Network Function Virtualization Enablement Within SDN Data Plane

Hesham Mekky*, Fang Hao†, Sarit Mukherjee†, T. V. Lakshman†, and Zhi-Li Zhang*
*University of Minnesota. †Nokia Bell Labs.

***Abstract*—Software Defined Networking (SDN) can benefit a Network Function Virtualization solution by chaining a set of network functions (NF) to create a network service. Currently, control on NFs is isolated from the SDN, which creates routing inflexibility, flow imbalance and choke points in the network as the controller remains oblivious to the number, capacity and placement of NFs. Moreover, a NF may modify packets in the middle, which makes flow identification at a SDN switch challenging. In this paper, we postulate native NFs within the SDN data plane, where the same logical controller controls both network services and routing. This is enabled by extending SDN to support stateful flow handling based on higher layers in the packet beyond layers 2-4. As a result, NF instances can be chained on demand, directly on the data plane. We present an implementation of this architecture based on Open vSwitch, and show that it enables popular NFs effectively using detailed evaluation and comparison with other alternative solutions.**

I. Introduction

Network Function Virtualization (NFV) revolutionizes the design, deployment, and consumption [1] in virtualized data centers. Conventionally, a Network Function (NF), such as load balancer, is often implemented as a specialized hardware device. NFV decouples the NFs from the hardware platform and makes them run on software like virtual machines running atop a hypervisor on a commodity server, which reduces cost and makes deployment easier. A network service is created by chaining a sequence of NFs together, and routing the packets through the NF chain. Ideally, such a service chain is constructed on demand according to dynamic policy settings. Different service chains should be allowed to share certain NF elements for better resource utilization.

Software Defined Networking (SDN) is an ideal candidate for dynamic packet flow management for NFV since the SDN routers or switches support packet forwarding based on more elaborate flow definition than L2 or L3 destination addresses. This enables fine-grained routing control based on policies. In addition, the centralized control plane with complete knowledge of the network makes it possible to better optimize the forwarding path. However, there are a few drawbacks: (1) As traffic must be chained through NF entities, policy routing becomes inflexible and traffic choke points get created in the network, which is harmful and unnecessary; (2) The controller does not have full visibility into the NFs, e.g., how many instances exist, placement of instances, and traffic volume a particular instance can handle. Likewise, the control plane of the NFs is often exposed to limited network information. Such isolated system architecture leads to non-optimal NFs and network utilization; and (3) The NFs tend to change the state of the packets, and such changes are invisible to the SDN controller. There are various types of state changes by NFs: changing the packet contents (e.g.,NAT changes addresses/ports), dropping packets (e.g., firewall), or absorbing packets and generating new ones (e.g., L7 load balancer terminates client's TCP session and establishes new session with the appropriate server). The SDN controller remains unaware of how packets are modified by the NFs in the middle, and may lose the capability to track flows [2].

Two approaches have been proposed in the literature that address these issues from two angles. OpenNF [3] proposes a virtualized NF architecture where NFs are controlled by a central OpenNF controller that interacts with the SDN controller. It maintains two distinct sub-systems and NFs remain separate entities outside the SDN. Flowtag [2] proposes to use SDN to support service chaining by redefining certain packet header fields as tags to track flows. This still keeps NFs outside the purview of SDN. It also requires customized changes to each NF to make them tag-aware, which introduces dependency between the processing logic at different NFs.

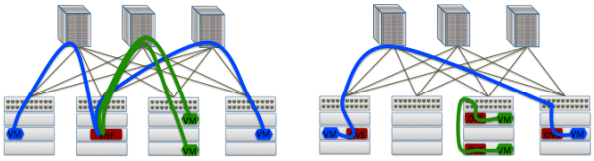
In this paper, we propose NEWS (**N**FV **E**nablement **W**ithin **S**DN Data Plane), a solution focusing on how SDN's complete knowledge of the network state can be maintained in a central controller while having it support NFs organically, efficiently and scalably. NEWS extends the current SDN architecture to make NFs integral parts of the SDN. This implies that there are no separate NF entities, and no separate control protocols for them. Chaining multiple NFs and scaling of the service happens natively within the SDN framework. We extend today's SDN architecture to address these issues. Our goal is to keep one controller in the network that is aware of all the states in the network, and can manage both the network and the network functions. Next, we present NEWS overview in Section II, the architecture in Section III and service chaining in Section IV. Then, we present our evaluation in Section V, related work in Section VI, and conclude in Section VII.

II. News Overview & Challenges

In NEWS, we address the following challenges.

Placement of NF: A naive solution is to implement NFs at the controller, and let switches forward packets to the controller. This is inefficient since the controller becomes a choke point leading to long delays and low throughput. NEWS extends SDN to support stateful packet processing natively at the hypervisor switches. This design takes advantage of the popularity of software switches in data centers [4], [5] as well as in service

978-1-5090-5336-0/17/$31.00 ©2017 IEEE



(a) Service chaining using NF VMs (traffic detouring & choke points).

(b) NFV enablement within SDN data plane.

Fig. 1. Current state of the art NFV vs. NEWS data plane.

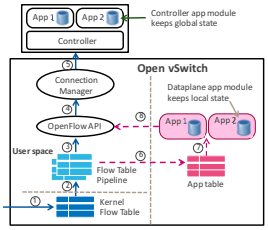


Fig. 2. NEWS System Architecture

provider's networks [6], where software switches run at the network edge to enable sophisticated policy and lower the cost, while conventional hardware switches are used in the core for simple and fast forwarding. Both the research community [7] and industry [6] have shown that software switches can reach processing capacity of hundreds of Gbps. The exponential growth in core density per host processor [8], [9] makes it feasible to run a number of data plane processing network functions at line rate within a server. While today's SDN allows the controller to install flow processing rules at the switches, in NEWS, we allow the controller to load application (app) modules at the switches; each app corresponds to a primitive function, e.g., a firewall consists of many primitive functions such as connection tracking, connection limiting, SNAT, etc. Apps are invoked in the same way regular flow actions are invoked (using OpenFlow vendor extensions), allowing NF logic to be executed natively in the data plane.

