A reactive SDN application running on the controllers. The application has a packet handler to handle *Packet In* messages and a RAL client for establishing web socket connections to the RAL server for querying.

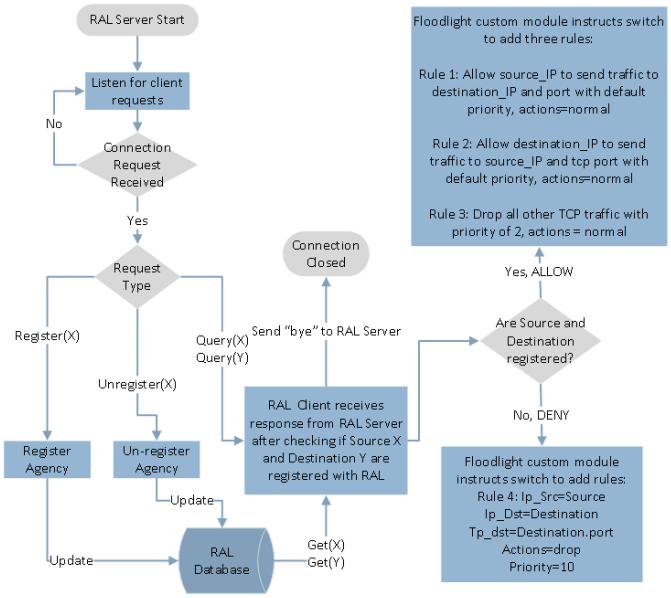


Fig. 4: Logic flow between RAL server, RAL client and SDN packet handler.

- OpenFlow as the SouthBound Interface for communicating with the SDN infrastructure layer. Currently, we have implemented our algorithm to support OpenFlow v1.0 [1] switches and use the OpenFlowJ-Loxigen library [19] for creating OpenFlow messages.

If the source and destination are both member of RAL, three rules are installed as shown in Figure 4. Assuming TCP traffic, the first rule allows TCP traffic from source to destination. Because TCP is bidirectional, the second rule allows TCP traffic from destination to source. A third rule will be installed to drop all other TCP traffic. The key thing to note here is that the priority of Rule 3 is very low. Rules 1 and 2 are installed at the default priority of 32,768. If either source or destination or both are not registered with RAL, Rule 4 is installed which instructs the switch to drop the traffic from that source to that destination. This algorithm is summarized in Algorithm 1.

IV. Delay Analysis

In this section, we analyze the expected processing delay of flows in the proposed reactive SDN application scenario. When a new flow arrives at the switch, the switch inspects the packet header and performs a look-up in the flow table to find matching flow rule(s) based on header fields like IP Protocol, IP Source, IP Destination, and Ingress Port. Flow rules when matched, guide the switch on what action to take on the packet, for example, forward to a specific port, drop the packet, or flood to all ports. Packet headers are matched to flow rules in the order of priority of the installed rules. The time it takes to do a search and match operation in the flow table depends upon the hardware, which for an OpenFlow switch is typically implemented by leveraging Ternary Content Addressable Memory (TCAM), where each header field is

```
Result: OpenFlow messages written to swtich
while packet_in message do
    if packet is IPv4 and protocol is TCP then
        source = payload.getSourceAddress();
        destination = payload.getDestinationAddress();
        destinationPort = tcp.getDestinationPort();
        if either source or destination are not in RAL then
            Add Rule 4 with Match:(src=source,
              dest=destination, ip_protocol=TCP,
              tcp_destination_port=destinationPort),
              actions=drop, priority=10;
        else
            Add Rule 1 with Match:(src=source,
              dest=destination, ip_protocol=TCP,
              tcp_destination_port=destinationPort),
              priority=default, actions=NORMAL;
            Add Rule 2 with Match:(src=destination,
              dest=source, ip_protocol=TCP,
              tcp_source_port=destinationPort),
              priority=default, actions=NORMAL;
            Add Rule 3 with Match:(src=source,
              dest=destination, ip_protocol=TCP,
              tcp_destination_port=destinationPort),
              priority=2, actions=drop;
        end
    else
        return Command.CONTINUE;
    end
end
return Command.CONTINUE;//allow processing by other
  handlers
```

Algorithm 1: SDN reactive application packet handler

searched in parallel in one clock cycle, resulting in look-up time in $O(1)$. This look-up delay of TCAM is typically in the order of nanoseconds. TCAMs have a limited size with typical switch chipsets storing about 2K flow entries [20] while typical data traffic in a data center can be in the order of 10,000 new flows per second [21]. Due to limited size, flow table occupancy is managed by installing flow rules with an expiration timeout, known as idle timeout set by the controller. This timer decays over time if no incoming flow is matched to this entry, until it finally reaches zero and results in the flow entry being evicted from the flow table due to being idle.

