A Web Server to Dynamically Generate Network Slices with User-Defined Virtual Network Function Service Chains

Submitted by

Sean K. Walker

in partial fulfillment of the requirements for the degree of Bachelor of Science in Computer Science.

Adviser: Professor Y. Richard Yang

Yale University

May 2019
ABSTRACT

5G networks promise to bring unparalleled scalability, speed, and stability to mobile network users all around the globe. However, the specifics of how these goals will be achieved are neither obvious nor fully classified. Although there is a wealth of research about potential methods of improving network architecture to achieve the 5G standard’s goals, one technique in particular—network function virtualization—will be vital to this end. Network function virtualization (NFV) is the usage of software to implement what is traditionally hardware-implemented network functionality (e.g., firewalls, load balancers, deep packet inspection, etc.) on standard commercial middleboxes. NFV allows network operators to significantly decrease costs while simultaneously enabling much higher elasticity and scalability to network traffic and resource demands. Despite this, however, many existing NFV frameworks make a significant tradeoff between performance and responsiveness, either having low performance metrics in favor of modularity to quickly changing service or vice-versa. In this paper, we leverage a recently-introduced, highly performant NFV framework in order to develop a system for dynamically generating virtual network function (VNF) service chains at runtime. The resulting system model allows for both impressive performance through its chosen framework as well as dramatically increased flexibility of service and responsiveness to arbitrary network demands, in terms of both traffic volume and type. We detail the motivation and background for the particular system, outline design choices and technologies used, give an overview of the system’s public API for interacting with an orchestration system, and offer suggestions for future related work and research in similar domains.
INTRODUCTION

A. Goals of 5G

The term “5G” is ambiguous and can refer to a variety of different network types. For the purposes of this paper, it will refer to the upcoming standards specified by the ITU in its IMT-2020 release. Although these precise requirements of 5G networks are yet to be formally specified by the ITU\(^1\), other organizations have created their own standards in the meantime. For instance, the 3GPP has created a standard known as “5G NR” (5G New Radio), which it will submit to be considered an IMT-2020-compliant 5G standard after the ITU releases its specification.

IMT-2020 posits a number of key metrics which any compliant network must support. To name a few, the network’s maximum data rate must be 20 Gbps, the radio network should contribute a maximum of 1ms to latency, energy consumption should be no more than 4G, and devices moving at up to 500kmph should not experience any degraded QoS. \(^1\) There have been many unique and creative proposals to achieving these goals, with a few common themes and techniques present throughout all the proposed solutions.

B. Cloud-Native Architecture

One of the biggest challenges of achieving the scale of speed, reliability, and accessibility 5G must offer is simply that the existing physical infrastructure is insufficient. \(^2\) 5G networks must be designed to support a radically larger number of devices than presently use mobile networks due to the rapidly expanding presence of Internet of Things (IoT) devices. \(^3\) One move toward being able to accomplish these goals is to favor software-defined networking (SDN) capabilities over hardware-defined ones as well as favoring virtual network functions over hardware boxes. SDN and NFV are almost always used in conjunction to allow network providers to circumvent nearly any need for specialized hardware functionality in deploying networks.

---

\(^1\) As its name suggests, ITU plans IMT-2020 to be formally released by 2020.
A popular example of NFV would be replacing physical routers with software routers. The benefits of leveraging NFV in such a scenario are numerous: for one, deployment of NFVs is much cheaper, as it can be as simple as installing software on a VM running in the cloud. Moreover, deploying updates is similarly much less costly and difficult than upgrading physical devices. Finally, it allows services to be remotely managed with greater ease by network administrators, especially using SDN tools to remotely manage groups of VNFs together.

In this way, 5G is called a “cloud-native” architecture: although on some level the networks must still be physically supported by links and radio towers, most all of the functionality of the networks will be programmed virtually to allow for extreme responsiveness to network traffic volume, mobility, and the like.

C. Challenges for VNF-Driven Networks

As outlined in [4], although many companies already use NFV and SDN systems to a significant degree, a variety of notable problems face the next step of the complete implementation of a cloud-native architecture. For instance, many NFV frameworks are not interoperable with others, use platform-specific code and are thus not portable, and often lack performance compared “one-on-one” to non-virtualized versions.\(^2\)

\(^2\) Even in those instances, NFV is still overall likely preferable due to scalability, but certainly one would seek the most performant single functions possible
BACKGROUND AND MOTIVATION

In this section we discuss motivation concerning easing development of VNF service chains and design choices made for bases of the project.

