Project: Reliable Data Transfer Protocol   
over Multiple Interfaces

# Prerequisites

1. The project is to be completed **by teams of two**.
2. Make sure your code is easily readable with sufficient explanations and comments. Also, use a proper and consistent naming convention. Do not copy/paste someone else’s code.
3. You can also discuss the project among yourselves or in the Google Classroom forum. Do not ask questions over email.
4. You must enclose instructions (a readme file) on how to compile, build and run your code (along with any system requirements).

# Submission

Create a single zip file that includes everything, name the file lastname1\_lastname2\_project2.zip and upload it to Google Classroom.

# Demo

You will show the demo and explain the code to the TAs at a designated time.

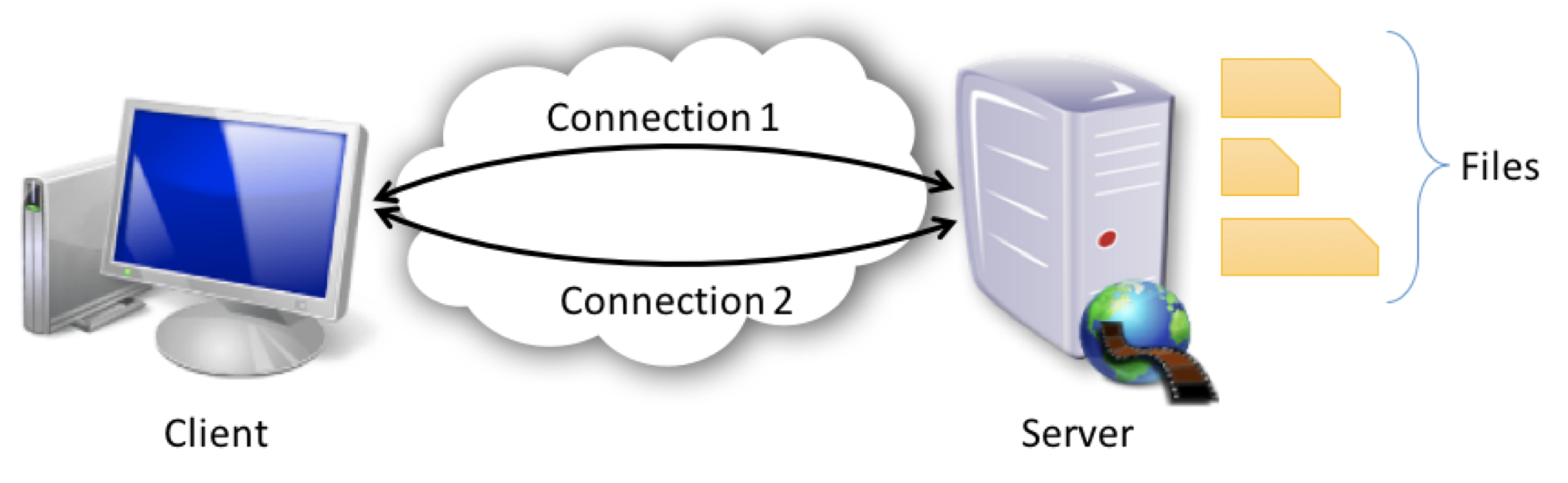
# Problem to Solve

You will design and implement a client-side protocol for reliable transfer of a given file from a server. The server side of the protocol is provided to you for testing purposes. Refer to <https://github.com/streaming-university/public-teaching-rdtp> for the server and an example client implementation. The server is dockerized for your convenience. For instructions, see the Readme file.

Your client will run on a host (your computer) with two network interfaces and it will use both interfaces simultaneously to download the file from the server. If you have a single network interface (e.g., just a WiFi interface), you can use the same interface over two different ports.

Your goal is to develop the fastest protocol that will download any requested file in the shortest amount of time. Your protocol will be built on top of UDP (you will have to design and implement the reliable data transfer protocol using UDP).

The scenario can be sketched as follows:



On the server side, we will apply throttling, rate limiting and/or packet loss to the transmission to induce problems on each connection. The client must overcome these issues and successfully transfer the whole file. If one connection is slower than the other one, the client should try to transfer more data on the faster connection. However, note that connection speeds (and loss rates) will vary over time, meaning that you need to monitor what is happening with your connections and alter your strategy accordingly. The server will host multiple files of different sizes (like 100 KB, 10 MB and 300 MB) and your client will be tested with each of them.

