

---

# COMP2017 / COMP9017

---

# Assignment 3

---

Full assignment due: May 31st, 11:59 pm AEST (Week 13 Sunday)

Milestone due: May 24th, 11:59 pm AEST (Week 12 Sunday)

*This assignment is worth 16% of your final assessment*

**This is the updated version of Assignment 3. Please see Ed for more details.**

## Task Description

In this assignment, you will create a networked server (“JXServer”) that sends files to clients in response to requests. Your server will support multiple connecting clients simultaneously as well as multiple simultaneous connections from the same client for increased transfer speeds.

Your assignment will also be tested for performance. You will need to correctly manage multiple parallel connections.

## Milestone

To ensure you are making progress on this assignment, you need to complete a minimum amount of work by May 24th, 11:59 pm AEST (Week 12 Sunday) to receive up to 4 out of 16 total marks.

2 out of 4 milestone marks are for passing specified Milestone automatic correctness tests on Ed.

By the Milestone deadline, you must also submit in your assignment workspace on Ed, a 300 word description of your design of your assignment so far (please use a plain text file). This short description should outline the general logical flow of your assignment, including initial socket setup and including the method by which you are handling multiple connections. You may wish to include an example scenario of a client request and the flow of data on the network and in your program under this scenario.

You must attend your Week 13 tutorial, where you will have a short 5 minute discussion with your tutor as to your progress and the design of your assignment submission, with reference to your written description and last submission before the Milestone. See the Submission section at the end of this document for further details.

The written description and review by your tutor in your Week 13 tutorial will contribute the other 2 out of 4 milestone marks.

See the “Submission and Mark Breakdown” section at the end of this document for further information.

## Working on your assignment

Staff may make announcements on Ed (<https://edstem.org>) regarding any updates or clarifications to the assignment. The Ed resources section will contain a PDF outlining any notes/changes/corrections to the assignment. You can ask questions on Ed using the assignments category. Please read this assignment description carefully before asking questions. Please ensure that your work is your own and you do not share any code or solutions with other students.

You can work on this assignment using your own computers, lab machines, or Ed. However, it must compile and run on Ed and this will determine the grade. It is important that you continually back up your assignment files onto your own machine, flash drives, external hard drives and cloud storage providers, **ensuring that it is private, and only accessible by you**. You are encouraged to submit your assignment while you are in the process of completing it to receive feedback and to check for correctness of your solution.

## Introduction

Typically, networking functionality is implemented by kernel space code. On Linux and other Unix-like operating systems, the kernel implements the various components required for networking, such as Wifi capabilities and the Internet Protocol. This is made available to userspace programs through special system calls, which interact with an abstraction known as a socket.

Sockets represent a connection to another endpoint on a network, but are analogous to file handles and are also described by a file descriptor. “Reading” from or “writing” to a socket corresponds to receiving or sending data over the network. However, as there are specialised operations that need to be performed with sockets, such as accepting or making connections to network destinations, there are special system calls that are generally used.

A full discussion of network stacks and protocols is out of scope for this unit. For this assignment, your server will use standard Transmission Control Protocol (TCP) connections on top of the Internet Protocol, version 4 (IPv4). This is a standard networking protocol combination for ubiquitous services such as HTTP which we use on the “World Wide Web”.

The combination of TCP and IP(v4), as implemented and exposed to userspace by the kernel, permits the formation of network connections. We will not be using any other version of IP in this assignment. A client initiates a connection to a server. Both client and server endpoints are defined by an IP address, which in IPv4 is a 32 bit unsigned integer, as well as a TCP port number, which is a 16 bit unsigned integer. Once the server accepts the connection, the endpoints exchange data to setup a reliable connection. Data that userspace programs send into the socket, at either end of the connection, is delivered and made available for receiving by the userspace application at the other endpoint. TCP guarantees that data is delivered reliably and in order over varying network conditions. It also permits either endpoint to send data simultaneously to each other. However, it treats data as a continuous stream rather than discrete messages; that is for example if an endpoint sends 5 bytes then 10 bytes, the other endpoint will be able to read the data as it is received, with no indication that there was originally a break in sending 5 bytes in.

Your server is responsible for creating a “listening” TCP socket. This means that it waits for inbound TCP connections from clients.

