Cloud-native Software-defined Mobile Networks

Assignment #5
P4-enabled Softwre-defined Networking

Sharif University of Technology
Department of Electrical Engineering

Due Date:30.05.2020

Problem 1

In this problem we're going to implement a novel load balancing scheme which leverages on the programmability of data plane based on P4 language .
For more detailed explanation please refer to the paper by Hsu et. al., "Adaptive Weighted Traffic Splitting in Programmable Data Planes", in Proceedings of the Symposium on SDN Research (pp. 103-109), March, 2020.

Preliminaries

Let us assume there exists several paths between two hosts (see Fig.1), then the authors in the above mentioned paper propose a Weighted-Cost Multi-Path (WCMP) approach to balance the load across these paths in the data plane, in which a unique region is assigned in the hash function (hash of some fields in the headers) output space in proportion to the weight of each path in the load balancing switch, called as Data-plane Adaptive Splitting with Hash threshold (DASH).

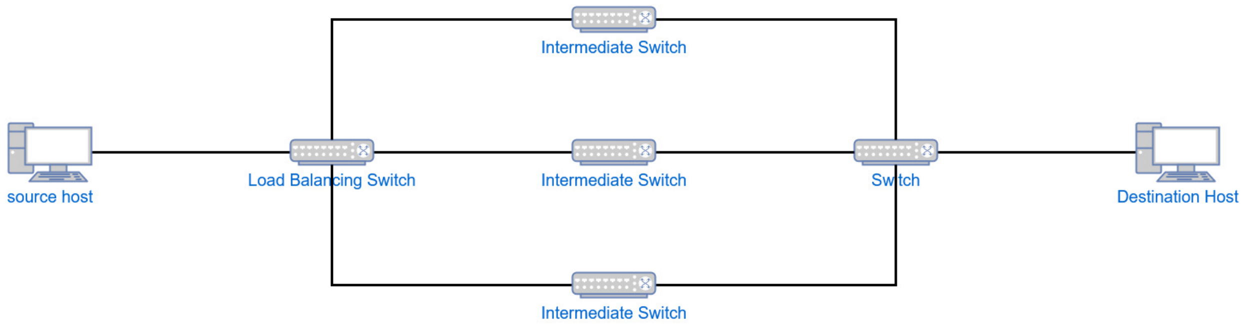

Figure 1: Sample topology

When an ordinary data packet arrives at the load balancing switch, the hash value generated from the packet header fields is compared against the region boundaries and determine the region it matches and the associated path. Consider a scenario where there are three paths towards a destination. We assign one path and the corresponding boundary to one stage in a multi-stage pipeline. As shown in Fig.2, a path's boundary is stored in the register memory mapped to a stage, and the boundary is compared against a packet's hash value. For instance, if the packet's hash value is less than the path boundary, then the packet is sent along that path.

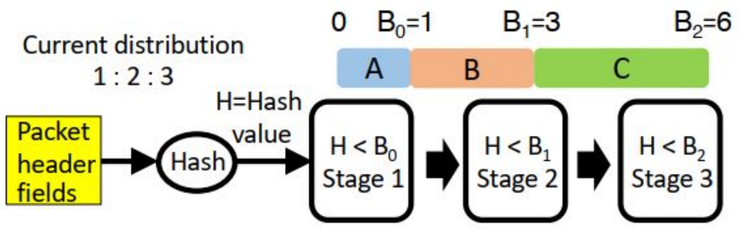

Figure 2: Packet-to-Path mapping using DASH

To get information about the load each path is experiencing, a DASH packet for each path is generated in the destination and forwarded to the source along these paths. When a packet arrives at each intermediate switch, that switch updates DASH headers using its metrics, e.g. queue length, as depicted in Fig.3.

