

# 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.

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.

## Practical Lab Structure

Practical 1 is split into a number of smaller tasks: ICMP Ping Client, Traceroute, and Paris-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 Task 1.2 will be fundamental to Task 1.3. Similarly, the work completed in Task 2.1 will greatly assist you in Task 2.2.

**For assessment purposes, only Task 1.3 and Task 2.2 will be marked.**

**Although Task 1.1, 1.2 and Task 2.1 will not be assessed, it is still vital that you complete them, as they will provide the foundation for the later tasks.**

For those tasks that are assessed, you will be awarded for meeting certain criteria. These are outlined in more detail within each task description. You are encouraged to progress as far as possible with each task. However, note that Task 1 and Task 2 are independent; attempting both is advised, even if you do not fully complete each.

This course uses a dedicated virtual machine image for the purposes of these labs. In previous years, this would have been accessible via the PCs in the physical labs.

## Running the Code Locally

It is also possible to run the code locally on your machine (directly and without a VM). To do so, you will need a regular install of Python 3 (version 3.11 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/>

Please note that you will still need the virtual machine (as described in the sections above and below) for the next coursework, starting in Week 17.

There are some caveats with this approach though (hence why we provide the virtual machine). Although the same Python code will run regardless of the environment, the underlying implementation (and therefore) behavior 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 variance that may occur when running code locally, we will **not** provide direct support for the method; only the virtual machine-based approaches will be supported by the teaching team.

If you do take this approach, also note that for the purposes of demonstration and marking, the provided virtual machine will (and must) be used. It is not an acceptable excuse to claim that it worked on your own machine, so at the very least, please ensure that it runs on the provided virtual machine image (using one of the methods described in the sections below or above) before submitting.

### Using a Remote Instance of the Virtual Machine

If the above options are not suitable, we have provided a remote instance of the virtual machine, running on infrastructure hosted at Lancaster University. This provides an identical experience and allows access to the environment without having to meet the hardware resource requirements of running the virtual machine or code locally. To access this, please use the following link and select the 'SCC 203':

<https://mylab.lancaster.ac.uk/>

These virtual instances can be accessed via a browser or a downloadable client; it is recommended that the client is used, as it offers a more stable experience. You may have to reload your browser once the client is installed for this to function correctly.

### Provided/Skeleton Code

To help you structure your code and get started with each task, we have provided you with some skeleton code, which should be available on the SCC.203 Moodle page. This file will 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. The skeleton code also contains some 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`, `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 (see T1.1 and T1.2). Furthermore, it also includes two methods which should be used to print out the results generated in T1.1 and T1.2: `printOneResult` and `printAdditionalDetails`.

Finally, it also includes five empty child classes (`ICMPPing`, `Traceroute`, `ParisTraceroute`, `WebServer` and `Proxy`) which are to be used for providing the solutions to each task (T1.1, T1.2, T1.3, T2.1 and T2.2 respectively). In the case of `ICMPPing` and `WebServer`, they also include some pseudocode to guide you (please see sections T1.1 and T2.1 for more details).

These classes inherit the `NetworkApplication` class, which will allow you to use the parent's helper methods (`checksum`, `printOneResult` and `printAdditionalDetails`). These can be called using the `super()` function. For more details on inheritance in Python, see here:

<https://docs.python.org/3/tutorial/classes.html#inheritance>

Each class contains an `__init__` method, which is 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. Please do not change the method signature for any of these `__init__` functions (always retain the passing of the `args` object).

To be clear: you can add new methods and code to each of the main classes (`ICMPPing`, `Traceroute`, `ParisTraceroute`, `WebServer` and `Proxy`), but the remaining structure, methods and code must remain intact.

For the purposes of plagiarism checking, this provided code will **not** be considered.

### Running your Python script

Once you have the virtual machine setup, you are ready to begin. For this practical, you will be building your applications using Python 3 (as installed on the virtual machine).