A. Probability of Hit

Let p_{hit} be the probability that a new flow arriving at the switch is matched with an installed flow entry and processed immediately, without any communication to the controller. This match occurs at line rate in TCAM hardware. On the other hand, if there is no flow table entry for the new flow in the current flow table with probability $1 - p_{hit}$, additional communication between switch and controller occurs to install a new flow entry for this new flow.

In order to find the probability of having a flow hit, p_{hit}, we need to consider multiple parameters at the same time, namely, the flow entry expiration timer τ (in seconds), the flow arrival rate λ (in flows per second) and the capacity of the flow table σ (in flows). By Little's theorem, on average there will be $\lambda\tau N$ flows arriving in a time interval of τ, where N is the total different number of flows possible. However, if the flow table size is not sufficiently large to hold all these flows, there will be at most σ flows existing in the flow table. Thus, when a new flow arrives, the probability of hit will be proportional to $\min\{\lambda\tau N, \sigma\}/N$.

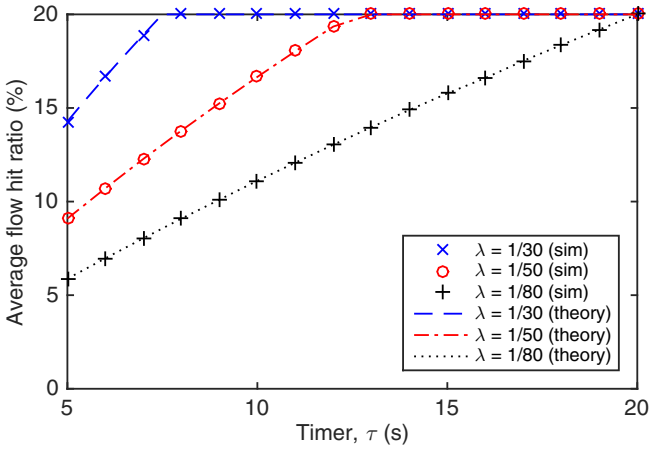


Fig. 5: Average flow hit ratio for different flow expiration timer (τ) values and flow arrival rates (λ), when N=1,000 and σ=200, where N is the total different number of flows possible, and σ is the flow table size.

On the other hand, it is possible that even though there is room in flow table for all arrivals, the old table entry may have expired for the newly arriving flow. Thus, we need to consider this possibility. Since the arrival rate of a flow is λ, the probability that another flow with same id will arrive before the timer expires (so, the probability of hit in this case) becomes the CDF of the exponential distribution by time τ:

$$p_{hit} = 1 - e^{-\lambda\tau}$$

Overall, the flow hit probability can be calculated by

$$p_{hit} = \min\left(\lambda\tau, \frac{\sigma}{N}, 1 - e^{-\lambda\tau}\right) \quad (1)$$

$$= \min\left(\frac{\sigma}{N}, 1 - e^{-\lambda\tau}\right) \quad (2)$$

Note that the first term in (1) will always be larger than the third term in (1) when $\lambda\tau > 0$, hence removed from the formula.

We have calculated p_{hit} in several scenarios using (1) as shown in Figure 5. We have also run simulations to compare with the theoretical results. To this end, we developed a Java based simulation environment and generated flows arriving with an exponential distribution with mean λ, to an SDN flow table of size σ. Once a new flow arrives and finds an existing entry in the flow table from previous arrival (as the timer of previous entry has not expired yet) it matches with the entry and a hit happens. Otherwise, a new entry is added by the controller to the flow table, with a higher processing delay. We have used least recently used (LRU) drop policy in the flow table. That is, if a new flow arrives and the flow

table is full already, the addition of the new flow's entry to the flow table causes the dropping of the least recently used flow entry. We ran each scenario for 10,000 seconds to get a stable behavior. The simulation results in Figure 5 show that average flow hit ratio for each scenario perfectly matches with analytical results, confirming the analytical model.

