nf.io: A File System Abstraction for NFV Orchestration

Md. Faizul Bari, Shihabur Rahman Chowdhury, Reaz Ahmed, and Raouf Boutaba
David R. Cheriton School of Computer Science, University of Waterloo
[mfbari | sr2chowdhury | r5ahmed | rboutaba]@uwaterloo.ca

***Abstract*—In recent years, Network Function Virtualization (NFV) has gained a lot of traction from both industry and academia. NFV promotes vendor-independence and rapid evolution through open source software, open standards, and open APIs. However, adopting these principles for virtual middleboxes or Virtual Network Functions (VNFs) is not enough. The VNF orchestration systems also need to adopt the same principles, otherwise a network operator may still face vendor lock-in. Moreover, standardization efforts take a long time to converge and are often futile. For this reason, we introduce `nf.io` that uses the existing well-known Linux file system interface for VNF orchestration. Different members of a DevOps team can readily utilize this tool without a cumbersome learning process. We have developed a prototype, and provided a set of example use-cases to demonstrate its effectiveness.**

I. Introduction

Middleboxes have become an integral part of modern enterprise and data center networks [1]–[3]. They are used for realizing various performance and security objectives [4]. Most middlboxes (*e.g.*, firewalls, Intrusion Detection Systems (IDSs), Network Address Translators (NATs), *etc*.) are dedicated hardware appliances. However, recent advances in cloud and virtualization technologies have fueled the concept of Virtual Middleboxes or Virtual Network Functions (VNFs) along with a new research field known as Network Function Virtualization (NFV) [5]. This area of research has gained a lot of traction from both industry and academia. The core concept of NFV is to move packet processing tasks from vertically integrated hardware middleboxes to software processes running on commodity (*e.g.*, `x86` systems) servers. This shift has been motivated by vendor lock-in, inflexible service chaining, and prolonged time-to-market for new services caused by the closed and proprietary nature of hardware middleboxes. NFV on the other hand, promises to solve these problems by promoting open source, open API, and standardized software solutions that can run on commodity servers [6].

Although much progress has been made in NFV technology [7]–[9], a crucial component for realizing the primary objective of NFV is still missing – a management and orchestration platform that offers open and standard APIs for configuring, chaining, and monitoring VNF instances. Without this feature, network operators may end-up with the same situation of vendor lock-in as with proprietary hardware middleboxes. A possible choice for VNF management is OpenStack [10]. However, it is more focused towards managing compute resources. In case of VNFs, we need fine grained control over both compute and network resources along with the APIs for orchestrating and monitoring service chains as well as individual VNFs. In recent years, there have been a number of proposals like Stratos [11], OpenNF [12], Split/Merge [13] for VNF management to fulfill these requirements. However, they propose incompatible northbound APIs. What is really needed is a standardized API, flexible enough to express a wide range of NFV management and orchestration operations [14]. Standardization efforts usually take a longtime and are often futile. Hence, we take a different approach, and propose to use an existing, well known, standardized interface for NFV management and orchestration: the Linux file system.

We call our proposed system `nf.io`. It utilizes the Linux file system as the northbound API for VNF orchestration. It adopts various operating system principles: (i) everything (resource, configuration) is represented as a file, (ii) well known utility programs (*e.g.*, `mkdir`, `cp`, `mv`, `ln`, *etc*.) are used for state manipulation, (iii) heterogeneous resource pools (*e.g.*, different networking tool-chains like Linux bridge [15] or Open vSwitch [16]) are controlled through a high-level abstraction, and (iv) resource specific drivers are developed similar to device drivers in an OS. Existing NFV management systems like Stratos or OpenNF can use the `nf.io` abstraction by developing their own resource drivers. Moreover, `nf.io` can be immediately utilized by different DevOps members without having to learn new languages and libraries.

Our key contributions in this paper are as follows: we (i) propose to use the Linux file system interface as a northbound API for NFV management and orchestration, (ii) define the file and directory structure semantics for performing different operations like VNF deployment, chain composition, configuration, and monitoring, and (iii) develop a proof-of-concept prototype that can deploy fully functional service chains composed of five different types of VNFs: Firewall, Load Balancer, Web Proxy, IDS, and NAT. However, `nf.io` is not restricted to these VNF types in anyway. Our prototype can be easily extended to support other VNF types.

The rest of the paper is organized as follows. Rationale behind choosing a file system based northbound API in Section II. The file system abstraction and its semantics are presented in Section III. The proposed system architecture is explained in Section IV. Section V presents an evaluation of the prototype through different use-cases. A brief discussion of related work is provided in Section VI. Finally, we conclude in Section VII with some future research directions.

