P4SFC: A tiny SFC implemented in P4

Ying-Chiao Chen
National Taiwan University
Taipei, Taiwan
b07901184@ntu.edu.tw

Kai-Hsiang Chou
National Taiwan University
Taipei, Taiwan
b07705022@ntu.edu.tw

Hsiang-Hsuan Fan
National Taiwan University
Taipei, Taiwan
r09942130@ntu.edu.tw

Abstract

In this project, we implemented a tiny SFC using p4 as the data plane and P4Runtime as the control plane. We design our headers based on previous works to not only meet our needs but also simplify the network model. The SFC we created can demonstrate all properties in our design. The SFC gateway can wrap a packet from the external network with our own SFC headers and remove them when leaving the network. The SFC classifier can generate proper information about the packet. Services in the network (including the gateway) are capable of deciding the next hop given information from itself and previous hops. With our implementation, we envision P4 as an alternative solution to SFC to provide better performance than NFV in many cases adopted in the current world.

CCS Concepts: • **Networks** → **Programmable networks**.

Keywords: Service Function Chaining, P4, Network Service Header

ACM Reference Format:
Ying-Chiao Chen, Kai-Hsiang Chou, and Hsiang-Hsuan Fan. 2022. P4SFC: A tiny SFC implemented in P4. In *NVS '22: Network Virtualization and Security Final Project, Jan 14, 2022, Taipei, Taiwan.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

End-to-end service delivery often requires several different service functions, such as firewalls, Quality of Service, load balancing, malware detection, and other application-specific functionalities. The ordered set of service functions together with the traffic routing handling is called **Service Function Chaining**, or **SFC**. SFC enables the composition of different service functions without direct dependency among any. Although SFC can be implemented in the traditional network, it has several downside such as restricted flexibility, network shutdown, and poor scalability.

In this work, we present a simplified model of SFC inspired by Quinn and Guichard's work[4] and [3]. We design our own headers to not only meet our needs but also simplify our model.

The headers comprises 3 parts: `sfc_header`, `sfc_service`, and `sfc_context`. `sfc_header` describes general information of the packet. `sfc_service` provides some information for the next service. `sfc_context` can do further than `sfc_service` by passing the information along the path. Using these 3 header types, we provide a strong flexibility and scalability by distributing functionalities into several instances while employing a state-machine like processing mechanism.

As for our implementation, we choose to implement it in P4. To demonstrate how our SFC model works, we also implemented a basic firewall and QoS in P4. The SFC is implemented entirely follow the design specification. We also open-sourced the implementation [1].

Due to time constraints, we left several components undone, and we still not able to integrate our project with SDN and NFV to make a more powerful system.

Our main contributions are:

- We implemented a general framework of SFC using P4.
- We open-sourced our project on GitHub.

2 Motivation

In the traditional networks, SFC exhibits several problems. First of all, the flexibility of SFC is highly related to the topology of the networks. Adding or altering any service function without causing downtime is difficult, as the network administers may need to install or rewire the physical devices. Also, SFC is not scalable and can't handle sudden peaks gracefully. Last but not least, handling different types of traffic flows with the same architecture can be difficult or even impossible. For example, one may require only some packets to pass through the firewalls, but this can be hard to handle in the traditional SFC.

With Software-defined networking (SDN), these problems can be easily solved, that is, with the flexibility of the SDN, we can enhance the functionality of the SFC. An SDN controller may apply a different chain of services to distinct traffic flows based on the source, destination, or type of traffic. This functionality automates a sequence of physical L4 to

NVS '22, Jan 14, 2022, Taipei, Taiwan

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$00.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

L7 devices to process the network packets without requiring different devices and physical wiring.

In recent years, another technology, NFV (network function virtualization), becomes popular to integrate with SDN to implement SFC.

On one hand, SDN helps create a holistic view of the entire network, which can optimize routing rules based on metrics collected in each switch instead of the use of some traditional routing protocol such as OSPF (open shortest path first), which calculates rules in a distributed way. On the other hand, NFV leverages virtualization technologies to implement NFs (network function) on commodity servers. The combination of the two makes the network extremely powerful and flexible.

However, although NFV provides a lot of flexibility, it also introduces more latency in packet processing because NFs are implemented in software rather than hardware. It is unacceptable for latency-sensitive applications. To overcome this problem, Ma, Xie, Zhao developed an SFC system written in P4 [3], which analyzed the offloadability of NFs from NFV to P4 and finally gained a latency reduction of around 20% and 46.5 % in throughput compared with a realistic SFC.

