Practical 1 – Network Application Development
Coursework Weight: 60%

In this practical, you will develop a number of small networking-based applications. These are designed to increase your competency in developing socket-based applications, as well as increasing your familiarity with a number of key technologies and measures. These are in widespread use, and commonly deployed to evaluate networks and to provide services over them.

Practical 1 is split into a number of smaller tasks: ICMP Ping Client, Traceroute Client, Web Server and Web Proxy. Importantly, the tasks build upon each other; the work you do in Task 1.1 will be fundamental to Task 1.2, and similarly, the work completed in Task 2.1 will greatly assist you in Task 2.2. Marks will be awarded for meeting certain criteria within each task. These are outlined in more detail within each task description. You are encouraged to progress as far as possible with each task. Do note however, that Task 1 and Task 2 are independent; attempting both of them is advised, even if you do not fully complete each.

To assist you in developing these applications, we have created a virtual machine image. This contains a number of useful tools, some of which you will be using in this practical. The image is deployed on all of the B076 lab machines. To use this, open a terminal window, and type:

```
svagrant up scc-203
```

The username and password for this machine is: `vagrant`, `vagrant`. This image will map your H: drive into your home directory (as `hdrive`). Be sure to save all of your work here, as the storage on the virtual machine is not persistent. This virtual machine has a graphical environment enabled. You a free to develop and work within this, but it recommended that you develop locally on your machine, using files shared with the virtual machine via the H: drive. In this case, test your application by executing it, either using a terminal within the graphical environment, or over an SSH connection to the virtual machine. To connect to the virtual machine using SSH, please use the following command executed in a terminal window:

```
ssh vagrant@127.0.0.1 -p 2222
```

For this practical, you will be building your applications using Python version 2.7 (as installed on the virtual machine). You are not expected to use any external libraries for this practical; doing so is strictly prohibited. All tasks can be fully achieved with the use of standard Python libraries.

During this practical, you will become familiar with the concept of 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 the packet. We can do this in much the same way as we would read and write to any file found on the local filesystem.

Submission and Assessment

The submission for all work completed in this practical is due by the end of Week 16. Please submit 4 distinct Python scripts, named according to each task. Even though you can reuse code from earlier tasks in the later tasks, it will simplify the marking procedure if you submit each solution independently.

During the marking session (scheduled for Week 17), you will be expected to demonstrate the functionality of each of these scripts. You will mainly be assessed on functionality, but expect to be able to walk-through and explain your code. As we will also be providing you with a few small snippets of code (to use in your own solution), you will not be expected to explain these in great detail. However, a general understanding of how these functions work will be beneficial to your overall learning and comprehension. There will also be a small proportion of marks available for a consistent code style and useful commenting. Resilient code, using `try` and `except` statements to catch errors is also preferred, and will be rewarded accordingly.

Task 1.1: ICMP Ping

The first task is to recreate the `ping` client discussed in Lecture 3: *Delay, Loss & Throughput*. 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 has reached that host, it is sent back to the sender. By measuring the amount of time taken to receive that response, we can determine the delay in the network. Similarly, by tracking the responses returned from our messages, we can determine if any have been lost in the network.

`ping` traditionally uses Internet Control Message Protocol (ICMP) messages to achieve this behaviour. More details can be found in RFC777. For this task, we will be sending *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 network delay between the two hosts.

Remember, you are recreating `ping` without the use of external libraries; they are explicitly prohibited!

Implementation Tips

There are a number of aspects to consider when writing your implementation. Carefully think about the logic required; use a whiteboard if need be. A `ping` client sends one ICMP *echo request* message at a time and waits until it receives a response. Measuring the time between sending the message and receiving it will give us the network delay incurred in transit. Repeating this process provides us with a number of delay measurements over time, showing any deviation that may occur.

