EAI Endorsed Transactions
on Internet of Things
Research Article EAI.EU

SAND: Smart and Adaptable Networking Design Using Virtual Slicing over Software-Defined Network

Farah I. Kandah[1,*], Steven Schmitt[2]

[1,2] Computer Science and Engineering Department
University of Tennessee at Chattanooga,
Chattanooga, TN 37403

Abstract

The importance of reliable and adaptable networks has become increasingly relevant with the escalation of connectivity in our lives. The growth of streaming of entertainment and development of always online software has created an environment of large data flows that need to be handled efficiently. Historically this problem has been solved with hardware-based load-balancers. These solutions often times are expensive and lack flexibility and scalability. With the use of Software-Defined Networking, a more dynamic solution can be created to meet network load balancing needs. We propose a Smart and adaptable network design seeking to utilize network resources more efficiently by identifying traffic patterns and analyzing network metric to dynamically build virtual slices. With this design, we were able to solve the aforementioned issues through minimizing packet loss, maximizing network link utilization, and efficiently reduce the load on the controller.

Received on 12 December 2017; accepted on 14 January 2018; published on 23 January 2018

Keywords: Load balance, SDN, Virtual Slicing.

doi:10.4108/eai.6-4-2018.155333

1. Introduction

The emergence of Software-Defined Networking (SDN) has brought along a wave of new technologies and developments in the field of networking with hopes of dealing with network resources more efficiently [1]. SDN allows for both flexibility and adaptability by separating the control and data planes in a network environment by virtualizing network hardware [2][3]. Through the programmablity features of SDN, handling data transfer between hosts become more efficient. The idea is to connect hosts through the use of virtual switches and virtual controllers that can be used to automate network functions programmatically. Our research seeks to develop an SDN system that uses metric-based analysis to virtually and automatically slice our network. These virtual slices will group hosts based on network activity aiming for efficient use of network resources. Our goal, by dynamically slicing the network based on metrics such as link utilization and packet drop rate, is to create an intelligent and adaptable network design that outperforms traditional methods.

The remainder of this paper is organized as follows: We will discuss the related work in Section 2 followed by our motivations and contributions in Section 3. Our problem statement is outlined in Section 4. Our proposed smart and adaptable network design (SAND) is presented and discussed in Section 5 followed by our analysis and performance evaluation in Section 6. We will summarize and conclude this paper in Section 7.

2. Related Work

Several solutions have been proposed recently with the aim on providing an efficient technique to balance the load on the network. Ghaffarinejad *et. al.* in [4] provided an SDN-based implementation of a load balancing scheme, which uses the flexibility of Openflow to

*Corresponding author. Email: farah-kandah@utc.edu

equally distribute the workload across the network servers. The authors in [5] proposed a cloud-based solution that focuses on equal distribution of the work over the network. The authors showcased both static and dynamic load balancing methods stating that dynamic load balancing performs better due to increased adaptability. We, in this work, present a smart and dynamic scheme based on network metrics to offer a more dynamic solution to load balancing by automating network decisions based on the current state of the network.

In [6], Cardellini *et. al.* presented a load balancing scheme, which consists of spreading network load across multiple web servers using a variety of methods such as client-based and DNS-based approaches. This approach does offer a solution to handle large flows of traffic at a macro level, but it fails to address the flexibility needed to survive in the current data climate. The solution also has a high entry cost due to the need for multiple web servers to achieve optimal load balancing. Our proposed scheme in contrast allows itself to be implemented in a variety of scenarios at low cost. The idea of virtually slicing a network using SDN technologies seeks to allow multiple tenants to occupy one physical network [7]. Our proposed scheme will use automate traffic pattern recognition to virtually slice the network, in order to create a dynamic network environment that can support the use of the network resources more efficiently.

3. Motivations

With many solutions focusing on balancing the load in the network, we observed that expensive hardware-based load balancing may be out of many customers' budget, while SDN based load balancing offers greater utility at a lower cost.

