COMPX234-A4: Assignment 4 specification

Integrity. Like in Assignment 3, you will be using **GitHub** to show evidence of the step-by-step development and demonstrate you understand the work.

You are **not allowed to** use LLM prompting for code generation. You must explain clearly each step of your design and implementation, with the steps documented via commits in GitLab. When the evolution is not demonstrated via the commit history, the student may get 0 marks for the assignment and/or be called for a viva voce to explain the code. Also, practical knowledge of the implementation needs to be demonstrated in the Exam. Any suspected cheating cases will be passed to the Student Disciplinary Committee for investigation. We acknowledge that code generation is a very powerful tool, but at this moment, students need to be able to develop their work without that kind of assistance. (For the record, we also generated versions of this assignment using different LLMs and we will be vigilant for signs of LLM usage.) Instead, get help from humans: ask the lecturer who is here to assist your work on the assignment.

Dates and submission. The specification is being released on 25 May 2025. The submission deadline is 14 June, 2025, which corresponds to *three weeks work* (and 24 hours of practical work in total). The submissions will be taken from GitHub. We will capture the most recent commit *prior* to when the assignment marking begins, which can be at any moment after the deadline (if you delay, you assume the risk).

Summary. In this assignment, you are going to develop a client/server networked program that implements mechanisms for reliable transmission over an unreliable channel (using UDP). The server provides files to clients: a client sends a request specifying the name of the file to download. The server transmits block by block the data to the client requesting the file; the file is treated as binary. The server will implement the sender side of a reliable protocol, and the client, the receiver side. Both sides will work together to ensure that the file is received correctly by the client. Like in Assignment 3, the server needs to deal with multiple clients at the same time. We will provide some code to get you started.

Synchronous client. The client downloads a single file at a time. This is what we call “synchronous behaviour”, which simplifies the client implementation. That is, the client is *not* multi-threaded. The client can download multiple files, but sequentially, and then exits. There is no concept of session.

Multi-threaded server. As stated above, the server needs to handle requests from multiple clients at the same time. For this reason, the server will use threads, each thread being created to deal with a *single request from a client*. This allows the server implementation to be much simpler than if having to interleave requests from multiple clients. The server will spawn a new thread whenever a new file request is received from a client. When the file has been successfully transmitted by the server, the corresponding thread at the server terminates.

Protocol. The protocol to be implemented uses ASCII text to encode the requests and responses in messages as follows. The client requests to download a file by sending a text message in the following format:

- DOWNLOAD **<filename>**

The server responds with a message (in text format) which can be either

- OK **<filename>** SIZE **<size_bytes>** PORT **<port_number>**
- ERR **<filename>** NOT_FOUND

The error NOT_FOUND occurs when the server does not have the file requested by the client. Note that in UDP, messages can be lost. So, if either the download message or the response (OK or ERR) is lost, then the client timeouts and retransmits the DOWNLOAD request (consider creating in the client a function "sendAndReceive" that will transmit a packet, set a timeout, and retransmit if a response message is not received within time; you can call this function when sending any request to the server).

If the message was OK**,** the message contains the size of the file (in bytes) and a port number (in the range 50000-51000) that will be used by the server to transmit this file. The server creates a new UDP socket (with this port) to receive the FILE requests from this client; you can thus call it e.g. a "clientSocket". This is necessary because the original socket in the server needs to be reserved for new file requests of other clients and cannot be mixed with data.

So, the client starts a cycle in which it repeatedly sends a request to the server specifying the byte range.

- FILE **<filename>** GET START **<start>** END **<end>**

Assuming the server receives the request, it replies with the corresponding data:

- FILE **<filename>** OK START **<start>** END **<end>** DATA **<encoded_data>**

Both messages start with the keyword FILE and are followed by the filename. Each packet transmitted by the client and response from the server contain information about the byte range (the first and last byte in the packet). For example, a packet with 1000 bytes of data whose initial sequence is 3000 will have 3000 and 3999 as start and end, respectively. This is equivalent to a sequence number, allowing the client to detect when data is duplicated, and the server to transmit packets with different amounts of data. A message will have up to 1000 bytes of binary data, which becomes 1336 bytes when encoded as a string in Base64. When the client has received all data, it sends a termination message to the server. At the client side, the file has been saved and can be closed.