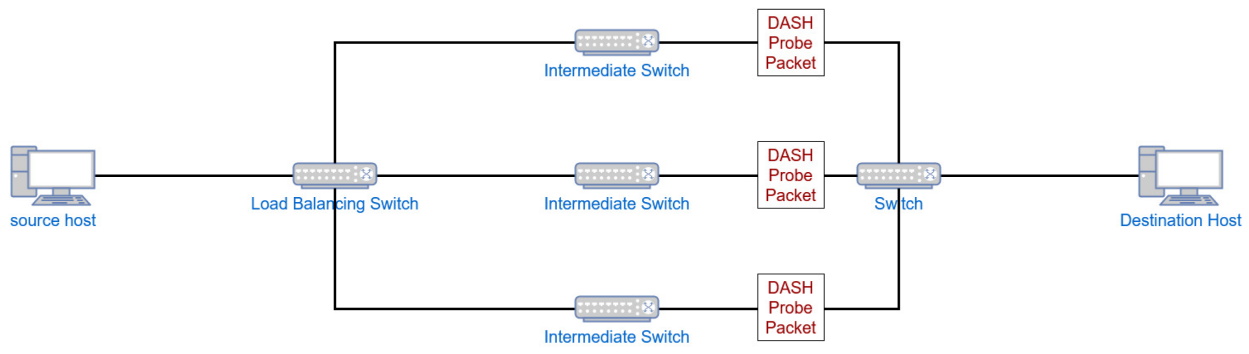

Figure 3: Sample topology: DASH Probe packet towards source host

Upon arriving all of these DASH probe packets for each period at the load balancing switch, the load balancer is informed about the load of each path. Hence, it is able to calculate a suitable boundary for each region to balance the load. Now it becomes easy to update the path boundaries with just a single packet as one could use the last probe in a given probe period to go through the stages and update corresponding boundary value (Fig.4).

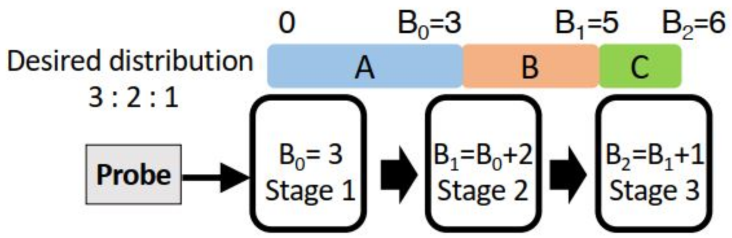

Figure 4: DASH weight update procedure

Scenario Description

The network we are going to simulate is shown in Fig.5. Consider a scenario in which h1 wants to send packets, for example UDP, to $h2$ and there are 4 possible paths from which $S1$ can choose using DASH load balancing scheme.

According to the DASH scheme, switches in the path, i.e. $S3, S4, S5$ and $S6$, update the DASH probe packets with their performance metrics. Based on these values, the load balancer, i.e. $S1$, calculates an appropriate distribution for stage boundaries regarding each path. However, due to arithmetic calculation limitations in P4, designing a suitable mapping logic is somehow challenging; therefor to sake simplicity, we assume that h2 sets the DASH probe packet values and the intermediate switches, i.e. $S3, S4, S5$ and $S6$, **don't change it at all** and $S1$ sets the boundaries using these values without using any algorithms to calculate a suitable boundary distribution. Note that we should generate appropriate DASH probe packets in $h2$ in order to correctly set the boundaries in $S1$.

Problem 1 continued on next page. . .