To assist you in your implementation, we have provided skeleton code for this task. This can be found on the SCC. 203 Moodle page. It contains suggested functions, as well as an overview of functionality to be implemented by each. These are given as comments and are to be treated as **guidance only**. Note that you may have to change the parameters passed to each function as you advance with the task. The following Python libraries will also be useful to your implementation:

https://docs.python.org/2/library/socket.html
https://docs.python.org/2/library/struct.html
https://docs.python.org/2/library/time.html
https://docs.python.org/2/library/select.html
https://docs.python.org/2/library/binascii.html

Note that to run a privileged socket, such as `socket.SOCK_RAW`, your script must be run with elevated privileges (such as `sudo`). Note that an alternative solution using unprivileged sockets, such as `socket.SOCK_DGRAM`, is also acceptable.

We have provided you with a checksum function (included in the skeleton code) which can be freely used in your solutions without penalty. It is important that when passing a packet to this function, the checksum field must contain a dummy value of 0. Once the checksum has been calculated, it can be immediately inserted in the packet to send.

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.

Debugging and Testing

Any host, whether this be a PC, laptop, phone or server, should respond to a message generated by your application. In reality, this is not always the case, as both networks and hosts can choose to disregard these packets, and may do so for a number of reasons (including security). For the purposes of this task, using any well-known server is acceptable. As an example, the following popular sites will respond to an *echo request*: lancaster.ac.uk, google.com, or bbc.co.uk. These will all return with relatively low delays. To rigorously test your application, using servers located further afield will usually return larger delays. For example, the US Department of Education (www.ed.gov) can be queried.

To confirm that the server you have chosen to test with does in fact respond to ICMP *echo request* messages, feel free to use the existing built-in `ping` tool to verify reachability.

Once you are sending packets, you can use the Wireshark tool to inspect these. Wireshark will also let you investigate the packets that you receive back. In both cases, it provides a useful method to ensure that these contain the expected information. This is a very useful tool for debugging, especially if you are getting unexpected errors; this will show exactly what is being sent from your script. Wireshark is installed in the virtual machine and can be started from the graphical interface. Once started, you can capture packets on the `eth0` interface (as highlighted in Figure 1). It may also be useful to filter packets to `icmp` only, using the filter bar found towards the top of the interface (also highlighted in Figure 1).

