CSE 3300 – Computer Networks and Data Communication

Professor Bing Wang

Assignment 2 – ICMP Ping Client

Nicholas Lambourne

ndl17004 – 2749404

DESCRIPTION

PROGRAM DESIGN

The program is provided as a self-contained, command-line python script. Given the basic nature of the program (and the limitations imposed by the task sheet), there was very little of the design that was left to be decided by me. The basic script was comprised of several methods that interacted with one another to, in a very simple fashion, mimic the core functionality of the GNU/Linux/DOS ICMP ping application. The dual-design format of the script allows the user to run it as a command line script (see README.txt) or as a set of individual, importable functions.

HOW IT WORKS

First, the script is run from the command line and is given a URL or IPV4 address as a command line argument. If there is no provided host IP/URL, the program will immediately exit, giving usage instructions. Next, a graceful shutdown function is imposed on a Keyboard Interrupt (SIGINT). The last step in the main program is starting the actual ping functionality to the specified host. This functionality involves (repeatedly, until deliberate interruption):

- Creating a socket: a raw socket is created and
- Sending One Ping: a dummy checksum is created and an ICMP_ECHO_REQUEST struct is created. The current timestamp is encoded. A “proper” checksum is calculated using the struct, encoded and added to the struct. When the request packet is completed, it is then sent over the previously created socket.
- Receiving One Ping: After the ping is sent, the program waits for a response using a non-blocking receive. If the time elapsed since sending exceeds one second it will stop listening and resend, assuming that the request is lost. If a response is received, it is unpacked into its various components and the validity of the response is checked (type and code equal to 0, correct ID and host address). If the response is valid, the difference between the time sent and received is returned.
- Printing the RTT: finally, the program will print out the RTT delivered by the receiveOnePing function before going on to start the request-response-print cycle again.

TRADE-OFFS

A number of trade-offs were made in the production of the code in this script, though many of these were due to limitations imposed by the assignment criteria to use the skeleton code and only:

- Not using OOP: this script was simple enough that I decided not to convert it to an OOP format for greater utility/reuse. I also thought this might contravene the instructions to only write code in the marked area.
- Not checking the checksum: We were instructed to ignore the checksum value, although normally it would be worthwhile checking the validity of the response message. In the skeleton code the checksum was not passed through to the function that we were to write, hence this was not possible without modifying code we were not instructed to.
- Checking the destination address: in addition to checking that the type and code were equal to zero and the ID number in the response was equal to the ID sent in the request, I also decided to check that the host who sent the response back had the same IPV4 address as the host that the request was sent to. This ensures that in the (admittedly unlikely) event that multiple ping instances were sent at

once to different hosts, their responses would not be mistakenly received by another instance. Depending on the complexity of the load-balancing functionality at the host, this could backfire in the instance that the host machine sending the response (for some reason) had a different IP to that of the receiving host. I gauged that the likelihood of this was remote enough to discount this possibility.

POTENTIAL EXTENSIONS

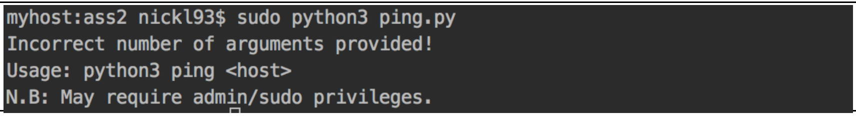
The ICMP Ping Client as provided is incredibly basic, potential extensions could include:

- Custom TTL: the standard Unix/DOS ping application allows the user to specify a maximum TTL, this could be achieved in this small script by modifying the headers before sending the ICMP request.
- Number of attempts: the standard Unix ping application allows the user to set a specific number of ping packets to be received before the program exits (the Windows version defaults to 4). Our script, like the Unix version, will continue until interrupted. We could modify the program to accept another (optional) command line argument accepting an integer for this value, which is then compared against a count of received responses.
- More comprehensive output: Both the Unix and Windows provided ping programs provide much more information in their output than ours (including the host, TTL, ICMP sequence number) while ours only provides the RTT. We have this information for each response, it would just be a matter of printing it for each.
- IPV6 Support: the script in its current form does not provide support for IPV6 addresses. In order to support them it would have to be modified to recognise IPV6 addresses at given in the command line arguments and set up an IPV6 specific connection in that case.