Bandwidth throttling, delay, packet loss, etc. are emulated on the server side using TC (<http://tldp.org/HOWTO/Traffic-Control-HOWTO/index.html>). See the example scripts at: <https://github.com/streaming-university/public-teaching-rdtp/tree/main/FileListServer/tc> (and feel free to modify them for testing purposes). During the grading, we may use other TC scripts, so your protocol must be robust and adaptive.

Your client should take the server’s IP address and port number pairs as an argument in the command line. E.g.,

**my\_client server\_IP1:5000 server\_IP2:5001**

When testing between two physical machines, make sure these values correspond to the server’s IP address(es) and port number(s), and that there are no NAT devices between your client and server. If you are using a single interface but different ports on the server (the case you will have when using the dockerized server), you will be using the same IP address in the command line above.

Once the client starts, it should first ask the server for the list of the files available to download. This will be achieved by sending a request with request\_type=1 (details below). Once the response is received from the server, the client should print out the file list on the screen and wait for the user’s selection.

**File List:**

**1 m1.dat**

**2 t.dat**

**3 u5.txt**

**Enter a number: \_**

The user picks one of the available files (by entering the file\_id) and the client queries the server about the size of this file by sending a request with request\_type=2.

**File 2 has been selected. Getting the size information…**

**File 2 is 63456123 bytes. Starting to download…**

At this point, the timer starts and your client tries to download this file as quickly as possible. During the download, the client should output useful data on the screen, such as transfer speed over each connection, percentage completed, elapsed time, packet loss rate experienced so far, current/average round-trip times, etc. Once the transfer is completed, a summary including the above information (and whatever else you think is appropriate to include) should be posted on the screen. The client must also compute the md5sum of the received file and output it on the screen. We will use this md5 hash to ensure the data is fully downloaded and identical to the source.

**File 2 has been downloaded in 32345 ms. The md5 hash is 595f44fec1e92a71d3e9e77456ba80d1.**

**[Other useful statistics about the download]**

If the md5 check fails, that means your data is corrupted and your protocol is broken, so you must fix it. After a download is completed, the client will go back to the first screen to ask the user to enter another file to download.

Note that any request or response may get delayed or lost, so there must be a timeout mechanism on the client side. Do not time out prematurely, as this will cause your client to use more traffic, and do not wait too long, as the download time will be longer.

# Request and Response Packet Structures:

A request packet (from the client to the server) has the following structure:

| **request\_type**  **1 byte** | **file\_id**  **1 byte** | **start\_byte**  **4 bytes** | **end\_byte**  **4 bytes** |
| --- | --- | --- | --- |

A response packet (from the server to the client) has the following structure:

| **response\_type**  **1 byte** | **file\_id**  **1 byte** | **start\_byte**  **4 bytes** | **end\_byte**  **4 bytes** | **data** |
| --- | --- | --- | --- | --- |

On success, response\_type will be set to the value of request\_type. On error, see the table below. There are three types of requests:

* **getFileList(uint8\_t request\_type):** request\_type is set to 1. Other fields are X (don’t care).

## Example Request:

| **1** | **X** | **X** | **X** |
| --- | --- | --- | --- |

## Response:

| **1** | **Total # of files** | **X** | **X** | **array of file\_descriptor** |
| --- | --- | --- | --- | --- |

The 1-byte field (uint8\_t) after the response\_type field denotes the total number of files on the server. The data field carries an array of file\_descriptor (defined below). You can assume that the response to a getFileList() request fits into one packet.

* **getFileSize(uint8\_t request\_type, uint8\_t file\_id):** request\_type is set to 2. In the response, the data field will be four bytes and it will return the size of the file with file\_id as uint32\_t. If file\_id is not a valid number, an error response (INVALID\_FILE\_ID) will be returned.

