Performance Evaluation Report

Stingescu Andrei Petrut 342C2

Which parts of the assignment were completed:

I. Prerequisites
II. Evaluation - System Limits Analysis
III. Implementation
IV. Documentation

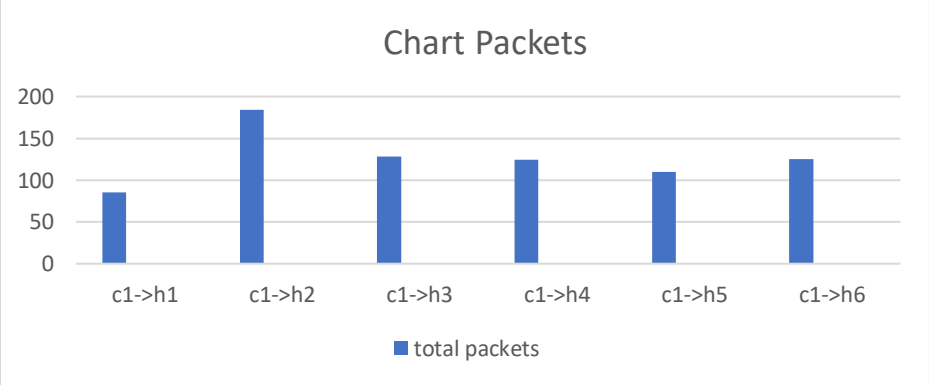
What grade do you consider that your assignment should receive?

I resolved all parts of this assignment 😊

II. Evaluation - System Limits Analysis

1) For this questions, I used the command **c1** ping -s 1000 -f **[hx]** to each worker and I tested the maximum load of requests each host can handle. For this purpose, I used 1024 byte packets and I observed the moment when the machine responded with the first error message.

	c1→ h1	**c1→ h2**	**c1→ h3**	**c1→ h4**	**c1→ h5**	**c1→ h6**
Total packets	**85**	**184**	**128**	**124**	**110**	**125**



Observation: If I tried to send again a flood ping to each worker, the values will be changed and the total packets will vary around **(300 – 400)** packets.

2) I tried firstly to verify the connectivity between the components of the network, using this command in mininet terminal **[rx]** ping -c 100 -q **[hx]**. I used 100 packets to receive certain values:)After using this command we can figure out the package loss and the delay for each route. The second packet taked a little time than the first packet, because a flow entry covering ICMP ping traffic was previously installed in the switches, so no control traffic was generated, and the packets immediately pass through the switches. I tried to send a single package in the routes to avoid that huge time and after use the **[rx]** ping -c 100 -q **[hx]**. The delay is (**avg time value**) / 2, because **avg time** is the time it took a packet to reach the destination and come back to the source. The latencies per region will be obtained using the average value of the assigned two workers of that region.

	r1 → h1	r1 → h2	r2 → h3	r2 → h4	r3 → h5	r3 → h6
packet loss	**6%**	**6%**	**3%**	**4%**	**2%**	**6%**
latency(ms)	**26.6705**	**20.658**	**24.458**	**11.4605**	**16.8585**	**25.2975**

Latency(ASIA) = [Latency(r1 → h1) + Latency(r1 → h2)] / 2 = 23.66425 ms
Latency(EMEA) = [Latency(r2 → h3) + Latency(r2 → h4)] / 2 = 17.96 ms
Latency(US) = [Latency(r3 → h5) + Latency(r3 → h6)] / 2 = 21.078 ms

3) I tested the connectivity between ***command unit*** and each ***worker***, using the command in mininet terminal: c1 ping -c 100 -q **[hx]*(hx: h1, h2, h3, h4, h5, h6)***. This command is useful for analyzing the packet loss and the latency, but these values can sometimes oscillate.

	c1→ h1	c1→ h2	c1→ h3	c1→ h4	c1→ h5	c1→ h6
packet loss	**7%**	**5%**	**5%**	**9%**	**6%**	**10%**
latency(ms)	**29.7865**	**23.9725**	**33.835**	**21.52**	**18.8285**	**26.976**