In software development, you will often be required to program against third party APIs and libraries. To practice this skill, for this assignment you will need to refer to the manpages for system calls mentioned to determine how to employ them, though guidance will be provided in this specification.

To create your TCP socket, you need to use **socket(2)**. For the domain argument, please use **AF\_INET**. For the type argument, please use **SOCK\_STREAM**. You can leave the protocol argument as 0.

You then need to bind your TCP socket to an address using **bind(2)**. This assigns an IP address and TCP port to your end of the socket.

You may have seen IPv4 addresses represented in “dotted quad notation”, such as “192.0.2.1”, which is simply 4 8-bit integers extracted in order from the 32-bit address. You may wish to use **inet\_aton(3)** to convert from dotted quad notation to the 32-bit integer representation, and **inet\_ntoa(3)** to convert in the other direction.

Next, you need to specify that your TCP socket will be listening for incoming connections, using **listen(2)**.

Finally, you will wait for inbound connections on your socket using **accept(2)**. The kernel will queue for your program connections to the IP address and TCP port combination that you **bind(2)** to. When **accept(2)** returns, it creates a new socket which allows your program to communicate with this particular accepted connection. Your original socket remains listening for further connections that can be accepted with **accept(2)**.

Once you have a connected socket, you can send data to the other endpoint using **send(2)**, and you can receive any data the other endpoint has sent using **recv(2)**.

To close a socket, preventing further communication on it, you can use **shutdown(2)** and/or **close(2)**.

This is a very basic sequence of system calls. To support multiple simultaneous connections, you may wish to use concurrent programming, such as threads or processes. You may also wish to use mechanisms by which you can be informed when a connection is waiting or data can be read. These include **select(2)** and **poll(2)**.

Please note the distinction between host and network byte order. Network protocols, including the one in this assignment, generally send data such as integers in big-endian (“network byte order”), whereas your host system generally stores them as little-endian. To help you convert between byte orderings, please review the manpages **byteorder(3)**, **bswap(3)** and/or **endian(3)**. For the avoidance of doubt, where applicable, data in this assignment is to be sent as big-endian on the network.

## Task

You will write your server in C, compiling to a single binary executable. It will accept exactly 1 command line argument as follows:

```
./server <configuration file>
```

The argument **<configuration file>** will be a path to a binary configuration file. The data of this file will be arranged in the following layout from the start:

- 4 bytes - the IPv4 address your server will listen on, in network byte order

- 2 bytes - representing the TCP port number your server will listen on, in network byte order
- All remaining bytes until the end of the file - ASCII representing the relative or absolute path to the directory (“the target directory”), from which your server will offer files to clients. This will not be NULL terminated.

The configuration file is binary, not plaintext, and does not have any delimiters or terminators (including newlines).

The full contents of an example configuration file, as a hexdump, are shown below:

```
c0 00 02 01 16 2e 2f 74 65 73 74 69 6e 67
```

The file is explained in order below:

- **c0 00 02 01** - 4 bytes which represent the IPv4 address “192.0.2.1” in network byte order
- **16 2e** - 2 bytes which represent the port number 5678 in network byte order
- **2f 74 65 73 74 69 6e 67** - 8 bytes of ASCII representing the string “/testing” which is the path of the target directory

Your server will listen for and accept connections from clients. Upon a successful connection, clients will send a number of possible requests, described in the below format. Please note that all integer fields are sent in network byte order. Your server should not send any data except in response to a client request.

When a client is finished making requests on a given connection, they will **shutdown(2)** the connection. When you detect this (refer to **recv(2)**), you should close that socket and clean up any associated data in your program.

The one exception to this is the client may send a “Shutdown command” (see section below).

All client requests and server responses consist of one or more structured series of bytes, called messages. Each message will contain the following fields in the below format:

- 1 byte - Message header; this describes details about the message as follows.
  - First 4 bits - “Type digit”: A single hexadecimal digit that defines the type of request or response. It is unique for different types of messages.
  - 5th bit - “Compression bit”: If this bit is 1, it indicates that the payload is compressed (see “Compression” section below). Otherwise, the payload is to be read as is.
  - 6th bit - “Requires compression bit”: If this bit is 1 in a request message, it indicates that all server responses (except for error responses) must be compressed (see “Compression” section below). If it is 0, server response compression is optional. It has no meaning in a response message.
  - 7th and 8th bits - padding bits, which should all be 0.
- 8 bytes - Payload length; unsigned integer representing the length of the variable payload in bytes (in network byte order).