Our main focus in this research is to develop a network that is able to smartly handle network traffic by creating an adoptable network that is capable of automatically coping with different changes in network traffic. We observed that allowing the network to automatically measure the network traffic by recognizing traffic patterns through monitoring the network activities will tailor the network to any evolving traffic needs. Also, with this technique, we can create a more adaptable network that is able to auto slice itself to manage network traffic more efficiently, reduce the load on the controller, and act as a cheaper alternative to expensive hardware-based load balancing solutions [8]. Besides that, we observed that balancing the network traffic evenly across network's resources creates congested areas and might prevent the network from handling more traffic, therefore, we decided to consider link utilization while balancing the load in the network to ensure that traffic flows are routed efficiently and avoid creating small bandwidth holes, where the links have small remaining unused space and are not able to handle any more traffic.

We summarize our contributions as the following:

- Develop a dynamic traffic pattern analysis technique that will form a base for our network slicing.
- Identify the common network patterns based on hosts communication and interactions.
- Develop a dynamic and adaptable networking design through auto slicing to achieve our main network resource utilization and use the network resources more efficiently.

4. Problem Statement

The main issues with current load balancing solutions are the high costs and low flexibility and scalability. In our current technological environment, networks must be able to rapidly adapt to new changes and threats that may be introduced. Our research seeks to solve these issues by implementing an SDN based scheme that can dynamically adapt to network changes and traffic rates.

The following definitions are used throughout our research:

Definition 1. A ***virtual switch*** is a software-based implementation of a hardware switch in a network. It often serves as a node to route traffic between hosts on a network. □

Definition 2. A ***virtual controller*** is a software-based implementation of a hardware controller in a network. A controller often serves as a primary point of control that can designate work to network switches. □

Definition 3. A ***network link*** is a route between any two virtual switches that allows for the transmission of traffic. □

Definition 4. A ***flow*** is defined by the source host (src), the destination host (dst) and the requested bandwidth (B_{req}), which is defined by the amount of traffic being generated during the flow. □

Definition 5. A ***Path*** ($Path_{src-dst}$) is defined by the set of switches that form a path from the source host (src) to the destination host (dst). □

Definition 6. A ***Residual bandwidth*** (T_{res}) is the remaining unused bandwidth on the network link, which is given in Eq. 4.1, where T_{max} is the full bandwidth capacity of the network link, and T_{cur} is the bandwidth amount being used by the current flow. □

$$T_{res} = T_{max} - T_{cur} \tag{4.1}$$

Definition 7. A ***flow collision*** is the event where two large traffic flows are routed through the same network link which does not have the available bandwidth to accommodate the flows. □

Our ***Smart and Adaptable Network Design (SAND)*** problem can be stated as: *Given a bi-connected network topology, our design seeks to create a dynamic load balancing scheme that smartly routes traffic flows by virtually slicing the network based on common traffic flows with the goal to minimize packet loss and maximize link utilization while minimizing the load on the controller.*

5. Design and Implementation

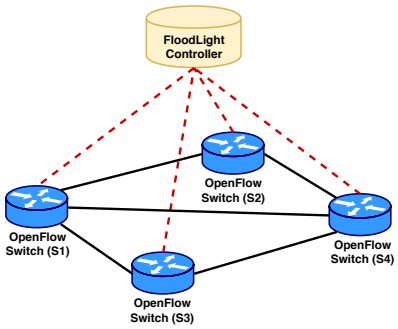


Figure 1. Biconnected Topology

The core idea of our research is to design a system that works with SDN to achieve dynamic resource utilization through virtual slicing. In order to achieve this we first designed a standard topology for testing that is at least bi-connected. This allows the formation of alternate routing paths when rerouting traffic flows. Fig. 1 shows the topology we will use throughout our discussion. Note that, our proposed design is not topology dependent and scales as needed by the network.