II. Design Rationale

Our design goal is to provide network administrators with a central point of management, hiding the underlying distributed nature of the deployment. Instead of reinventing the wheel to achieve this goal, we tried to find an existing wheel that can fit the task. We found the Linux file system interface to be a perfect fit. The rationale behind choosing the Linux file system interface as the API is as follows:

- **Centralized view:** The Linux file system interface gives the user a centralized view of the underlying file system through a well defined semantics of system calls. It hides the possible distributed nature and heterogeneity of file system technologies from the user. This will allow a user to perform operations on the file system without worrying about under-the-hood details. An NFV orchestration system consists in configuring and managing a wide range of heterogeneous resources. Exposing a file system like interface to NFV orchestration will allow the users to perform different management tasks without worrying about the exact implementation details of how things are deployed. For example, a user can easily deploy a service chain without knowing the complex details of routing and connectivity in the underlying network fabric.
- **Centralized vs. distributed management:** Using the Linux file system as a medium for management provides the opportunity to separate the configuration storage from the management agents. These agents simply take the role of separate processes accessing and modifying a file system. The file system itself can be deployed in a distributed manner as well. A network operator may choose to deploy one or multiple management agents based on requirements like fault-tolerance or scalability.
- **Rich Set of File system features:** There are innumerable file system architectures with various features [17]–[19]. A file system can be centralized or distributed, provide features like fault-tolerance, scalability, low-latency access, *etc*. There is no need to re-implement these features.
- **Rapid application development:** A well defined management API is necessary for developing higher level management applications that can perform a wide range of management tasks. Having a familiar API like the file system will make the task of application development easier, and would allow developers to leverage the file system utilities and libraries from the current ecosystem.
- **The hierarchical NFV elements:** The NFV elements have a natural hierarchy that can be represented as a tree-like directory structure, *e.g.*, a service chain can be represented as a collection of network function instances, a network function instance can in-turn be represented as a collection of configurations, logs, and some VM or container running the corresponding software.
- **Leverage rich set of existing applications:** There are a number of popular configuration management tools (*e.g.*, Chef, Puppet, *etc*.) that support a wide range of file system management tasks. Exporting a file system interface to NFV management and orchestration will allow us to leverage these tools, and take advantage of the already developed routines. In addition, a properly redefined semantics of file system interface for NFV orchestration will allow us to leverage the file system manipulation commands available on Linux systems to perform different management tasks.

III. File System Abstraction

`nf.io` uses a simple and intuitive directory hierarchy to store states regarding VNF deployment, configuration, and chaining. A high-level view of the `nf.io` directory hierarchy is shown in Figure 1.

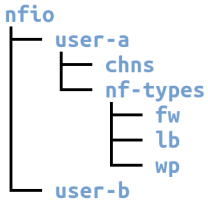


Fig. 1. A High-level View of `nf.io` Directory Structure

The `nfio` directory is the root of the `nf.io` file system. Under this directory there is one subdirectory for each user (*e.g.*, `user-a` and `user-b` directories in Figure 1), which marks the mount-point of that user's home directory. Here, 'user' is any party that uses `nf.io` to deploy VNF(s). A user is restricted within his home directory in `nf.io`. In other words, a user cannot browse or access the `nf.io` directory structure outside his home directory. This can be ensured by using utility programs like Limited Shell (lshell) [20]. A user's home directory contains the state information for all VNF instances and VNF chains deployed by that user. These states are organized into directories and files mounted inside the corresponding user's home directory. Each user's home directory contains two subdirectories: (i) `nf-types` – contains information about different VNF types along with deployed instances and (ii) `chns` – contains information regarding the VNF chains deployed by the user. The `nf-types` directory contains one directory for each VNF type (*e.g.*, `fw`, `lb`, and `wp` in Figure 1). These directories contain one subdirectory for each VNF instance deployed by the user.

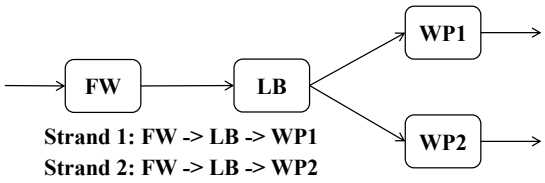


Fig. 2. A Multi-Strand Chain

Let us assume that `user-a` has deployed the VNF chain as shown in Figure 2. The chain has four VNF instances: one firewall, one load-balancer, and two web-proxies. Traffic first goes through the firewall `fw`, then the load balancer `lb`, and finally through one of the web-proxies (`wp1` or `wp2`). Figure 3 shows the directory hierarchy for the `nf-types` directory.