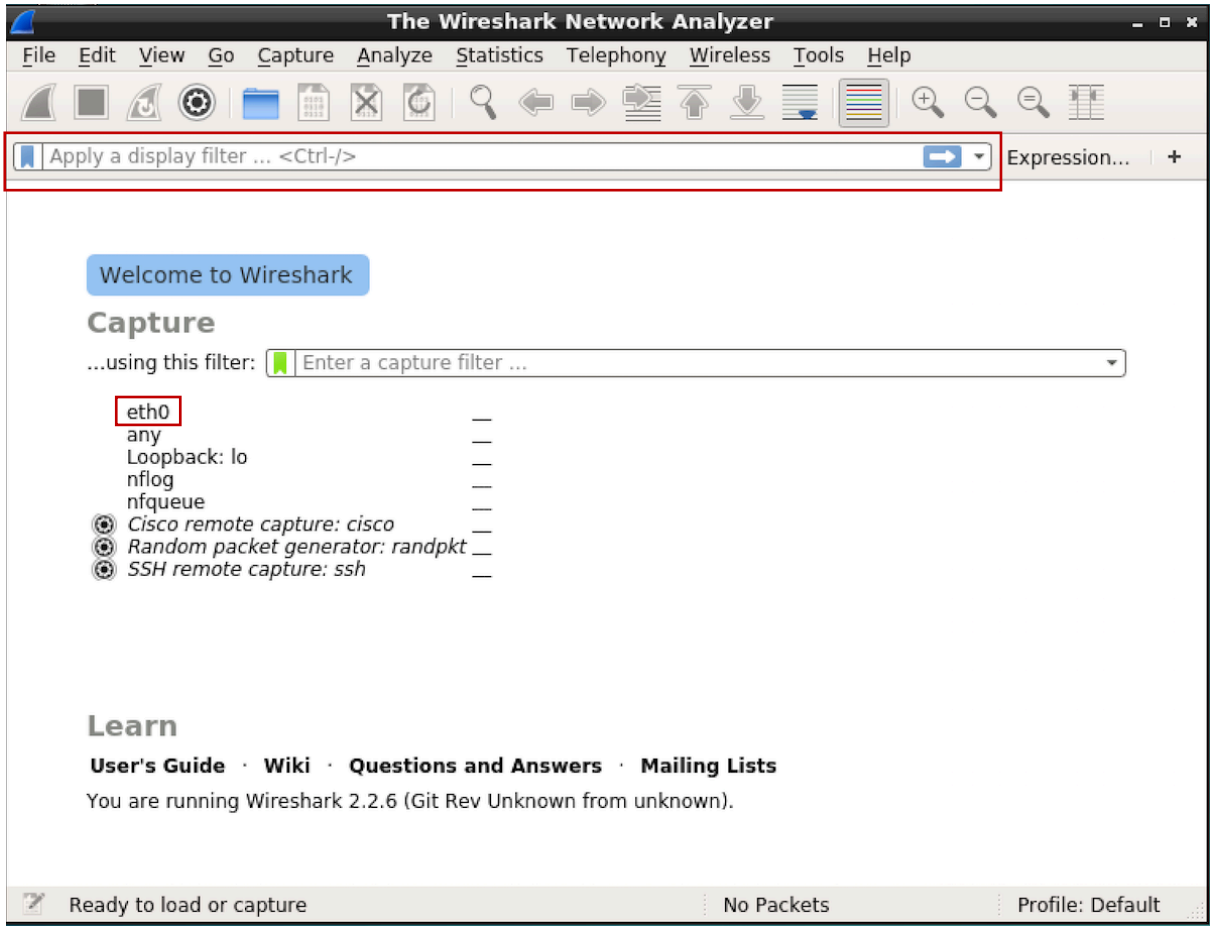


Figure 1: Wireshark Interface

Marking Criteria

You will be awarded the majority of marks for a functioning replica of the `ping` tool. That is, you can successfully send and receive ICMP *echo* messages, timing the delay between. This is then reported in the terminal window. Your application should continue to perform these measurements until stopped.

If you are unsure about the accuracy of the delay measured by your own tool, use the built-in `ping` tool to confirm your results. We're not expecting the results to be perfectly identical (delay changes all the time), but showing that they are close is expected.

Additional marks will be awarded for the following aspects:

- Taking an IP or host name as an argument
- Once stopped, show minimum, average and maximum delay across all measurements
- Configurable measurement count, set using an optional argument
- Configurable timeout, set using an optional argument
- Measuring and reporting packet loss, including unreachable destinations
- Handling different ICMP error codes, such as *Destination Host Unreachable* and *Destination Network Unreachable*

Please note that the features mentioned above are considered supplementary; you do not have to complete them all, and you can still receive a satisfactory mark without completing *any* of them. They are intentionally challenging and designed to stretch you.

Task 1.2: Traceroute

The second aspect of this task is to recreate the `traceroute` tool, again in Python. As discussed in Lecture 3: *Delay, Loss & Throughput*, this is used to measure latency between the host and each hop along the route to a destination. This too uses an ICMP *echo request* message, but with an important modification: the Time To Live (TTL) value is initially set to 1. This ensures that we get a response from the first hop; the network device closest to the host we are running the script on. 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 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, delay can be calculated at each hop in the network. This process can be repeated, increasing the TTL each time, until we receive an *echo reply* back (with an ICMP type of 0). This tells us that we have reached the destination, so we can stop the script.

Implementation Tips

As with the previous task, make sure you think carefully about the logic here. Remember you can build upon your Task 1.1 implementation, although you should submit two separate scripts; one for each subtask.