Once the OpenFlow switches are identified in the network, a pulling system is used to periodically retrieve network metrics to identify the traffic patterns. The metrics we gather consist of link utilization, network utilization, packet rate, bit rate, throughput, and packet loss.

For our evaluation, we will introduce three traffic routing systems; First, we will present the Floodlight's default forwarding system as a baseline for our comparison. Next, we will introduce the reactive based load balancing system presented in [5][9]. Finally we will introduce our smart networking design system that uses virtual slicing to make efficient use of the network.

5.1. Floodlight Forwarding

The Floodlight forwarding algorithm offers a routing system that finds a route between any two switches by always choosing the shortest available path [10]. An example of the floodlight forwarding scheme is illustrated in Fig. 2. It can be seen that no matter which hosts connected to S_1 are transmitting to hosts connected to S_4, the Floodlight routes traffic from S_1 to S_4 (highlighted with blue color) down the same link due to it is being the shortest available path (depicted with dotted lines). In this scenario if the link cannot accommodate the amount of traffic produced by both traffic flows packet loss will occur.

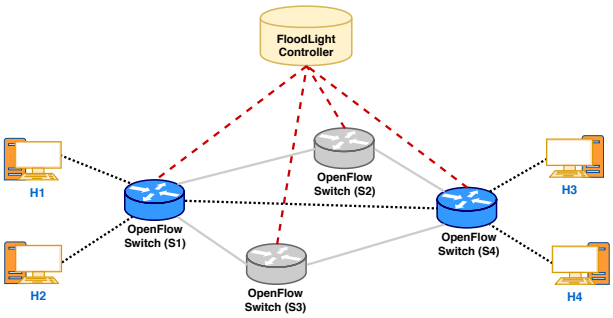


Figure 2. Floodlight Forwarding

5.2. Reactive Load Balancing

The reactive load balancing algorithm offers a routing scheme that routes new flows down the shortest available path [5]. The difference between the reactive and Floodlight algorithms is that when a link reaches a predetermined utilization cap the reactive algorithm will then reroute the flows to be evenly distributed throughout the network. This in fact avoids major packet loss, but does not attempt to efficiently use network links. An illustration of this scheme is given in Fig. 3. For instance, let us assume that host (H_1) is sending to host (H_4), and host (H_2) is sending to host (H_3), the reactive algorithm will reroute both traffic flows from S_1 to S_4 when the link cannot accommodate both of them (depicted by the dashed lines and blue highlighted switches). This in fact avoids major packet loss, but does not attempt to efficiently use network links.

5.3. SAND: Smart and Adaptable Network Design

Our smart and adaptable network design (SAND) begins with an ***initialization phase*** that initializes the controller (Algorithm 1). The controller will listen for OpenFlow switch connections on the network and find all available paths between each set of switches. This produces an ordered list of routes based on shortest path (Algorithm 1: Lines 1 - 3). The final step in the initialization phase is to setup the flow table entry (FTE) (see table 1) at the switches which will be based on whether there was any previous communications in the

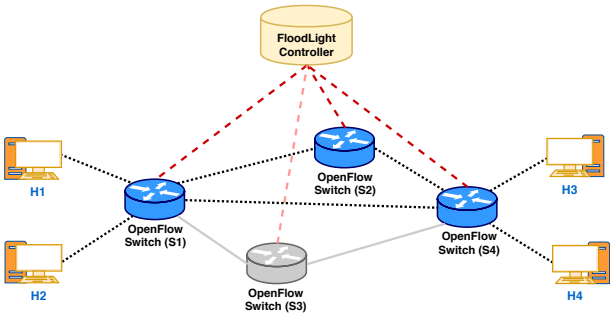


Figure 3. Reactive Load Balancing

network. With these flow entries we will setup different thresholds at which rerouting will be initiated.