NF Primitives: NEWS takes advantage of the redundant functionalities implemented by different network functions to reduce the memory footprint, and break down NFs to basic primitives. For instance, the `iptables` firewall implements SNAT, DNAT, and ACLs, which are also implemented by load balancers such as HAProxy [10]. Therefore, NEWS apps are implemented based on the primitive building blocks of NFs, and they can be dynamically loaded by the controller.

Chaining NFs: Sophisticated network services require combining multiple NFs in a sequence. Since NFs are implemented as app actions in NEWS, NF chaining simply involves calling such app actions in a sequence. Within a switch, we create a logical chain of different app modules for each flow that requires a network service. In most cases, the flow gets complete treatment for the service in a single switch, avoiding inefficient detouring, choke points and packet copies (see Figure 1). Section IV discusses NEWS support for cases where the chain is long and involves multiple switches on the path.

Scalable Deployment: In NEWS, the controller installs an app module chain and the flow matching rules in a switch for network services. Scalability and elasticity are achieved by dynamically configuring the number of switches supporting a specific network service, and sending the flows to those switches using ECMP or other load balancing techniques [11]). This ensures that our proposed solution scales out with both the number of composed services and the traffic.

Dynamic Service Creation: The central controller must be aware of all services in the network and be able to add/delete services on demand. As evident from the above description, the central controller in NEWS is in charge of app module activation at the switches and sending the appropriate policy to the app using OpenFlow, e.g., "*load the connection limiting app and limit TCP connections to 100.*" We implement app modules as dynamically loadable software libraries that can be enabled/disabled remotely by the controller at runtime. Therefore, it remains aware of the network state, and creates/destroys NFs when needed. Thus, both network traffic and NFs are managed from one point of control.

We extend Open vSwitch (OVS) [12], for implementing NEWS. We quantify the amount of processing power needed at the switches to perform these additional functions. Note that some amount of computation power is needed to implement NFV, be it performed at virtual machine NFs or at extended OVS instances. Through experiments, we establish that in order to implement the same network service, the extended OVS instances are not using extra processing power than NFs and unmodified OVS instances combined, i.e., processing power is utilized in a different fashion at different places. Our experiments show that network services composed using NEWS offer competitive performance compared to existing solutions. For instance, the firewall service composed using NEWS is much faster than the conventional virtual firewall that runs in VMs, achieving performance very close to, and sometimes better than, the recently developed OVS conntrack [13]. Likewise, the Content Aware Load Balancer based on NEWS has much less delay and better scalability compared to the popular L7 load balancer HAProxy [10].

III. SYSTEM ARCHITECTURE

Virtual switches such as OVS are commonly available in data centers. They offer an ideal platform for our design since they open up the opportunity of inserting app logic in the data path via software extensions. We design a solution that extends OVS to be application-aware while conforming with the OpenFlow protocol. We also attempt to make the design modular, with a clean interface to OVS to allow new apps to be plugged in easily. Obviously, loading a full-blown NF into OVS is not going to be efficient unless OVS is given more cores that were originally given to the NFs, and even in that case, the binary can get bigger and may not fit in the cache. Therefore, in NEWS we break down these NFs into smaller building block functions that can be loaded independently. For

instance, iptables extensions, e.g., **connlimit**, **hashlimit** , **conntrack**, etc., are implemented as separate modules in NEWS. Similarly, other NFs such as LB and IDS can be broken down into basic building blocks such as L4 LB, L7 LB, and RegEx matching, where each block implements one app. In addition, the policy and initial state for the modules comes from the controller, e.g., number of connections to limit or RegEx values to use. For simplicity, we will refer to all firewall app modules as FW and all load balancer modules as LB.

A. Existing Open vSwitch (OVS) Design

Figure 2 shows the high-level architecture for NEWS. The four main components of the existing OVS are shown on the left, including connection manager, OpenFlow API, userspace flow table pipeline and the kernel flow table. The flow table pipeline contains one or more flow tables, each with flow rules that specify how matched packets are processed. A packet can be processed by multiple flow tables using the "goto" instruction in the rules. To speed up packet processing, the kernel flow table caches the flow actions so that active flows can be processed in the kernel. The solid blue arrows (steps 1-5) show how packets are processed by the application at the controller in OVS. An incoming packet is first matched with the kernel table. If there is no match, the packet is sent up to the flow table pipeline in the userspace. A *table-miss* rule is contained in each table, which tells what to do when the packet does not match any other rules in the table, e.g., goto to another table or to the controller. In the latter case, the OpenFlow API calls the connection manager to encapsulate the packet in a **Packet_In** message and send it to the controller. When the controller receives the **Packet_In**, one or more applications on the controller may process the message and install rules in the switch flow tables via a **Flow_Mod** message so that later packets are processed on the switch.

B. Design Choices

We intercept the packet before sending it to the controller, so that application logic can be applied to the packet without leaving the switch. The controller determines the rule on what apps are applied to the packet, and should also dynamically load and initialize the required apps as needed while conforming with OpenFlow. To do so, we need to first decide where to intercept the packet and where to maintain the application state. We consider the following three choices in turn:

Option 1: Controller proxy or message filter. The first option is to implement a controller proxy or message filter, which sits between the controller and OVS. It can be a standalone proxy colocated with OVS in the same host, e.g., FlowVisor [14]. Since the proxy monitors messages exchanged between the controller and switch, it can invoke the application logic to process packets locally before contacting the controller. To reduce overhead, we can implement a message filter module inside OVS, to intercept messages at the OVS connection manager, and call application logic. This option provides nice isolation between applications and OVS code. However, it leads to significant redundancy in packet processing. For instance, when the application intercepts the **Packet_In** message from the switch, it needs to decapsulate the already encapsulated message and recover the original data. The application also needs to implement its own flow table so that it can look up policy rules. Both message decap/encap and flow lookup add extra performance overhead and development cost.

