# COSC 264 Assignment

Andreas Willig
Dept. of Computer Science and Software Engineering
University of Canterbury

July 17, 2019

## 1 Administrivia

This assignment is part of the COSC 264 assessment process. It is worth 10% of the final marks. It centers around socket programming using the Python programming language. Your program should be a text mode program that can run from the command line.

You will need to find out some details about socket programming on your own. The Internet offers gazillions of examples about socket programming in all sorts of languages, including Python, and it is important for you to get some practice in using that resource. For Python, a good starting point is to familiarize yourself with the `socket` module, which offers many low-level operations on sockets.

Please use the "Question and Answer Forum" on the Learn platform for raising and discussing any unclear technical issues. Please **do not** send emails with technical questions directly to me or the tutors, instead use the learn forum. This way other people can benefit from the question (and the answer).

Another piece of advice: socket programming is systems programming, and there can be sometimes subtle differences in the precise operation of socket calls between different (versions of) operating systems. I strongly suggest that you develop the assignment under Linux, as this is what we are most familiar with.

## 2 Plagiarism Warning

Your submissions are logged and originality detection software will be used to compare your solution with other solutions. Dishonest practice, which includes

- letting someone else create all or part of an item of work,
- copying all or part of an item of work from another person with or without modification, and

- allowing someone else to copy all or part of an item of work,

may lead to partial or total loss of marks, no grade being awarded and other serious consequences including notification of the University Proctor.

You are encouraged to discuss the general aspects of a problem with others. However, anything you submit for credit must be entirely your own work and not copied, with or without modification, from any other person. If you need help with specific details relating to your work, or are not sure what you are allowed to do, contact your tutors or lecturer for advice. If you copy someone else's work or share details of your work with anybody else, you are likely to be in breach of university regulations and the Computer Science and Software Engineering department's policy. For further information please see

- Academic Integrity Guidance for Staff and Students
  `www.canterbury.ac.nz/ucpolicy/GetPolicy.aspx?file=Academic-Integrity-Guidance-For-Staff-And-Students.pdf`
- Academic Integrity and Breach of Instruction Regulations in the University Calendar
  `www.canterbury.ac.nz/regulations/general-regulations/academic-integrity-and-breach-of-instruction-regulations/`

You will have to sign a plagiarism declaration upon submission of your assignment report.

## 3 Problem Description

You will write a client and a server application which allows a client to download a file of its choosing from the server. The client and the server will communicate through stream / TCP sockets, exchanging both control and actual file data.

### 3.1 Server

The server is a command line application and will operate as a TCP server. It will accept one parameter, to be read from the command line. This parameter is the port number to which the server will bind the socket on which it accepts incoming file transfer requests. The port number should be between 1,024 and 64,000 (including). If it is not, then the server should print an error message and exit. Otherwise, the server will perform the following steps:

- The server creates a socket and attempts to bind it to the port number given on the command line. If this does not work, then the server should print an error message and exit.
- The server then calls (the Python equivalent of) `listen()` on the socket. If this does not work, then server should print an error message, close the socket and exit.
- The server enters an infinite loop, in which the server performs the following steps:

  - It first `accept()`s a new incoming connection. Recall that `accept()` returns a new socket for that connection. For logging purposes, the server prints a message which indicates the current time and the IP address and port number of the client from which the incoming connection originated. You will have to find out how to obtain that data from the socket returned by `accept()`.
  - It then tries to read a `FileRequest` record from the connection. The record includes the filename to be read (see Section 3.3 for a detailed description of the record format). More precisely, the server attempts to read as many bytes from the socket as are needed for the `FileRequest`, stores them in a byte array and then checks the validity of the `FileRequest` record (see Section 3.3). If the `FileRequest` record is not valid, or if the required number of bytes for the complete `FileRequest` record cannot be read from the socket within a maximum of one second after `accept()` has returned with a socket (see Section 3.4), then the server prints an appropriate error message, closes the socket obtained from `accept()` and goes back to the start of the loop.
  - If the `FileRequest` record is correct and contains a filename, the server tries to open the file for reading.
    * If the file does not exist or cannot be opened, the server sends an appropriate `FileResponse` message back to the requesting client, closes the socket obtained from `accept()`, prints an informational message and goes back to the start of the loop.
    * If the file can be opened for reading, the server sends an appropriate `FileResponse` message back to the client, followed by the actual contents of the file (the server should count the actual number of bytes transferred). Once the file transfer has been completed, the server closes the file, it closes the socket obtained from `accept()`, prints an informational message which includes the actual number of bytes transferred, and goes back to the start of the loop.