As we are aiming to slice the network and utilize the resources more efficiently, we will define the *link utilization* $[L_U]$ as the current consumed throughput of a link $[T_{cur}]$ divided by the maximum available throughput of the link $[T_{max}]$ (Eq.5.1).

$$L_U = \frac{T_{cur}}{T_{max}} \tag{5.1}$$

With that, we also can define the *network utilization* $[N_U]$ as the summation of all current link throughput $[T_{cur}]$ divided by the max possible throughput across the network $[N_{max}]$ (Eq.5.2).

$$N_U = \frac{\sum T_{cur}}{N_{max}} \tag{5.2}$$

The flow table entry (FTE) at each switch will be checked on whether there was any previous entries. If found, the utilization threshold ($thresh_i$) will be set based on the previous traffic data requirement (T_{cur}), which is defined by the amount of required throughput that was consumed to satisfy that communication. This will allow us to allocate specific portion of the network that is being used and will lead to the creation of virtual slices in the network. If no prior traffic data exists on the OpenFlow switches, then all link utilization thresholds will be equal across all links.

Our SAND design starts with the initialization phase (Algorithm 1). An (available paths) set will be created to store all the available paths between the network switches, which will define the network topology and cover any future updates such as new available paths or in case of any paths failure (Algorithm 1 - Line 1). Another set (available links) will be maintained to contain all the available network links that will be used to form paths between sources and destinations based on the network flows and requests (Algorithm 1 - Line 2). A polling interval for network statistics collection will be defined, which will allow the monitoring of the network and cover any updates happen to the network

Table 1. Switch's flow table entry (FTE)

Flow ID	*src*	*dst*	in	out	*T*	Dur
ID	x	y	S_i	S_j	T_{cur}	TTL
ID	y	w	S_i	S_k	T_{cur}	TTL

that will drive the rest of the SAND design (Algorithm 1 - Line 3).

Algorithm 1 - SAND: *Initialization phase*

1: $AvailPaths \leftarrow \emptyset$
2: $AvailLinks \leftarrow \emptyset$
3: Define pulling interval: $I \leftarrow \Delta$;
4: $Prev_{flow} \leftarrow \emptyset$
5: Set the controller (C) to listen for connections;
6: **for** each switch (S_i) connected to C **do**
7: Assign switch ID
8: Identify available port(s)
9: **end for**
10: **for** each switch (S_i) **do**
11: $AvailPaths \leftarrow$ available path(s) to other switches
12: set $T_{cur} = 0$
13: **end for**

We will keep track of all the flows coming and leaving the network through the $Prev_{flow}$ set in which we will compare and update the network to make it self intelligent and be able to adopt based on the traffic being supplied to the network (Algorithm 1 - Line 4). At the initial stages of the network, the controller will communicate with the switches to realize the topology and maintain the set of all available paths which will drive the network resource utilization in the rest of the design (Algorithm 1 - Lines 5 - 13).

$$TLL = \frac{B_{req}}{D_{rate}} + \beta \tag{5.3}$$

$$T_{cur} = (T_{cur} + |B_{req} - T_{cur}|) * \alpha \tag{5.4}$$

After finishing with the initialization phase, we will move to the ***execution phase***, which actively identify the traffic patterns through monitoring the network activities and building initial slices (Algorithm 2).

We start by accepting flows in the network, which are defined by the source host src, destination host dst, and the requested bandwidth B_{req}. Since we already pulled the list of available paths (Algorithm 1 - Line 11) in the network, we can simply identify the path that will be used to support this flow ($Path_{src-dst}$). With that, we can start the process of updating the flow tables for all the switches within the $Path_{src-dst}$.