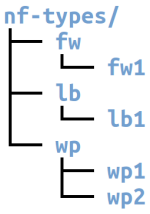
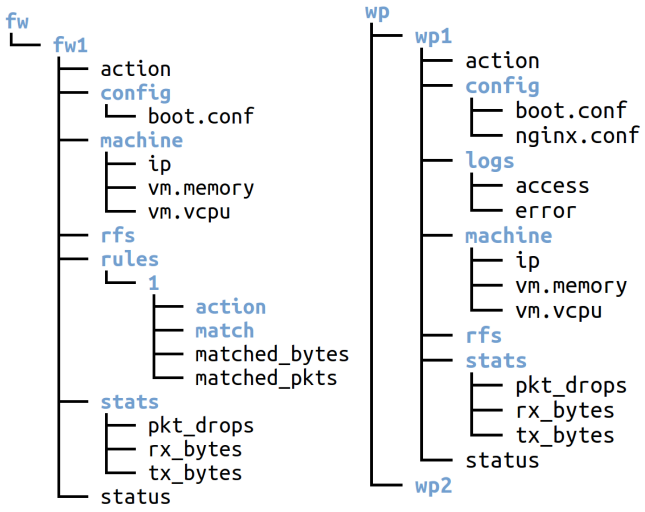


Fig. 3. The `nf-types` Directory



(a) Firewall (b) Web-proxy

Fig. 4. Two Sample VNF Instances

There is one sub-directory for each VNF type: `fw`, `lb`, and `wp`. The directories representing the deployed VNF instances (*e.g.*, `fw1`, `lb1`, `wp1`, and `wp2`) reside under the corresponding VNF type directories. Next we will explain the directory and file organization for individual VNF instances.

Figure 4 shows the directory and file organization for a firewall and web-proxy. Due to space limitations we have refrained from showing the directory and file organization of a load-balancer. As we can see from Figure 4(a), the directory `fw1` contains a number of sub-directories: (i) `config` – contains a file named `boot.config` that specifies the parameters required during VNF boot up, (ii) `machine` – contains files representing host and VM information, (iii) `rfs` – this is a special directory that marks the mount point for the file system of the VNF instance itself, (iv) `rules` – represents the match $\rightarrow$ action rules for the firewall, and (v) `stats` – contains files that represent different counters and statistics. The `stats` directory contains files like `pkt_drops`, `rx_bytes` and `tx_bytes`. As the name suggests the user can read the number of dropped packets, received and transmitted bytes from these files, respectively. These files are actually placeholders. When a user issues a read command the data is collected from the VNF instance using various drivers, and returned to the user. For example, `nf.io` fetches the packet drop counter of the VM running a VNF instance when the corresponding `pkt_drops` file is read. Aggregate statistics such as total number of packet drops along a chain can also be represented using such file based abstraction. In the later case, `nf.io` will collect and aggregate the required information from different VNF instances and will return the data as text records to the caller. Such file system abstraction enables users to use utilities such as `grep`, `awk`, `sed`, *etc.* along with I/O redirection (pipes) to collect and process information from VNF instances.

The user provides the firewall rules using the `rules` directory. Each rule is represented by a separate subdirectory under the `rules` directory. Each rule directory contains two subdirectories: `match` and `action`. The `match` directory contains one file for each packet header field that should be checked. So, if the user wants to match the source IP address to `10.10.0.5`, then this value should be written in a file named `src_ip` under the `match` directory. The `action` directory must contain one file. It can be a file named either `allow` or `drop`. If it contains a file named `allow` then the matched packets are allowed to go up the network stack, otherwise they are dropped. The rule directory also contains two files: `matched_bytes` and `matched_pkts`. These files can be read to collect the number of bytes and packets that matched this rule, respectively. This is a very simple abstraction that we are currently using for our proof-of-concept prototype. We are working towards a better abstraction that can easily represent different types of complicated firewall rules.

The directory and file organization of a web-proxy is quite similar to that of a firewall (Figure 4(b)). However, web-proxy does not contain a `rules` directory. Instead, the configuration file for the web-proxy can be found under the `config` directory – the `nginx.conf` file in Figure 4(b). A user can modify this file to change the configuration of the proxy. The web-proxy also has a directory called `logs`. This directory contains two files: `access` and `error`. The `access` file provides information about the successful web requests served by the web-proxy. The `error` file contains information regarding errors encountered by the web-proxy.

A VNF instance directory (*e.g.*, `fw1`, `wp1`) also contains two files: (i) `status` – indicates the current status of the VNF (*e.g.*, `initialized`, `running`, `paused`, *etc.*) and (ii) `action` – a placeholder file that triggers VNF manipulation operations. For example, to deploy and activate a VNF instance the user needs to write "`activate`" in this file. The other supported operations include: `pause` and `destroy`.