B. Reactive SDN Overhead

Referring to Figure 3 of the proposed architecture with two agencies A and B, with B wishing to provide access to a resource within its network to agency A, we have the following time delays:

Let the delay $t_{A_{RAL}}$ be the time for a message exchanged between agency A's SDN application and the RAL server; the RAL server may not be running at the same location as agency A. Assume that the SDN application runs on the same host as the SDN controller. Then, the time delay for a message between the SDN application and the SDN controller in agency A's network is $t_{A_{cap}}$. The delay $t_{A_{pki}}$ refers to the time for the switch to generate a *Packet In* message and send it to the controller, $t_{A_{fmd}}$ refers to the time for a *FlowMod* message sent between the SDN controller and the switch, $t_{A_{lkp}}$ be the lookup time in the flow table for finding a flow-rule match, and $t_{A_{fin}}$ is the flow installation time. Let $t_{A_{hst}}$ denote the time for message sent between the host and the switch in agency A. Similarly, for agency B, we have the following corresponding time delays, which may be different from agency A as B's network is owned and managed by a different entity: $t_{B_{RAL}}$, $t_{B_{cap}}$, $t_{B_{pki}}$, $t_{B_{fmd}}$, $t_{B_{lkp}}$, $t_{B_{fin}}$ and $t_{B_{hst}}$. Agency B also has a resource rendering time of $t_{B_{rsc}}$, which is the time it takes for the host machine (server) in agency B to fetch and process the data requested by the host (client) in agency A. Finally, let t_{net} be the propagation delay of communicating packets between the edge switch of agency A and the edge switch of agency B over the Internet.

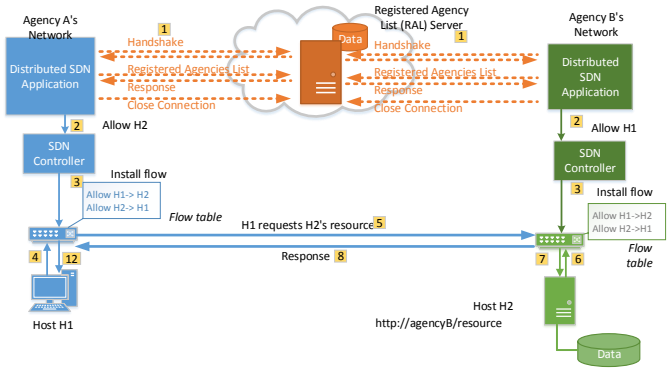


Fig. 6: Proactive SDN application setup, where flow rules are proactively pushed and installed from the controller to the switches when the network instantiates.

1) Proactive Scenario: In the proactive scenario (Figure 6), rules are pre-installed by the network admin into the flow tables. For the purpose of this analysis, we consider the flow entries in the proactive application to be permanent where both the idle timeout and the hard timeout are set to 0. If we assume the flow tables to be sufficiently large to allow pre-population of all flow rules required, then, the *end-to-end data access* delay from the host machine in agency A, h1, to the data repository behind the host machine in agency B, h2, is:

$$t_P = t_{A_{lkp}} + t_{B_{lkp}} + n(t_{A_{hst}} + t_{B_{hst}} + t_{net} + t_{B_{rsc}})$$

where $n \geq 2$ is the number of times communication exchange between A and B occurs. For simplification, assume that network setup is the same at both agencies with notations summarized in Table I. Thus, we can rewrite t_P as:

$$t_P = 2t_{lkp} + n(2t_{hst} + t_{net} + t_{rsc}), \quad n \geq 2. \tag{3}$$

TABLE I: Delay notations used in SDN overhead analysis

Symbol	Meaning
t_{RAL}	Communication b/w SDN app and RAL server
t_{cap}	Communication b/w SDN controller and app
t_{hst}	Time between A's host and switch
t_{lkp}	Switch flow table lookup time
t_{pki}	Switch *Packet In* message to controller
t_{fmd}	Controller's *FlowMod* message to switch
t_{fin}	Switch flow rule intallation time
t_{net}	Communication delay b/w A and B edge switches
t_{rsc}	Server resource rendering time