Option 2: Stateful application module in kernel. The second option is to implement application logic in the kernel as a special action. This achieves good performance since packets are processed without leaving the kernel. The main challenge is the complexity of kernel development: beside writing the application code, one has to figure out how to integrate the kernel code with OVS in a modular and extensible way. One example of this approach is the recently developed OVS conntrack module, which supports a stateful firewall and NAT. A new **conntrack** action is added in OVS to track the flow state (new or established). A new **conn_state** metadata is added as part of the flow rule so that the flow connection state can be taken into consideration when packets are processed. The conntrack action is implemented using the existing conntrack module in Netfilter. Although this implementation has nicely integrated conntrack functionality into OVS, it has several limitations. First, conntrack is an independent module that is controlled by Netfilter, so the switching and firewall rules are installed by different entities. Second, this approach only works for Linux kernel since it uses conntrack kernel module. It will be difficult to port the solution to other data path implementations such as DPDK. Also, this approach is mostly suitable in cases where kernel modules are already available.

Option 3: Stateful application module in userspace with a stateless kernel. The third option is to intercept the packet at the end of userspace flow table lookup, and implement the application logic and state in userspace. To intercept the packets right after the standard flow table matching, we use the last flow table in the pipeline as a special *app-table*, shown in Figure 2. All table-miss rules with the action of "*output to controller*" are modified to "*goto app-table*", so that all packets that are originally processed by the controller will instead first pass through the app-table. Unlike the standard flow table, special *app actions* can be called from the app-table to handle the packets. Such app actions are implemented as OpenFlow vendor extensions. The dotted arrows (steps 6-8) in Figure 2 show how the packets are "detoured" to the app-table in the our architecture. For simple actions that require fast processing but does not require maintaining state, we can still implement them as kernel actions, e.g.,TCP splicing in an example of such action, shown in Section V. We believe this option achieves a good balance between ease of implementation and performance. Hence we use this approach in NEWS.

C. Integrating with OVS flow processing

In many cases, part of the app logic can be executed using existing OVS flow tables. For example, if a FW app decides to drop a flow after processing the first packet, it can generate a regular flow rule and insert it into the OVS flow table, then later packets can be handled directly by the standard flow

tables. This is done using OvS flow insertion APIs from the app to insert the rule into the userspace flow table. However, our experiments showed that userspace flow insertion is a significant performance bottleneck because of OvS locking for the userspace flow tables to support multi-threading; each thread acquires a lock before reading/writing to the flow tables.

We instead insert the rules directly into the kernel table via **`netlink`**. Although the kernel flow table also has a lock, the lock operates much faster in the kernel than in the userspace. However, the kernel flow is only a temporary cache, in which the flow entries may be replaced based on their access patterns. To maintain flow state consistently, the flow rules generated by the apps are also kept in an interim rule cache in the userspace. If the kernel flow entry is deleted before the flow ends, the next packet that arrives will be delivered to the userspace and the app rule will be invoked. Before invoking app actions, we check to see if the packet matches any existing flow rule in the interim rule cache. If so, the matching rule is inserted back into the kernel; otherwise the packet is processed by the apps and a new flow rule can be generated. The implementation of the app interim rule cache is much simpler than the OvS flow table since there is no need for pipelining or priority matching. Read and write locks are used for the interim rule cache so that multiple reads can be executed in parallel. To reduce the locking granularity, a hash function is used to partition the entire interim rule cache into multiple hash tables; each table is implemented as a separate hash table that can be locked individually. This new design has improved the performance significantly over our initial design, e.g., the LB that used to support fewer than 100 concurrent flows, can now support one to two orders of magnitude more flows.

To make the apps dynamically extendable and loadable, the app actions are made transparent to OvS. Only one wrapper action **`news`** is exposed to OvS, as an OpenFlow vendor extension. The individual app actions are implemented as subactions, which are in turn invoked from **`news`** engine. For example, when OvS invokes the app action **`fw`**, it invokes it as **`news(app=fw)`**, invoking action **`news`** and passing **`(app=fw)`** as the parameter. The **`news`** action handler then calls the actual app based on the subaction contained in the parameter, in this case **`fw`**. In this way, apps can be added or removed from NEWS at runtime without requiring OvS recompilation. Since **`news`** action always precedes any NEWS apps, i.e., the **`news`** action is a demultiplexer. We omit **`news`** when writing the app actions in the rest of the paper for simplicity.

D. App Table and App Actions

The app table operates similar to the standard flow table. Packets are matched with OpenFlow rules, and the corresponding actions are executed for the matching rule. If matching fails in the app table, the table-miss rule is used to send the packet to the controller. The controller installs/removes rules from the app table using the standard **`Flow_Mod`** command.

App actions are specified as *vendor actions* in the OpenFlow protocol [15]. The app table rules run only at the userspace; they are not installed in the kernel to avoid slowing down the fast path. An app action may do any combination of the following operations: (1) determine actions for the current packet; (2) modify its local state; (3) generate or modify the rule to be installed in the kernel table for processing this flow and also update the interim rule cache entry in userspace; (4) remove flows from kernel table and the userspace interim rule cache; (5) generate a packet out to other switches; and (6) send **`Packet_In`**, **`Flow_Removed`**, or **`App_Update`** vendor messages to the controller. Operations (3) to (6) are done by calling the set of APIs exposed from the OpenFlow API module and OvS APIs (Figure 2 step 8), which include the following: **`add_flow`**, **`del_flow`**, **`packet_in`**, **`packet_out`**, and **`flow_removed`**. Each app also provides a message handler, so that it can be called from **`news`** when the controller sends messages to the app. Message exchanges between the controller and local app modules are encapsulated in OpenFlow vendor messages, and they are passed to the app through **`news`**, and therefore OvS does not need to understand the format of these messages, or be changed after adding new apps or messages.