Following the concept, we want to create an SFC entirely using P4 and SDN and acquire benefits from both of them. To make our implementation more powerful, we add the notion of NSH (network service header) from Quinn and Guichard's work[4] to our work.

3 Design

In this section, we present the design of our SFC model. We first introduce components of our models in Section 3.1 and the custom protocol in Section 3.2. The design of our demonstrating service functions is presented in Section 3.3.

3.1 Components

Chou: maybe this should be mentioned in the introduction?

A common structure of the SFC model includes four major components: SFC classifier, Service Function Instance (SFI), Service Function Forwarder (SFF), and Service Function Proxy (SF Proxy).

SFC classifier SFC classifier is the entrance of the SFC. Every packet entering the SFC will first go through the SFC classifier. In the common cases of SFC, there would be several different paths. For example, some packets may need to go through the firewall, while others don't. The SFC classifier is responsible for identifying which path should the incoming packet take. Specifically, we will add some entries to the header, which is called *SFC Header* and will be further discussed in Section 3.2, and the rest of the SFC would redirect and parse the packet according to the SFC header.

Service Function Instance The Service Function Instance (SFI) is the instance of service function, such as firewall and QoS. It can be a process or a physical device.

Service Function Forwarder The Service Function Forwarder (SFF) provides service layer forwarding. It is responsible to redirect the packet to the corresponding SFIs according to the SFC header. In many cases, including our implementation, this is combined with either SFC classifiers or SFIs.

Service Function Proxy Suppose that one wants to use the legacy device or service that doesn't understand the SFC protocol, the Service Function Proxy (SF Proxy) is responsible for handling these devices or services. The SF Proxy would remove the SFC header and other custom parts of the packet before sending the packet to the legacy device or service. After receiving the return of the device or service, the SF Proxy would restore the SFC header and the custom part, and send it back to the SFF or the next SFI.

3.2 SFC Header

We use header entries to define how the SFC handles the packet. The header is injected into the packet by the SFC classifier to hint at how the packet should be forwarded, as well as the metadata that would be helpful to the SFIs.

The SFC header includes `sfc_header_t` (Table 1), `sfc_service_t` (Table 2), `sfc_context_t` (Table 3).

3.2.1 sfc_header_t.

`sfc_header_t` is the overall information of a packet. It contains the general information of this packet after being classified by the SFC classifier. These fields will not be changed throughout the journey of SFC.

- **version**: the version of the SFC protocol, which can help SFC to gracefully migrate from one version of the protocol to another.
- **max_size**: the maximal size of SFC context, which is further explained in 3.2.3.
- **type**: the type of this packet. For example, it can be general, web, mobile, and so on. This field may be used by some SFC instances.
- **qos**: the QoS level of this packet. This field is used in the QoS instance.
- **dst_id**: the final destination of the packet. The final destination is determined by the classifier in the beginning. Each packet may differ in paths, but the destination must be the same if given the same dst_id.

3.2.2 sfc_service_t.

`sfc_service_t` carries information needed by the next hop by the current hop. It is just like function calls in programming languages. You give the name of an SFC instance, some parameters, and the possible actions to do. The next SFC instance will first verify the information and do something that provided the information from the previous instance.

- **type**: the type of SFC instance. It is a synonym of the name of the instance. However, what we care about is just the functionality of an instance, not its name. Moreover, there may be several instances with the

Table 1. The fields in `sfc_header_t`

`version`	2
`max_size`	4
`type`	4
`qos`	2
`dst_id`	4

Table 2. The fields in `sfc_service_t`

`type`	6
`status`	2
`act`	4
`params`	4

Table 3. The fields in `sfc_context_t`

`bos`	1
`content`	15

same capabilities. Therefore, we choose to use `type` instead of `name` as the name of this field.

- **status**: the status of this packet. A packet may be classified into some status, like running, failed, in the previous instance. The next instance can do some recovering or simply discard it in terms of this field.
- **act**: the possible actions that the next instance can take. The next SFC instance may take this as a suggestion when making a decision. By the way, the naming of this field is that `action` is a keyword of P4.
- **params**: the parameter for the next instance. This field increases the diversity of an instance can do by adding some modes to choose from.

3.2.3 sfc_context_t. `sfc_context_t` contains information of each hops in a trace. Each instance the packet passed can add some context to it. It provides a way of message passing beyond 2 hops, which is the main difference from `sfc_service_t`. It is worth noting that usually the context is the final decision of each instance, so it should be very condensed and can be categorized into 15 bits. Each category is defined by users and should be explicitly put into the source code of each P4 program.