2) Reactive Scenario: In the reactive scenario, when there is a hit, the delay is $t_{Rh} = t_P$. And when there is a miss, the end-to-end delay will include all the transactions 1-17 in Figure 3. We can express this delay as:

$$t_{Rm} = 10t_{RAL} + 4t_{cap} + 2(t_{lkp} + t_{pki} + t_{fmd} + t_{fin}) + n(2t_{hst} + t_{net} + t_{rsc}), \quad n \geq 2. \tag{4}$$

Using the probability of hit from (2), we, then, write the average delay for our reactive SDN setup as:

$$\begin{aligned} t_R &= p_{hit} \times t_{Rh} + (1 - p_{hit}) \times t_{Rm} \\ &= p_{hit}t_P + t_{Rm} - p_{hit}t_{Rm} \end{aligned} \tag{5}$$

We can formulate the additional delay overhead of the reactive SDN setup as $t_R - t_P$. Subtracting t_P from (5) and simplifying, we get:

$$t_R - t_P = (t_{Rm} - t_P)(1 - p_{hit}) \tag{6}$$

Using (3) and (4), we write the first term of (6) as:

$$t_{Rm} - t_P = 2(5t_{RAL} + 2t_{cap} + t_{pki} + t_{fmd} + t_{fin}) \tag{7}$$

Thus, substituting (7) in (6), the overhead is:

$$t_R - t_P = 2(1 - p_{hit})c \tag{8}$$

where

$$c = 5t_{RAL} + 2t_{cap} + t_{pki} + t_{fmd} + t_{fin} \tag{9}$$

is the additional delay caused by agency A's flow table miss due to the reactive operation of informing the other agency as well as installing a new flow entry to the tables at both sides.

According to a study done by Cedexis Inc., the average cloud latency of major cloud service providers ranges between 63 ms and 104 ms [22]. If we assume RAL to be located in the largest average latency region, then we can consider $t_{RAL} = 104$ ms. The delay between the switch and controller is based on several factors such as a) the switch generating a *Packet In* message to be sent to the controller, b) time to send the message to the controller, c) time for the controller to process the *Packet In* message and communicating with the SDN reactive control application on how to proceed, d) time to generate OpenFlow control messages to send back to the switch, e) delay at the switch to receive a response from the controller in the form of *FlowMod* OpenFlow message(s), f) delay in installing the rule(s) which has to take into consideration the priority of rule(s) being installed and existing rules, and g) controller determination of evicting an existing rule (for example, using a Least Recently Used strategy) to make space for this new entry, should the flow table be at full capacity. These delays are dependent upon the optimizations implemented by the chipset vendor. For example, on Intel, the average delay of a)-b) is 8 ms per packet; and on Broadcom, the average delay of e)-f) varies between 3 ms and 30 ms [23]. The delay incurred by a TCAM-based flow table for rule installation varies between 33 ms to 400 ms in current OpenFlow implementations [24], with optimizations that improve the update delay to a median of less than 12 ms as shown in [25]. Using the end values of these ranges, we assume $t_{pki} = 8$ ms, $t_{fmd} = 30$ ms and $t_{fin} = 400$ ms.

Most reactive SDN applications are implemented in the same virtual machine or sometimes as an extension of the controller thread. In Floodlight, which is what we used in this paper, there is no additional network delay for communication between controller and reactive application since it requires the reactive application to reside in the controller's process thread as an extension. So, we assume the delay between the SDN controller and the reactive application t_{cap} is negligible. Bringing all of these time delays together, we anticipate $c \approx 1$ s in the current OpenFlow implementations and cloud services.

Figure 7 plots the overhead defined in (8) against varying λ and τ, and shows that as the idle timeout is increased, the reactive SDN application delay decreases and converges to the proactive case.

