Network storage with P4

Tamas Lengyel
Noel Hetei
Peter Kokai
d1b5hi@inf.elte.hu
njoim8@inf.elte.hu
kokaipeter@gmail.com

ABSTRACT

Using P4, python and MiniNet our task was to create a network storage which could use any custom topology, given a few constraints. With a simple python client, data could be uploaded, queried and removed from the network.

KEYWORDS

p4lang, network storage, mininet, bmv2

ACM Reference Format:
Tamas Lengyel, Noel Hetei, and Peter Kokai. 2018. Network storage with P4. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

P4 introduces a standardized way to data plane programming. It defines how a switch proccesses incoming and outgoing packets. Using python scripting language and MiniNet[2], which creates a virtual network, we created a network storage system where packets are circulating from switch to switch. The goal of the project is to be able to upload, query and remove data from a custom topology using a simple python client. Certain methods and implementation details are going to be described in this paper, along with some of the challenges we faced and how we overcame them.

2 OVERVEIW

In this section we are going show what environment we used to implement the project, and what were the main features we were planning to implement.

2.1 Preparation

We used a premade virtual machine environment we found at **p4lang/tutorials** [6]. Setting up *VirtualBox* with *Vagrant* tool was required to access the environment. This environment uses the **P4 v1.0 switch model** [8] and the **BMv2 behavioral model** [1].

2.2 Tasks

Initially, we had the following features in mind for the program:

- `Storing data`: Where and how to store the incoming data?
- `Implementing operation`: How to handle incoming operations?
- `Handling Parallel Requests`: What happens when multiple clients connect?
- `Data Identification`: How to identify the data?
- `Handling Large Data`: What happens when large amount of data arrive to a switch?

3 CHALLENGES & SIMPLIFICATIONS

The design and the capabilities of a database was kept minimal, as the main aim was to explore the possibility of implementing a database on top of a P4 capable switch. A complex database feature are just a combination of these simplified functions.

3.1 Storing Data

Storing data has raised a lot of questions that had to be considered, for example *What kind of data structure should we use?* and *How the servers positioned in the topology?*. We found that the header is capable of storing data, which was enough for us. We decided to simplify the second question by using only simple topologies where each switch has two or maximum three connections. The data can circulate inside the topology using any port.

3.2 Implementing Operations

We decided to implement three operations: *PUT, GET, REMOVE.* Since the clients require some kind of feedback from the server for all of these operations, we had to use package cloning, which became quite a challenge. We decided to **reserve port 3** in the topology for **client-server communication** only. This was necessary due to way cloning works: it uses a port that is predefined for the network.

Conference'17, July 2017, Washington, DC, USA
© 2018 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM. . . $15.00
https://doi.org/10.1145/1122445.1122456

3.3 Handling Parallel Requests

Handling multiple request at the same time is hard to implement. One possible solution is to use working queues. We had simplified the task so we have assumed that only one user sends data from a single computer into the topology.

3.4 Handling Large Data

If a pack of data cannot be fit inside one package the question is raised how should we handle them. The first method is splitting the data into separate packets, however we have found another, easier solution, where we are going to implement a size limit. It can be found both on the client and server side, when a larger data arrives it drops them.

3.5 Unique ID

In order to maintain consistency and a way for clients interact with the packages in the database, a unique id had to be assigned for each package. We found *hashing* to be the go-to solution.

4 METHODS

Cloning, id hashing and data forwarding were the most challenging parts of the project.

4.1 Cloning

In order to give a response for the clients the data has to be cloned. We use two headers to differentiate between data that are circulating and data that are used to give response or uploaded. Firstly, we clone in the *PUT operation* to send back the a result if the method was succesful. During this operation we copy all the incoming id, and data to the circulating header type. Second time for the *GET operation* becuase if we send back the data that is circulating in the topology we are going to remove it. This operation we follow the same logic that was behind the *PUT operation*, however, we use it in a revesre way. From the header of the circulating data we copy the data into the API header that is responsible for the communication with the clients. [3] [4]

