Coursework 1 – Network Application Development (20%)

In this coursework, you will develop several networking-based applications. These are designed to increase your competency in developing socket-based applications and increase your familiarity with several key technologies and measures. These are widely used and commonly deployed to evaluate networks and provide services.

During this practical, you will become familiar with network sockets and begin to understand how they are used. Sockets are a programming abstraction designed to assist us in building applications that use the network. We can treat these the same as any other resource; writing to a socket sends a packet into the network, whilst reading from a socket provides us with the contents of a packet. We can do this in much the same way as we would read and write to any file found on the local filesystem.

Practical Lab Structure

Practical 1 is split into two parts: Tasks 1 and 2. We also recommend that you complete the partially implemented Traceroute class before starting Task 1. For more, details, see Example Code 2 section on page 5.

For the tasks assessed, you will be awarded points for meeting certain criteria. These are outlined in more detail within each task description at the end of this document. You are encouraged to progress as far as possible with each task. However, note that Tasks 1 and 2 are independent; attempting both is advised, even if you do not fully complete each. This course uses a dedicated virtual machine image for these labs.

Running the Code

Running the code locally on your machine is also possible. To do so, you will need to install Python 3 (version 3.10 or newer is recommended). As this task does not use any external libraries (see 'Python Library Usage' section below), this should be relatively easy to achieve. Details on how to install Python can be found here:
`https://www.python.org/downloads/`
Although the same Python code will run regardless of the environment, the underlying implementation (and therefore) behaviour can be different, including for many of the

network- related libraries used in this practical. Particularly on the Windows platform (rather than MacOS or Linux), the results may be different.
Given the multitude of potential discrepancies and variances that may occur when running code locally, we will try our best to provide direct support for different systems.
Note that for marking, we encourage you to include system information and a readme file when submitting the coursework.

Provided/Skeleton Code

To help you structure your code and get started with each task, we have provided you with a skeleton code (NetworkApplications.py), which is available on the LZSCC.231 Moodle page. This file should eventually contain all the code you have developed in this lab and will be the one submitted at the end of your work. It contains a structure for the code, including some helper functions and empty classes. It also contains pseudocode for some of the tasks to get you started.
Firstly, there is a function included to parse command line arguments given to the script (`setupArgumentParser`). This includes determining which function is called (`ping, traceroute, mtroute, web, proxy`), as well as additional parameters used for each task. Some of these may be optional, particularly if they are part of the additional features associated with each task.
The code also includes the `NetworkApplication` class, which will be a parent to the classes that you will develop in the coursework. It contains some useful methods that are common across tasks. This includes the checksum function, which calculates a checksum to be used in an ICMP packet. Furthermore, it also includes two methods which are used to print out the results generated ping (see the Example Code 1 section below): `printOneResult` and `printAdditionalDetails`.
Finally, it also includes five classes: `ICMPPing`, `Traceroute`, `MultiThreaded`, `Traceroute`, `WebServer`, and `Proxy`. In the case of `ICMPPing` and `WebServer`, they include the complete code as examples for you to study. The `Traceroute` class is partially implemented; it contains a fully functional UDP traceroute, but not ICMP traceroute.
These classes inherit the `NetworkApplication` class, allowing you to use the parent's helper methods (`checksum`, `printOneResult`, `printAdditionalDetails`, and `printMultipleResults`). These can be called from the child classes. For more details on inheritance in Python, see here:
`https://docs.python.org/3/tutorial/classes.html#inheritance`
Each class contains an `__init__` method, called when the object is created (which is done once the command line arguments have been parsed, in this case). This is the starting point for writing the code in each class, but additional methods can be created as required.

Important: changing the method signature for any of these `__init__` functions will result in zero marks, so please retain all the arguments, including the args object.
To be clear: you can add new methods and code to each of the main classes: `Traceroute`, `MultiThreaded`, `Traceroute` and `Proxy` (and even to the fully-implemented classes Ping and Webserver), but the remaining structure, methods and code must remain intact.