E. App Chaining and Execution

Multiple apps can be chained together to implement complex network services. The order of app actions are determined based on the service policies. Each app action may modify the packet header and its metadata, and also generate or modify the actions that are to be installed in the kernel flow table. Such actions are stored in the interim rule cache that we described in Section III-C. The first app that processes a flow creates an entry for the flow in the interim rule cache. This entry stores the rule that is generated by the apps for this flow. The last app action in any app chain is always **`Install`**, which installs the rule set into the interim rule cache and kernel table. The flow's metadata contains the index of the flow in the interim rule cache and also the **`break`** flag, which can be used to prevent later apps in the chain from execution. For example, the firewall app can decide to drop the packet and set **`break=true`**, so that the following LB app can ignore the packet. Note that all the app actions are still invoked – we do not change the OpenFlow semantics, but the apps ignore the packet when they see **`break`** being set. However, the **`Forward`** and **`Install`** apps always execute regardless of the flag. **`Forward`** determines the output port for the packet unless there is a prior action **`drop`**. **`Install`** calls OvS APIs to install the rule unless the action set is empty.

F. Flow Installation and Removal

All the flows in the userspace, standard flow table or app table, can be installed/removed by the controller. The kernel flow entries are installed by the userspace OvS daemon, either triggered by the standard flow table rules or by the app table. Likewise, when the kernel flows are removed, the corresponding standard flow table rules or app table rules are invoked to update statistics and counters. We extend OvS with a flag to indicate that the current execution is part of flow-removal process, and therefore NEWS apps can perform any required logic that is part of this flow removal. An example of this process is explained

later in the firewall app (Section V). Flows in the interim rule cache are maintained based on the app logic.

G. State Management

Apps reside in the controller and the OvS data plane, where each data plane app module maintains local state accessed during packet processing. The controller app maintains the aggregated state for the app to allow for a global view of the NF. The controller app is responsible for managing the global policy/state. We used vendor extensions in OpenFlow to add a new message that is used to push/pull state between the controller app and their corresponding slaves in the switch data plane. This message includes the following fields: **app_id** , **msg_id**, and **data**. We assume that the controller app developer will agree with the switch app developer on the messages that is used to communicate state. The **app_id** is used by NEWS to pick the app that is responsible for decoding this message, and the **msg_id** is used by the app to decode the actual **data** sent by the controller app, which enables the app on the controller to send different messages to the app on the switch. For instance, the **app_id=2** identifies the L7 LB, and the **msg_id=1** identifies the **path_end** message, which is used to match against HTTP GET requests ending with a specific value, and the rest of the **data** is just the array of URLs and the end-servers IPs.

H. An Example: Firewall & Load Balancer

The following example shows how NEWS architecture works. Suppose the network policy is *"web traffic to server x needs to go through the FW then the LB"*. The FW rule specifies that only web traffic is allowed, and maximum number of active TCP connections is 1,000. The LB rule is to distribute the load to servers s1 and s2 by hashing according to the source IP address. The controller sets up the policy by inserting the rule **(dst_ip=x,tcp,dport=80: fw,lb,fwd,install)** to the app table, where **fw**, **lb**, **fwd**, and **install** will call Firewall, Load Balancer, Forward, and Install apps, respectively. When a new flow **(src_ip=a,sport=6000,tcp,dst_ip=x,dport=80)** arrives, it gets sent from the kernel to the flow table in userspace. The flow table table-miss rule will send the packet to the app table, then **news** invokes the apps as follows.

1. Firewall keeps track of the number of active flows using a counter **n_flows**. Suppose currently **n_flows < 1000**, then the following rule is generated and stored in the interim rule cache: **(src_ip=a,sport=6000,tcp,dst_ip=x,dport=80: [])** and **n_flows** is incremented. The empty action set indicates the flow is accepted but no actions produced. Firewall also sets **break=false** flag and stores the rule's index in the packet metadata.

2. Load Balancer looks up the server IP address using **hash(src_ip)**. Suppose the hash result is **s1**, then the rule in the interim rule cache is updated: **(src_ip=a,sport=6000,tcp,dst_ip=x,dport=80: set dst_ip=s1)**. It also changes the **dst_ip** of the packet from **x** to **s1**.

3. Forward looks up the routing table to find the output port **pt1** based on **s1**, then it updates the rule in the interim rule cache: **(src_ip=a,sport=6000,tcp,dst_ip=x,dport=80: set dst_ip=s1, out=pt1)**.

4. Install retrieves the rule from the interim rule cache and installs it in the kernel and the userspace interim rule cache, and sends the packet.

As a result, the apps jointly generated the flow rule according to the network policy without going to the controller. Note that Firewall drops the flow if **n_flows > 1000**, and it sets **break=true** flag in metadata and generates the rule: **(src_ip=a,sport=6000,tcp,dst_ip=x,dport=80: drop)**. This causes LB and Forward to ignore this packet. At the end, Install sets up the drop rule into the kernel table and drop the packet. One issue not covered so far is handling flow termination. When a flow ends either by a FIN or RST packet or timeout, the firewall must decrement **n_flows**. Hence the firewall generates a slightly different rule: **(src_ip=a,sport=6000,tcp,dst_ip=x,dport=80,tcpflag=-fin-rst: null)**; therefore, regular packets without FIN or RST flags being set are forwarded, while the FIN or RST packets will not match this rule and will go to userspace. This will trigger the app table rule and cause the firewall app to decrement **n_flows**. The firewall app also maintains flow idle timer and removes the rules and decrements **n_flows** when the flow expires.