4.2 Hashing

To be able to interact with data and maintain consistency we have given each datum a unique ID that is generated with the built-in CRC16 hashing [5] algorithm. We have used hashing during *ingress* and *egress processing.* First, we check which kind of header is active, then calculating the ID. Hashing has to be implemented twice in the ingress and egress for the two kind of header. As stated above we use cloning to send back data, however the cloned data is not containing the generated ID. In the egress processing we recalculate the hashed id to assign it to the clone. [7]

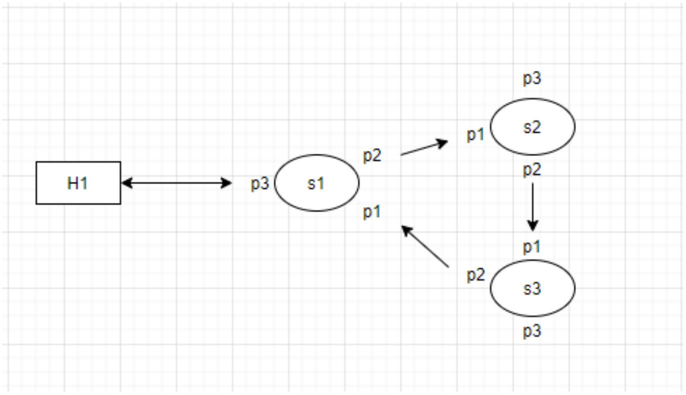
4.3 Forwarding

To make the data circulating in the given topology we have implemented two tables. One is for handling the different operations, *PUT, GET, and REMOVE*, the second table is for forwarding the messages. We use a boolean variable to mark data on *PUT and GET* operations to be circlulated. The key of the table is the ingress port. If the key is equal to port #1 or #3 data is going to be forwarded on port #2. The only incoming parameter for the actions is the port number where data are going to be forwarded.

5 TESTS & RESULTS

For testing we have used a linux virtual machine that has MiniNet pre-installed. To start the project we have to run the *init_netstore.py* to initialize the port where the communication with the client happens. After that the P4 program can be started. We used a triangle topology to test the program. It has one host and three swicthes all of which are circulating the data using the same port number. Communication with clients happens on port #3.

Figure 1: Topology graphical represenation



With the help of the log files we can check whether the results meet our expectations. As we could see the switches are forwarding the packets which are circulating in the topology as expected. During client side testing we have tried to cover as many branches as possible. We have used manual testing only for both the server and the client side.

6 CONCLUSION

Programing switches can be hard. Our task was to create a network storage in P4 language with the help of MiniNet and python. In the topology the data had to be circulated through switches. We have simplified some parts of the project, for example: the port where the communication executed with clients is hardcoded. Three operations had to be implemented: *PUT, GET and REMOVE.* The first two used cloning the give proper response to the clients. All data has unique IDs using the built-in hashing algorithm. With manual testing we tried to cover as many branches and test for all the possible inputs. The results met our expectations. The client side dropped every message that is larger than the limit, also responded correctly for different operations. Furthermore, using wrong format for the input, for instance misspelling the input command, results in execution break or prints a help message. Taking a look at the logs for the P4 program show that when puting data in the topology that is circulating through the switches, as expected. All things considered, we have achived our goal to store data.

REFERENCES

[1] *Behavioral model v2*. Accessed at 2020-06-28 21:07:00. URL: https://github.com/p4lang/behavioral-model.

[2] *Mininet documentation*. Accessed at 2020-06-28 21:07:00. URL: http://docs.mininet.org.

[3] *P4 cloning*. Accessed at 2020-06-28 21:07:00. URL: https://p4.org/p4-spec/docs/PSA.html#sec-clone.

[4] *P4 cloning*. Accessed at 2020-06-28 21:07:00. URL: http://csie.nqu.edu.tw/smallko/sdn/p4-clone.htm.

[5] *P4 hash functions*. Accessed at 2020-06-28 21:07:00. URL: https://p4.org/p4-spec/docs/PSA.html#sec-hash-function.

[6] *P4 lang tutorials*. Accessed at 2020-06-28 21:07:00. URL: https://github.com/p4lang/tutorials.

[7] Dominik Scholz et al. "Cryptographic hashing in p4 data planes". In: *2019 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*. IEEE. 2019, pp. 1–6.

[8] *V1model*. Accessed at 2020-06-28 21:07:00. URL: https://github.com/p4lang/p4c/blob/master/p4include/v1model.p4.