- Variable byte payload - a sequence of bytes, with length equal to the length field described previously. It will have different meanings depending on the type of request/response message.

The full contents of an example message, as a hexdump, are shown below. Note that it is a valid example of a message in this network protocol, but does not correspond to any meaningful request or response you actually have to implement in this assignment.

**d8 00 00 00 00 00 00 07 ab ab ab ab ab ab ab**

This message is explained in order below:

- **d8** - 1 byte message header. This is represented by the binary **11011000**.
  - **1101** - 4 bits representing the hexadecimal type **0xD**
  - **1** - 1 bit flag indicating this payload is compressed
  - **0** - 1 bit flag which means, if this message is a request, the response does not have to be compressed. If this message is a response, it has no meaning.
  - **00** - 2 bits of 0 padding
- **00 00 00 00 00 00 00 07** - 8 bytes representing the payload length 7 in network byte order
- **ab ab ab ab ab ab ab** - the 7 byte payload

On any single connection, clients will only send one request at a time, before waiting for the appropriate response. That is, after sending a request, the client will wait for the server to send a complete response before sending the next request. For some requests, the server will need to send multiple response messages. An example is file retrieval, where the server may need to send a file to the client split over many response messages. In this case, the client will wait until all appropriate response messages are received before sending the next request, if any.

## Error functionality

If you receive a client request with invalid (unknown) type field, your server is to send back a response with type digit 0xf, with no payload (payload length field 0), and then close the connection. You should also send this error message if there are any other errors that arise in requests with valid request type fields. You are to also send back this error response if you receive a client request with a type field that must only be used in server response messages. Error messages are never compressed, even if the original request indicated compression is required.

## Echo functionality

Clients may request an “echo”. The request type digit will be 0x0. There will be an arbitrary sequence of bytes in the payload of the request.

In response to this request, your server is expected to send back a response with type 0x1. The payload of your response should contain the same payload you received in the request. Note that if

the original request requires compression, then you need to compress the payload before returning it in the response. However, after decompression, the payload should be identical to the one you received.

## Directory listing

The request type will be 0x2. There will be no payload and the payload length field will be 0. The client is requesting a list of all files in the server's target directory.

In response to this request, your server is expected to send back a response with type 0x3. The payload of your response should contain the filenames of every regular file in the target directory provided in the command line arguments to your server. (You do not have to return subdirectories, links, or any other type of entry other than regular files). The filenames can be returned in an arbitrary order.

These filenames are to be sent end to end in the payload, separated by NULL (0x00) bytes. Include a NULL (0x00) byte at the end of the payload. You will need to set the payload length appropriately. If the directory is empty, send a single NULL (0x00) byte as a payload.

## File size query

The request type will be 0x4. The request payload will be a NULL terminated string that represents a target filename, for which the client is requesting the size.

In response to this request, your server is expected to send back a response with type 0x5. The payload of your response should contain the length of the file with the target filename in your target directory, in bytes, represented as a 8-byte unsigned integer in network byte order. If the requested filename does not exist, return an error response message (see "Error functionality").

## Retrieve file

The request type will be 0x6. This is a request for part or whole of a file in your server's target directory. The payload will consist of the following structure:

- 4 bytes - an arbitrary sequence of bytes that represents a session ID for this request. Please see below for uniqueness requirements.
- 8 bytes - the starting offset of data, in bytes, that should be retrieved from the file
- 8 bytes - the length of data, in bytes, that should be retrieved from the file
- Variable bytes - a NULL terminated string that represents a target filename

The filename will not contain any relative or absolute path components, you only need to search in the target directory, and no subdirectories.