C. Optimizing the Idle Timer

As we have been studying, a key factor in the design of our reactive SDN application is the idle timer τ. From Figure 7, larger idle timer reduces the overhead of the distributed reactive SDN setup's end-to-end delay in serving the application running in agency A. As τ increases, the additional delay due to the reactive operation reduces and the overall delay in the reactive design gets closer to the delay in the proactive case. However, the other metric that plays a key role here is the flow table size. Given the probability of hit for a newly arriving flow is p_{hit} from (2), we can express the average flow table size as:

$$F_{avg} = p_{hit} N \tag{10}$$

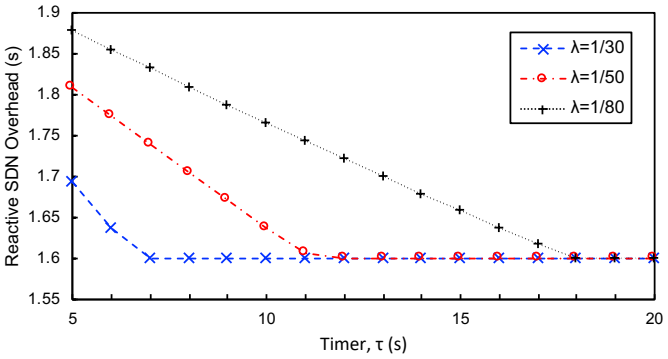


Fig. 7: The reactive SDN overhead (in terms of delay) for different flow expiration timer (τ) values and flow arrival rates (λ), when N=1,000 and σ=200, where N is the total different number of flows possible, and σ is the flow table size.

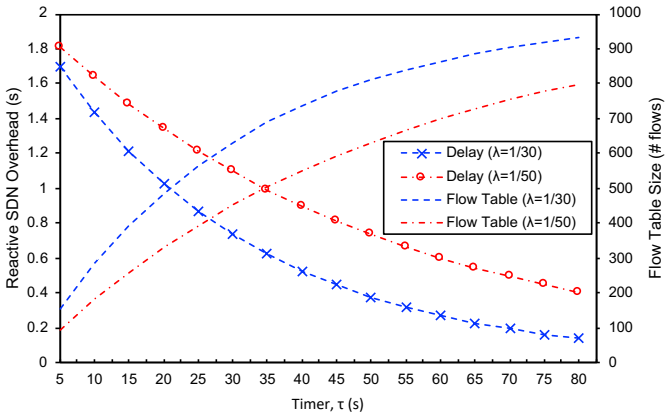


Fig. 8: The reactive SDN overhead vs. average flow table size (F_{avg}), for different flow expiration timer (τ) values and flow arrival rates (λ), when N=1,000 and σ=1,000, where N is the total different number of flows possible, and σ is the flow table size.

The above formulation also expresses the number of flows that are active in steady state.

In the proactive case, we assume the flow table size to be equivalent to all possible flows N. Yet, this is not realistic as a typical SDN setup may encounter millions of different flows. To find a middle ground, we can optimize the idle timer of our reactive SDN design so that both the reactive SDN delay overhead and the average flow table size are minimized. To do so, we update the performance formulations of the reactive case so that the flow table capacity is not an issue, i.e., $\sigma = N$ and compare the reactive delay overhead to the average flow table size as shown in Figure 8.

V. Simulation with Mininet

In this section, we describe our reactive SDN simulation setup and present results from our experiments.

A. Simulation Setup

The simulation setup consists of four VirtualBox Virtual Machines (VMs) on a host laptop as shown in Figure 9. Two of the VMs are running Floodlight ver. 1.0 controller [26] each with our custom reactive SDN application. Each Floodlight VM serves as the remote controller for a Mininet VM with a custom topology of a switch and two hosts. The two Mininet VMs represent a network belonging to different agencies and are able to send packets to each other over a Generic Routing Encapsulation (GRE) Tunnel. The RAL server, implemented as a multi-threaded Java server application, resides at the host machine where the VirtualBox is running.