The structure of the `chns` directory is shown in Figure 5. It contains a directory for each VNF chain (*e.g.*, `chain-alpha`, `chain-beta`, *etc.*). Keeping a separate directory hierarchy for the VNF chains enables us to share a VNF instance between multiple chains. Each chain directory contains two files: `status` and `action`. The `status` file indicates the current status of the VNF chain, and the `action` file can be used to trigger operations like `activate`, `pause`, and `destroy` on the entire chain. A VNF chain can contain multiple traffic flow paths that we call *strands*. A chain directory contains one subdirectory for each strand. In Figure 5 we have two strands: `strand-1` and `strand-2`. The strands directories contain additional directories, files, and symbolic links to specify the VNF instances that comprise the chain. This directory semantics enable users to specify their chains in a simple manner as it resembles the way traffic is supposed

```
chns/
├── chain-alpha
│   ├── action
│   ├── status
│   ├── strand-1
│   │   ├── fw1
│   │   │   ├── fw1 -> /nfio/user-a/nf-types/fw/fw1
│   │   │   └── next -> /nfio/user-a/nf-types/lb/lb1
│   │   ├── lb1
│   │   │   ├── lb1 -> /nfio/user-a/nf-types/lb/lb1
│   │   │   └── next -> /nfio/user-a/nf-types/wp/wp1
│   │   ├── start -> /nfio/user-a/nf-types/fw/fw1
│   │   └── wp1
│   │       └── wp1 -> /nfio/user-a/nf-types/wp/wp1
│   └── strand-2
│       ├── fw1
│       │   ├── fw1 -> /nfio/user-a/nf-types/fw/fw1
│       │   └── next -> /nfio/user-a/nf-types/lb/lb1
│       ├── lb1
│       │   ├── lb1 -> /nfio/user-a/nf-types/lb/lb1
│       │   └── next -> /nfio/user-a/nf-types/wp/wp2
│       ├── start -> /nfio/user-a/nf-types/fw/fw1
│       └── wp2
│           └── wp2 -> /nfio/user-a/nf-types/wp/wp2
└── chain-beta
```

Fig. 5. Directory Hierarchy of a VNF Chain

to flow through a chain.

The strand directory contains one directory for each VNF instance in the strand. In Figure 5, `strand-1` contains three instances: `fw1`, `lb1`, and `wp1`. Each of these directories contain two symbolic-links: (i) the first link has the same name as the directory (*e.g.*, `fw1`) itself and points to the current VNF instance, (ii) the second link called `next` points to the next VNF instance in the chain, *e.g.*, under the `fw1` directory the `next` symbolic-link points to the directory `nfio/user-a/nf-types/lb/lb1` that represents the load-balancer placed right after the firewall in Figure 2. The strand directory also contains a file called `start` that is a symbolic-link to the first VNF in the chain. In Figure 5, `start` points to the directory `fw1` representing the firewall. It is worth mentioning that, these directories only contain symbolic-links to other directories. We have modified the `symlink` file-system call to make it easy for the user to create these directories.

Figure 5 shows the directories for both strands of the VNF chain. The firewall and load-balancer appear in both strands. The rationale behind this design choice is to make it easy for the user to understand the directory hierarchy. It closely resembles the way traffic is supposed to flow through the chain. We have implemented a command called `mkchn` to create the directory and file hierarchy for a chain. The chain in discussion can be created with the following commands:

```
$ mkchn --chain-name=chain-alpha
 --strand-name=strand-1 fw1 lb1 wp1
$ mkchn --chain-name=chain-alpha
 --strand-name=strand-2 fw1 lb1 wp2
```

The `mkchn` command searches the user's `nf-types` directory for matching VNF instances and then populates the `chain-alpha` directory. The user can add multiple strands to a chain. He can also use the same VNF instance for multiple chains. We also implemented `rmchn` and `rmins` commands to remove an entire chain and a single VNF instance from a chain, respectively. `rmins` removes a VNF instance from a chain by properly updating the symbolic-links. A user can also modify a chain manually by using Linux commands like `rm`, `cp`, `ln`, *etc*.

Typically, a user deploys a chain by writing the string "`activate`" in the `action` file under the chain's directory (*e.g.*, under `chain-alpha`). `nf.io` then traverses through the symbolic-link structure under each strand and deploys the required VNF instances and the network connections between the instances. Similarly, the chain can be paused and destroyed by writing "`pause`" and "`destroy`" in the `action` file.

IV. System Architecture

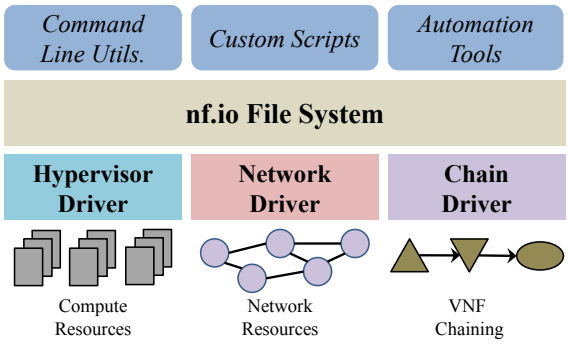


Fig. 6. `nf.io` Architecture

A high-level view of the `nf.io` architecture is shown in Figure 6. The `nf.io` File System is a virtual file system that runs on top of the traditional OS file system. VNF operations are triggered when a user writes a operation string in the `action` files. `nf.io` performs these operations by using three resource drivers: (i) Hypervisor Driver, (ii) Network Driver, and (iii) Chain Driver. The hypervisor and network drivers manage the compute and network resources, respectively. The chain driver manages VNF chains by configuring traffic forwarding rules between VNFs.