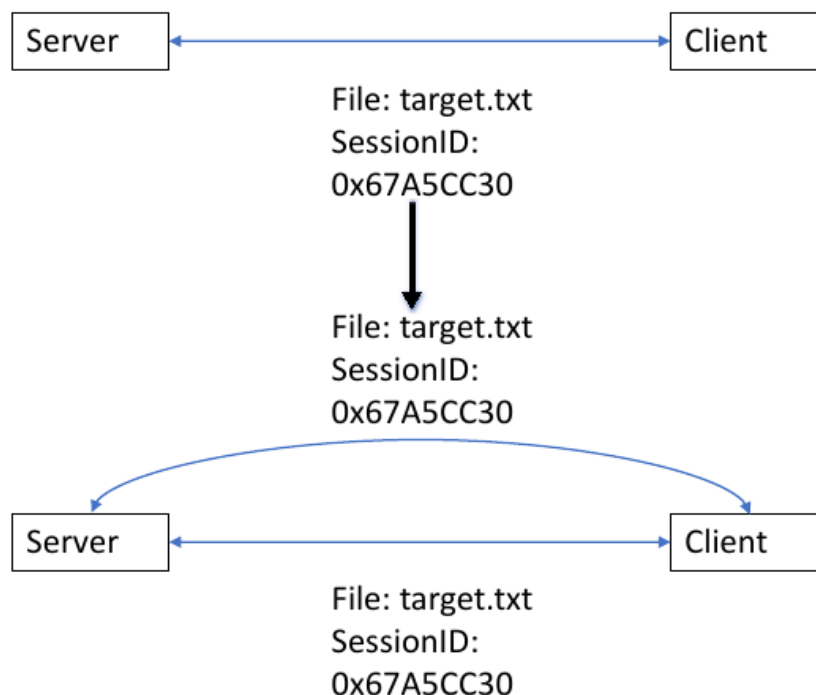
In response to this request, your server is expected to send back one or more response messages with type 0x7. Each response of type 0x7 may represent a portion of the requested file data. It is up to you how many and how large these portions you send are. It is also up to you the order in which you

send these portions; the start of the file does not need to be sent first, as long as all requested data is eventually received by the client. Different portions which you send corresponding to the same original request must not overlap in the byte ranges from the target file they contain. Each payload must consist of the following structure:

- 4 bytes - the same session ID as was provided in the original request
- 8 bytes - a starting offset of data, in bytes, from the target file, that this response contains
- 8 bytes - the length of data, in bytes, from the target file, that this response contains
- Variable bytes - the actual data from the target file at the declared offset and length in this response

The client may open several concurrent requests for the same filename on different simultaneous connections, with the same session ID. If you receive multiple connections with requests for the same file range with the same session ID, it means you are able to multiplex your file data across those connections; a single requesting client is unifying the data at the other end. If you choose to do this, you need to ensure that across all connections sharing the same session ID, the whole requested range of the file is eventually reconstructed. The client may make an extra concurrent connection for a given file at any time.

In the diagram below, the blue double headed arrows indicate a successful connection. Originally, the client has opened one connection, requesting the file target.txt, with session ID 0x67A5CC30. The client then opens a second connection, requesting the same file. Because the session ID is the same, the server accepts the connection, and is able to return file data simultaneously over the two connections (note that the requested file range must also be the same, but this is not shown in the diagram).