I calculated the latencies from **c1 →(h1, h2, h3, h4, h5, h6)**:
The server with the smallest response time is **c1→ h5(18.8285 ms)**
The server with the slowest response time is **c1→ h3(33.835 ms)**
4) The path that has the greatest loss percentage is **c1→ h6(10% package loss)**
5) I need to test the connectivity of r0 with each devices which is binded**(c1 r1 r2 r3)**
packet loss: **0%** || delay: **7.397 / 2 = 3.6985 ms (r0→r1)**
packet loss: **1%** || delay: **21.554 / 2 = 10.777 ms (r0→r2)**
packet loss: **2%** || delay: **6.323 / 2 = 3.1615 ms (r0→r3)**
packet loss: **0%** || delay: **0.162 / 2 = 0.081 ms (r0→c1)**
After several runs in which I analysed the latencies provided from the router r0, I came to the conclusion that the introduced latency is:
The latency value for r0 → c1 is to small and can be skipped(= 0)
Latency(r0) =[Latency(r0→r1) + Latency(r0→r2) + Latency(r0→r3)] / 3 = 5.879 ms
6) I think the bottleneck is caused by the difference in the bandwidth. The bandwidth is calculated as the size of packet divided by average latency. For **c1 → r0,** the bandwidth is :
551,72 bytes/ms, for **r0 → r1:** 17.30 bytes/ms**, r0 → r2:** 5.93 bytes/ms**, r0 → r3:** 20.24 bytes/ms and for **r1 → h1:** 2.399 bytes/ms**, r1 → h2:** 3.09 bytes/ms**, r2 → h3:** 2.616 bytes/ms**,**
r2 → h4: 5.584 bytes/ms**, r3 → h5:** 3.796 bytes/ms**, r3 → h6:** 2.529 bytes/ms
As you can see, the value of the maximum bandwidth has been shown in the **c1 → r0,** after that, while the flow of data parse the topology to the hosts, the follow links have decreasing values. A good solution for fixing this bottleneck si to update the infrastructure of the topology in order to have an increased bandwidth in all links
7) The estimation for the latency is calculated as the average value of the all latencies from the **command unit** to all hosts of the topology. I used the values obtained from the table 😉
Latency(total) = 25.81975 ms
8) The downsides of this architecture are given by the weak links between the components of the topology. The poor performance was analyzed from the huge percentage of loss packets in the chosen routes**(5–10%)**, the unequally distributed bandwidth over the network and the lantencies

III. Implementation

In this part, I implemented 5 policies, one of them is implemented using threads, three of them are implemented asynchronously and the last one is serial.
If you want to run the client.py to test the policies, first you need to enter in mininet and run the command in his terminal: **c1 python3 client.py -p http -n 100, -n** specify the number of requests. After that, you have to choose the policy specified by the number in the console.

First Policy:
This policy sends asynchronous requests to random machines. In this case, It will generate a random index from the list where the URL of the host can be found. The correct address where I have to send http requests is obtained as the current index from the loop divided by number of hosts. The requests will be sent using coroutines and in the end it will display the total time. This policy can handle a total of 300 – 400 requests.
Second Policy:
In this policy, I implemented the Robin Round algorithm asynchronously, using aiohttp and async. In this algorithm, I used a queue in which all URLs of hosts were inserted. This policy proved to be almost the fastest in terms of execution time, but it can handle a total of 350 – 500 requests.
Third Policy:
In this case, I just remake the first policy using an method to send synchronous http requests
Forth Policy:
This policy is implemented using threads, the number of requests is splitted into the chunks of requests and every chunk is assigned to each host. It will be 6 threads and every thread will have an URL of assigned host. The policy is very efficient if you want to send a huge number of requests, for example, I could send 2000 requests but it takes 20 min :(
Fifth Policy:
In the final policy, I used the Robin Round algorithm, sending requests http synchronous. The correct address policy where I have to send http requests is obtained like in the previous one as the current index from the loop divided by number of hosts(I used the provided function http_get). This policy can handle 200 – 300 requests but it is slowest than the other solution with aiohttp and async