A. Hypervisor Driver

In `nf.io`, network functions can be deployed in a number of ways. They can run as processes on a physical machine, VMs on a hypervisor like Xen or KVM, or as lightweigh containers provided by Docker [21] or Linux Container (LXC) [22]. The hypervisor driver abstracts the underlying diversity in these virtualization technologies and provides a uniform interface to `nf.io`.

B. Network Driver

`nf.io` requires support for certain networking functionality from the underlying physical infrastructure. Figure 7 shows a typical network configuration required for `nf.io`. In each physical machine, `nf.io` must have the ability to: (i) setup bridges, (ii) create IP links between virtual ethernet (`veth`) pairs, (iii) setup tunnels (*e.g.*, VXLAN or GRE), and (iv)

install forwarding rules. The first three are usually supported by most Linux networking stacks. Forwarding rules can be configured in two different ways: using `iptables` in Linux or using OpenFlow [23] rules in Open vSwitch (OVS) [16]. `nf.io` can work with both setups, as long as the OVS version supports some tunneling technology.

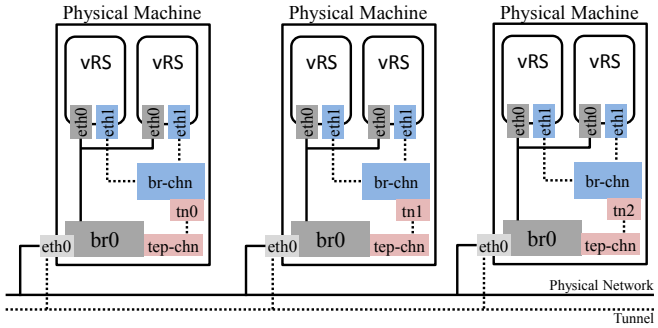


Fig. 7. Network Setup for VNF Chaining

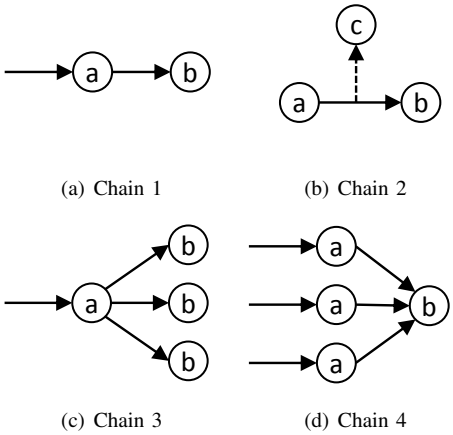
Figure 7 shows a network configuration where each physical machine has one NIC. The configuration with multiple NICs is much simpler and thereby not explicitly included in the paper. In each physical machine, `nf.io` needs to setup two bridges `br0` and `br-chn` that are used for basic network connectivity and VNF chaining, respectively. `tn0` and `tep-chn` are `veth` pairs used to connect the two bridges. Then GRE or VXLAN tunnels are setup between the `tn*` interfaces to connects all the `br-chn` bridges on different physical machines – creating a separate network for the `eth1` interface of the *vRS*s. *vRS* indicates isolated virtual compute resources (*e.g.*, VMs, containers, *etc*.). If we have a second physical NIC than this `veth` pair is not required, and the `br-chn` can directly connect to the second NIC. Each vRS is connected to the network with two interfaces. `eth0` is used as the front-facing interface (*e.g.*, one with a public IP in a NAT/Firewall). It is also used to provide SSH access to the vRS. `eth1` is used for setting up routing paths between VNFs for chaining. The mechanism used for chaining VNFs with these interfaces will be explained in detail in Section IV-C.

Similar to the hypervisor driver, the network driver hides the underlying heterogeneity and complexity. `nf.io` only needs to work with simple abstractions like network interfaces (`eth0` and `eth1` in vRS) instead of bridges and tunnels.