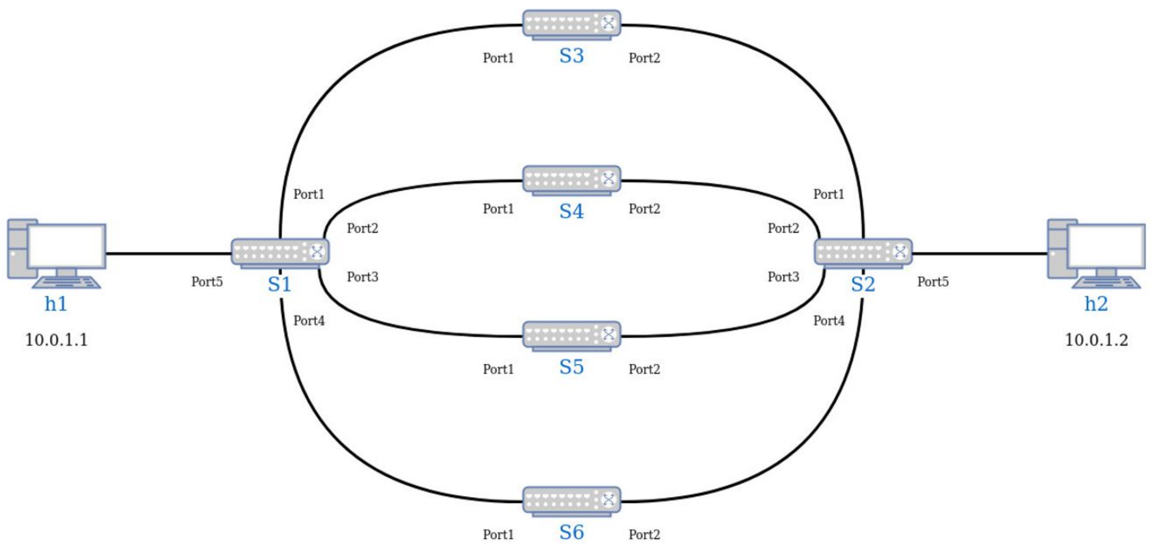

Figure 5: Our network

Consider a fixed UDP flow from $h1$ to $h2$, meanwhile DASH probe packets are sent from $h2$ to $h1$ in each possible path in four rounds. (In each round four DASH probe packets are sent in total, one for each path) The weight of these DASH Probe packets are shown in the table below. In this simple scenario between the first and second round, all of the UDP packets originating from h, would be forwarded via $S3$, whereas $S4$ is responsible for packet forwarding between the second and the third round, accordingly.

Path		S3	S4	S5	S6
Weight	Round 1	30	0	0	0
	Round 2	0	30	0	0
	Round 3	0	0	30	0
	Round 4	0	0	0	30

Functionalities

- **DASH Probe Packets**
  - As already denoted, according to our simplified scheme, weight header in the DASH Probe packets are set by the destination and remain unchanged through intermediate switches.
  - In order to route from each possible route, use ”Source routing” scheme.
- **Intermediate Switch**
  - Here, intermediate switches just relay both DASH Probe packets and UDP Packets.
- **Load Balancing Switch**
  - This carries out the main functionality of balancing as described above.

Needed Materials

- **dash.p4**
  - We have provided you a skeleton for the code. All parts except "Ingress Processing" and "Headers" are completed and you are assigned to complete these two parts and leave other parts unchanged. Don't change the code, instead add your code to the file in specified places. Please refer to the comments in the file for further details.
  - Despite the difference between switch functionalities, we use a single P4 code for each of them intentionally. You are assigned to differentiate between them by designing appropriate tables and appropriates rules for these tables.
- **sx-runtime.json**
  - Add table rules for each of six switches in these files.
- **topology.json**
  - This is an ultimate topology and you should not change it.
- **src.py**
  - This script generates a UDP packet every 0.5 seconds. Run this in $h1$ to test your work.
- **dst.py**
  - This script generates mentioned DASH probe packets. Run this in $h2$ to test your work.
- **Makefile**
  - Here we use utilities provided by https://github.com/p4lang/tutorials. So, in order to run your code place it under tutorials/exercises folder and use the command *make run*. Furthermore, you can use *make clean* to clean up the run-time files.

Deliverables

You are assigned to complete dash.p4 file as well as sx-runtime.json files. Please note that one can easily evaluate the work using Wireshark to capture the packets on the switches' interfaces.

Problem 2

The objective of this problem is to simulate an *abstracted* Quality of Service (QoS) model for Long Term Evolution (LTE) mobile networks according to 3GPP standard. In LTE mobile broadband networks, Depending on whether the service is a voice call using Voice-over-IP, a video streaming service, a file download, a chat application, etc., the QoS requirements for the IP packet transport are different. The services have different requirements on bit rates, delay, jitter, etc.
“EPS bearer” (also denoted as “bearer” for simplicity) is a central concept in LTE for providing the IP connection as such and for enabling QoS. In fact bearer provides a logical transport channel between the UE and the PDN for transporting IP traffic. Each bearer is associated with a set of QoS parameters that describe the properties of the transport channel, for example bit rates, delay and bit error rate, scheduling policy in the radio base station, etc. All conforming traffic sent over the same EPS bearer will receive the same QoS treatment. In order to provide different QoS treatment to two IP packet flows, they need to be sent over different bearers (Fig.6).
There are two different types of bearers in the LTE standard, namely, default and dedicated bearers. Default bearer is established when UE is initially attached to the LTE network while dedicated bearer is only established when there is need to provide QoS to specific service like VoIP, video etc.