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
```

**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 first start this task, the provided code will run successfully (defaulting to pinging `lancaster.ac.uk`) but do nothing.

### Python Library Usage

You are not expected to use any external libraries/modules 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 believe 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 a 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 the end of Week 16. All code should be included in a single file, titled: `NetworkApplications.py`.

### Automated Testing

To improve the efficiency of the marking process, we will be using some simple automated testing of your code to help guide the in-lab sessions.

To help us in using this, we would request that you follow the guidance described in the 'Provided/Skeleton Code' and 'Submission and Assessment' sections regarding modifying provided classes, methods and filenames.

Failure to adhere to this will result in a slowed marking process, which will be detrimental to all. If you are in any doubt about whether a change or modification is acceptable, please contact the teaching team.

### Marking Session

During the marking session (scheduled for Week 17), you may be expected to demonstrate the functionality of each of these scripts, as directed by the teaching team. 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 [RFC792](https://tools.ietf.org/html/rfc792). 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 pseudocode for this task. This can be found in the provided code, specifically in the `ICMPPing` class. 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/3/library/socket.html>  
<https://docs.python.org/3/library/struct.html>  
<https://docs.python.org/3/library/time.html>  
<https://docs.python.org/3/library/select.html>  
<https://docs.python.org/3/library/binascii.html>

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 (for better or worse!). As such, it requires `sudo` to work.

In normal circumstances, a non-privileged version would probably be preferable; you don't

always have privileges on a system. This is where `SOCK_DGRAM` comes in. However, it appears that `SOCK_DGRAM`, when used specifically to send ICMP packets, is also a privileged operation within some flavours of Linux (and on the provided VM).

For the purposes of this practical, it is therefore recommended that you use `SOCK_RAW`, and run your Python using `sudo` privileges.

We have provided you with a checksum function (included in the `NetworkApplication` parent class) 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.

Be warned that some servers will reject ICMP requests with an identifier of 0, so ensure that this is set to a value  $> 0$ .

Please ensure that the `printOneResult` and `printAdditionalDetails` methods (and only these methods) are used for reporting the results.

## Debugging and Testing

In theory, any host, whether this be a PC, laptop, phone or server, should respond to a ping message generated by your application. However, 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).

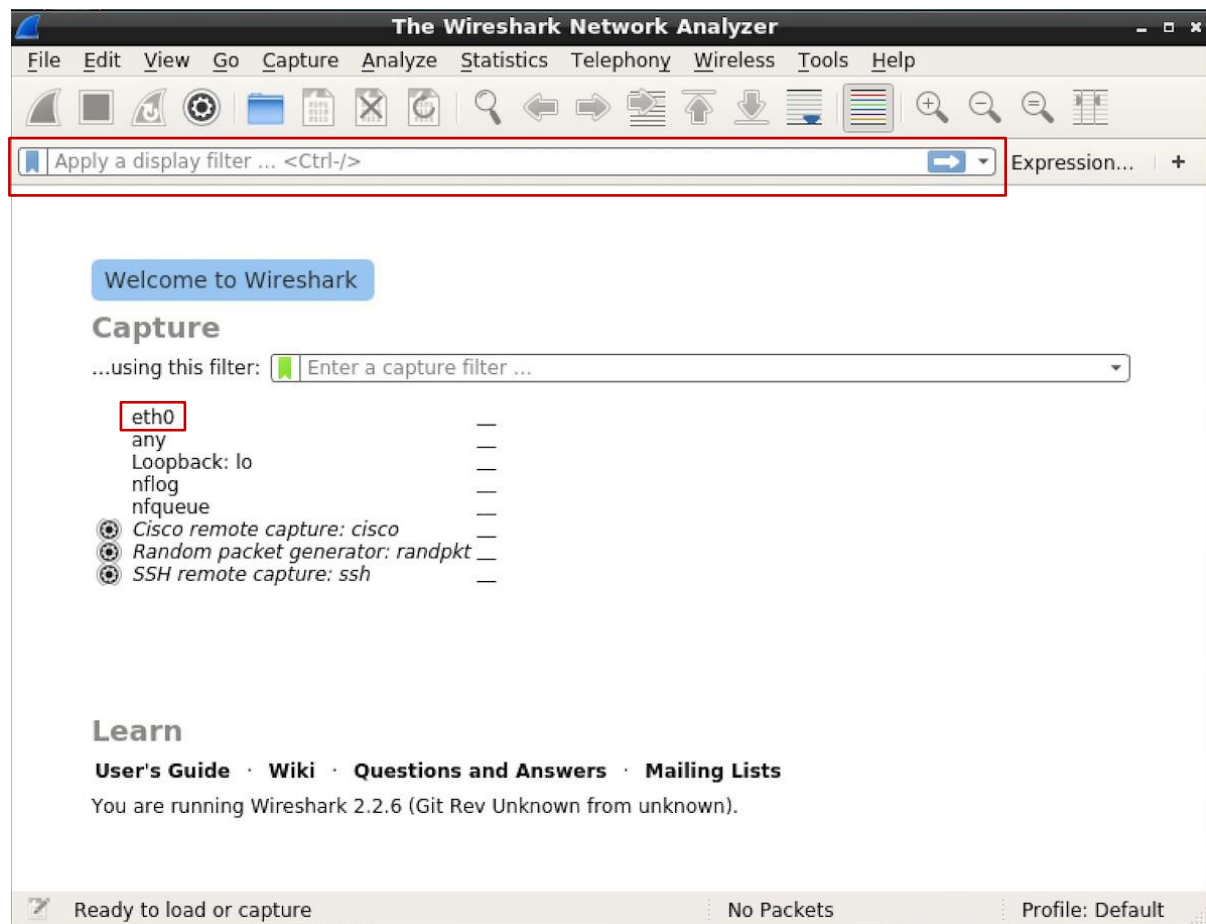


Figure 1: Wireshark Interface

## Completion Criteria

For this task, we are looking 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 are not expecting the results to be perfectly identical (delay changes all the time) but showing that they are close is expected.

Potential additional features include:

- Once stopped, show minimum, average and maximum delay across all measurements (use `printAdditionalDetails` method)
- Configurable measurement count, set using an optional argument (use `count` positional argument)
- Configurable timeout, set using an optional argument (use `timeout` positional argument)
- Measuring and reporting packet loss, including unreachable destinations (use `printAdditionalDetails` method)



- Handling different ICMP error codes, such as *Destination Host Unreachable* and *Destination Network Unreachable*

As with the rest of this task, you do not have to completely implement these features, as **no marks will be awarded for Task 1.1**. The features listed above may assist you in Tasks 1.2 and 1.3 though; they are intentionally challenging and designed to stretch you.

## Task 1.2: Traceroute

Building on Task 1.1, 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.

As before, the provided checksum function included in the skeleton code can be used without a 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/3/library/socket.html#socket.socket.setsockopt>