C. Chain Driver

The chain driver interconnects different types of VNFs. It provides a single function `chn-cnct(vnf1, vnf2)` to the `nf.io`, where `vnf1` and `vnf2` are two arbitrary VNFs. For a chain like a $\rightarrow$ b $\rightarrow$ c, this function must be called twice: first for a $\rightarrow$ b, and again for b $\rightarrow$ c. The task of Interconnecting two VNFs depends primarily on their types, and whether their network interfaces are on the same network (IP subnet) or not. Figure 8 shows four possible chaining scenarios:

In Figure 8(a), VNF a forwards traffic to VNF b. This is an example of how a NAT or Firewall is connected. VNF a receives traffic on one interface, and after some processing (*e.g.*,



(a) Chain 1 (b) Chain 2

(c) Chain 3 (d) Chain 4

Fig. 8. Four Chaining Scenarios

port mapping, applying security rules) forwards the packet to its second interface. The second scenario represents VNFs that do not actively forward traffic, like an IDS. In Figure 8(b) VNF c just listens to the traffic forwarded by VNF a to b. In the third scenario, a VNF forwards traffic to multiple VNFs. Figure 8(c) shows one such example, where VNF a is forwarding traffic to three VNFs. Here, VNF a can be a load-balancer, and b can be web-proxies. Finally, multiple VNFs can forward traffic to a single VNF, as shown in Figure 8(d). For example, multiple firewalls can forward traffic to a single web-proxy. The current implementation of the `nf.io`'s Chain Driver can configure all possible connection scenarios between five different VNFs. We do not include all of them in the paper due to space limitations. It is worth mentioning that `nf.io` is not restricted to these VNF types in anyway.

V. Prototype Evaluation

We evaluate the effectiveness of `nf.io` through various use cases (Section V-B). Before going into the details of these use cases, we first briefly describe our prototype implementation (Section V-A).

A. Implementation

The `nf.io` prototype is implemented using the python API binding for FUSE [24]. We rewrote a number of Linux file system calls like `mkdir`, `read`, `write`, `symlink`, *etc*. to implement the `nf.io` file system semantics. The Hypervisor Driver currently supports KVM, Xen and Docker. We use libvirt [25] and Docker Remote API to control VMs in KVM/Xen and containers in Docker, respectively. The Network and Chain Drivers currently support two configurations: (i) Linux `iptables` with Linux bridge and (ii) Open vSwitch (OVS). In the first configuration, the `route` command is used to forward traffic from one VNF instance to the next. `iptables` is used for setting up NAT rules and also for blocking unwanted traffic. Linux bridges are used to setup a separate network for VNF chaining. In the second configuration, OVS itself provides all required functionalities: (i) traffic is forwarded or blocked using OpenFlow rules, (ii) NATing is also performed by setting up OpenFlow rules to

change the source or destination IP address and port numbers, and (iii) OVS bridge is used to deploy a separate network for chaining. In both cases we use GRE tunnels to connect vRSs deployed on different physical machines. Finally, we remotely mount a vRS's file system under the directory `rfs` (Section III) using `sshfs` [26]. A demonstration of `nf.io` is available at http://faizulbari.github.io/nf.io/.

B. Use Cases

We demonstrate the capabilities of `nf.io` by showcasing use cases focused on three primary areas: (i) deployment, (ii) configuration and (iii) monitoring of VNF instances and chains. In the following examples we assume that `nf.io` is mounted at `/nfio` and all the paths are relative to this mount point.

1) Deployment: In this section we first show how to deploy a single VNF instance, followed by the deployment of a chain.

The following `mkdir` command creates the directory structure (similar to Figure 4) for a Bro IDS instance:

```
$ mkdir nf-types/bro/ids-a
```

Automation tools like Chef [27] can use `nf.io` for NFV management. For example, the following Chef recipes first creates the directory structure for a VNF instance and then configures it to be deployed on a physical machine with IP address `10.0.0.11`:

```
directory "nf-types/bro/ids-a" do
  owner 'user-a'
  group 'user-a'
  mode '0755'
  action :create
end
file "nf-types/bro/ids-a/machine/ip" do
  owner 'user-a'
  group 'user-a'
  mode '0755'
  content "10.0.0.11"
end
```

We can create a VNF chain after creating the VNF instances. First, we need to create a directory (`chain-a`) under `chns`. Then we add symbolic-links to the VNF instances that are part of `chain-a`. We also need to add the `start` and `next` symbolic-links as explained in Section III. The following example creates the chain firewall → webproxy → load balancer:

```
$ mkdir chns/chain-a
$ mkdir chns/chain-a/strand-1
$ cd chns/chain-a/strand-1
$ ln -s nf-types/fw-ufw/fw-a
$ ln -s nf-types/proxy-nginx/proxy-a
$ ln -s nf-types/balancer/lb-a
$ ln -s nf-types/fw-ufw/fw-a start
$ ln -s nf-types/proxy-nginx/proxy-a
fw-a/next
$ ln -s nf-types/balancer/lb-a
proxy-a/next
$ echo 'activate' > chns/chain-a/action
```

The `echo 'activate' > chns/chain-a/action` command triggers the deployment of the VNF instances included in `chain-a`. `nf.io` traverses the symbolic-link structure to get the necessary configurations, and performs the following operations: copy VM or container images to target machines, boot the VNF instances, and configure the network connectivity to create the chain.