Note that the file can be of any size and it needs to be transferred completely and without errors, and without any un-necessary gaps. The file is to be treated as a binary file (see Section 3.5). To quickly check whether the received file on the client side is completely identical to the file on the server side, you can use the `md5sum` command line tool (see the `man` page for this command).

## 3.2 Client

The client is a command line application and will operate as a TCP client. The client will accept three parameters, all to be read from the command line:

- The first parameter is either a string giving an IP address in dotted-decimal notation (e.g. "130.66.22.212"), or it is the hostname of the computer running the server (e.g. "fileserver.mydomain.nz"). The client will attempt to convert this parameter to an IP address using the Python equivalent of the `getaddrinfo()` function. If this conversion fails (e.g. because the hostname does not exist or an IP address given in dotted-decimal notation is not well-formed) then the client should print an error message and exit.

- The second parameter is the port number to use on the server. The port number should be between 1,024 and 64,000 (including). If it is not, then the client should print an error message and exit.
- The third parameter is the name of the file that the client wishes to retrieve from the server. The client should check whether the indicated file actually exists and can be opened locally. If this is the case, then the client should print an error message and exit, to avoid over-writing local files of the same name.

If there are fewer or more than three parameters on the command line, the client should print an error message and exit. If the parameters are all ok, the client will go through the following steps:

- The client creates a socket. If this does not succeed, then the client prints an error message and exits.
- Next, the client calls `connect()` to connect with the server, using the IP address and port number inferred from the command line parameters. If the `connect()` call does not succeed, then the client closes the socket, prints an error message and exits.
- Next, the client prepares a `FileRequest` record and sends this to the server over the socket.
- Next, the client will read a `FileResponse` record from the server. More precisely, the client attempts to read as many bytes from the socket as are needed for the `FileResponse`, stores them in a byte array, and checks the validity of the `FileResponse` record. If it is not valid or if there is a gap of more than one second while reading the `FileResponse` record (see Section 3.4), then the client prints an error message, closes the socket and exits.
- If the `FileResponse` record is valid, its contents are checked:
  - If the `FileResponse` record indicates that no file data is following (e.g. because the file does not exist on server side), then the client prints an appropriate informational message, closes the socket and exits.
  - Otherwise, the client performs the following steps:
    - It opens the file with the indicated filename locally for writing. If that does not work, the client prints an error message, closes the socket and exits.
    - The client then processes the file data. The file data is read in blocks of up to 4,096 bytes size, such that each block is first transferred into a

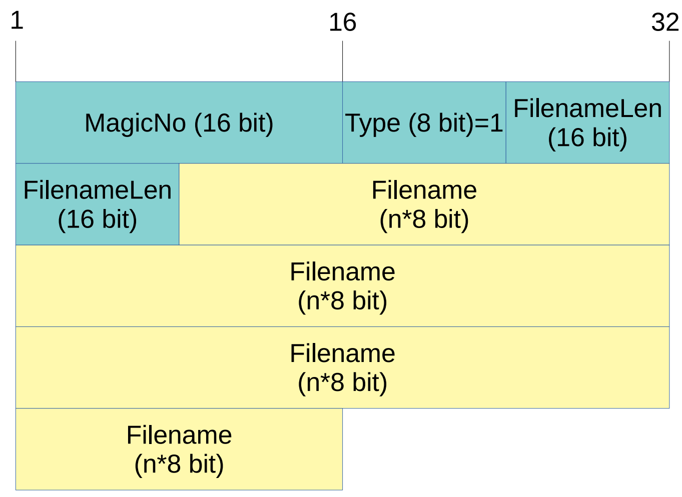
byte array, and from there written to the file. While doing this, the client counts the number of bytes received. This continues until no further data is received from the socket (how will you know?). If an error occurs when reading the data from the socket or when writing it to a file, or if there is a gap in the received data of more than one second, then the client prints an error message, closes the socket, closes the file and exits. It is also important to check that there are **exactly** as many data bytes as indicated in the 'DataLength' field. If there are more or fewer data bytes, error processing shall be carried out.

∗ At the end of the file transfer the client prints an informational message (which includes the number of bytes received), closes the file, closes the socket and then exits.

## 3.3 Message Formats

### 3.3.1 The `FileRequest` Record

The format of the `FileRequest` record is shown in the following figure:



It consists of the following fields:

- The first 16 bit field 'MagicNo' is taken up by a magic number, which needs to have the value `0x497E` in network byte order. This is a simple safeguard to check whether the received data could actually be a `FileRequest` record.