- **bos**: to tell the parse whether this context is at the end of the stack.
- **context**: the contexts in a trace. Each context is a final decision of an instance. Along the SFC, the headers start to grow, and it can contains many contexts no more than `max_size` mentioned in 3.2.1. In addition, there is still a global maximal size of contexts. The program will take the smaller one as the constraint.

3.3 Service Functions

While not the core components of this project, to demonstrate how our SFC implementation can be used, we designed and implemented some service functions based on p4tutorial [2]. In this section, we will explain the functionality and the improvement of our design.

3.3.1 Firewall. We implemented a stateful firewall that can block all incoming requests while allowing the outgoing ones, and are capable of handling TCP connection termination (i.e., **FIN** and **FIN ACK** packets.) Moreover, we use the bloom filters to provide high-performance queries and updates.

Chou: TODO: maybe add more SF?

4 Implementation

We use the structure of p4tutorial [2] as our template. To simplify the implementation, we assume all switches in our SFC are specialized P4 switches. Therefore, we can install a P4 program on each switch separately to make the network more flexible.

4.1 Topology

We choose this tiny topology as our implementation due to time constraints. As you can see in the figure 1, we have 2 gateways serving as ingress and egress gateway respectively. There are 2 services, firewall, and QoS. The firewall service will add a `sfc_context_t` in the headers and then the QoS classifier will determine which port to forward to. We will further explain the configuration in 4.4.

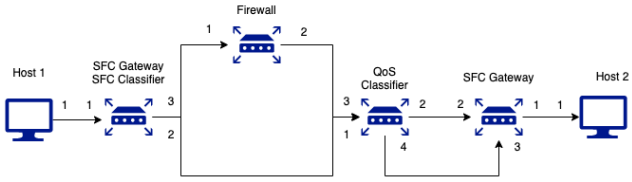

Figure 1. A simple SFC

4.2 Switch Type

We divided switched into 2 categories: edge switch and internal switch.

4.2.1 Edge Switch. Edge switches are located at the boundary of our SFC. An SFC can have several ingress gateways and egress gateways. Each gateway should be able to add SFC headers before the packet enters the network and strip SFC headers when the packet leaves the network. The SFC gateway is combined with the SFC classifier to make the network more condensed. Therefore, the gateway has to populate the headers based on the information provided in the original packets and then determine the first hop to forward to.

4.2.2 Internal Switch. Internal switches are switches within the network. They assume all incoming packets have SFC headers. When a packet arrives, a service first checks it contains SFC headers, then check if the service type is correct, and finally process the packet.

4.3 Control Plane

To simplify our development, we use P4Runtime as our control plane instead of SDNs like ONOS. We found that ONOS is too hard to integrate with P4 currently (at least in the time we surveyed). Hence, we use the p4runtime control plane from p4tutorial [2] as our template and modify the code for our use.

In figure 2, our program (`test.p4`) is first compiled by p4 compiler. It generates 2 files: `test.p4.info` and `test.json`. Ths `test.p4.info` is target-independent, it contains gRPC client and server stub and it is of protobuf format. Thus, the client and the server can communicate through gRPC. It defines the schema of pipeline for runtime control, such as tables and actions. `test.json` is target-dependent. In this case, it is compiled into the format of BMv2, but it also can be compiled to another format, like ASIC. This file determines the actual behavior of the switch, which is used to realize switch pipeline.

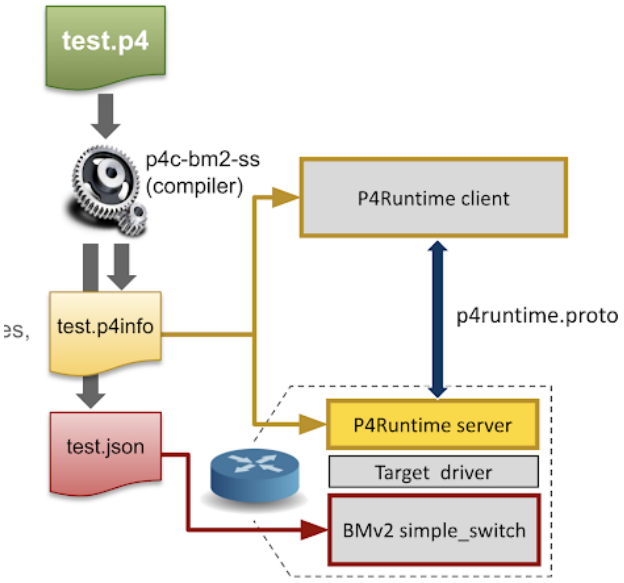

Figure 2. P4Runtime.

4.4 Demo

In this section, we demonstrate how our implementation works. Host 1 wants to communicate to Host 2, and it has 2 options: using a firewall or not. With a firewall, Host 2 cannot initiate a connection with Host 1 but can still make a reply when the initiator is Host 1. The topology is the same as 4.1.