I. Loadable App Actions

NEWS design allows new apps to be loaded at runtime without the need to restart the switch or recompile the code. We implement an OpenFlow vendor action **news**, and invoke all the apps as subactions inside **news**. To understand the dynamic app loading process, let us start from a clean state with no apps. To load an app, the controller issues a custom OpenFlow vendor message **Load_Module(app_id,app_name)**, which triggers NEWS to lookup the file **libapptable-<app_name>.so** in a pre-determined path. Then, NEWS loads the app dynamic library and stores the **(app_id,app_name)** mapping, and invokes the app **init** method to initialize the internal state of the app. On success, **news** sends back to the controller app success message, and afterwards the controller can push policy to the app. All apps are required to export three functions:

- **init(handlers, revalidators)**: initializes the app with the number of threads for packet processing (**handlers**) and cleanup threads (**revalidators**).
- **xlate_actions(flow, actions, packet)**: executes actions for the given flow in the interim rule cache.
- **destroy()**: destroys the app and cleans internal state.

As a result, apps are developed independent of OvS and are loaded during runtime. The controller dynamically enables/disables apps on the data path for a flow based on the flow's service requirement and resource availability at various switches. Furthermore, this design is by and large independent of OvS data path implementations, and can be easily adapted to run on other OvS data paths such as DPDK or a userspace data path. As we will show in Section V-C, the overhead for loadable modules is negligible compared to compiled modules.

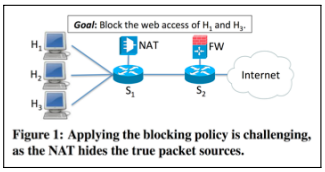


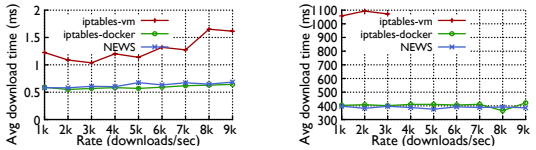
Fig. 3. Reproduced from [2]: Service chaining example 1

IV. Service Chaining

A service chain is created by routing flows through a sequence of NFs. The NFs in a chain modifies the packet headers without coordination with other routers and NFs. This makes steering a flow through a service chain hard as the downstream routers or NFs may depend on the unmodified headers. Figure 3 shows an example where a NAT and a firewall are chained together. It is hard for the firewall to implement the policy "*block web access of the internal hosts H_1 and H_3*" since it can only see the address set by the NAT, not the original address. Flowtags [2] proposes to use a flow tag to track flows. NAT tags the packet according their original address, and the firewall sets up the rule according to the tags. Different NFs must agree on the definition of tags, and unused fields in the header must be agreed upon for reading/writing tags. This scheme accommodates hardware appliances and/or software instances of stand-alone NFs after modifying them generate and/or consume flow tags.

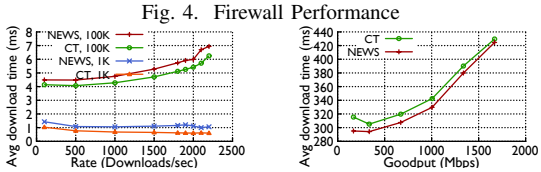
NEWS is a complete software-defined approach for integrating the NFs into the network itself by installing app modules in the virtual switches running on servers. The exponential growth in core density per host processor [8], [9] makes it feasible to run a number of data plane processing NFs at line rate within a server. Therefore, NEWS tries to fit the whole service chain within a single software switch. This gives us an opportunity to address the issue in Figure 3 differently: instead of using dependent flow tags, we adjust the placement of NFs in the service chain to make sure each NF is exposed to the information that it needs to access. NEWS solves that example by adding the following match/action rule: **`(in_port=1: fw, nat, fw, fwd, install)`**. We insert another firewall action at the left side of NAT, which can see the original host addresses and hence can enforce the rule. Note that we cannot simply move the firewall from the right side of NAT to its left because the firewall is also needed for filtering external traffic from the Internet. As a result, there are two calls to the firewall app in this chain. The left firewall enforces internal rules and the right one enforces external rules. Although this approach seems simple, it may be too expensive to implement if the NFs are conventional hardware appliances or even virtual instances, since duplication of NFs would incur significant hardware expenses and also introduce delays in packet processing. In NEWS, since NFs in a service chain are simply software modules that can be invoked within the same thread, they can easily be duplicated without adding overhead.

We believe normal service chains in practice (consisting of 3-4 NFs) can be easily accommodated within a software switch



(a) Small flows (1KB) (b) Large flows (10MB)

Fig. 4. Firewall Performance



(a) Small/Medium (1/100KB) flows (b) Large flows (10MB)

Fig. 5. Connection Tracking Performance

running in a single server. For elasticity, the server instance can be horizontally scaled out to handle loads. NEWS does not necessarily restrict the chain to one server. There may be cases where the service chain may be too long [1] or the chain's processing may be too heavy duty to fit in a one server. In such cases, the chain is subdivided into two or more sub-chains such that each sub-chain fits into a server. Each sub-chain instance can scale out independently as needed. When packets need to be forwarded downstream based on packet headers that is already modified by an upstream sub-chain, tunnels can be used to connect the sub-chains together. In NEWS, the tunnel header is added in the **`forward`** action shown in Figure **??**. Note that the NF modules along the chain do not need to be aware of the tunnels. The controller determines if tunnels are needed and if so, what tunnel id should be used.

V. Evaluation

We built two NFs in NEWS: a firewall and a content-aware LB. We choose to build a firewall since it is commonly used in practice, and to compare NEWS with OvS conntrack. The content-aware LB is used to demonstrate the benefits of the scale out architecture. We also evaluate the impact of the dynamic enablement of apps on data path performance. We used five Linux (Ubuntu 12.04) machines with Xeon(R) CPU (12 hyper-threads @2GHz, 15M cache, and 16G RAM). Each machine has an Ethernet Quad 1Gb NIC. One machine acts as a middlebox running the system under test, e.g., NEWS, iptables, etc. Two machines act as clients, each running 5 containers generating load, and two machines act as servers, each running 5 containers running Apache servers. We implemented our apps by extending OvS v2.3.0 and the Floodlight SDN controller.