For each coming flow we need to check whether this flow was previously appeared in the network and it is still alive (*the TTL in the switch's flow table entry has not*

Algorithm 2 - SAND: *Execution phase*

1: **for** each $flow_{src-dst}$ **do**
2: **for** each switch $(S_i) \in path_{src-dst}$ **do**
3: **if** $flow_{src-dst} \in Prev_{flow}$ **then**
4: **if** $T_{cur} \leq B_{req}$ **then**
5: update FTE with T as given in Eq. 5.4
6: **end if**
7: **else if** $T_{res} \geq B_{req}$ **then**
8: $Prev_{flow} \leftarrow flow_{src-dst}$
9: Add FTE and set T as given in Eq. 5.4
10: **else**
11: Go to Algorithm 3
12: **end if**
13: **end for**
14: **end for**
15: Go to Algorithm 4

expired for that flow), Algorithm 2 - Lines(3-13). If the flow is still alive and the current threshold is less the the requested bandwidth, then we need to update the flow table entry with a new threshold as given in Eq. 5.4, otherwise, we will go through the updating phase (Algorithm 3). If this is a new flow and have never been identified previously, we will pull the best path from the *AvailPaths* set and assign it as a path for the flow as long as the residual bandwidth on the path's links satisfies the requested bandwidth, otherwise, we will go through the updating phase (Algorithm 3).

Algorithm 3 - SAND: *Updating phase*

1: **for** each network link $e \in AvailLinks$ **do**
2: **if** $e_{res} \leq B_{req}$ **then**
3: $SL \leftarrow e$
4: **end if**
5: **end for**
6: from all network links in $Succ_{links}$ find a path $path_{src-dst}$ between *src* and *dst*
7: $AvailPaths \leftarrow path_{src-dst}$
8: **for** each switch $(S_i) \in path_{src-dst}$ **do**
9: update FTE with T as given in Eq. 5.4
10: **end for**

If the flow was not previously identified in the $Prev_{flow}$ set, then we will revisit the network links to find a best path that can support this flow (Algorithm 3). We will start by pulling all the network links form the *AvailLinks* set, and check the residual bandwidth available on these links. If a link cannot support the flow request, due to unavailability of enough bandwidth, then the link will be excluded, otherwise it will be added to the successful links (SL) set (Algorithm 3 - Lines (1-5).

From all the links in the successful links set, we will find a path that can support the flow and add that path to the *AvailPaths* set (Algorithm 3 - Lines (6-7). With that, the last step will be to update the flow entries for the switches within the path to accommodate the changes in the network paths ((Algorithm 3 - Lines (8-10).

Algorithm 4 - SAND: *Re-engineering phase*

1: **for** each $flow_{ij} \in Prev_{flow}$ **do**
2: $S_i \leftarrow$ Switch connected to source i
3: $S_j \leftarrow$ Switch connected to source j
4: $Curr_{flow} \leftarrow flow_{ij}$
5: $Curr_{path} \leftarrow path_{S_i-S_j}$
6: **for** all $Paths_{S_i-S_j} \in AvailPaths$ **do**
7: Sort paths based on T_{res} in ascending order
8: **if** $T_{res} \geq B_{req}$ **then**
9: Assign this to be the path for this flow
10: update FTE with T as given in Eq. 5.4
11: **end if**
12: **end for**
13: **end for**

After assigning paths for the network flow, our design will go through the process of re-engineering and reorganizing the flows in the network to define common flows, build virtual slices and utilize the network resources more efficiently (Algorithm 4). The idea here is to look into the flows that are available in the network and reorganize them in an efficient way taking into consideration the links capacity, by switching the flows to different paths to utilize the network capacity more efficiently, and help the network to handle more future traffic with a lower drop rates.

We will start by looking into each flow, and identify the first switch (S_i) that is connected to the source host of the flow and the last switch (S_j) that is directly connected to the destination host (Algorithm 4 - Lines 1-5). Later, we will take the paths between S_i and S_j that are currently identified in the *AvailPaths* set and sort them in ascending order based on the residual bandwidth, and check if the requested bandwidth can be satisfied by the residual bandwidth, then the flow will be routed and the flow table entity for all the switches in the new path will be updated accordingly (Algorithm 4 - Lines 6-12).