For this task, we rely on ICMP Type 11 error messages (TTL Exceeded). Unlike Type 0 messages (Echo Reply), these do not contain an identifier field in the response. Any checking you might do in respect to this identifier will therefore fail (as it is not present).

Please ensure that all code pertaining to this task is included in the `Traceroute` class.

### 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.

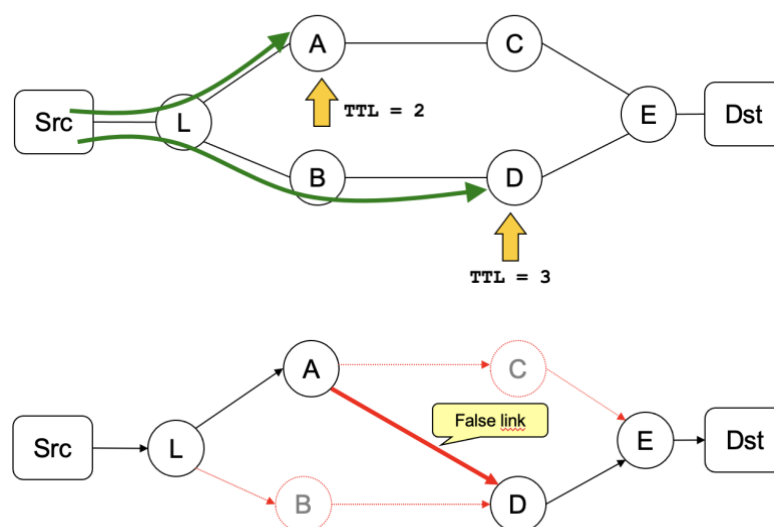
The `traceroute` utility can be used to confirm the results generated by your own application. This is installed on the virtual machine if you wish to use it. Be aware that by default, this tool actually sends messages over UDP instead of ICMP; this is done to avoid the blocking discussed earlier. To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

### Task 1.3: Paris-Traceroute

A well-known limitation of trace route is that it may indicate a path that does not actually exist in the presence of “load-balancers” in the network. Consider the example below where a source host `Src` sends traceroute traffic to a destination host `Dst`.



Today, most commercial routers have load balancing capabilities. If this capability is enabled, a router acts as a **load balancer**. In the example above, node `L` is a load balancer. A load balancer splits incoming traffic across a set of outgoing interfaces. In the above example, `L` splits incoming packets from `Src` across two interfaces: the outgoing interface facing node `A` and the outgoing interface facing node `B`.

A load balancer typically uses flow-based splitting of traffic, which uses the same output interface for all packets that have the same “flow identifier”; that is, the 5-tuple: 1) IP source address, 2) IP destination address, 3) protocol, 4) source port and 5) destination port. When flow-based splitting is applied to a traceroute traffic between two hosts, each packet can be assigned to a different path, because the traceroute tool changes the flow identifiers (i.e., the destination port) of subsequent packets sent to a destination. As a result, the traceroute tool incorrectly conclude “false links” between nodes.

The *Paris-traceroute* is a modified traceroute that does not modify the flow identifiers of subsequent packets (unlike traceroute), and therefore a paris-traceroute traffic between the same source and destination follows the same path even in the presence of flow-based splitting of traffic by the load-balancers.

For this task, you are expected to recreate the paris-traceroute tool. You can build upon your Task 1.2 implementation. You are strongly advised to capture the traffic generated by the paris-traceroute tool with WireShark to inspect the packets generated by this tool.

### Marking Criteria

The majority of marks will be awarded for ensuring that your implementation behaves in a way similar to the `paris-traceroute` utility. This includes providing three delay measurements for each node (i.e., for each TTL) between your machine and the chosen destination host. You are expected to increase the TTL of each message, until you reach the final destination, as similarly done for the original trace-route.

Additional marks will be awarded for the following aspects:

- Measuring and reporting packet loss, including unreachable destinations (use `printAdditionalDetails` method)
- Repeated measurements for each node (you can present these either as separate results using the `printOneResult` method, or use the `printMultipleResults` to display all measurements for each hop at once)
- Configurable timeout, set using an optional argument (use `timeout` positional argument)
- Configurable protocol (UDP or ICMP), set using an optional argument (use `protocol` positional argument)
- Resolve the IP addresses found in the responses to their respective hostnames (see optional `destinationHostname` variable in `printOneResult` method)

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 `127.0.0.1:8080/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. Your implementation needs only serve files from the same directory in which the Python script is executed.

### Implementation Tips

As before, we have provided pseudocode that can be used to aid you in this task. This can be found in the provided code, specifically in the `WebServer` class. It contains suggested a suggested function, as well as an overview of functionality to be implemented. This is 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:

<https://docs.python.org/3/library/socket.html>

<https://docs.python.org/3/library/socketserver.html>

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 `curl` utility can be used to generate a request (presuming your web server is running on port 8080):