A. Firewall Connection Limiting & Connection Tracking

The firewall service is composed using the following app chain in NEWS: **`(connlimit=<limit>,fwd,install)`**. As described in Section III-H, a firewall app is implemented to support **`connlimit`** action. The app maintains a firewall rule table, and applies the rules to the input packet when called.

[1] Although chains with more than 4 NFs are rare in practice.

fwd action determines the output port for this packet, and **install** action inserts the rule into the OvS kernel table. The userspace flow table contains ten rules [2] each one handles load generated from one client, and a miss in the fast path will occur when a new micro-flow comes in. For instance, in the x-axis in Figure 4a 5k/sec load corresponds to 5k micro-flow misses in the kernel fast path that are matched against the ten rules in the userspace slow path.

We compare our prototype with the Linux **iptables** under two common setups: (1) **iptables** in a Virtual Machine (referred as iptables-vm); and (2) **iptables** in a container (referred as iptables-docker). In both cases, OvS runs in the host to "chain" the service, i.e., forwards the packet to and from **iptables**. To compare the three firewall services: NEWS based, iptables-vm and iptables-docker, we run each of them one at a time, in the same host with the same resources. To simplify the experiment, only one firewall app is used in the chain (**connlimit**): a new TCP connection is rejected if the number of active connections to a server exceeds a threshold; the connection is accepted otherwise. A RST packet is sent back to the source when the connection is rejected. To improve performance, the internal state of the firewall app is partitioned into multiple hash tables protected by read/write locks.

Figure 4a shows the end-to-end performance of the three firewall services with small files (1KB). The HTTP requests are generated at rates ranging from 1K to 9K requests/second. We observe that NEWS and iptables-docker have similar performance. iptables-docker and NEWS are much better than iptables-vm. iptables-docker has the best performance because there is no packet copying in that configuration. Since all packets are handled in the kernel, and there is no copying between OvS kernel and iptables kernel module, the data path is very efficient. On the other hand, iptables-vm introduces the most overhead because packets have to be copied between the host and the guest VM, which slows down the data path significantly. To understand why NEWS is close to iptables-docker but slightly worse, recall that in NEWS, the first packet of a flow is forwarded to userspace, and then a micro-flow kernel rule gets installed. Later packets of the flow are then handled within the kernel, and therefore NEWS incurs slightly more overhead for the first packet and none for other packets. Figure 4b shows the performance for large files (10MB). Similar to the small file size case, iptables-vm has the worst performance. In fact, in iptables-vm many downloads cannot complete due to packet losses and timeouts when the rate reaches beyond 3K downloads/sec. Hence the corresponding curve only shows three data points. Interestingly, we find that NEWS is slightly better than iptables-docker for large file downloads because when the flow contains many packets, the overhead of first packet is less significant. In iptables-docker, even though packets do not leave the kernel, they are processed by both OvS and iptables, which incurs slightly more overhead compared to NEWS, where packets are only processed by OvS.

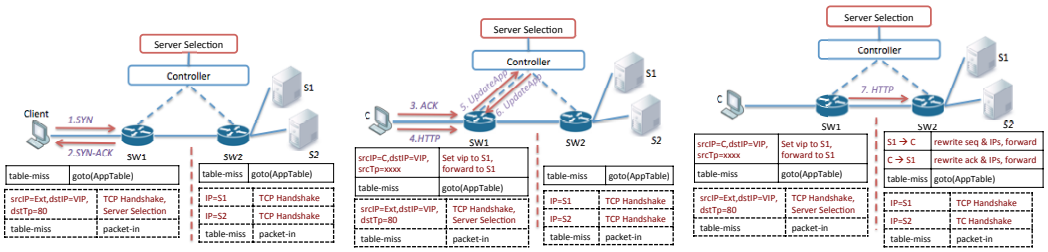
[2]In the experiment, we vary the number of rules ranging from 10 to 50K and the performance remains almost the same.

We also compare the performance of NEWS and OvS conntrack [13] implementation which integrates iptables conntrack into OvS. In connection tracking, connections from clients to servers are accepted, while servers to clients are rejected. Figure 5a shows the average download time under various rates. For file sizes of 1KB and 100KB, we find that OvS conntrack is slightly better than NEWS, although the difference is not significant. This is not surprising since the OvS conntrack uses the same conntrack module from iptables, hence its performance should be comparable to iptables-docker. We also observe that the difference in download time for the two implementations are close (about 0.5ms) under the two significantly different flow sizes. This difference is likely caused by the processing delay of the first packet of each flow at userspace in NEWS. Figure 5b compares the average download time vs. goodput for conntrack [13] and NEWS when downloading 10MB files. Goodput is defined as the ratio of the amount of user data transmitted successfully divided by the time duration. We find that NEWS is slightly better than conntrack, which is consistent with our observation that OvS conntrack incurs more processing overhead in the kernel since the packets are processed by both OvS and the conntrack module. In NEWS, packets are only processed by the OvS data path module. For large flows, the extra kernel processing overhead in OvS conntrack is more significant than the overhead from the first packet userspace processing in NEWS.

B. Content-aware Server Selection

To enable Content-aware Server Selection, we extended the OvS kernel module to support TCP SEQ/ACK rewriting actions, since they are simple/stateless and has to be applied to all packets. More complex but less frequently invoked actions including TCP handshake and server selection are implemented as app actions in the userspace. In server selection, a client's request to a virtual IP address, VIP of a server pool, is redirected to an appropriate server dynamically based on the request URL. This is done in two steps: (1) the data center gateway distributes requests to multiple front-end vSwitches using ECMP; (2) the front OvS selects the server based on the request URL, and forwards the packet to the back-end OvS, then to the server. On the return path, the packet is forwarded by the back-end OvS directly to the client. Figure 6 shows a simple mapping of the app onto the proposed architecture with intermediate message flow. We omit the gateway and first stage ECMP, and just show the two Open vSwitches: front-end (SW1) and back-end (SW2). Standard OpenFlow flow tables are shown in solid lines, and app tables are shown in dashed lines. In our implementation the server selection app resolves the VIP to either S1 or S2. Initially the controller adds default rules to submit table-misses to the app table in SW1 and SW2. In addition, the server selection controller app adds flow entries into the app table as shown in the figure.