We, with our smart, dynamic and adaptable design (SAND), can reserve link capacity for common network flows between hosts, while routing smaller network flows to fill in utilization gaps. In Fig. 4 we can see an example of how our algorithm slices the network based on prior traffic flows. In this scenario since hosts H_1 and H_3 commonly use a large portion of the link utilization between S_1 and S_4, we can reserve the space for future routing. The benefit of this is since we know an approximate amount of link utilization that can be expected, we can then reroute smaller traffic flows to

Table 2. Evaluation Flows

	Flow 1 (2 hosts)	Flow 2 (4 hosts)	Flow 3 (6 hosts)	Flow 4 (8 hosts)
Hosts	**Range (Mbps)**	**Range (Mbps)**	**Range (Mbps)**	**Range (Mbps)**
H1	2 - 4	4 - 6	6 - 8	6 - 8
H2	4 - 6	4 - 6	6 - 8	6 - 8
H3		2 - 4	4 - 6	4 - 6
H4		2 - 4	4 - 6	4 - 6
H5			2 - 4	2 - 4
H6			2 - 4	2 - 4
H7				2 - 4
H8				2 - 4

fill any gaps in link utilization, which allows us to leave more free links to accommodate future traffic flows in the network.

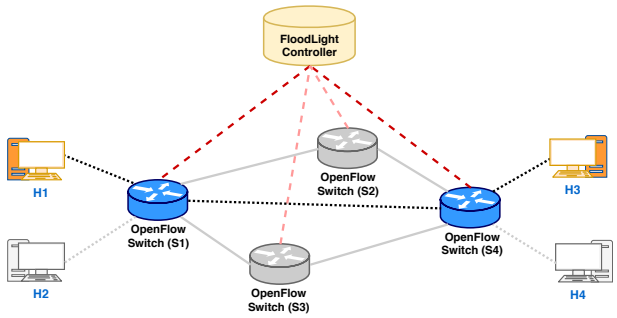


Figure 4. Our Smart Load Balancing

6. Analysis and Performance Evaluation

The foundation of this research uses SDN technologies to provide the functionality needed to implement solutions to dynamic load balancing. At the heart of software-defined networking is the functionality between the virtual controllers and virtual switches. For this research a Floodlight controller was used [10], along with OpenFlow switches to structure the topology of our network. The OpenFlow switches share direct connections with the Floodlight controller. In order to benchmark our network the network simulator Mininet was used [11], along with the Distributed Internet Traffic Generator (D-ITG) [12] to allow hosts to populate the network with multiple types of traffic.

To illustrate the performance of our scheme, we implemented our solution (denoted by **Smart** in the figures), and compared it with the default Floodlight forwarding scheme (denoted by **Normal** in the figures) and the reactive load balancing scheme in [4][5] (denoted by **Reactive** in the figures). To perform our experiments, we considered the network topology presented in Fig. 5. All links in our test network have a maximum throughput of 9.89 *Mbps*.