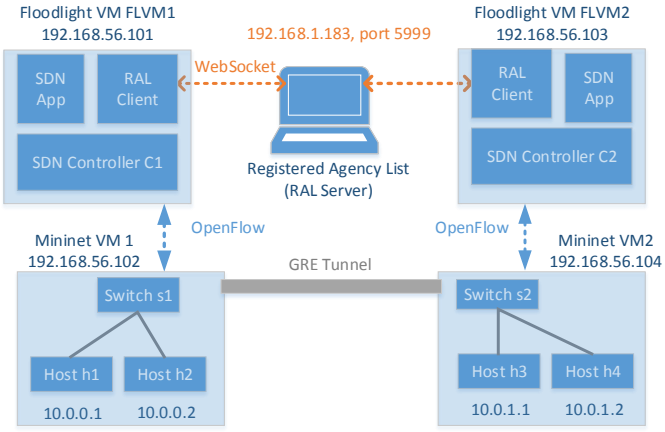


Fig. 9: Reactive SDN setup with two Floodlight VMs, two Mininet VMs, and a host running the RAL server.

B. Experiments and Results

Two different experiments were conducted to show that our reactive SDN application works successfully.

1) Test 1 – SimpleHTTPServer: In the first test, we registered hosts h1 and h4 with the RAL server, with h4 running SimpleHTTPServer at port 80. Then, we used the `wget` command to access this server from host h1 and then from h2. As expected, Figure 10 shows that h1 `wget` h4 is successful and we can see the page was returned. However, the request from h2 failed because h2 is not registered with the RAL server. The `dump-flows` of the two switches after the successful `wget` request from h1 and h4 shows that both have three rules installed to allow TCP traffic from h1 to h4 and from h4 to h1 and to drop any other TCP traffic with a low priority.

2) Test 2 – `iperf` Server: Second experiment was done by running `iperf` server on port 5,001 on the registered host h4 and running the `iperf` client first between the registered hosts h1 and h4 and then between the unregistered host h2 and the registered host h4. As expected, the request for registered resource h1 to h4 was successful but for unregistered resource h2, it timed out and failed. Dump of the flow tables revealed the allow rules for this flow that was installed on both Mininets automatically by our distributed reactive SDN application.

Extending this Mininet implementation to a more realistic setting is straightforward. To do so, one needs to replace the GRE tunnel between the two VMs with Internet and making the VMs reachable from the public Internet.

C. Challenges to Practice

In the reactive SDN application, the inter-play between idle timeout of different rules as well as the priority of the rules determines the overall behavior, and therefore, both of these should be carefully selected.

If a new agency is registered with the RAL after a drop rule was already inserted into the switch for that agency, then a smaller idle timeout would ensure that we get to query the RAL server again for a flow involving this newly registered agency and then have an opportunity to add new flow rules.

Assume now that the flow table is empty and a valid TCP request comes in for a source and destination registered with the RAL server. The reactive SDN application module installs three rules in the switch, one each to allow bidirectional TCP traffic flow from the registered source and destination, and a third one to drop any other packet. The problem here is again the same: If an invalid request follows, then, Rule 3's idle counter gets reset while Rule 1 and Rule 2 remain idle and get expired earlier than Rule 3. This may result in denial of service, if back-to-back invalid requests are received. This denial of services is not because the controller cannot respond to the frequency of *Packet In* messages but because the *Packet In* messages are not being sent to the controller as they keep getting matched on Rule 3. A new agency might have already registered with the RAL server but Rule 3's idle timeout may not have expired yet and would not allow the reactive application to query the RAL server to install the allow rules Rule 1 and Rule 2. The solution here is to keep the idle timeout of Rule 3 very small.