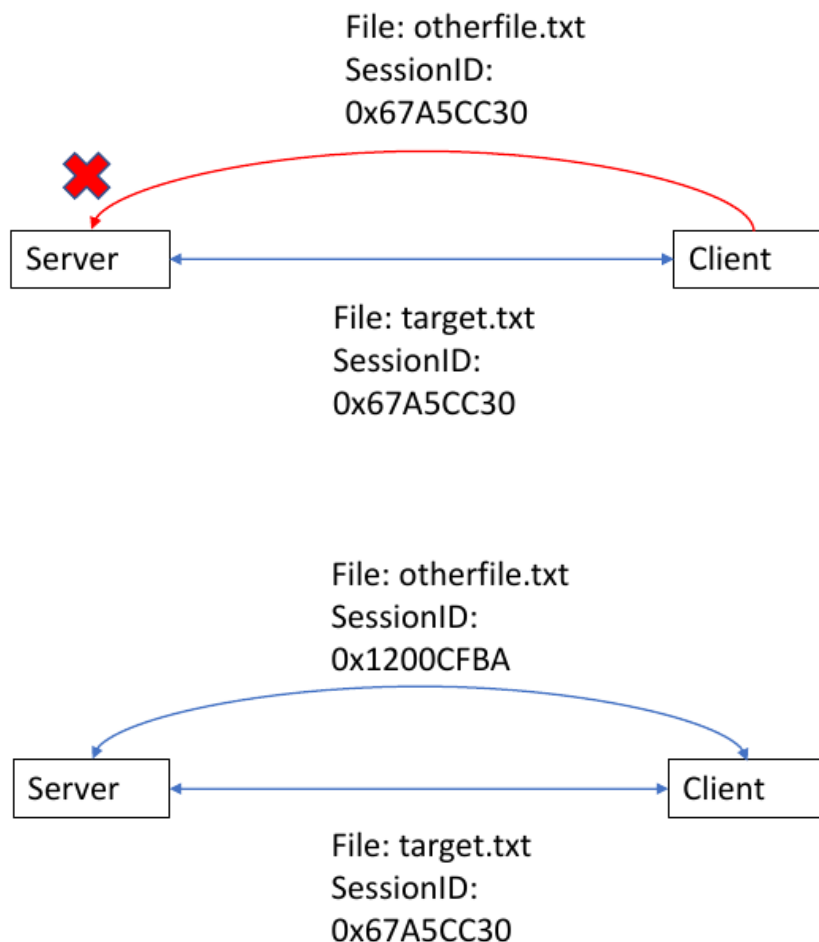


You do not have to multiplex your file response across multiple connections. If so, for connections on which you will not be returning data, you can send a response with type 0x7 with empty payload. However, your program must be returning the requested response on at least one connection among those the client opens.

If you would like to achieve higher performance, you will need to implement multiplexing of your file response across multiple connections.

It is not valid to receive a request for a different file, or the same file with a different byte range, with the same session ID as a currently ongoing transfer. If this occurs, you should send an error response message (see “Error functionality”). However, once the entirety of a file is transferred, the session ID may be reused for different files or the same file with a different byte range, in subsequent requests.

In the below example diagram, the red arrow indicates a failed connection where an error response should be sent. The client has an existing connection requesting the file target.txt, with session ID 0x67A5CC30. It has attempted to open a new connection requesting the different file otherfile.txt, with the same session ID. This is invalid; however the client is able to make a request for otherfile.txt, shown using the different session ID 0x1200CFBA. The server is then expected to service these two requests simultaneously.



You may receive a request for the same file with a different session ID while that file is being transferred under a first session ID. This is considered a separate client that requires a separate copy of the



file and should receive the appropriate response.

If you receive any other invalid request, such as the filename not existing, or the requested byte range being invalid for the size of the target file, you must send an error response message (see “Error functionality”).

## Shutdown command

The request type digit will be 0x8 and the payload will be 0 length. Your server does not send any response. Instead, your server will **shutdown(2)** and **close(2)** all current connections, and exit, cleaning up/freeing all threads/processes/memory, as required. You are guaranteed there will be no further new connections or requests after this command. You are guaranteed a Shutdown command will only be sent after your server completes processing all other requests. After all processing has been completed and resources freed, your server should terminate with exit code 0.

## Lossless compression

For any message where the compression bit is set in the message header, the variable length payload is losslessly compressed, which means it is encoded in a way that completely retains the original payload information, but aims to reduce the size by applying a compression algorithm to the data. The payload will have the following structure in order (note that the sections are not necessarily aligned to bytes):

- Variable number of bits - Compressed payload. See below for details. It is not necessarily aligned to a byte boundary.
- Variable number of bits - padding with 0 bits to the next byte boundary. This ensures that the structure is aligned to a byte boundary.
- 1 byte - an unsigned integer representing how many bits of 0 padding were required in the previous field

Note that the payload length field of the compressed message will contain the length of the compressed payload in bytes. Note that the padding in the compressed payload ensures it is aligned to whole bytes in size.

Your server may also receive request messages (which may also be compressed) where the “Requires compression” bit is set in the message header. This means that any message(s) that your server sends in response to such messages must be compressed. Your server should never set this bit in a response message; it is only valid in request messages. If this bit is not set for a request, then it is up to you whether to compress response(s) to that request. Compression can reduce the amount of data needing to be sent over the network, but requires a processing time tradeoff to compute the compressed payload.