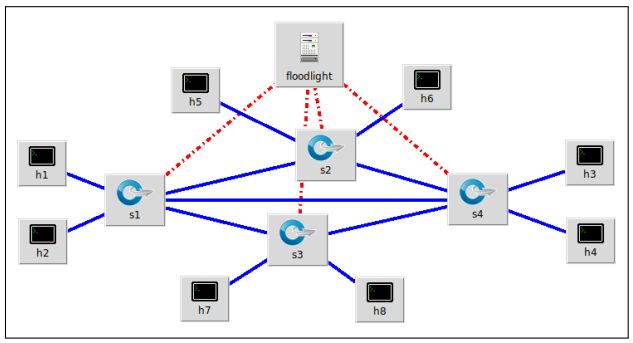
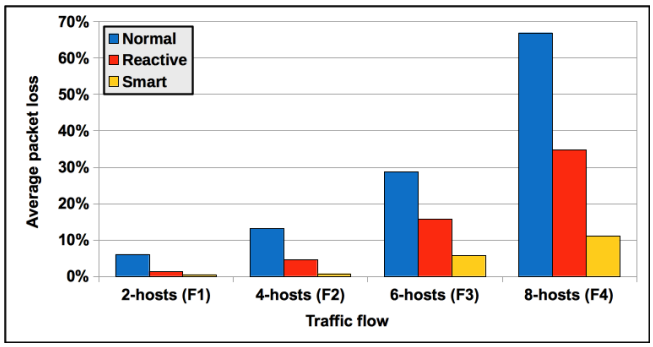
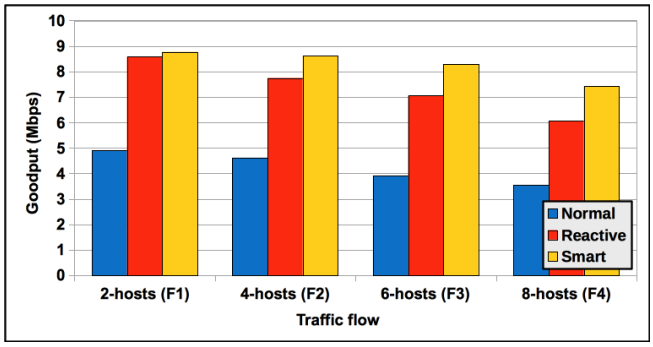


Figure 5. Test network architecture – Mininet

For our evaluation purposes, we defined four test cases consisting of different flows that simulate traffic being transmitted on the network for ten minutes as shown in Table 2. In each case we set each host to transmit data at a set rate with an average deviation of 1 *Mbps*. The deviation allows to simulate the instability in the throughput in the network. We considered flows with 2, 4, 6, and 8 hosts with each Openflow switch containing two hosts as shown in Fig. 5.



(a) Average packet loss



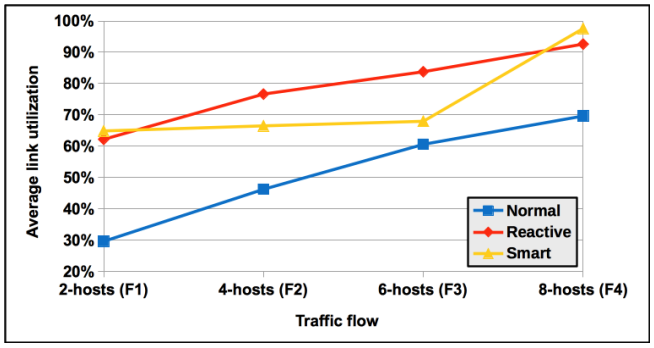
(b) Average goodput

Figure 6. Network performance

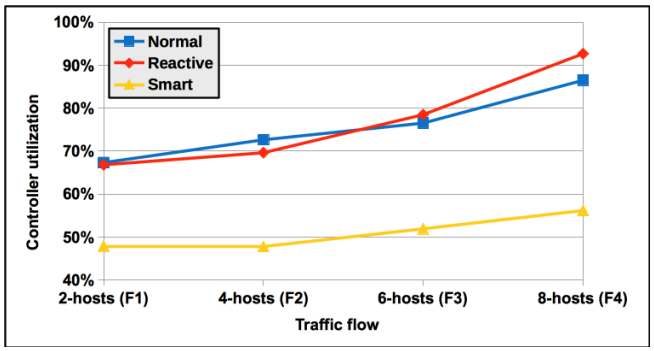
To show the performance of our proposed scheme compared to previous schemes, we considered the ***Average packet loss*** as our first metric, which is defined

as the amount of packets received at the destination host compared to the amount sent from the source host. Our results are presented in Fig. 6(a). It can be seen across all tests that our smart design outperforms previous schemes, which is attributed to our scheme's ability to use network links efficiently across the network, thus allowing the network to handle more traffic efficiently.