## Example Request:

| **2** | **8** | **X** | **X** |
| --- | --- | --- | --- |

## Response:

| **2** | **8** | **X** | **X** | **Size of file (4 bytes)** |
| --- | --- | --- | --- | --- |

* **getFileData(uint8\_t request\_type, uint8\_t file\_id, uint32\_t start\_byte, uint32\_t end\_byte):** request\_type is set to 3. In the response, the requested range of the file with file\_id will be returned in the data field. Note that both the start\_byte and end\_byte are included in the range.

## Example Request:

| **3** | **8** | **501** | **2500** |
| --- | --- | --- | --- |

## Response 1 (assumes maximum payload size is 1000 bytes):

| **3** | **8** | **501** | **1500** | **data (1000 bytes)** |
| --- | --- | --- | --- | --- |

## Response 2:

| **3** | **8** | **1501** | **2400** | **data (900 bytes)** |
| --- | --- | --- | --- | --- |

In this project, the maximum data size in a single packet will be assumed to be 1000 bytes. However, the client must always check the response’s start\_byte and end\_byte fields as the range of the data sent by the server could be different than the requested one. If the requested start\_byte or end\_byte value carries an invalid value, the server will return an error response (INVALID\_START\_OR\_END\_BYTE).

# Data Type Definitions:

| request\_type | 1-byte unsigned integer  0: RESERVED  1: getFileList  2: getFileSize  3: getFileData |
| --- | --- |
| file\_id | 1-byte unsigned integer |
| start\_byte | 4-byte unsigned integer |
| end\_byte | 4-byte unsigned integer |
| response\_type | 1-byte unsigned integer  0: RESERVED  1: getFileList success  2: getFileSize success  3: getFileData success  100: INVALID\_REQUEST\_TYPE  101: INVALID\_FILE\_ID  102: INVALID \_START\_OR\_END\_BYTE |
| file\_descriptor | file\_id (1-byte unsigned integer)  file\_name\0 (null terminated string) |

# 

# Clarifications

1. **First things first: Set up your development environment**

You need to come up with a smart algorithm and implement it. But your first task is to set up your development environment correctly. So please take time to get it right and then start thinking about your implementation and coding. Please do not leave it to the last days if you want to get a good grade.

1. **Our evaluation method**

You need to show the integrity (file correctness) and the performance of your application by providing some statistics. These are:

- **Integrity**: md5 verification result (true or false)

- **Performance**: Download time (the sooner, the better), file size/total downloaded bytes ratio (1.0 is perfect)

- **Information**: Size of the downloaded file, total downloaded bytes, total number of part requests, etc.

**Integrity is the first thing we will check**. Because even if your implementation is the fastest, it is meaningless if the downloaded files are corrupt.

1. **Do not use Windows**

On Windows 10 using docker, we encountered that TC does not work properly, just causing delay but not loss. This is not acceptable on our end. There are two solutions for that. Either install a virtual machine running Ubuntu or some Linux distribution on your Windows or open a tiny server on a cloud provider such as AWS, Google Cloud or Azure (You can create a free account on some of them using your student credentials).

1. **Be sure that *tc* works**

In your demos, **we expect you to demonstrate your implementation on a fully working testbed**. The TC program must work and throttle the network properly by introducing delay and loss.

When you run FileListServer (either as a docker container or in a VM), you should see the *tc* logs.

The log lines are as follows:

setting rate on lo 2 1Mbps

setting rate on eth0 4 1Mbps

setting loss on lo 2 10%

setting loss on eth0 4 10%

setting rate on lo 1 5Mbps

setting delay on lo 2 0ms

setting delay on eth0 4 0ms

This shows that tc scripts are running correctly and limiting the traffic. Make tc work first if you do not see any of those.

1. **How to use test files**

Three test files are under FileListServer/files with different sizes. You are supposed to test your implementation with at least those three files. If you want to add new files to test, copy them under FileListServer/files folder and restart the FileListServer (for docker execute docker-stop.sh and then execute docker-start.sh).

1. **IDE is your choice**

JBuilder, Eclipse or VS Code. VS Code with the Java Extension Pack on Mac runs well. Try other IDEs if you encounter a problem (compilation problem, etc.) when building the client project.

1. **Questions**

If you have any problems, please **ask in Google Classroom** instead of sending individual emails. You can schedule a call with the TAs to show your work beforehand if absolutely necessary.