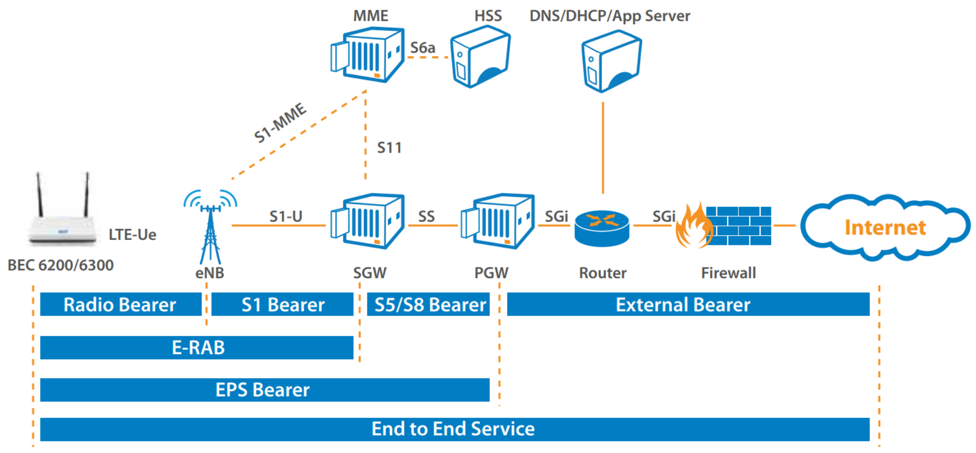

Figure 6: Quality of Service in the LTE Standard

Part 1- Abstracted Bearer Establishment

Assuming a simplified model, here we consider the dedicated bearer as a tunneling protocol which encapsulate packets with a unique identifier and then assign each ID a dedicated service with specific parameters. Hence, the objective of this part is to define an innovative L3 encapsulation to assign users different service parameters based on associated protocol headers.

Deliverables:

- A P4 code for a software switch that parses packets and extract their headers (Especially the header of the new protocol) defined in the new format.
- A JSON file that creates the topology described in Fig.7.

Problem 2 continued on next page. . .

- Two Python scripts, one for the sender the receiver, respectively. These codes use Scapy (a packet manipulation tool for computer networks) to create packets in the new format compliant with your bearer implementation approach.
- JSON files, one for each switch to add flows.

Please note that your protocol should include a Unique Identifier (UID) field together with other fields if necessary. Moreover, packets have to be routed with the new header based on their UIDs not Ethernet and/or IP address.

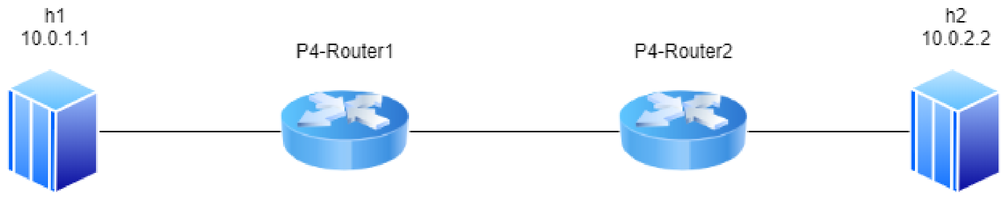

Figure 7: Network topology for Part 1 and 2

Part 2- Dedicated and Default Bearer Co-existence

Consider IP-based routing protocol as the default bearer while the implemented tunneling protocol in part 1 as the dedicated bearer, run the following scenario:

1. The sender should send a hello message with IP protocol that has a payload in a form of: "hello message, <sender-IP>".
2. The receiver has to responses with an IP packet with payload: "answer message, Hash <sender-IP>".
3. Sender uses this hash in the response massage as the unique identifier for dedicated bearer and sends a packet applying the protocol implemented in part 1.

Deliverables:

- Modified send.py and receive.py. Bear in mind to add appropriate flows before running send.py and receive.py.

Part 3- Service Differentiation

Fig.8 demonstrates a tree network where the root of the tree refers to packet date network gateway (P-GW) in the Evolved Packet Core. The serving gateways (S-GW) and the eNodeBs are connected to the leaves and hosts, respectively.
The objective is to develop a Python script, controller.py, running in two different phases as follows to add flows to the switches.

- **Initialization phase:**
  1. Run receive.py (modify if needed) on the root node connected to the packet date network gateway.
  2. Install IP flows on the switches for routing default bearer (IP protocol) to the packet data network gateway.
  3. Run send.py (modify it if needed) on the terminal of each host that are demonstrate as eNodeBs in Fig.8 .