```
curl 127.0.0.1:8080/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:

```
127.0.0.1:8080/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:

```
python3 -m http.server --protocol HTTP/1.1 6789
```

You can specify the port number for the simple HTTP server runs on. In the above example, port 6789 is provided. Using python version 3.11, you can also specify the http protocol version (HTTP/1.1 in the example above). Requests to this server can be made using the methods described previously.

### Completion Criteria

For this task, you expected to build 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.

Potential additional features include:

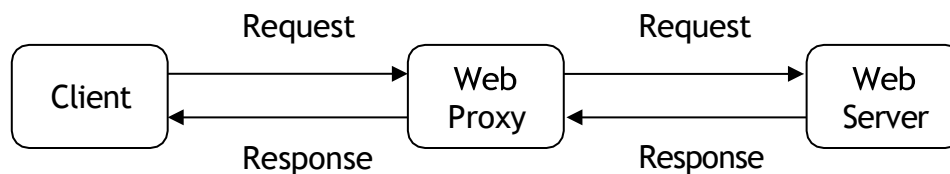
- Binding the Web Server to a configurable port, defined as an optional argument (use `port` positional argument)
- When a requested file is not available on the server, return a response with the status code *Not Found* (404)
- Create a multithreaded server implementation, capable of handling multiple concurrent connections

As with the rest of this task, you do not have to completely implement these features, as **no marks will be awarded for Task 2.1**. The features listed above may assist you in Task 2.2 though; 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.

Please ensure that all code pertaining to this task is included in the `Proxy` class.

### 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 `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](https://tools.ietf.org/html/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 scope for this practical.

Thanks to the proliferation of HTTPS (this is a good thing, just not for this practical!) the list of live websites that 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 virtual machine and test it from that. This could be your web server implementation from Task 2.1. Alternatively, Python also has a simple-to-use 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 also 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 using `curl`. 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 (use `port` positional argument)
- Support for other HTTP request types (PUT, DELETE, etc.)
  - This can be tested using the *postman-echo* service, details of which can be found here: <https://docs.postman-echo.com/?version=latest>
- 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.