When a client request arrives at SW1, the app table rule fires and SW1 performs the TCP handshake with the client to advance to the HTTP GET request. During this phase (steps 1-3), SW1 uses SYN cookies to preserve the connection state,



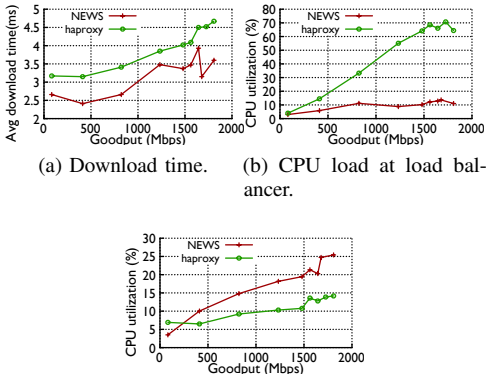
(a) *App-rule installation and TCP handshake with client* (b) *Flow-rule insertion at SW1* (c) *TCP handshake with S1 and flow-rule insertion at SW2*

Fig. 6. **Content-aware server selection.**

and stops execution of the server selection app by inserting a "**break=true**" after packet-out-ing the SYN-ACK packet. Only after the HTTP GET request packet arrives, the execution "continues" to server selection which extracts the requested URL and uses it to select a server from the pool. If the mapping is not available locally at the switch, it sends the packet to the controller, which resolves and returns the mapping back to SW1 (say S1), as shown in steps 5 and 6 of Figure 6b. Server selection app writes a forwarding rule for the rest of the connection into the kernel flow table of SW1 to rewrite the destination VIP to S1 and forwards the packet towards S1. When the switch in front of S1 (i.e., SW2) receives the packet, it matches the app table rule. It then invokes the TCP handshake app that plays back the handshake with S1 based on the headers of the packet. During this phase, right after receiving the SYN-ACK from S1, it can compute the deltas used for TCP splicing, and therefore it installs the appropriate rewrite rules in the kernel flow table of SW2 (see Figure 6c). When S1 replies, SW2 performs TCP splicing to adjust the sequence and acknowledgement gaps for the connection to go through transparently between the client and S1. SW2 also rewrites the source address back to the VIP.

We compare NEWS server selection with HAProxy [10]. We set up two client hosts, two server hosts and one load balancer. Each client or server runs two containers. First, we run HAProxy at the load balancer node, and use **httperf** [16] to generate load. Both directions of the flow go through HAProxy since it needs to splice the connection. In the second experiment, we run NEWS front-end OVS at the load balancer node, and run back-end OVS at each server host. In this case, the reverse flow from the server to the client do not have to go through the front-end OVS as we explained before, which allows reverse traffic to be sent separately.

Figure 7a shows the average download time vs. goodput under different rates. We observe that NEWS introduces much less delay on the data path than HAProxy since in HAProxy every packet goes to the userspace, but in NEWS only the first few packets are sent to userspace. Figure 7b shows the average CPU utilization at the load balancer node. As load increases, the increase in CPU utilization is much more significant with HAProxy than with NEWS, indicating NEWS has much better scalability than HAProxy. Figure 7c shows the CPU utilization at the server host, which shows that NEWS incurs more server



(a) Download time. (b) CPU load at load balancer.

(c) CPU load at server host.

Fig. 7. NEWS L7 Load Balancer App vs. HAProxy.

TABLE I
IMPACT OF LOADABLE MODULES ON DATA PATH

Rate (pkt/s)	Avg RTT(ms)	
	Loaded	Compiled
1k	0.7	0.7
10k	0.3	0.4
100k	0.5	0.6

CPU overhead than HAProxy, although the extra overhead is much less compared to the gain at the load balancer host. The overhead is caused by the TCP handshake app that runs in the userspace. This experiment demonstrates that NEWS can scale the capacity of the load balancer by distributing the processing load onto each server host. As a result, the distributed framework eliminates "choking points".

C. Loadable App Modules

We evaluate the overhead of loadable app modules to examine latency in loading an app, and whether loadable apps affects the data path performance. Loading time is measured using the system clock. Our benchmarks show that the average loading time is about 1ms, and it varies across apps, depending on their internal state initialization. To examine the impact on data path latency, we used **hping** to measure the RTT between sending a SYN packet and receiving the ACK for both cases of loadable modules and compiled modules. As shown in Table I, the difference in RTT between the two is negligible.

D. Discussion

Deep Packet Inspection. Apps in NEWS process the first few packets of a flow to determine the action. Once the action is determined, the app inserts a micro-flow into the kernel data path to speed up packet switching. However, there is a class of NFs that examines every packet, e.g., intrusion detection systems. Therefore, in NEWS, each packet of a flow will be propagated up to the userspace where the app examines it, which slows down switching and reduces throughput. We intend to use the OVS recent release for a DPDK [17] data path that receives packets directly from the NIC to the userspace, and thus NEWS apps executes on them without performance penalty (i.e., NEWS does not depend on a specific data path).
Distributed State Management. App state is distributed across the controller and multiple data plane instances. The controller app initializes the instances and regularly communicates with them to push/pull current state, e.g., a **`connlimit`** policy is populated by the controller app into data plane instances. For correct operation, state across instances must be synchronized. This issue is not unique to NEWS and it was studied in the cloud context [18]. While we do not prescribe any specific solution here, we believe that existing state management solutions [3] can be adapted in NEWS.