In this assignment, you will apply compression by replacing bytes of uncompressed data with variable length bit sequences (“bit codes”). This works by aiming to encode more frequently used bytes to shorter bit codes, and less frequently used bytes to longer codes. Therefore, it is possible to have data which does not compress at all, or in fact “compresses” to a larger size. The compression dictionary defines the mapping of bytes to bit codes and consists of 256 segments of variable length. Each

segment corresponds in order to input byte values from 0x00 to 0xFF. Each segment is not necessarily aligned to a byte boundary. However, at the end of the 256 segments, there is padding with 0 bits to the next byte boundary. This means the entire compression dictionary is aligned to a byte boundary overall. The overall structure follows:

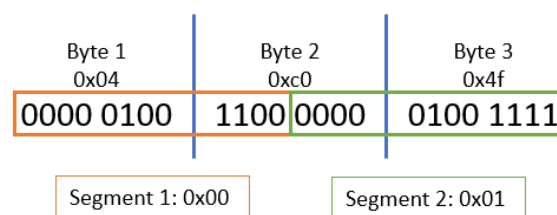
- 256 of the following segments, with no padding in between (including no padding to byte boundaries):
  - 1 byte - unsigned integer representing the length of this code in bits; this is equal to the length of the next field in bits. It is not necessarily aligned to a byte boundary.
  - Variable number of bits - the bit code that is used to encode the byte value corresponding to this segment. It is not necessarily aligned to a byte boundary.
- Variable number of bits - padding with 0 bits to the next byte boundary. This ensures that the entire dictionary is aligned to a byte boundary.

You will be provided with a binary file that contains a compression dictionary in the structure described above. This binary file will always be called **compression.dict** and be present in the same directory as your server executable during testing. The compression dictionary stays the same for all requests handled each time your server runs. That is, your server only needs to read the file once at start up to obtain the dictionary, then use that dictionary to handle all compressed requests or responses.

An example of the start of a compression dictionary is shown below. Note that this only includes a few segments and the full dictionary would include all 256 segments plus 0 bit padding at the end as required:

**04 c0 4f ...**

This data is explained in order below:



- The first segment corresponds to the byte **0x00**
- **04** - the first byte is the length of the code for **0x00** in bits (i.e. 4 bits)
- **c0** - the binary for this byte is **11000000**. The first 4 bits, **1100**, is therefore the bit code for **0x00**. The next 4 bits is the start of the second segment, corresponding to the byte **0x01**. The size of the entire first segment for byte **0x00** is 12 bits.
- **4f** - the binary for this byte is **01001111**. Remember that segments are not necessarily aligned to byte boundaries. Therefore, the byte representing the length of the code for **0x01** is comprised of the second 4 bits from **c0** and the first 4 bits from **4f**. The binary is therefore

**00000100**, which corresponds to a code length of 4 bits as well. This means that the second 4 bits of **4f**, **1111**, is the code for **0x01**.

- The size of the second segment is also 12 bits. Note that segments will vary in size. Depending on the size of each segment, the boundary of each segment can be at any bit offset within a byte, not necessarily 0 or 4 bits like in this short example.

To create the compressed payload, for each byte in the original payload, obtain the segment in the compression dictionary corresponding to the byte value. In your compressed output, the bit code for this segment should be output for this input byte.

Using just the segments shown in the example, you could compress data containing bytes **0x00** and **0x01**. For the following example uncompressed data:

**01 00 00 01 01 00**

Simply replace each uncompressed byte with the bit code from the compression dictionary. That is, the compressed binary would be:

**1111 1100 1100 1111 1111 1100**

This would result in the final compressed bytes:

**fc cf fc**

The compressed payload is output as is. However, at the end of the compressed payload, there is 0-bit padding to the nearest byte boundary, and a single byte indicating how much padding there was (see above).

To decode compressed data, as you read in bits from the compressed payload, simply reverse the process by using the compression dictionary to find the original byte values. When you decompress a payload, interpret that payload as per the other functionality of your server, depending on the message type digit contained in the message header.