A. Building Network Functions

All this begs the question of how one might go about writing and deploying a virtual network function to be used in a real production scenario. Many companies have already developed and run their own proprietary VNFs, such as Cisco’s “Enterprise NFV” and AT&T’s “Integrated Cloud” data centers. [5, 6] On the open source side, a number of options exist for SDN and NFV development. Examples of popular NFV development libraries include ClickOS [7] and NetBricks [8]. In this project, we build off of the later framework due to its impressive performance metrics.

B. NetBricks

NetBricks was introduced in 2016 as a novel framework for writing VNFs in Rust. [8] In comparison to competing frameworks, NetBricks boasted much higher performance metrics, with VNFs written in NetBricks even outperforming non-NetBricks implementations of certain VNFs, such as Google’s Maglev.

NetBricks achieves this performance through several interesting techniques. Firstly, NetBricks makes a modularity tradeoff in favor of performance. Unlike other libraries, NetBricks does not require a single VNF be isolated in a single container, but instead dedicates a single process to each VNF chain, of arbitrary length. Thus, although using the library as-is one must predict the exact service chains which will be required, the result is that VNFs running without such overhead in NetBricks can be up to 11x faster. [8]

However, even though NetBricks is not as modular on the face as other frameworks, the modularity tradeoff does not equate to a lack of isolation. By leveraging Rust’s strong static type safety and memory management system, NetBricks also guarantees packet isolation between NFs
through Rust’s notion of ownership: because a piece of memory can only be “owned” by one function at a time, it is impossible for multiple VNFs to be able to manipulate a single packet at once. The NetBricks authors dub this technique “zero copy software isolation” (ZCSI). [8]

C. Network Slicing

In 5G, it is vital that one user’s service does not degrade as a result of a change in the network usage by other users. In order to ensure this, researchers have proposed the theoretical technique of network slicing. A network slice is an entire logical network devoted to a single UE; all these slices, however, operate on the same physical network. [9] That is, from the UE’s point of view, it is the only UE using an entire mobile network, when in reality, it is still sharing resources with every other device connected to the network, but just separated logically. Isolating networks in this way allows for service isolation: if each slice is fully configurable, then for any given class of UE any number of different parameters, e.g. latency, bandwidth, etc. can be guaranteed.

D. Docker

Docker is an extremely popular tool for building and managing containers. A container, in short, is an isolated userspace. In practice, containers can often act as lower-overhead substitutes for virtual machines (VMs). Because containers all use the same kernel, there is no overhead of a hypervisor required to use them, and communications are much more direct.

Here we leverage Docker as a tool to build network slices. Each Docker container comprises a single logical slice, and within each container we run a single process, a NetBricks binary which contains the code for a single VNF service chain.
In this section we discuss decisions made about the structure and function of the project.

A. Introduction to the Project

Although NetBricks offers a performant and user-friendly way framework of VNF development, as previously stated, it comes with the tradeoff of modularity. Since VNF service chains are run in a single process in order to increase performance, if one wishes to use a single NF in different service chains, two separate source files using that NF must be written, compiled, and distributed to any middleboxes which would potentially need to run that service chain. We believe there is a middle ground between the performance of single-process service chains and single-container/single-VM NFs. We present an HTTP server which can create arbitrary VNF service chains from a library of NF source files given a configuration file specifying a directed graph representing the service chain to be created. See Figure 1 on the next page for a visualization of the functionality of the system.

**Figure 1.** A diagram of the overall structure of the server. Traffic is routed to the middlebox by the routing/load balancing protocol used by the orchestrator. Requests for allocating a new user equipment (UE) to a slice are sent to the server, which confirms and responds with routing information. Traffic for existing UEs is known by the router and sent accordingly.
B. Overview of Traffic Flow

To initialize, the datacenter (DC) middleboxes should start the server software. This opens a dedicated port to listen to new requests for connections. Traffic coming to the data center, as dictated by the orchestrator, will then be directed to a middlebox with sufficient capacity for another connection. When the target middlebox is determined, it is then up to the orchestrator to create a configuration file, in JSON format, to specify the VNF service chain desired. After this is done, it is sent over HTTP to the target server, which parses the configuration file and creates a new container for this new device connection to be run through. Here, each container represents a single logical “network slice,” where each slice’s functionality is completely configurable by the orchestrator according to the specific demands of the UE. For instance, while traditional data connections might have slices which have a virtualized evolved packet core (EPC), IoT UEs would likely utilize a simpler slice with perhaps just some shared data store which a massive number of devices can access and update.