Running your Python script

To run a Python script, open a Terminal window and navigate to the directory in which the file you want to run is located. To run the script, simply use the following command:
`python3 NetworkApplications.py`
The above will print the usage information for this program.
Please note the use of python3 rather than python: the default command refers to Python 2.7, which we are not using in this course. Using this will result in a different outcome and potential incompatibilities. When you start this task, the provided code will run successfully and print the help information but do nothing.

Python Library Usage

You are not expected to use any external libraries for this practical; doing so is strictly prohibited. All tasks can be achieved fully with the use of standard Python libraries.
We are also aware of a number of network and IP-orientated libraries that are included within the standard Python distribution. These could potentially be used in different ways to assist in your implementation. However, as we are trying to build your understanding around the fundamentals of computer networks, we ask that you do not use these for this practical, either.
The teaching team believes it is vitally important that you grasp the technical details behind many of these libraries, which do a good job of abstracting and obscuring the details. It is, of course, perfectly acceptable to use these libraries in any future software development you may do, whether this be as part of an upcoming course module or even after graduation.
By following the provided structure and guidance, you will not need to use any of these.
If you are in any doubt about whether or not you can use a particular library, please contact the course tutors to confirm.

Submission and Assessment

The submission for all work completed in this practical is due by **3 pm on December 13th (Friday, Week 10)**. All code must be included in a single file titled NetworkApplications.py and submitted on Moodle.

Example Code I: ICMP Ping

A `ping` client implementation is provided in the skeleton code. We have covered ping in week 2. Remember that ping is a tool used to measure delay and loss in computer networks. It does this by sending messages to another host. Once a message reaches that host, it is returned to the sender. We can determine the sum of nodal delays in the network by measuring the amount of Round-Trip Time (RTT) taken to receive that response. Similarly, by tracking the responses returned, we can determine if any have been lost in the network.

`ping` traditionally uses Internet Control Message Protocol (ICMP) messages to achieve this behaviour. The example code sends echo request messages (with an ICMP type code of 8). These requests are useful to us because on reaching the client, the client will respond with an echo reply message (with an ICMP type code of 0). By timing the period of time elapsed between sending the request and receiving the reply, we can accurately determine the RTT between the two hosts.
In general, you can get help with running each of the classes: traceroute, mtroute (multithreaded traceroute), webserver, and proxy by issuing a `-help` (or -h) argument in the terminal:
`python3 NetworkApplications.py ping -help`
Or
`python3 NetworkApplications.py ping -h`
which will print usage information. In the case of ping,
python3 NetworkApplications.py ping – count 5 –timeout 1 lancs.ac.uk
will send five ping probes to Lancaster university server with one second timeout on the receipt of responses. Note that the count and timeout are optional arguments and their default values are defined in `setupArgumentParser` method. A timeout on a socket defines the maximum time the socket will wait while trying to receive data before raising an exception. If data is not received within the specified timeout period, the recv and recvfrom methods will throw a socket.timeout exception. This ensures that your program does not get stuck indefinitely waiting for a response and allows you to handle network delays or packet loss more effectively. You can set the timeout using `socket.settimeout(seconds)`.
It is possible to use both `socket.SOCK_RAW` and `socket.SOCK_DGRAM` when

creating sockets. SOCK_RAW requires privileges, as it gives you a huge amount of control and power over the type and content of packets sent through it. As such, it requires sudo to work. In normal circumstances, a non-privileged version would probably be preferable; you don't always have sudo privileges on a system. This is where SOCK_DGRAM comes in. However, it appears that SOCK_DGRAM, when used to specifically to send ICMP packets, is also a privileged operation within some flavours of Linux.
For the purposes of this practical, it is therefore recommended that you use SOCK_RAW. Note: normally, using SOCK_RAW requires sudo privileges. The ICMP header contains both an identifier and a sequence number. These can be used by your application to match an echo request with its corresponding echo reply. It is also worth noting that the data included in an echo request packet will be included in its entirety within the corresponding echo reply. Use these features to your advantage.