- The next 8 bit field 'Type' needs to contain the fixed value '1' for the `FileRequest` record.

- The next 16 bit field 'FilenameLen' contains the length of the following filename as a number of bytes. The length value is given in network byte order, and for a valid `FileRequest` record the length field should at least be one and should be no larger than 1,024. Denote by $n$ the value stored in the 'FilenameLen' field.

- Finally, the `FileRequest` record contains $n$ bytes for the actual filename.

We refer to the first five bytes of the record (comprising the 'MagicNo', 'Type' and 'FilenameLen' fields) as the **fixed header**. The `FileRequest` record is being sent from the client to the server. To receive and process a `FileRequest` record the server performs the following steps:

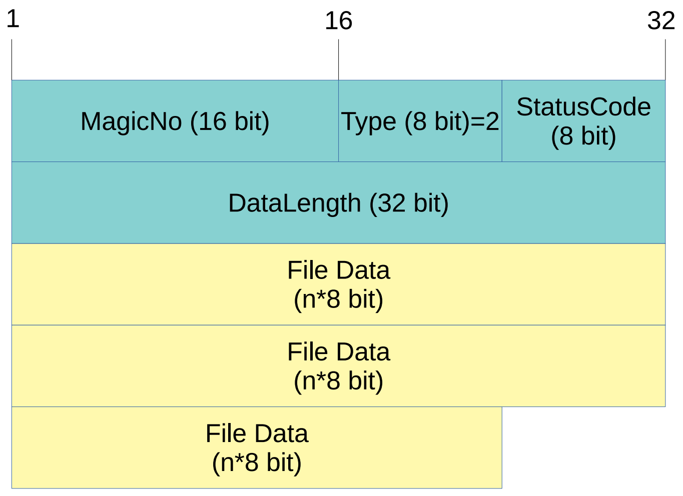
- First the server attempts to read the five bytes of the fixed header into a byte array. If that is not possible without gap (see below), then the server concludes that the received `FileRequest` is erroneous and performs error processing as described in Section 3.1 (i.e. printing an error message on the terminal, closing the socket obtained from `accept()` and going back to the start of the loop). If this is possible, then the server performs the following checks:
  - The contents of the 'MagicNo' field must equal `0x497E`.
  - The contents of the 'Type' field must equal 1.
  - The contents of the 'FilenameLen' field must be at least one and must not be larger than 1,024.

  If any of these conditions is not true, then the server concludes that the received `FileRequest` is erroneous and performs error processing.

- Then the server attempts to read exactly $n$ further bytes from the `FileRequest` record. These bytes are read into a byte array. If reading from the socket is not possible without gap (see Section 3.4), then the server concludes that the received `FileRequest` is erroneous and performs error processing as above. If the server reads fewer than $n$ bytes or more than $n$ bytes, then again the server concludes that processing failed and performs error processing.

### 3.3.2 The `FileResponse` Record

The format of the `FileResponse` record is shown in the following figure:



It consists of the following fields:

- The first 16 bit field 'MagicNo' is taken up by a magic number, which needs to have the value `0x497E` in network byte order. This is a simple safeguard to check whether the received data could actually be a `FileResponse` record.
- The next 8 bit field 'Type' needs to contain the fixed value '2' for the `FileResponse` record.
- The next 8 bit field 'StatusCode' contains information about whether the server was able to successfully open the requested file for reading. It contains the value '0' if the file does not exist on the server or the server was not able to open it for reading, and it contains a '1' if the server was able to open it for reading, in which case the remaining `FileResponse` record contains actual file data.
- The next 32 bit field 'DataLength', stored in network byte order, contains the length of the requested file and consequently also tells how many data bytes follow. If the value in the 'StatusCode' field is 0, then the client should ignore the value of this field. The server should set it to zero in this case. Otherwise, if the 'StatusCode' is 1, then this field can contain any unsigned integer value, including 0 (which corresponds to an existing but empty file).
- Finally, the 'FileData' field contains as many bytes as indicated in the 'DataLength' field if the 'StatusCode' is 1, otherwise it contains no data bytes.

We refer to the first eight bytes of the `FileResponse` record (comprising the 'MagicNo', 'Type', 'StatusCode' and 'DataLength' fields) as the **fixed header**. The `FileResponse` record is being sent from the server to the client (in response to a `FileRequest` record). To receive and process a `FileResponse` record the client performs the following steps:

- First the client attempts to read the eight bytes of the fixed header into a byte array. If that is not possible without gap (see Section 3.4), then the client concludes that the received `FileResponse` is erroneous and performs error processing as described in Section 3.2 (i.e. printing an error message on the terminal, closing the socket and exiting). If this is possible, then the client performs the following checks:
  - The contents of the 'MagicNo' field must equal `0x497E`.
  - The contents of the 'Type' field must equal 2.
  - The contents of the 'StatusCode' field must be either 0 or 1.