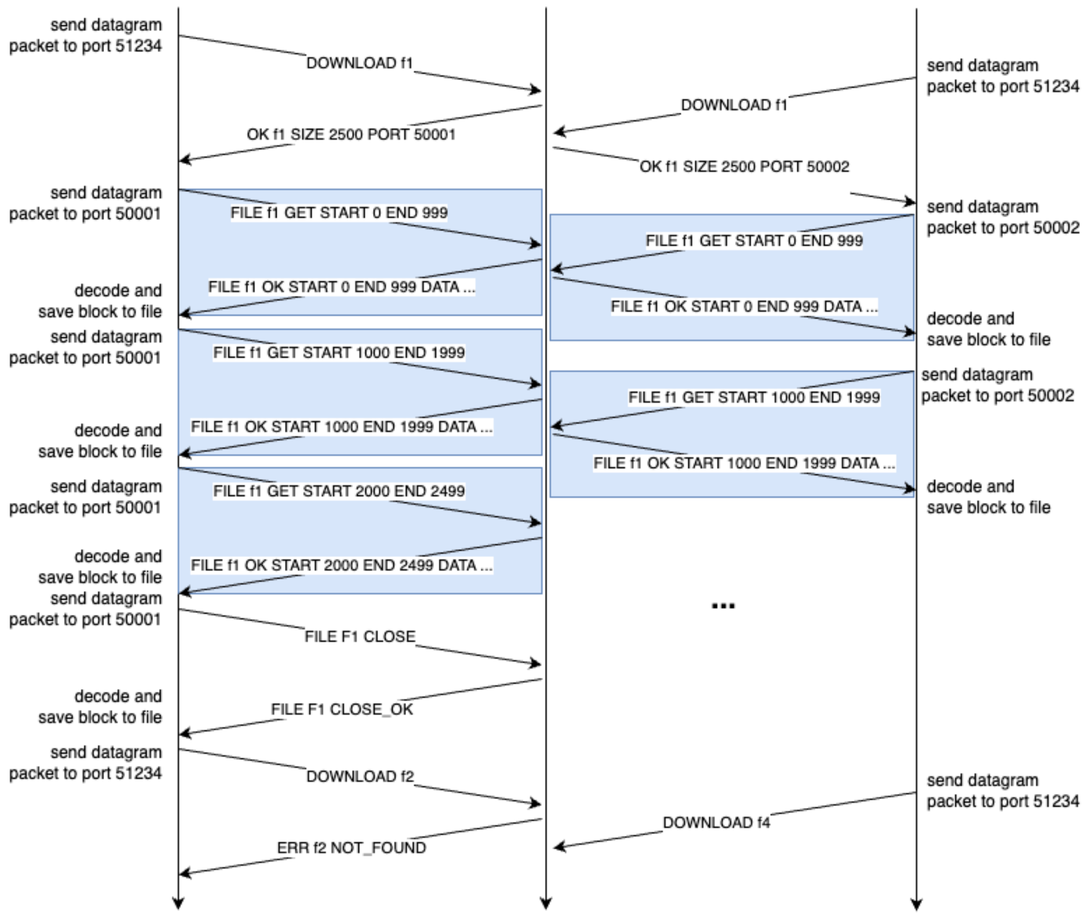
- FILE **<filename>** CLOSE

The server responds to the close message by sending back:

- FILE **<filename>** CLOSE_OK

Note that this protocol has a problem and it will not work if the CLOSE_OK is lost!

To illustrate the workings of the protocol, consider the time diagram with two clients asking the server to download files f1, f2... The server is running and waiting for download requests on port 51234.


send datagram packet to port 51234
DOWNLOAD f1
send datagram packet to port 51234
DOWNLOAD f1
OK f1 SIZE 2500 PORT 50001
OK f1 SIZE 2500 PORT 50002
send datagram packet to port 50001
FILE f1 GET START 0 END 999
send datagram packet to port 50002
FILE f1 GET START 0 END 999
decode and save block to file
FILE f1 OK START 0 END 999 DATA ...
FILE f1 OK START 0 END 999 DATA ...
decode and save block to file
send datagram packet to port 50001
FILE f1 GET START 1000 END 1999
send datagram packet to port 50002
FILE f1 GET START 1000 END 1999
decode and save block to file
FILE f1 OK START 1000 END 1999 DATA ...
FILE f1 OK START 1000 END 1999 DATA ...
decode and save block to file
send datagram packet to port 50001
FILE f1 GET START 2000 END 2499
decode and save block to file
FILE f1 OK START 2000 END 2499 DATA ...
...
send datagram packet to port 50001
FILE F1 CLOSE
decode and save block to file
FILE F1 CLOSE_OK
send datagram packet to port 51234
DOWNLOAD f2
send datagram packet to port 51234
DOWNLOAD f4
ERR f2 NOT_FOUND

Reliability mechanisms. The protocol uses a *stop-and-wait* approach. The client requests a chunk of data and gets the corresponding data, which works as an acknowledgment. If the client does not get a response in time, it can retransmit the request. If the error persists, the client will timeout and retransmit a number of times (5), until it aborts and gives up. If consecutive retransmissions are needed, the client should make the timeout longer and longer every time, in case the timeout was set too short.

