CS331 Project 4 Write Up – Ben Webb

The purpose of this project was to implement a UDP-based ping program using socket programming in python. The task was to create the client side of the socket interface with the server side being provided.

This socket connection is different than previous exploration with socket programming because UDP connections do not have prior arranged connections, i.e. there is never a connection call made and the server is not listening. The implementation for the client side was to create a UDP socket:

```
clientSocket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM
```

SOCK_DGRAM is the connection type for UDP, which is datagrams. The client then enters a loop for as long as many pings as is specified. I implemented this with both a for and an infinite while loop, for a finite and infinite number of pings to be sent. Because UDP is an unreliable protocol, we were instructed to have a timeout of 55 ms. This was done by setting the socket timeout which throws an exception if the timeout for a message to be returned is exceeded. This was how I processed losing UDP datagrams in a lossy network. The code to implement this is seen below.

```
clientSocket.settimeout(0.055)
while:
 try:
       message, address = clientSocket.recvfrom(8000)
 except socket.timeout:
```

This allowed for events to be controlled based on how long they took. This is important because the recvfrom function of a socket idles until a message is received. If the message is lost (the timeout is exceeded), an exception occurs, and this is printed to the terminal:

```
56 bytes lost: icmp_seq=2 ttl=55
```

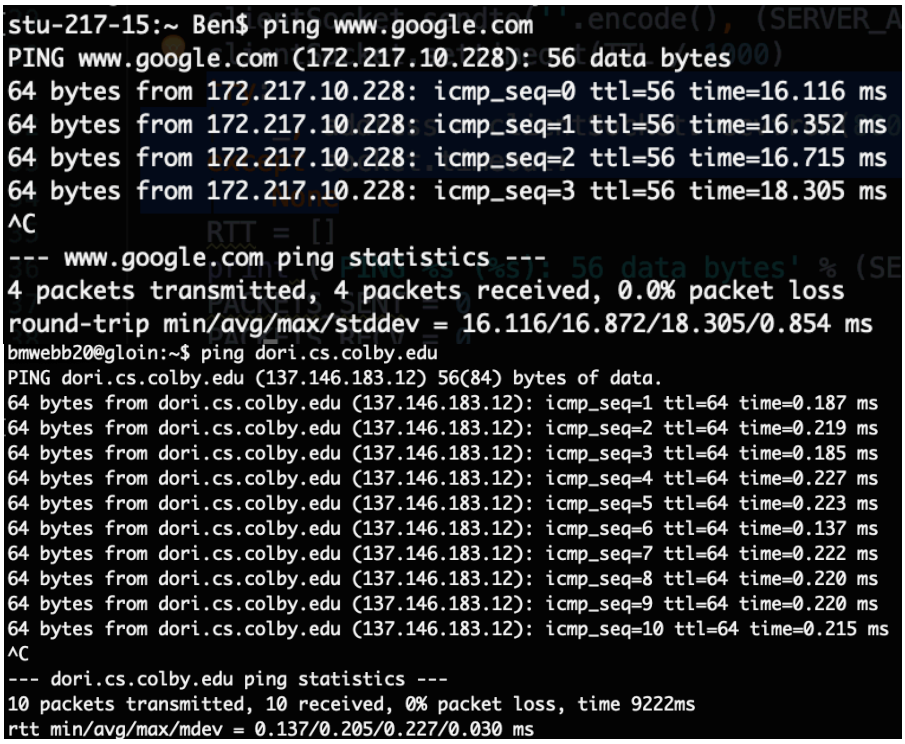
Depending on which side of the try/except statement that the packet was, the RTT was either recorded in a list or dropped. After sending 30 packets, the program then returns the statistics of the ping program. Then using built in functions and the statistics package, the min, average, max, and stdev of RTT list was printed. An example of such is seen below:

```
30 packets transmitted, 28 packets received, 6.7% packet loss, time 10176.615 ms
round-trip min/avg/max/stddev = 30.349/33.637/40.098/2.235 ms
```

Extensions

My overall goal for my extensions was to make my ping client similar to the ping program built into the command line. This was done by doing a couple of things. The first was I made sure that my print statements echoed that of the ping statement. This meant that I lied a little bit about what a was actually transmitted, specifically the size of the packet being sent and

the type of sequence (we did not send ICMP ECHO_REQUESTS). I did this by capturing what the output from the PING program looks like:



The second screenshot is from running ping from one of the dwarfs to the other; while the first is pinging www.google.com from my computer. The dwarves have a slightly different ping program from the one that is installed directly on my computer. The two differences are the dwarves includes the IPv4 hostname for each ping and the total time spent sending the pings.

To make my code mimic this, I send an initial ping to the server to get the IP address of the server. This is done by using a try/except statement outside of my while loop. That allows me to print the first line of the ping statement that is outside of the loop:

```
PING localhost (127.0.0.1): 56 data bytes
```

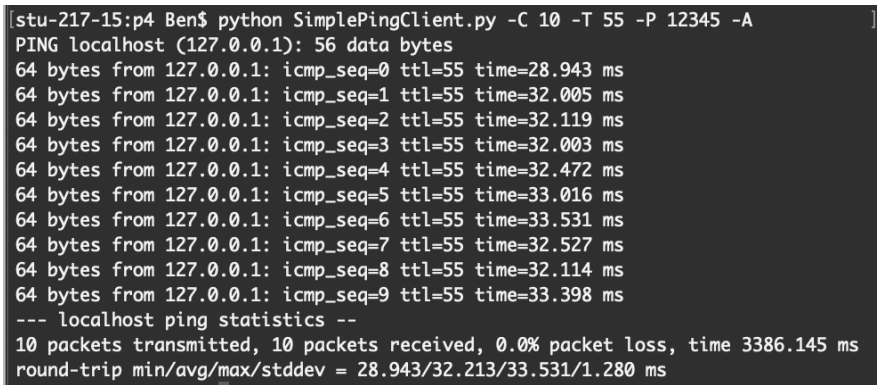
The next part was to make each ping response appear the same as the ping statement. The print statement to implement each ping response was:

```
print '64 bytes from %s: icmp_seq=%i ttl=%i time=%.3f ms' %
(address[0], PACKETS_SENT, TTL, RTT[PACKETS_SENT-PACKETS_LOST]))
```