  If any of these conditions is not true, then the client concludes that the received `FileResponse` is erroneous and performs error processing.
- Then, if the value of the 'StatusCode' field is 1, the client attempts to read and process the actual file data bytes as described in Section 3.2.

## 3.4 Reading from a Stream/TCP Socket

For a stream socket, when a sending station sends 50 bytes by writing them in one go into its socket buffer, the underlying TCP protocol implementation has full discretion over when the data is sent, how many "packets" it sends and what the size of these packets is. So theoretically, the TCP implementation on the sender can choose to send the first three bytes immediately, then wait three seconds, send the next thirteen bytes, then wait another five seconds and then send the remaining 34 bytes. In practice such large time gaps are unlikely and point to problems in the underlying network. It is required that you detect gaps of one second or more and abort operation of either client or server should that occur (of course, when aborting you should not forget to print an informative error message and clean up / return all the resources like closing sockets, files etc).

To achieve this, as a preparation you will have to set a socket option on the socket you want to read data from, this can be done using the function `setsockopt()`, with the specific option `SO_RCVTIMEO`, or you can use the Python library function `settimeout()` on the socket. This should be done immediately after creating the socket (i.e. after a `socket()` or `accept()` call). Then, when actually reading data from a socket, e.g. using `recv()` or `read()`, in Python you will have to check if this operation throws an exception. If it does, then you will need to determine the reason for the exception (a reading timeout on the socket could be one of the reasons) and print an informational message, then do cleanup and exit.

## 3.5 Data Representation

There are a few things to consider regarding data representation:

- The filename that the client reads from the command line will be treated by Python as a string. Strings in Python are represented using the UTF-8 character encoding system, and in this system a printed character may require a variable number of bytes to store in memory. Hence, a string which looks like having $m$ characters may in fact need a number of $n \geq m$ of bytes to represent it, and your client program needs to transmit all $n$ bytes. To achieve this, you can use the `encode` method in Python to convert a string to a byte array. For example:

  ```
  ...
  mystring = "hello world"
  mybytes = mystring.encode('utf-8')
  ...
  ```

  and then you can use the `len` function to find out the length $n$ of the byte array `mybytes`. You will transmit the complete byte array in the 'Filename' field of the `FileRequest` record, and the value of $n$ will be filled into the 'FilenameLen' field.

- All 16- or 32-bit fields in the fixed headers of the `FileRequest` and `FileResponse` records are encoded in network byte order. For a 16-bit field this means that the first or leftmost eight bits are occupied by the highest-valued bits and the last or rightmost eight bits are occupied by the lowest-valued bits. Similarly for a 32-bit field.

- The file contents should be treated as bytes throughout. In other words, when the server reads bytes from the file, they should be stored in a byte array, and then the byte array contents should be sent over the socket. Similarly, on the client side the file data bytes should be storec in a byte array, and from there written to the file.

## 4 Deliverables

**Each student** has to submit **a single pdf file** which includes the following items:

- A cover sheet with your name and student-id.
- A listing of your source code. We would appreciate if you use a pretty printer. **Do not use screenshots!**
- You have to print the plagiarism declaration form from the learn page of COSC 264, sign it, scan it, convert the scanned form into a pdf and include this into your submission.

**Warning**:

- Submissions that are not in pdf format or which contain more than one file automatically receive 0 marks!!
- Submissions which do not include the **signed** plagiarism declaration form automatically receive 0 marks!!

The pdf file has to be submitted via the `learn` page of COSC 264 (see `https://learn.canterbury.ac.nz/course/view.php?id=542&home=1`). Please submit no later than **Sunday, August 18, 2019, 11:59 pm**. Late submissions are **not** accepted, except through the special consideration process.

## 5 Marking

Marking will be based on the source code. In particular, we will mark it for its ability to produce the right results (e.g. is packet processing correctly implemented, are all the right socket calls there, is the gap detection implemented correctly), and for the amount of error / consistency checking you do – if we find that you use system-/socket calls without any error checking or that you do not check incoming data / records for correctness, we will apply deductions. We will check whether you have returned all resources (sockets, files) to the operating system, wherever sensible. We will also have an eye on style, and may apply deductions if your code is particularly ugly or messy.