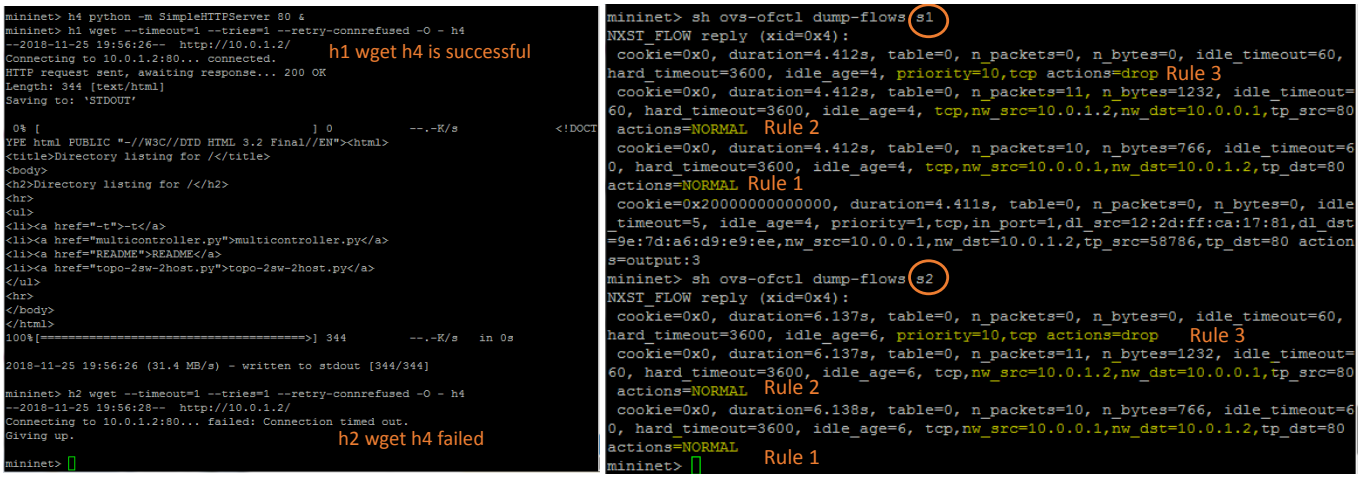
VI. Future Work

In future, we would like to explore Quagga [27] for a multi-domain simulation in Mininet with various routing protocols like Open Shortest Path First and Border Gateway Protocol and various traffic generators. One of the key challenges in the reactive SDN application is to intelligently insert new rules while resolving conflicts with existing rules added by other modules, for example, the Firewall module or Access Control List module. Furthermore, application-aware optimization of τ can be done through optimizing an objective function based on the reactive SDN overhead and average table size.

Consideration of other architectures without the RAL server will be a worthy effort to establish East/West interfaces among the SDN applications distributed at different domains. Finally, extending the architecture to more than two domains is of interest in terms of scaling the design for emerging SD-WANs.

VII. Concluding Remarks

In conclusion, we have presented a distributed SDN application architecture to allow logically distributed controllers to jointly install flow rules within their respective networks by talking with a central server. We have shown how such an architecture can be used to allow access to datasets owned



(a) Request and response

(b) Flow entries in flow tables

Fig. 10: With h1 and h4 registered and h2 not registered with RAL server, `h1 wget h4` is successful while `h2 wget h4` times out as shown in (a). Rules 1, 2 and 3 are installed on both switches after `h1 wget h4` as shown in (b).

by different domains, for example, between different research institutions, hospital networks, and public safety agencies during disaster situations. We anticipate that this approach is scalable to many agencies and does not suffer from traditional multi-controller design challenges like controller-controller coordination and placement. Furthermore, the controller in one agency does not need to have the global view of the networks of both agencies which was the case with the East-West Bridge [18]. Also, agencies can register at any time with the proposed RAL server, which allows ad-hoc inter-agency public safety communication networks to be set up and hook into the multi-domain data access framework.

VIII. Acknowledgement

This work was supported in part by NSF awards 1647189 and 1647217, and NIST grant 70NANB17H188.

References

[1] "OpenFlow Protocol," http://bit.ly/2DiGOZr.

[2] Brad Casemore, Petr Jirovsky, Rohit Mehra, "Worldwide Datacenter Software-Defined Networking Forecast, 2018-2022," Jun 2018.

[3] D. Raumer, L. Schwaighofer, and G. Carle, "MonSamp: A distributed SDN application for QoS monitoring," in *2014 Federated Conference on Computer Science and Information Systems*, Sept 2014, pp. 961–968.

[4] D. Levin, A. Wundsam, B. Heller, N. Handigol, and A. Feldmann, "Logically Centralized?: State Distribution Trade-offs in Software Defined Networks," in *Proc. of the First Workshop on Hot Topics in SDN*, ser. HotSDN '12, 2012, pp. 1–6.

[5] J. Yu, Y. Wang, K. Pei, S. Zhang, and J. Li, "A load balancing mechanism for multiple SDN controllers based on load informing strategy," in *2016 18th Asia-Pacific Network Operations and Management Symposium (APNOMS)*, Oct 2016, pp. 1–4.