For the “Requires compression” bit, you do not ever compress already compressed data. For example, if you receive an echo request with compressed payload, and “Requires compression” set, you do not compress the payload again in your response.

As you may note, input bytes are encoded to variable length bit codes (“variable length coding”). It is guaranteed that your compression dictionary gives you bit codes that are **uniquely decodable**. This means there is only one way to decode a compressed payload, even though you are not explicitly informed where the variable length bit codes start and end.

## Notes and Hints

- For performance reasons, test clients will begin to make connections within approximately 1 second of your server startup. Please ensure that any setup you do is finished by this point, and your server is ready to accept connections. If your server is not ready to accept connections, you may receive a “Connection refused” error from the test client.
- You may find it helpful to use fixed-width integer types such as **uint8\_t** and **uint64\_t**

- When manipulating individual bits, you will find it helpful to review bitfields and/or bitwise operations.
- Depending on what your server is sending, the test client may not always be able to immediately validate the data sent. For example, if you declare a certain payload length, but only send part of it, the client may wait indefinitely for the remainder of the payload to be sent. There is no timeout built into the assignment protocol. This means that you may receive a “ran too long” error due to inherent limits in the testing system.

## Submission and Mark Breakdown

Submit your assignment via Git to Ed for automatic marking. Your server program will be executed, and automatic client programs will connect to it to carry out testing.

Your code will be compiled with the following options. Please note you may not use variable length arrays.

```
-O0 -Wall -Werror -Werror=vla -g -std=gnull -lm -lpthread -lrt -fsanitize=address
```

The mark breakdown of this assignment follows (**16 marks total**).

- **7 marks** for correctness, assessed by automatic test cases on Ed. Some test cases will be hidden and will not be available.
  - **2 mark** of these 7 correctness marks will be due for assessment at 11:59 pm AEST, Sunday May 24th as the Milestone. These will correspond to a subset of the final test cases, concerning starting your server, reading the configuration file, “Error functionality” and “Echo functionality” only (no compression required). Specifically, only test cases with names starting with **milestone** will count towards these marks. This will be assessed against your last submission before this Milestone time.
  - There will be no opportunity to regain this mark after the Milestone.
- **5 marks** for performance. This will assess the speed of your submission only. You must pass all correctness test cases before your submission will be assessed for performance.
- **4 marks** from manual marking by teaching staff.
  - **2 marks** of these 4 manual marks will be based on your 300-word description, which must be submitted before the Milestone deadline, combined with oral assessment by your tutor during your tutorial in Week 13.
  - There will be no opportunity to regain these marks after the tutorial assessment.
  - The other **2 marks** will be from manual marking by staff after the final assignment deadline. This will assess the style and structure of your code. Feedback will also be provided.

**Warning:** Any attempts to deceive or disrupt the marking system will result in an immediate zero for the entire assignment. Negative marks can be assigned if you do not follow the assignment specification or if your code is unnecessarily or deliberately obfuscated.

## Academic Declaration

By submitting this assignment you declare the following:

*I declare that I have read and understood the University of Sydney Student Plagiarism: Coursework Policy and Procedure, and except where specifically acknowledged, the work contained in this assignment/project is my own work, and has not been copied from other sources or been previously submitted for award or assessment.*

*I understand that failure to comply with the Student Plagiarism: Coursework Policy and Procedure can lead to severe penalties as outlined under Chapter 8 of the University of Sydney By-Law 1999 (as amended). These penalties may be imposed in cases where any significant portion of my submitted work has been copied without proper acknowledgment from other sources, including published works, the Internet, existing programs, the work of other students, or work previously submitted for other awards or assessments.*

*I realise that I may be asked to identify those portions of the work contributed by me and required to demonstrate my knowledge of the relevant material by answering oral questions or by undertaking supplementary work, either written or in the laboratory, in order to arrive at the final assessment mark.*

*I acknowledge that the School of Computer Science, in assessing this assignment, may reproduce it entirely, may provide a copy to another member of faculty, and/or communicate a copy of this assignment to a plagiarism checking service or in-house computer program, and that a copy of the assignment may be maintained by the service or the School of Computer Science for the purpose of future plagiarism checking.*