Introduction to Computer Networks - Programming Assignment 1

Anastasia Riana 邱慧莉 - 109006234

## Part 1. UDP Pinger

The first part of the program, we need to declare a few things. Firstly, we need to import both *time* and *socket* modules. *Time* module provides users with time-related functions and *socket* module provides Berkeley Software Distribution (BSD) socket interface for Python. BSD is usually configured for internet, web hosting and for hosting multiple servers on one system. Next, we need to define our remote IP address and port in a tuple data type.

In line 5, we are specifically creating a server socket, denoted by **AF_INET. SOCK_DGRAM** indicates that we are using IP4/IP6 as our UDP socket. Then, we define a few variables. In line 8, **count** will be our ping counter. **lossCount** variable is to count how many packets were lost during the transmission. Lastly, we define **rttList** to append all RTT time which will become useful in calculating RTT's average.

As the program requires us to ping the server for ten times, we use a for-loop. Next, we write a message which is "PING (1-10)". Then, we move forward to count how much time needed to send the message. In line 15, we set the starting time by utilizing the *time* module. In the next line, we send the message to the address and its port. As the first parameter only takes bytes data type, we can use **encode()** to convert string into bytes. Next, we need to read information from the UDP socket, which is the message and IP address. As the messages and IP address are the same, we just overwrite it with the same variable. In addition to that, we set the number of bytes to be read to be 1024 bytes. Then, we end the time count by storing the current time to the **end** variable.

Then, we can count the RTT time by subtracting the end and **start** variable and store it to the **elapsed** variable. Then, as the value of **elapsed** might have multiple decimal places, we round the value to three decimal places and store it to **rttTime**. Next, based on line 24, we print out both the m

essage and the value of **rttTime**. Lastly, we append **rttTime** to **rttList** to make it easier for us to compute average RTT. We must not forget to increment the **count** variable, that is our ping counter. In case a packet is lost in transmission, we print "Request timed out." and we increment both **lossCount** and **count** variables.

In the last part of the code, we are going to print the average RTT and packet loss rate. The way we can count packet loss rate is to divide the **lossCount** with 10 as we have 10 pings. To calculate the average RTT, we take the sum of **rttList** and divide it with the length of **rttList**. Lastly, we must not forget to close the socket that we made earlier.

Result:

```
riana@riana-GF63-Thin-9RCX:~/Documents/Coding$ sudo python3 UDPPingerClient.py
[sudo] password for riana:
PING 1 1.804
PING 2 1.156
Request time out.
Request time out.
PING 5 1.355
Request time out.
Request time out.
PING 8 8.213
PING 9 1.194
PING 10 1.182
Result:
Average RTT 2.4839999999999995
Packet loss rate 0.4
```

## Part 2. ICMP Messages

The first part of the program, we need to declare a few things. Firstly, we need to import both ***time*** and ***socket*** modules, which have the same function as Part 1. Then, we bind both the destination IP address and the port. After that's done, we make a socket to send messages using the UDP, denoted by **AF_INET** and **SOCK_DGRAM**, which indicates that we are using IP4/IP6. It is also required by the client specification to set the timeout time to 1 second. Next, we need another socket to receive if there are any ICMP replies, so we declare it, denoted by **AF_INET, SOCK_DGRAM,** and **IPPROTO_ICMP**, which is a protocol that allows us to create ICMP packets. Like the client specification, we set the ICMP socket to time-out after 1 second. Then, we create an additional thread, which means that thread will have two things happening at the same time. In this thread, we will send a packet through a U

DP server while also listening if there's ICMP replies, through the **check_icmp** function. Then, we start opening the thread.

Next, we want to send a message ten times and check if there's any ICMP packets. We don't have any specific message to send to the UDP server, therefore, we just send an empty string to the address. Then, we have to check if we receive any ICMP replies with **recvfrom()** and parse the return value to **icmp,** that is some data received and **address.** We can be sure that address and **address port** hold the same value. Then, we slice the **icmp** variable to another variable that's **icmpHeader,** from the twentieth to the twenty-eighth item. Then, we need to unpack the ICMP header, to its specified functions. To use **struct.unpack()**, we need to have a format character to convert between C language and Python values. In this case, we use **'bbHHh'** in which b means conversion from signed char type to int, H means unsigned short type to int and h means short type to int. We unpack the ICMP header to several important datas, such as **type, code, checksum, packetID, sequence.** As we only need **type** and **code** variables, we can delete the remaining variables to save some memory. In the condition of a timeout, we continue.