Our second performance evaluation metric is the ***Goodput***, which is defined as the amount of network throughput that resulted in successful transmission. Our results are presented in Fig. 6(b). Compared to the previous schemes, it can be seen that we were able to avoid drops in the goodput due to high link congestion, by dynamically slicing the network and allowing the reserved links to transmit data uninterrupted.



(a) Average link utilization



(b) Average controller utilization

Figure 7. Network resource utilization

To show the efficiency of network resource utilization, we considered the ***Average link utilization*** as our third performance evaluation metric, which is defined as the current consumed throughput compared to the maximum throughput for a link (See Fig. 7(a)). It can be seen that our smart scheme outperforms the reactive scheme by only utilizing links when necessary. For instance, in Flow 4 with 8 hosts, compared to the Floodlight and reactive schemes, it can be seen that our scheme uses the network's links more efficiently which resulted in maintaining more unused links' capacity to accommodate more requests and reduce the packet loss rate.

Our fourth performance evaluation metric is the ***Controller utilization***, which is defined as a combination of controller memory usage along with the amount of hits/requests the controller has to process during the test in which it will have to update the flows entries. Our results in Fig. 7(b) shows that, on average, our proposed schemes is able to produce less load on the controller compared to that with the other schemes. This is due to the initial virtual slicing that is done to the network. Since the reservations are already in place the Openflow switches can rely less on the controller for routing decisions. It can be also seen that both the Floodlight and reactive algorithm's controller usage scales heavily with the number of hosts involved.

7. Conclusion

In this work, we presented a smart, dynamic and adaptable network design (SAND) that is able to effectively mitigate large traffic flow while minimizing packet loss and maximizing goodput. By relying on the current state of the network we were able to automate the process of network decisions involving routing. In comparison to other related work, we have shown that our dynamic load balancing scheme has increased scalability and adaptability to network changes. We have shown that our research is a step in the direction of building more reliable and resilient networks that are able to adapt and scale based on the evolving data climate.

Acknowledgements. The authors acknowledge the support of the University of Tennessee at Chattanooga. Research reported in this publication was supported by the 2018 Center of Excellence for Applied Computational Science competition.

References

[1] Cisco. The zettabyte era: Trends and analysis. *Cisco*, page 7, June 2017.

[2] D. Kreutz, F. M. V. Ramos, P. E. VerÃŋssimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig. Software-defined networking: A comprehensive survey. *Proceedings of the IEEE*, 103(1):14–76, Jan 2015.

[3] T. Wang, F. Liu, and H. Xu. An efficient online algorithm for dynamic sdn controller assignment in data center networks. *IEEE/ACM Transactions on Networking*, PP(99):1–14, 2017.

[4] A. Ghaffarinejad. *Comparing a Commercial and an SDN-based Load Balancer in a Campus Network*. 2015.

[5] Pabitra Mohan Khilar Padhy, Ram. Load balancing in cloud computing. *International Journal of Recent Trends in Engineering and Research*, pages 260–267, 2017.

[6] V. Cardellini, M. Colajanni, and P. S. Yu. Dynamic load balancing on web-server systems. *IEEE Internet Computing*, 3(3):28–39, May 1999.

[7] Zdravko Bozakov and Panagiotis Papadimitriou. Autoslice: Automated and scalable slicing for software-defined networks. In *Proceedings of the 2012 ACM Conference on CoNEXT Student Workshop*, CoNEXT Student '12, pages 3–4, New York, NY, USA, 2012. ACM.

[8] Wolfgang Braun and Michael Menth. Software-defined networking using openflow: Protocols, applications and architectural design choices. *Future Internet*, 6(2):302–336, 2014.

[9] Hong Zhong. An efficient sdn load balancing scheme based on variance analysis for massive mobile users. *Mobile Information Systems*, page 9, 2015.

[10] Floodlight. http://www.projectfloodlight.org/floodlight/, 2017.

[11] Mininet. http://mininet.org/, 2017.

[12] Traffic generator D-ITG. http://www.grid.unina.it/software/itg/, 2017.