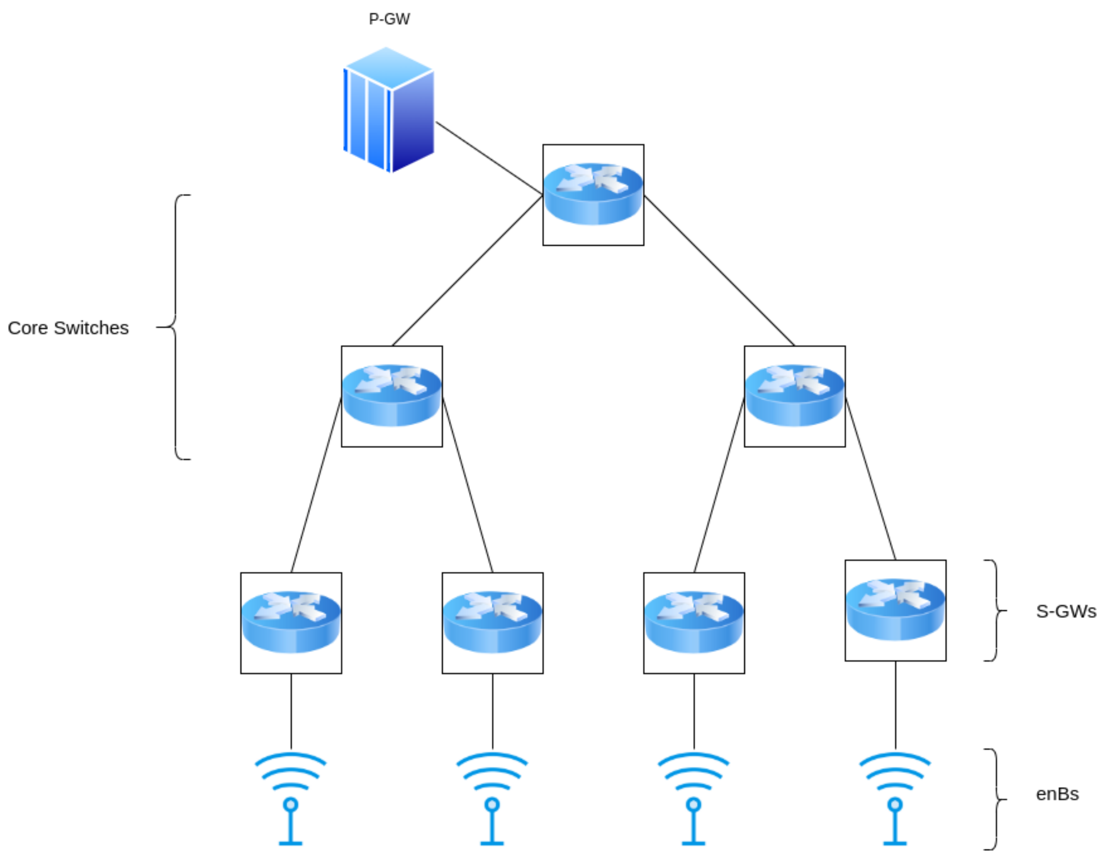

Figure 8: LTE Network

4. Receiver that is running on the root host should run handshake scenario described in the part 2 and create text file (.txt) containing the UIDs and associated arbitrary service type for each UID.

- **Secondary phase:**

  1. The controller in this phase should read the text file created by receiver.py at the previous phase and properly install the flows for dedicated bearers on the switches.
  2. eNodeBs send packets through dedicated bearers to the root host connected to P-GW.

  It is important to note that the packets should be routed based on their UIDs and the corresponding service class, hence, distinguishing the service classes in the switch pipelines in an appropriate manner is highly demanded.

Deliverables:

- P4 code for the switch which parses packets defined in the new format, processes, and routes each packet based on its class of service (in the way you choose to implement QoS).
- Modified send.py running on eNodeBs and receive.py running on the P-GW.
- A python script, Controller.py which applies to P4-runtime to implement the control policies according to the scenario.
- JSON file creates the topology described in Fig.8.

Problem 2 continued on next page. . .

Part 4- Counter Implementation

In this part you have to add a counter to your switch pipelines to count the packets based on their type of services and UIDs.

Deliverables:

- P4 code, modified to count packets based on UIDs and service classes.
- controller.py, modified to fetch counters information from the switch.