Each service works as mentioned in 3. In detail, SFC gateways determine the overall information of this packet, populate the next hop information, and add SFC gateway context in the headers. The firewall and QoS also service decide the next hop and modify the field in the SFC service header and add their own context as well.

4.4.1 Without Firewall. In figure 3, after passing the SFC classifier, the packet is forwarded to port 2, where it connects to the QoS classifier without the firewall. After that, the QoS classifier determines to forward this packet through port 2 to the destination SFC gateway. In the opposite direction, the packet just follows the path of the original one but in the reverse direction.

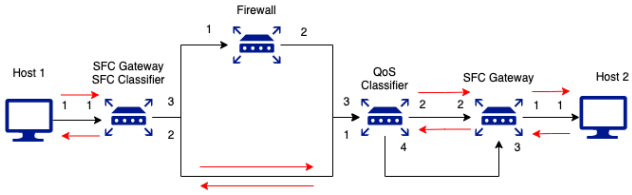

Figure 3. Routes without a firewall.

4.4.2 With Firewall. In figure 4, after passing the SFC classifier, the packets are forwarded to port 3. Contrary to the above scenario, this time the packet has to pass through the firewall. The firewall will add a firewall context and then forward it to the QoS classifier. Seeing the firewall context appended by the firewall, the QoS classifier forwards the packet to port 4 rather than port 2 in the above example. This case demonstrates the use of the SFC context header to pass information. Although the message can be passed using the SFC service header, the benefit of the SFC context is that the information is kept along the path and is visible to later service, SFC Gateway in this example.

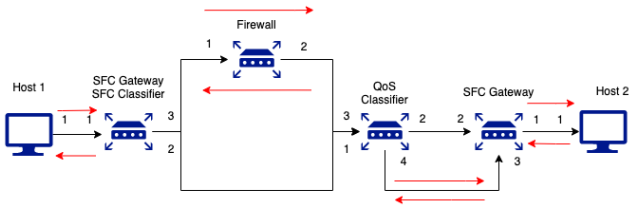

Figure 4. Routes with a firewall.

5 Future Works

5.1 More Complex SFC

Because of the time limit of this project, we are not able to implement all features we want, like the following figure 5 shows:

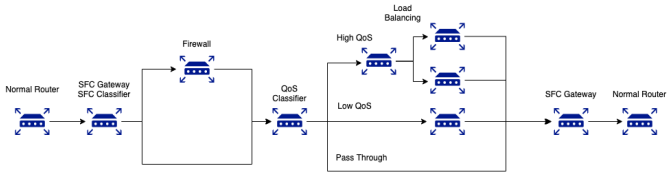

Figure 5. A more complex SFC.

The QoS classifier can distribute service into 3 categories: High QoS, Low QoS, and passthrough. For High QoS, we even integrate the path with a load balance to share traffic among several instances to improve performance. For Low QoS, we only have 1 instance. As for passthrough, it's about the packet that does not require another level of service, such as encryption, so the packet can be passed directly to the egress SFC gateway.

5.2 Integration with SDN and NFV

In the future, this project may integrate SDN and NFV to form a more powerful and flexible network system. Furthermore, it is even possible to create a component management system to automatically create, configure, and destroy the whole system while collecting metrics for visualization.

6 Conclusion

We implement a simplified model of SFC using P4 as the data plane and P4Runtime as the control plane. In addition, we designed our own SFC headers to pass the information along the routing path. As a result, we create a simple SFC with 2 gateways and a firewall, and a QoS classifier to form a working tiny example SFC.

7 Work Contribution

- Ying-Chiao Chen: Implement the architecture of SFC, including sub-services like SFC gateway, SFC classifier.
- Kai-Hsiang Chou: Implement the service function instances and write the report.
- Hsiang-Hsuan Fan: File editing and topic research.

References

[1] 2022. P4-SFC. https://github.com/s3131212/P4-SFC.
[2] 2022. p4tutorial. https://github.com/p4lang/tutorials.
[3] Junte Ma, Sihao Xie, and Jin Zhao. 2020. P4SFC: Service Function Chain Offloading with Programmable Switches. In *2020 IEEE 39th International Performance Computing and Communications Conference (IPCCC)*. 1–6. https://doi.org/10.1109/IPCCC50635.2020.9391530
[4] Paul Quinn and Jim Guichard. 2014. Service Function Chaining: Creating a Service Plane via Network Service Headers. *Computer* 47, 11 (2014), 38–44. https://doi.org/10.1109/MC.2014.328