As before, the checksum function included in the skeleton code can be used without penalty.

The same Python documentation as noted in Task 1.1 will be useful for this task too. Of particular note is the `socket.setsockopt(level, optname, value)` function, which can be used to set the TTL of a socket (and thus the packets leaving it):

https://docs.python.org/2/library/socket.html#socket.socket.setsockopt

Debugging and Testing

As with the previous task, every host on the path to your chosen destination should respond to your *echo request* message. In reality, these messages 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 test with a closer endpoint that is reachable: lancaster.ac.uk. Although the number of hops is small (~5), 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 that at home, your results likely differ. You will also be able to reach external hosts more easily.

The `traceroute` utility can be used to confirm the results generated by your own application. This is installed on the virtual machine if you wish to use it. Be aware that by default, this tool actually sends messages over UDP instead of ICMP; this is done to avoid

the blocking discussed earlier. To force `traceroute` to send packets using ICMP, the `-I` flag can be used. See the Linux `man` page for more details:

https://linux.die.net/man/8/traceroute

As with Task 1.1, Wireshark can be used to inspect the packets leaving your application. Comparing these to those created using the `traceroute` utility will provide you with a meaningful comparison.

Marking Criteria

The majority of marks will be awarded for ensuring that your implementation behaves in a way similar to the `traceroute` utility. This includes providing delay measurements for each of the nodes between your machine and the chosen remote host. You are expected to increase the TTL of each message, until you reach this final destination.

Additional marks will be awarded for the following aspects:

- Measuring and reporting packet loss, including unreachable destinations
- Repeated measurements for each node
- Configurable timeout, set using an optional argument
- Configurable protocol (UDP or ICMP), set using an optional argument
- Resolve the IP addresses found in the responses to their respective hostnames

As before, please note that the features mentioned above are considered supplementary; you do not have to complete them all, and you can still receive a satisfactory mark without completing *any* of them. They are intentionally challenging and designed to stretch you.

Task 2.1: Web Server

For the second task of this practical, you will be building a simple HTTP 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 Lecture 6: *Web & HTTP*. Fundamentally, a web server receives a 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.

As with the previous task, we will be using network sockets to build our application and to interact with the network. 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 the virtual machine. For interest, application developers can register port numbers with the Internet Assigned Numbers Authority (IANA), reserving them for their application's use:

https://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.xhtml

The application you build should respond to HTTP GET requests, and should be built to HTTP/1.1 specification, as defined in RFC2616. These requests will contain a *Request-URI*, which is used to define the path to the object requested. For example, a request with a URI of `127.0.0.1:8000/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
- `8000:` 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. Your implementation needs only serve files from the same directory in which the Python script is executed.

Implementation Tips

As before, we have provided skeleton code that can be used to aid you in this task. This can be found on the course's Moodle page. It contains suggested functions, as well as an overview of functionality to be implemented by each. These are given as comments and are to be treated as **guidance only**. Note that you may have to change the parameters passed to each function as you advance with the task. An example HTML file (`index.html`) is also provided in the same location. The following Python library and its documentation may also serve as a pointer to helpful functions (the latter will be of particular interest):

https://docs.python.org/2/library/socket.html
https://docs.python.org/2/library/socket.html#socket.socket.accept

As a baseline, your implementation needs only to be single-threaded. This allows a maximum of one request to be handled at a time.

Debugging and Testing

To test your web server application, you must generate a valid request. There are a number of tools to achieve this. For example, the `wget` utility can be used to generate a request (presuming your web server is running on port 8000):

```
wget 127.0.0.1:8000/index.html
```

An equally valid method is to use a web browser, such as the Chromium Web Browser installed on the virtual machine. Simply point the browser to the same URL:

```
127.0.0.1:8000/index.html
```

If you are unsure about what a HTTP request should look like, Wireshark can again be used to inspect packets. This includes both the HTTP request and response. This will help you debugging the form and structure of your requests, identifying any issues that may be

present. If you are still using Wireshark from the previous task, make sure to remove the `icmp` filter! `http` can be used instead. It will also be necessary to capture packets on the loopback interface (`lo`), rather than the external interface (`eth0`).