Now, we are going to make a few if-statements to show a message of what ICMP type that is received. For example: type 0 means echo reply, type 3 means destination unreachable, type 4 means source quench and type 5 means redirect. Then, we print the output according to client specifications. Then, we close the thread with the join function from the ***threading*** library. Lastly, we cannot forget to close both the

**Result:**

```
riana@riana-GF63-Thin-9RCX:~/Documents/Coding$ sudo python3 UDPClient.py
[sudo] password for riana:
ICMP Info: type=3 code=3, message: destination port unreachable.
ICMP Info: type=3 code=3, message: destination port unreachable.
ICMP Info: type=3 code=3, message: destination port unreachable.
ICMP Info: type=3 code=3, message: destination port unreachable.
ICMP Info: type=3 code=3, message: destination port unreachable.
ICMP Info: type=3 code=3, message: destination port unreachable.
ICMP Info: type=3 code=3, message: destination port unreachable.
ICMP Info: type=3 code=3, message: destination port unreachable.
```

## Part 3. UDP Traceroute

The first part of the program, we need to declare a few things. Firstly, we need to import both **time**, **struct** and **socket** modules, which have the same function as Part 1 and 2. In addition to that, we import an additional

module for our data structure for printing the output, which is the **collection** module. Next, we declare the destination address and the message that we want to send as a global variable.

Next, we need to try to listen if there's any ICMP messages, called ***listenICMP***. We create a function to do just that. First, we start the time and we try to retrieve anything from the ICMP socket, like packets and addresses. Then, we need to check a few conditions, such as if it's TTL or destination unreachable, using the ***check_info*** function. We first check if the 20th bit is 3, then destination is unreachable and if the 20th bit is 11, then it's TTL. If it's not TTL and if the destination is reachable, then we continue. Next, we also check if the port we are using is the correct one, since it is not mentioned that the port is in what specific number, we assume that it's using port 1.

After listening to the ICMP messages, we try to trace for 1 time using the ***trace*** function. In this function, we need to check again if we're using the right port, using the ***checkPort*** function. We also initialize both the UDP and ICMP socket for us to use. We create a UDP socket, indicated by the function ***socket*** with **AF_INET** and **SOCK_DGRAM** protocol, same as the Part 1. Then, because we don't want to make the packet ourselves, we use the **setsockopt**. Then, we create an ICMP socket. Just like in Part 2, we use the **socket** function with **AF_INET, SOCK_RAW** and **IPPROTO_ICMP** protocols. In addition to that, we set the ICMP socket to have a timeout of 1 second, with **settimeout**. Then, we iterate through the next port and we start the time. Using the UDP socket, we send a message to the IP address and its destination port. Then, we call the function ***listenICMP*** which returns an address. Then, we end the time count. We use another variable to store the subtraction of the end time and start time. Then, we set this function to return an address and elapsed time in a tuple data type. Finally, if both the UDP and ICMP socket have some values after the traceroute is done, we close both sockets respectively.

Next, we will try to do the traceroute, using the ***traceroute*** function. This function also lets us retry TTL time 3x in each router visited. In this function, we create an empty list to store our values. Then inside a for loop, which loops for 3 times, we call the ***trace*** function. ***Trace*** function will return an IP address and TTL time. We then append these two values to the empty list. Before appending, we make sure to already call a function called ***collect_ttl*** to make it easier for us to print the output required. In case of a time out, we will still append '*' and None to the empty list. Finally, the ***traceroute*** function will return the list that is appended with TTL values and addresses.

Next, we create a function called ***output*** to print the required values. We first initiate a data structure that is a defaultdict. A defaultdict works exactly like a normal dict, but it is initialized with a function that takes no arguments and provides the default value for a nonexistent key. We then iterate through the list given in the parameter and append items to the defaultdict.

Again, we iterate through the items of the defaultdict to extract all needed information. We try to extract the IP address and check if the IP address has an available domain name. Then, we check if we received 3 messages. If not, we need to print '*' after the sequence number. Other than that, we also need to check if we don't receive anything. In this case, if the first item of the time list is False, it means that we don't receive anything. If this happens, we need to print 'Request timed out.' If everything is fine, then we print out the sequence, domain name (if there's any), IP address, and lastly all 3 TTL times.

Lastly, we initiate our main driver. We use a for loop that loops until 30 which is our maximum hops. We call the ***traceroute*** function which returns a list. That list will be used to print out the desired output. We break the loop if the address is the same as the destination address.

Result:

```
riana@riana-GF63-Thin-9RCX:~/Documents/Coding$ sudo python3 traceroute.py
1 XiaoQiang (192.168.31.1) 4.191 ms 23.695 ms 0.829 ms
2 140.114.204.254 (140.114.204.254) 1.639 ms 1.524 ms 1.489 ms
3 140.114.3.117 (140.114.3.117) 1.618 ms 1.490 ms 1.595 ms
4 140.114.1.29 (140.114.1.29) 1.616 ms 1.582 ms 1.481 ms
5 140.114.2.141 (140.114.2.141) 1.729 ms 1.372 ms 1.448 ms
6 tesla.cs.nthu.edu.tw (140.114.89.43) 1.656 ms 1.284 ms 1.270 ms
```

Problem Faced

The problem I faced when I did this homework was getting used to Python again after a year of coding in C++ language. However, after having a good understanding of programming by now, adjusting to Python's syntax was not that difficult as it is way simpler and more human-friendly compared to C++. Using Python, I had to readjust some things that I usually have to take care of when using C++ to code, such as memory allocation, variable declaration, etc. Other than that, as Python has so many libraries that we can utilize, I had to spend a lot of time researching what libraries I would need to use.