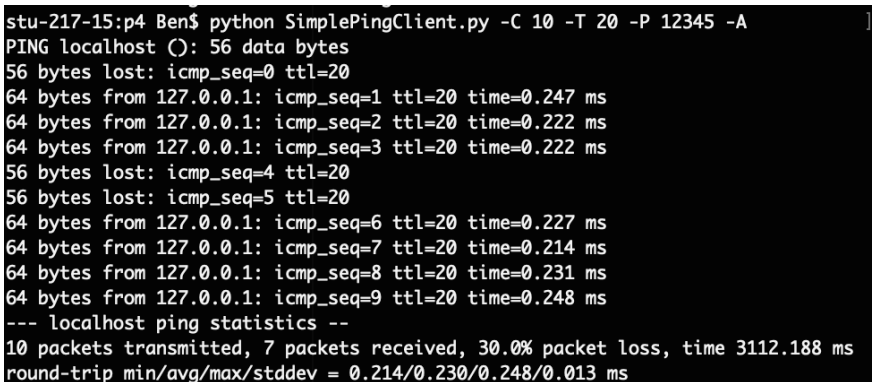
Where address is the port/addr from the UDP datagram, PACKETS_SENT is the total number of packets sent, TTL is timeout, and RTT[PACKETS_SENT-PACKETS_LOST])) is the most recent received packet's round trip time accessed from a list.

The actual ping program exists in an infinite loop and exits when the user presses ^C. I added compatibility for this by putting the loop for each ping inside another try/except statement looking for the KeyboardInterrupt exception. After the exception is reached, or if there is a finite number of pings to be send, the final statistics from the program are printed to the terminal.

I also went through and added some flags that are standard in the ping program. The first one I added is -T, this sets the time to live in milliseconds, the default is 55 ms. The second one I added is -A, this is for verbose output, in ping this prints any lost packets which are rare, but our server is very lossy and this is important to have to print the lost packets. The third flag I added was -P which allows for the client to set the server port. The final flag I added was -C which is the maximum number of pings to send. If -C is not set, the maximum number of pings is defaulted to infinity. Below is the full output of the program with all flags set to the values that were specified in the project.



This is a hybrid of the two ping programs that are on the dwarf and my computer as seen in the screenshots in the start of my extensions. I also decided to play around a little bit with the time to live. Specifically, I set it to be below the standard response time from the server (~30 ms). Below is the output:



What is interesting is that the round-trip time is much smaller when the TTL is lowered. This is a product of how I calculated the RTT. Because each UDP packet does not contain which

number it is in sequence, the packets are assumed to be received in order. But because the timeout occurs faster than the server can send a response, two datagrams are sent to the server before (40ms) a response is sent from the server. That means that the first packet is always going to be lost, and the second packet will be received nearly instantly, because the client perceives the response to the first packet as the response to the second packet which was the one it just sent. That error is propagated by the delay timer which is used to ensure that not all packets are sent at too fast. When I first added the delay, it was set to 30 ms. The RTT is actually the RTT of sending a packet, the packet timing out, the program sleeping for 30ms, sending another ping, and then receiving a response to the prior datagram. When I set the sleep time to 50 ms, all packets were lost because the response was sent before a second packet was sent. The output of this can be seen below:

```
[stu-217-15:p4 Ben$ python SimplePingClient.py -C 10 -T 20 -P 12345 -A          ]
PING localhost (): 56 data bytes
56 bytes lost: icmp_seq=0 ttl=20
56 bytes lost: icmp_seq=1 ttl=20
56 bytes lost: icmp_seq=2 ttl=20
56 bytes lost: icmp_seq=3 ttl=20
56 bytes lost: icmp_seq=4 ttl=20
56 bytes lost: icmp_seq=5 ttl=20
56 bytes lost: icmp_seq=6 ttl=20
56 bytes lost: icmp_seq=7 ttl=20
56 bytes lost: icmp_seq=8 ttl=20
56 bytes lost: icmp_seq=9 ttl=20
--- localhost ping statistics --
10 packets transmitted, 0 packets received, 100.0% packet loss, time 5276.500 ms
```

The total length timer (5276.500 ms) can be understood by the timeouts. 5000 ms was spent sleeping, 200 ms was spent waiting for a response, and 76.5 ms was spent actually running the code.

This had me interested in the time spend idle in the actual ping program. I was also interested because when we pinged one dwarf from the other, the response time was similar to when I had the timeout set to 20 ms.

That had me interested in the time that the ping program normally spends sleeping. To find this I estimated the time the computer spent processing code from when all my timeout occurred. In my python program that was about 7.65 ms per ping (76.5 ms over 10 pings).

```
41 packets transmitted, 41 received, 0% packet loss, time 40958ms
37 packets transmitted, 37 received, 0% packet loss, time 36866ms
```

I noticed that the ping program took about as many seconds as if it were to spend one second throughout the processing of each packet, so I decided to do the same with my sleeping time. I achieved something similar using this expression: `time.sleep(1 - time.time() + sendTime - 0.00765)` which essentially determines the time sing the packet was sent and assume the processing time to attempt to make each process take just under one second.

```
10 packets transmitted, 10 packets received, 0.0% packet loss, time 9312.416 ms
```