Example Code 2: UDP Traceroute

We have also provided you with a partially implemented (a single-threaded) `Traceroute`. In particular, the UDP traceroute implementation is provided. You are encouraged to add the ICMP Traceroute implementation to make this a fully functional Traceroute tool. Please carefully go through the provided UDP implementation before you start working on the ICMP traceroute. Although, the ICMP traceroute is not marked, we strongly recommend that you work on ICMP traceroute, as your code for this part can be re-used in Task 1: `MultiThreadedTraceroute`. As discussed in week 2, traceroute measures RTT between the host and each hop along the route to a given destination. By default, this tool uses UDP messages with a Time-To-Live (TTL) value initially set to 1. This ensures we get a response from the first hop, the network device closest to the host on which we run the script. When the message arrives at this device, the TTL counter is decremented. When it reaches 0 (in this case, at the first hop), the message is returned back to the client with an ICMP type of 11. This indicates that TTL has been exceeded. As with the previous task, by measuring the time taken to receive this response, RTT can be calculated at each hop in the network. This process can be repeated, increasing the TTL each time, until we receive a destination unreachable message (with an ICMP type of 3). This tells us that we have reached the destination, so we can stop sending traceroute probes. To run Traceroute, you can check its usage by:
python3 NetworkApplications.py traceroute –help
The traceroute expects a hostname (similar to ping), and you can provide optional parameters. As an example, to run UDP traceroute to lancs.ac.uk, do:
python3 NetworkApplications.py traceroute –protocol udp lancs.ac.uk
or

python3 NetworkApplications.py traceroute -p udp lancs.ac.uk The ICMP variant of traceroute operates similarly but uses ICMP Echo Request (Type 8) messages instead of UDP packets. The Time-To-Live (TTL) value is still incremented with each probe, and when the TTL expires at each hop, the router sends back an ICMP Time Exceeded (Type 11) message. The round-trip time (RTT) is measured based on the time it takes for this response to return. Once the probe reaches the final destination, an ICMP Echo Reply (Type 0) is received, indicating that traceroute has successfully completed the path discovery. When you eventually implement ICMP traceroute, you should be able to run it by:
python3 NetworkApplications.py traceroute -p icmp lancs.ac.uk
You can run the real traceroute tool using the traceroute command on the terminal. Note that by default, traceroute uses UDP probes. To send ICMP probes, you must use the -I flag: `traceroute -I lancs.ac.uk`

Example Code 3: Web Server

Web Servers are a fundamental part of the Internet; they serve the web pages and content that we are all familiar with. You will be learning more about web servers and the operation of the HTTP protocol in Week 3: Web & HTTP Basics. Fundamentally, a web server receives an HTTP GET request for an object (usually a file) located on the web server. Once it receives this request, the web server will respond by returning this object back to the requester.
The Web Server differs from the ICMP Ping application in that it will bind to an explicit socket, identified by a port number. This allows the Web Server to listen constantly for incoming requests, responding to each in turn. HTTP traffic is usually bound for port 80, with port 8080 a frequently used alternative. For the purposes of this application, we suggest you bind to a high-numbered port above 1024; these are unprivileged sockets, which reduces the likelihood of conflict with existing running services on your machine.
For example, a request with a URI of 127.0.0.1:8080/index.html, will serve a file name index.html found in the same directory as the Python script itself. The URI is broken down as follows:

- 127.0.0.1: Hostname of web server
- 8080: Port number that web server has bound to
- index.html: File to be served

On successfully finding and loading the file, it will be sent back to client with the appropriate header. This will contain the Status-Code 200, meaning that the file has been found OK, and that it will be delivered to the client as expected. This implementation

only serves files from the same directory in which the Python script is executed. To start the WebServer, simply do:
python3 Network Applications.py web -p 8081
Then copy the following URI and paste it into the address bar of your favourite browser: 127.0.0.1:8081/index.html. You must also store an index.html file in the same folder where you run your script. A dummy index.html file is provided in Moodle.