2) Configuration: `nf.io` provides the facility to modify different configuration parameters. For example, a VNF instance can be migrated to a different physical machine as follows:

```
$ echo 'pause' > chns/chain-a/action
$ echo '10.0.0.15' > chns/chain-alpha/
fw-alpha/fw-a/machine/ip
$ echo 'activate' > chns/chain-a/action
```

3) Monitoring: `nf.io` exposes a file system interface to collect information ranging from that of individual VNFs to end-to-end service chains. This enables the users to use utility programs like `grep`, `sed`, `awk`, *etc*. to collect and process monitoring data. It also facilitates rapid development of monitoring applications that can easily monitor a distributed NFV deployment through simple file system operations in `nf.io`. The following examples demonstrate its monitoring capabilities:

Total number of packet drops along a chain:

```
$ find -L chns/chain-a pkt_drops |
xargs cat | awk '{total += $1}
END {print total}'
```

List of all VNF instances deployed on a physical machine with IP address `10.0.0.17`:

```
$ grep -R '10.0.0.17' nf-types/
```

List of all chains a VNF instance is part of:

```
$ find chns -lname nf-types/fw-ufw/fw-a
```

Remotely mounting a VNF's file system (under `rfs`) enables us to directly collect and process their logs from `nf.io`. For example, we can monitor the number of active connections to a web server from the logs of a Bro IDS as follows:

```
$ tail -n +8 chns/chain-a/ids-a/ids-a/rfs/
/logs/current/conn.log | awk '{print $8}' |
 grep -c 'http'}
```

VI. Related Work

Current state-of-the-art NFV management solutions include projects like Stratos [11] and OpenNF [12]. Stratos proposes an architecture for orchestrating VNFs outsourced to a remote cloud by taking care of traffic engineering, horizontal scaling *etc*. On the other hand, OpenNF proposes a converged control plane for VNFs and network forwarding plane by extending the centralized SDN paradigm. Both of these projects propose

different northbound APIs that are specific to their individual feature set. They are not inter-operable in any way. nf.io proposes an open, standardized and well-known API for VNF management and orchestration that has passed the test of time and proven to be very expressible. Adopting the nf.io abstraction is quite easy as everything is represented as files.

The "everything as a file" design principal of Linux virtual file system has inspired many other systems with file system like interfaces. Good examples of such systems are sysfs and procfs. sysfs exports various system configuration parameters to the userspace as a hierarchical structure of virtual directories and files. Similarly, procfs exports various process related parameters to the userspace through such virtual directory structure. This kind of interfaces allow users to easily read and tune system parameters using regular file system operations. Recently, an SDN controller with a file system like interface was proposed in [28]. It represents the network topology and forwarding rules in switches with a hierarchy of virtual directories and files.

Cloud management systems like OpenStack [10], CloudStack [29], and SaltStack [30] can be candidates for VNF management. However, they are more focused towards managing compute resources. They do not provide adequate networking support for VNF chaining [31]. nf.io provides a very simple and well known interface to work with. The user do not need to know about the underlying complex technologies like routing, bridges, tunnels, or tunnel-end-points. He can orchestrate a VNF service chain by populating the proper directory hierarchy using common Linux utility programs.

VII. Conclusion and Future Work

In this paper, we have presented the design of nf.io – a file system abstraction for managing and orchestrating VNFs. nf.io exposes the Linux file system interface as the northbound API, enabling users to manage and orchestrate VNF service chains by performing simple file and directory manipulation operations. We have shown the effectiveness of nf.io through several use-cases that demonstrate its flexibility, expressiveness, and user-friendliness.

nf.io is currently under active development. Our next steps include: (i) improve the file system semantics, (ii) provide better isolation by deploying a separate network namespace for each user, (iii) integrate a resource scheduler for improved automation and utilization, (iv) add support for Cosmos [32] that simplifies deployment of ClickOS [7] based VMs on Xen, and (v) integrate nf.io with cloud management frameworks like OpenStack. nf.io marks our first step towards designing a cutting-edge middleware for NFV management and orchestration. During the initial development of nf.io we have found the file system abstraction to be a great fit as a northbound API for VNF management and orchestration. We believe that, this design choice will avoid the hassle of going through tedious standardization processes in the long run, and will promote vendor-independence and rapid evolution in NFV management and orchestration.

References

[1] W. Liu, H. Li, O. Huang, M. Boucadair, N. Leymann, Q. Fu, Q. Sun, C. Pham, C. Huang, J. Zhu, and P. He, "Service Function Chaining Problem Statement," *draft-liu-sfc-use-cases-08 (work in progress)*, 2014.