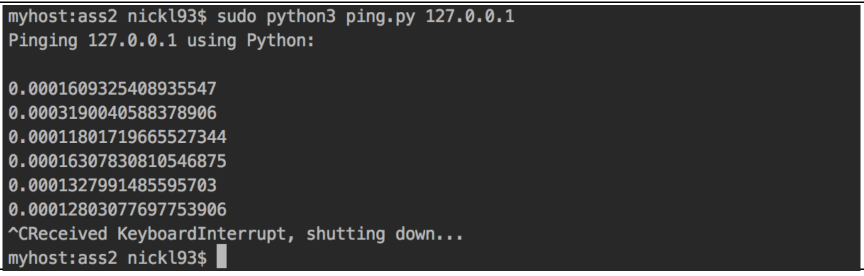
TEST CASES

Overall Design:

The test cases were designed to test common usage situations, including likely misuse. I wanted to ensure that as much of the 'regular' functionality of existing ping applications was replicated by this program, within the limitations set by the assignment criteria.

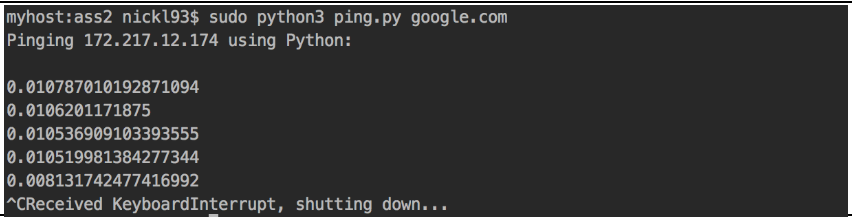
Test Case 1	
Description:	No Input (No Host Provided).
Rationale:	Likely to be many users first attempt at using the program.
Input:	sudo python3 ping.py
Output:	

```
myhost:ass2 nickl93$ sudo python3 ping.py
Incorrect number of arguments provided!
Usage: python3 ping <host>
N.B: May require admin/sudo privileges.
```

Test Case 2	
Description:	Localhost (127.0.0.1)
Rationale:	Test case requested by assignment specification.
Input:	sudo python3 ping.py 127.0.0.1
Output:	

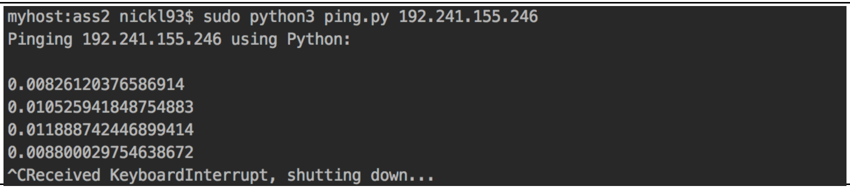
```
myhost:ass2 nickl93$ sudo python3 ping.py 127.0.0.1
Pinging 127.0.0.1 using Python:

0.0001609325408935547
0.0003190040588378906
0.00011801719665527344
0.00016307830810546875
0.0001327991485595703
0.00012803077697753906
^CReceived KeyboardInterrupt, shutting down...
myhost:ass2 nickl93$
```

Test Case 3	
Description:	Common website (google.com)
Rationale:	Ping should work for any website/IP, especially common ones.
Input:	sudo python3 ping.py google.com
Output:	

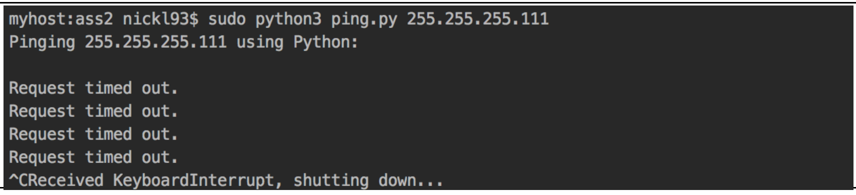
```
myhost:ass2 nickl93$ sudo python3 ping.py google.com
Pinging 172.217.12.174 using Python:

0.010787010192871094
0.0106201171875
0.010536909103393555
0.010519981384277344
0.008131742477416992
^CReceived KeyboardInterrupt, shutting down...
```

Test Case 4	
Description:	Raw IPV4 (no DNS required - 192.241.155.246 – my Jenkins build server).
Rationale:	The ping program should work for raw IPV4 addresses as well.
Input:	sudo python3 ping.py 192.241.155.246
Output:	

```
myhost:ass2 nickl93$ sudo python3 ping.py 192.241.155.246
Pinging 192.241.155.246 using Python:

0.00826120376586914
0.010525941848754883
0.011888742446899414
0.008800029754638672
^CReceived KeyboardInterrupt, shutting down...
```

Test Case 5	
Description:	Invalid IP (255.255.255.111)
Rationale:	The ping program should timeout (and continue checking) an invalid IP until interrupted.
Input:	sudo python3 ping.py 255.255.255.111
Output:	

```
myhost:ass2 nickl93$ sudo python3 ping.py 255.255.255.111
Pinging 255.255.255.111 using Python:

Request timed out.
Request timed out.
Request timed out.
Request timed out.
^CReceived KeyboardInterrupt, shutting down...
```