If you wish to observe how a Web Server should behave (and examine the packets generated by such), Python provides a handy way of starting a very simple HTTP server implementation:

```
python -m SimpleHTTPServer
```

Requests to this server can be made using the methods described previously.

Marking Criteria

For this task, you will be awarded marks for a functioning Web Server, capable of handling requests for content. You should be able to demonstrate that, given a request, the Web Server will return the correct file, as well as producing a well-formed response header with protocol version and response code set correctly.

Additional marks will be awarded for the following aspects:

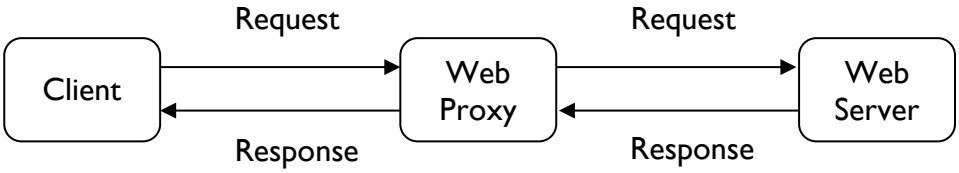
- Binding the Web Server to a configurable port, defined as an optional argument
- When a requested file is not available on the server, return a response with the status code *Not Found* (404)
- Create a multithreaded server implementation, capable of handling multiple concurrent connections
- Write your own HTTP client to query the web server (this should be submitted as an additional standalone Python file)

Please note that the features mentioned above are considered supplementary; you do not have to complete them all, and you can still receive a satisfactory mark without completing *any* of them. They are intentionally challenging and designed to stretch you.

Task 2.2: Web Proxy

Building on the Web Server described in Task 2.1, this task is concerned with building a Web Proxy. This operates in much the same way as 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:



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.

Debugging and Testing

As with Task 2.1, there are a number of ways to test your Web Proxy. For example, to generate requests using `wget`, we can use the following:

```
wget lancaster.ac.uk -e use_proxy=yes -e http_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 lancaster.ac.uk.

A web browser can also be used to the same effect. Many web browsers support the use of a web proxy through configuration. For example, the Chromium browser included in the virtual machine image can be started in the following way to utilise the Web Proxy:

```
chromium-browser --proxy-server="127.0.0.1:8000"
```

A caveat when testing your Web Proxy: some websites have enabled HTTP Strict Transport Security (HSTS) (RFC6797). This forces clients (including both `wget` and a web browser) to use HTTPS rather than HTTP. HTTPS is a secure version of HTTP, but we will consider this out of scope for this practical. To check if a website has this feature enabled, use the following tool, and ensure that the website you are using for testing purposes is **not** listed:

https://hstspreload.org/

As with the other tasks, Wireshark can be used to capture and investigate packets sent to and from your proxy. As the proxy will be receiving local requests from the web browser, as well as making external requests to fetch content, it is necessary to capture packets on both the external (`eth0`) and loopback (`lo`) interfaces.

Marking Criteria

For this task, the majority of marks will be awarded for demonstrating a working Web Proxy. You are expected to show the functionality of such through the use of either `wget` or a properly configured web browser. Note that you are not expected to demonstrate the Web Proxy using a website with HSTS enabled (see above).

Additional marks will be awarded for the following aspects:

- Binding the Web Proxy to a configurable port, defined as an optional argument
- Support for other HTTP request types (PUT, DELETE, etc.)
- 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 how it is handled 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 main memory; there is no need to make it persist.

As before, please note that the features mentioned above are considered supplementary; you do not have to complete them all, and you can still receive a satisfactory mark without completing *any* of them. They are intentionally challenging and designed to stretch you.