Task 1: A multithreaded Traceroute (weight: 50% of the total mark of cw 1)

Building on example code 2, this task aims to create a multithreaded traceroute tool.

Implementation Tips

In general, you can find lots of information about the socket library documentation: `https://docs.python.org/3/library/socket.html` In this task, you will implement a multithreaded version of the traceroute tool (mtroute), which sends probes and receives responses in parallel using separate threads. A skeleton code is provided in the `MultiThreadedTraceRoute` class to help you get started. Your task is to complete the implementation of mainly the two methods: `send_probes` and `receive_responses`, using the ICMP and UDP protocols (you can also modify the __init__ method and include additional methods as you need). Your implementation must:

- Use one thread to handle the sending of traceroute probes, incrementing the TTL as described earlier.
- Use another thread to handle the reception of responses, calculating the round-trip times (RTT).
- Ensure thread safety by using locks for shared data, and use thread events to signal when sending is complete (which is already provided in the skeleton code).

Your final implementation should display the traceroute results, including the IP of each hop and the RTT, using the `printMultipleResults` method.

Debugging and Testing

As with the previous task, every host on the path to your chosen destination should respond to your probes. In reality, these probes are often filtered, including within the lab network. As a result, it is especially difficult to test this tool with a remote host. Instead, it is suggested that you initially test with a closer endpoint that is reachable: lancs.ac.uk. Although the number of hops is small, it can still be used to demonstrate the working of your application. If you run your script whilst attached to a different

network, such as at home, your results will likely differ. Once, traceroute to lancs.ac.uk works, you should also test destinations such as `google.com` that are farther away.
To run UDP multithreaded traceroute with lancs.ac.uk as the target:
python3 NetworkApplications.py mtroute -p udp lancs.ac.uk
To run the ICMP version of the multithreaded traceroute, simply update the -p option:
python3 NetworkApplications.py mtroute -p icmp lancs.ac.uk
Note that your implementation must configure a timeout period to ensure that your program does not get stuck indefinitely waiting for a response. Please see the traceroute implementation on how to configure a timeout. The threading library is imported into the skeleton code. For more details on the use of threading, please refer to: `https://docs.python.org/3/library/threading.html` Wireshark can be used to inspect the packets leaving your application. Comparing these to those created using the original traceroute utility will provide a meaningful comparison.

Marking Criteria

The marks will be awarded for ensuring:
1) A multithreaded UDP (30%) and ICMP (40%) traceroute implementation that sends a train of probe packets while simultaneously receiving and processing responses. (70%)

Note I: Your implementation will be tested on the SCC.231 VM with at least two destinations: lancs.ac.uk and another one such as google.com. I will check the packets sent by your program to determine whether your implementation is correct; that is, the packet formats, sequence number handling, TTL management, and so on are accurate.

Note II: your program must use the –protocol(or -p) positional argument to configure the protocol from the command line. The configurability is required to test both protocols.

2) Similar to the original traceroute utility, display three delay measurements for each node between your machine and the chosen destination. You are advised to use the `printMultipleResults` method in the skeleton code to print the output. (30%).

Note III: Your final submission should not print extra (e.g., debugging) information other than the traceroute output.