[2] W. Haeffner, J. Napper, M. Stiemerling, D. Lopez, and J. Uttaro, "Service Function Chaining Use Cases in Mobile Networks," 2014.

[3] S. Surendra, M. Tufail, S. Majee, C. Captari, and S. Homma, "Service Function Chaining Use Cases in Mobile Networks," 2014.

[4] J. Sherry and S. Ratnasamy, "A Survey of Enterprise Middlebox Deployments," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2012-24, Feb 2012.

[5] ETSI, "Network Functions Virtualisation – Introductory White Paper," https://portal.etsi.org/NFV/NFV_White_Paper.pdf, 2012.

[6] ——, "Network functions virtualisation (nfv); management and orchestration," http://www.etsi.org/technologies-clusters/technologies/nfv, 2014.

[7] J. Martins, M. Ahmed, C. Raiciu, V. Olteanu, M. Honda, R. Bifulco, and F. Huici, "ClickOS and the art of network function virtualization," in *Proceedings of NSDI '14*. USENIX Association, 2014, pp. 459–473.

[8] J. Hwang, K. K. Ramakrishnan, and T. Wood, "NetVM: High performance and flexible networking using virtualization on commodity platforms," in *Proceedings of NSDI '14*. USENIX Association, 2014, pp. 445–458.

[9] R. Jain and S. Paul, "Network virtualization and software defined networking for cloud computing: a survey," *Communications Magazine, IEEE*, vol. 51, no. 11, pp. 24–31, 2013.

[10] "OpenStack Open Source Cloud Computing Software," http://openstack.org/.

[11] A. Gember, A. Krishnamurthy, S. S. John, R. Grandl, X. Gao, A. Anand, T. Benson, A. Akella, and V. Sekar, "Stratos: A network-aware orchestration layer for middleboxes in the cloud," *CoRR*, vol. abs/1305.0209, 2013. [Online]. Available: http://arxiv.org/abs/1305.0209

[12] A. Gember-Jacobson, R. Viswanathan, C. Prakash, R. Grandl, J. Khalid, S. Das, and A. Akella, "OpenNF: Enabling innovation in network function control," in *Proc. of SIGCOMM*. ACM, 2014, pp. 163–174.

[13] S. Rajagopalan, D. Williams, H. Jamjoom, and A. Warfield, "Split/merge: System support for elastic execution in virtual middleboxes." in *Proc. of USENIX NSDI*, 2013, pp. 227–240.

[14] M. F. Bari, S. R. Chowdhury, R. Ahmed, and R. Boutaba, "On orchestrating virtual network functions in NFV," *CoRR*, vol. abs/1503.06377, 2015. [Online]. Available: http://arxiv.org/abs/1503.06377

[15] "Linux Bridge," http://www.linuxfoundation.org/collaborate/workgroups/networking/bridge.

[16] "OVS: Open vSwitch," https://linuxcontainers.org/.

[17] A. Traeger, E. Zadok, N. Joukov, and C. P. Wright, "A nine year study of file system and storage benchmarking," *ACM Transactions on Storage (TOS)*, vol. 4, no. 2, p. 5, 2008.

[18] R. Hasan, Z. Anwar, W. Yurcik, L. Brumbaugh, and R. Campbell, "A survey of peer-to-peer storage techniques for distributed file systems," in *Information Technology: Coding and Computing, 2005. ITCC 2005. International Conference on*, vol. 2. IEEE, 2005, pp. 205–213.

[19] M. Satyanarayanan, "A survey of distributed file systems," *Annual Review of Computer Science*, vol. 4, no. 1, pp. 73–104, 1990.

[20] "Limited Shell (lshell)," https://github.com/ghantoos/lshell.

[21] "Docker," http://docker.com/.

[22] "LXC: Linux Containers," https://linuxcontainers.org/.

[23] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: enabling innovation in campus networks," *SIGCOMM CCR*, vol. 38, no. 2, pp. 69–74, 2008.

[24] "fusepy," https://github.com/terencehonles/fusepy.

[25] "libvirt: The virtualization API," http://libvirt.org/.

[26] "sshfs," http://fuse.sourceforge.net/sshfs.html.

[27] "Chef," https://www.chef.io/chef/.

[28] M. Monaco, O. Michel, and E. Keller, "Applying operating system principles to SDN controller design," in *Proc. of HotNets*. ACM, 2013.

[29] "Apache CloudStack: Open Source Cloud Computing," http://cloudstack.apache.org/.

[30] "SaltStack automation for CloudOps, ITOps & DevOps at scale," http://saltstack.com/.

[31] "OpenStack TelcoWorkingGroup," https://wiki.openstack.org/wiki/ TelcoWorkingGroup.

[32] "Cosmos," http://cnp.neclab.eu/clickos/.