[6] A. Tootoonchian and Y. Ganjali, "HyperFlow: A Distributed Control Plane for OpenFlow," in *Proc. of the 2010 Internet Network Management Conference on Research on Enterprise Networking*, 2010, pp. 3–3.

[7] T. Koponen *et al.*, "Onix: A Distributed Control Platform for Large-scale Production Networks," in *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'10, 2010.

[8] H.-z. Wang, P. Zhang, L. Xiong, X. Liu, and C.-c. Hu, "A secure and high-performance multi-controller architecture for software-defined networking," *Frontiers of Information Technology & Electronic Engineering*, vol. 17, no. 7, pp. 634–646, Jul 2016.

[9] "Project Floodlight," http://bit.ly/2HlARib.

[10] P. Berde *et al.*, "Onos: Towards an open, distributed sdn os," in *Proc. of the Third Workshop on Hot Topics in SDN*, 2014, pp. 1–6.

[11] Y. E. Oktian, S. Lee, H. Lee, and J. Lam, "Distributed SDN controller system: A survey on design choice," *Computer Networks*, vol. 121, pp. 100 – 111, 2017.

[12] K. Phemius, M. Bouet, and J. Leguay, "DISCO: Distributed multi-domain SDN controllers," in *2014 IEEE Network Operations and Management Symposium*, May 2014, pp. 1–4.

[13] M. A. S. Santos, B. A. A. Nunes, K. Obraczka, T. Turletti, B. T. de Oliveira, and C. B. Margi, "Decentralizing SDN's control plane," in *39th Annual IEEE Conf. on LCN*, Sept 2014, pp. 402–405.

[14] S. Hassas Yeganeh and Y. Ganjali, "Kandoo: A Framework for Efficient and Scalable Offloading of Control Applications," in *Proc. of the First Workshop on Hot Topics in SDN*, ser. HotSDN '12, 2012, pp. 19–24.

[15] D. Marconett and S. J. B. Yoo, "FlowBroker: A Software-Defined Network Controller Architecture for Multi-Domain Brokering and Reputation," *J. of Network and Systems Management*, vol. 23, no. 2, pp. 328–359, Apr 2015.

[16] F. X. Wibowo, M. A. Gregory, K. Ahmed, and K. M. Gomez, "Multi-domain Software Defined Networking: Research status and challenges," *J. of Network and Computer Applications*, vol. 87, pp. 32 – 45, 2017.

[17] Y. Jiménez, C. Cervelló-Pastor, and A. J. García, "On the controller placement for designing a distributed SDN control layer," in *2014 IFIP Networking Conference*, 2014, pp. 1–9.

[18] P. Lin, J. Bi, and Y. Wang, "East-West Bridge for SDN Network Peering," in *Frontiers in Internet Technologies*, 2013, pp. 170–181.

[19] R. Izard, "How to use OpenFlowJ-Loxigen," http://bit.ly/2CoB8vc.

[20] G. Lu *et al.*, "Serverswitch: A programmable and high performance platform for data center networks," in *Proc. of the 8th USENIX Conference on Networked Systems Design and Implementation*, 2011, pp. 15–28.

[21] T. Benson, A. Akella, and D. A. Maltz, "Network traffic characteristics of data centers in the wild," in *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement*, 2010, pp. 267–280.

[22] Brandon Butler, "Who's got the best cloud latency?" Jul 2016. [Online]. Available: http://bit.ly/2LagaaT

[23] K. He *et al.*, "Measuring control plane latency in sdn-enabled switches," in *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, 2015, pp. 25:1–25:6.

[24] M. Kuźniar, P. Perešíni, and D. Kostić, "What you need to know about sdn flow tables," in *Passive and Active Measurement*, J. Mirkovic and Y. Liu, Eds. Cham: Springer Int'l Publishing, 2015, pp. 347–359.

[25] X. Wen *et al.*, "Ruletris: Minimizing rule update latency for tcam-based sdn switches," in *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*, June 2016, pp. 179–188.

[26] R. Izard, "Floodlight v1.0," http://bit.ly/2Fx4GdZ.

[27] "Quagga Routing Suite," https://www.quagga.net/.