Task 2: A Web Proxy (weight: 50% of the total mark of cw 1)

Building on the Web Server example, this task concerns building a Web Proxy. A Proxy operates similarly to a web server, with one significant difference: once configured to use the Proxy Cache application, a client will make all requests for content via this proxy. Normally, when we make a request (without a Web Proxy), the requests travels from the host machine to the destination. The Web Server then processes the request and sends back a response message to the requesting client.
However, when we use a Web Proxy, we place this additional application between the client and the web server. Now, both the request message sent by the client, and the response message delivered by the web server, pass through the Web Proxy. In other words, the client requests the objects via the Web Proxy. The Web Proxy will forward the client's request to the web server. The web server will then generate a response message and deliver it to the proxy server, which in turn sends it to the client. The message flow is as below:

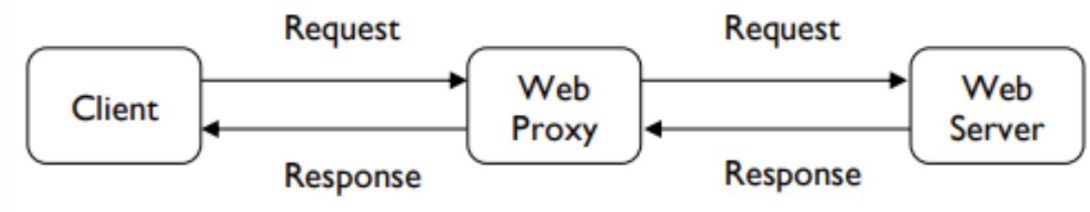

Figure 1: Caption

As with the Web Server, your Web Proxy application is only expected to handle HTTP/1.1 GET requests. Similarly, the Web Proxy will also bind to a specific port (this can be the same as the Web Server) and continue to listen on this port until stopped.
Please ensure that all code for this task is included in the `Proxy` class.

Debugging and Testing

As with the webserver, there are a number of ways to test your Web Proxy. For example, to generate requests using curl, we can use the following:
curl neverssl.com –proxy 127.0.0.1:8000
This assumes that the Web Proxy is running on the local machine and bound to port 8000. In this case, the URL requested from the proxy is neverssl.com.
A caveat when testing your Web Proxy: some websites have enabled HTTP Strict Transport Security (HSTS) (RFC6797). This forces clients (including both curl and a web browser) to use HTTPS rather than HTTP. HTTPS is a secure version of HTTP, but we will consider this out of the scope of this practical. Thanks to the proliferation of HTTPS (this is a good thing, just not for this practical!) the list of live websites you can test the Proxy with is quite limited. A few include:

- `http://neverssl.com` (see above)
- `http://captive.apple.com`

It is also possible to run a webserver locally on your machine and test it from that. This could be the web server implementation provided in the code. Alternatively, Python also has a simple implementation that can be run directly from the terminal: `https://docs.python.org/3/library/http.server.html`
As with the other tasks, Wireshark can be used to capture and investigate packets sent to and from your proxy.

Marking Criteria

For this task, marks will be awarded:

1) A correctly implemented Web Proxy. You are expected to test the functionality of your Proxy using `curl`, as shown above. Note that you are not expected to use a website with HSTS enabled (see above). (60%)

2) Binding the Web Proxy to a configurable port, defined as an optional argument (use port positional argument). (10%)

3) Object caching: A typical Web Proxy will cache the web pages each time the client makes a particular request for the first time. The basic functionality of caching works as follows. When the proxy gets a request, it checks if the requested object is cached, and if yes, it returns the object from the cache, without contacting the server. If the object is not cached, the proxy retrieves the object from the server, returns it to the client and caches a copy for future requests. In practice, the proxy server must verify that the cached responses are still valid and that they are the correct responses to the client's requests. You can read more about caching and its handling in HTTP in RFC2068. Add the simple caching functionality described above. You do not need to implement any replacement or validation policies. Your implementation, however, will need to be able to write responses to the disk (i.e., the cache) and fetch them from the disk when you get a cache hit. For this you need to implement some internal data structure in the proxy to keep track of which objects are cached and where they are on the disk. You can keep this data structure in the main memory; there is no need to make it persist. (30%)
Note: If you implement caching, ensure that your program does NOT use absolute paths (e.g., /home/Alex/project/cache) to the location of your cache file or folder. Doing so will not be compatible with marking and result in zero marks. Instead, you must use relative paths (e.g., ./cache or subdir/cache).