Once a slice is created, it is given a MAC address which is in turn relayed to the orchestrator. Thereafter, the orchestrator should address frames based on that MAC address when sending traffic to the DC for that same UE. Thus, when the packets are received at the middlebox, it will be sent to the MAC address of the slice running the proper service chain for that UE.
IMPLEMENTATION

In this section, specific notes about how the system was implemented are made.\(^3\)

A. Server

1. Overview

The server is implemented in JavaScript, leveraging Node.js and the Express library for their easy programmability and low programming overhead. More performant servers could certainly be written (and likely should be, for real implementations), but this server merely serves as a model of the functionality and not a paragon of performance. One of the server’s API endpoints is used by the orchestrator to initialize new slices. This endpoint takes POST requests with a mandatory argument of a JSON configuration file specifying the service chain to be put in the new slice.

2. Configuration File Format

Configuration files, as previously stated, are written in JSON (JavaScript Object Notation), partially for its simplicity and readability and partially because it is a natural complement to a server written in JavaScript. Each configuration file should be an array of objects, where each object consists of a single key, “name”, which should be the exact name of the NF desired to be run. This configuration file specification is minimal and extensible to leave room for improvements to the system as needed; in its current form, it merely provides the required minimum functionality to show feasibility of this software model.

\(^3\) Note that all code is available on GitHub at https://github.com/seankwalker/cpsc-490
3. API

The server supports multiple API endpoints for the orchestrator to interact with, in addition to the /start endpoint discussed above. Those endpoints are listed here:

- **POST /capacity**
  - Query parameters: **data (.json file upload)**: a configuration file
  - Description: responds with list of remaining connection capacity for all containers currently running specified service chain.

- **GET /does-support**
  - Query Parameters: **nf**: a network function name
  - Description: responds with true or false reflecting whether the server supports the specified network function name.

- **POST /start**
  - Query Parameters: **data (.json file upload)**: a configuration file
  - Description: creates a new slice running the service chain specified in the provided configuration file. Responds with the MAC address of the created slice.

- **GET /supported**
  - Query Parameters: none
  - Description: Responds with a list of the server’s supported NFs.

B. Slices

Each slice can support a server-specified number of connections; this allows for better performance, as different service chains will have very different traffic loads. For instance, whereas a slice dedicated toward streaming may have to consistently be serving data and thus
adding multiple clients could very quickly degrade QoS, a slice made for periodic, predictable IoT communication will likely have significant periods of downtime and can easily interleave requests of different clients. As implemented, each slice is merely a Docker container in which the source code for the specified service chain is built and run. Because every function chain is built in NetBricks, the containerization’s isolation of packets may seem redundant, but it does offer some further benefits. For one, note that while NetBricks may offer memory safety through Rust, it offers no controls on CPU usage. As such, if one attempted to naively run multiple NetBricks VNF chains on a box without containerization, it’s very possible that resource usage will be unequally and unfairly distributed. Managing containers, however, allows for realtime monitoring of resource usage and limiting if need be.
FUTURE WORK

This project offers a step toward realizing easily configurable, dynamic allocation of slices dedicated to VNF chains. However, it is far from the end of progress in this direction. There is much potential for future work in this area. We offer a number of ideas here:

- Developing an orchestrator system to complement the middlebox server software offered here. Although this project is functional, it’s really only half of the requisite software to be used in a real-life scenario.

- Alternatively, investigating interoperability with existing orchestrators would also be viable. Systems like E2 [10] have shown to be effective systems in this regard, and if this server system could be made to work with these already-proven orchestrators, it would be perhaps a better choice than attempting to design a new orchestrator.

- Either way, benchmarking this server model, especially with an orchestrator system. As can be seen from the code on GitHub, only experimental testing was able to be performed, and benchmarking was unable to be performed on this setup. However, without benchmarking the whole theoretical system—including a complete orchestrator and dedicated packet generation server—benchmarking results would be hard to extrapolate anyhow. This sort of data would be immensely valuable in evaluating various aspects of how this software model could be improved going forward and would be invaluable in focusing future research.
CONCLUSIONS

Although many companies have already been launching what they claim are 5G-compliant networks, only time will tell if they can live up to the IMT-2020 standard. It is clear, though, that no matter what exact shape 5G-compliant networks end up taking, NFV and network slicing will be vital parts of the architecture. In this paper, we present a software model which can facilitate the easy deployment of user-specified network functions to remote middleboxes via an orchestrator controlling servers at each middlebox. As such, specifically any future VNFs written in NetBricks can be easily made to fit with such a model to be quickly and easily deployable, and additionally a similar model could be adapted for any other VNF framework if necessary.

There is still much work to do for this model to completely ready for deployment and usage in real-world situations. For one, a minimally-functional orchestrator should be written. After that, thorough benchmarking would allow insight into what parts of the model do and do not work, where improvements can be made, and the overall viability of this model for further research and investigation.
REFERENCES