VI. Related Work

Network services are implemented by steering flows through NF chain. SDN enables dynamic service chaining, although tracking flows traversing through middleboxes is a challenge since the packets may be altered along the way. Novel approaches uses statistical inference [19] or tags in the packet [2]. Inference may cause errors and finding fields in the packet for tags may not be possible. NEWS avoids flow tracking by careful placement of NFs that ensures that NFs are exposed to the information they require. Also, most service chaining is done locally by chaining apps, which enables flexible processing logic. Middlebox virtualization has also been studied extensively [20], [21]. ETTM [22] uses special end-host modules for middleboxs. OpenNF [3] proposes an architecture based on virtual NFs and SDN focusing on using SDN to assist NF state management [3]. This is complementary to our work since similar state migration techniques can be applied in NEWS. ClickOS [23] uses minimal OS image for middleboxes, where service chaining is done by forwarding traffic between these middleboxes. In NEWS, NFs are collection of modules, at finer granularity than individual NFs, e.g., one can compose **`connlimit`** (part of firewall) and DNAT (part of NAT), which makes service composition flexible and efficient. In general, we propose to integrate services into SDN data plane for unified control. Prior work studied SDN traffic engineering [24], [25], L3 [26], [27] and L4 load balancing [11]. NEWS builds on that to address application-awareness in data plane, and addresses L7 load balancing problems and other NFs, which require application-awareness. AVANT-GUARD [28] has been proposed to protect SDN from SYN floods using TCP splicing in SDN switches. NEWS is a general framework for different types of NFs, and can address security NFs as well.

VII. Conclusion

NEWS enables native NFV within SDN by extending the data plane and allowing NFs to be dynamically enabled on the data path at the most appropriate/efficient locations according to policies and network resources. Our results show that NEWS offer competitive performance compared to other approaches. More importantly, NEWS is designed to be independent of the NFs data path implementations and NFs are added to the system without recompilation or restarting. We plan to further optimize performance and explore other services.

Acknowledgment

This research was supported in part by NSF grants CNS-1411636, CNS-1618339 and CNS-1617729, DTRA grant HDTRA1-14-1-0040 and DoD ARO MURI Award W911NF-12-1-0385.

References

[1] R. Jain *et al.*, "Network Virtualization and Software Defined Networking for Cloud Computing: A Survey," *IEEE Communications*, 2013.
[2] S. K. Fayazbakhsh *et al.*, "Enforcing Network-Wide Policies in Presence of Dynamic Middlebox Actions using FlowTags," in *NSDI*, 2014.
[3] A. Gember-Jacobson *et al.*, "OpenNF: Enabling Innovation in Network Function Control," in *SIGCOMM*, 2014.
[4] T. Koponen *et al.*, "Network Virtualization in Multi-tenant Datacenters," in *NSDI*, 2014.
[5] A. Greenberg, "SDN for the cloud," http://conferences.sigcomm.org/sigcomm/2015/pdf/papers/keynote.pdf, 2015.
[6] "Alcatel-Lucent joins virtual router race," lightreading.com, 2014.
[7] M. Honda *et al.*, "mSwitch: A Highly-Scalable, Modular Software Switch," in *SOSR*, 2015.
[8] G. Blake *et al.*, "A Survey of Multicore Processors," *IEEE Signal Processing*, 2009.
[9] H. Sutter, "Design for Manycore Systems," http://www.drdobbs.com/parallel/design-for-manycore-systems/219200099, 2009.
[10] HAProxy, "High Performance Load Balancer," haproxy.org/, 2015.
[11] P. Patel *et al.*, "Ananta: Cloud Scale Load Balancing," in *SIGCOMM*, 2013.
[12] B. Pfaff *et al.*, "The Design and Implementation of Open vSwitch," in *NSDI*, 2015.
[13] "Stateful Connection Tracking and NAT," openvswitch.org/support/ovscon2014/, 2014.
[14] R. Sherwood *et al.*, "FlowVisor: A Network Virtualization Layer," *OpenFlow Switch Consortium*, 2009.
[15] ONF, "OpenFlow Switch Specification," www.opennetworking.org/, 2012.
[16] D. Mosberger *et al.*, "httperf: A Tool for Measuring Web Server Performan," in *Internet Server Performance Workshop*, 1998.
[17] DPDK, "Data Plane Development Kit," http://www.dpdk.org/, 2015.
[18] B. Raghavan *et al.*, "Cloud Control with Distributed Rate Limiting," in *SIGCOMM*, 2007.
[19] Z. A. Qazi *et al.*, "SIMPLE-fying Middlebox Policy Enforcement Using SDN," in *SIGCOMM*, 2013.
[20] A. Gember *et al.*, "Toward Software-defined Middlebox Networking," in *HotNets*, 2012.
[21] V. Sekar *et al.*, "Design and Implementation of a Consolidated Middlebox Architecture," in *NSDI*, 2012.
[22] C. Dixon *et al.*, "ETTM: A Scalable Fault Tolerant Network Manager," in *NSDI*, 2011.
[23] J. Martins *et al.*, "ClickOS and the Art of Network Function Virtualization," in *NSDI*, 2014.
[24] S. Jain *et al.*, "B4: Experience with a Globally-deployed Software Defined Wan," in *SIGCOMM*, 2013.
[25] C.-Y. Hong *et al.*, "Achieving High Utilization with Software-driven WAN," in *SIGCOMM*, 2013.
[26] R. Wang *et al.*, "OpenFlow-based Server Load Balancing Gone Wild," in *HotICE*, 2011.
[27] N. Handigol *et al.*, "Aster* x: Load-Balancing Web Traffic over Wide-Area Networks," 2009.
[28] S. Shin *et al.*, "AVANT-GUARD: Scalable and Vigilant Switch Flow Management in Software-defined Networks," in *CCS*, 2013.