Implementation. The implementation uses UDP sockets. The server creates an UDP socket on a port that is informed via the command line, like in Assignment 2. For example,

```
mbarcell@lab-rg06-02:~$ java UDPserver 51234
mbarcell@lab-rg06-02:~$ python3 UDPserver.py 51234
```

As mentioned above, the server uses threads. The server creates a thread to deal with each new DOWNLOAD request. The thread will operate on a new UDP socket, bound to (associated with) a high port (50000-51000) that will be randomly chosen by the server. The thread opens the file and waits for chunk requests from the client. This chosen port is informed to the client.

The client, on its turn, is run with three arguments in the command line:

- **hostname** where the server resides (can be "localhost" if client and server are in the same host);
- **port number** to send a download request to the server;
- **filename** of the download list file; this is a text file containing a list of filenames to be downloaded, one per line.

For example, the client might be run using one of the command lines below:

```
mbarcell@lab-rg06-01:~$ java UDPclient lab-rg06-02 51234 files.txt
mbarcell@lab-rg06-01:~$ python3 UDPclient.py lab-rg06-02 51234 files.txt
```

where "files.txt" is a list of files to be requested from the server, such as:

```
07.pdf
08.pdf
IM-NZ.mov
```

These files (yours) need to be available on the server. A copy will be created locally at the client. Check the integrity by running md5. Here is the command I used when testing locally:

```
mbarcell@lab-rg06-01:~/COMPX234/COMPX234-A4/Client$ md5sum *pdf
*mov ../Server/{*pdf,*mov}
f1417a627378dfbca28444a265a480e4  07.pdf
c7b55638cdd87498aec85f0ff5f2caa8  08.pdf
9599bd118f889da03344f3e7355e2571  IM-NZ.mov
f1417a627378dfbca28444a265a480e4  ../Server/07.pdf
c7b55638cdd87498aec85f0ff5f2caa8  ../Server/08.pdf
9599bd118f889da03344f3e7355e2571  ../Server/IM-NZ.mov
```

The client creates an UDP socket to send and receive packets to/from the server. The client does *not* need to bind the socket to an address; instead, the client will inform the destination IP address and port at every packet transmission.

Outputs. There are no necessary outputs to be shown by the server or the client. Feel free to display information (especially during debugging), such as "sending file <filename> to client at <hostname:port>" and each message sent or received by client and server. At the client, display the filename being downloaded, the total size, and how much data has been downloaded (e.g. print a * on the same line for every block received).

Language. The code can be developed in either Java or Python (which may be "object-oriented" if desired). The development must be based on features explored/explained in the lectures, such as the "synchronized" keyword if developing in Java.

To give you a reference, while the assignment is not trivial, it is not large either. My implementation in Java has a client with ~150 LoC and a server with ~130 LoC, while the implementation in Python has ~130 LoC and ~105 LoC for client and server, respectively.

Assistance/help. Snippets of code will be provided in both Python and Java to help students achieve the goal of completing the assignment. I will let students know via a post on Canvas; also, please check the Canvas page where Assignment 4 files are located.

How to test your networked system. We provide example workload files to be given to the clients, as well data files stored at the server (e.g. an image library). In the test, each client requests the download of a number of files.

Follow these steps:

1) copy a set of files to the directory where the server is run (needs to be different from the client if in the same host)
2) in the directory where the client is run (different from the server), create a text file "files.txt" containing a list of the filenames of the files available in the server. Initially, there can be a single file on the list.
3) start the server using a high port, e.g. 51234
   `java UDPserver 51234` **or**
   `python3 UDPserver.py 51234`
4) start the client using as arguments "hostname port files.txt", where hostname is the host where the server is, port. I suggest you do all the testing locally on the same machine and when it all working, start testing with two different machines in the lab.
   `java UDPclient localhost 51234 files.txt` **or**
   `python3 UDPclient.py localhost 51234 files.txt`
5) as a result, all the files listed in files.txt should be downloaded from the server
6) run "`md5sum -b *`" over the set of files and compare the hash with the files in the server (they should match!).
7) repeat the test using two machines in the Linux labs. Do not forget to run the clients in different directories, otherwise the clients will write over the same files.

Advanced

8) *in different folders,* run multiple instances of clients like above, to download the same set of files every time (clients need to be in different folders!)
9) repeat the test with “md5sum -b” over the set of files.

Marking rubric.

Your project in GitHub must be named “COMPX234-A4”.

Assessment	**Marks**
development history (small steps at each commit, with detailed information)	20
code quality (documentation, indentation)	10
client/server structure correct, correct use of UDP sockets	20
can download 1 file	10
can correctly download 1 file despite losses	20
can download multiple files	10
multiple clients can download files at the same time	10