



BESE-9A
COMPUTER
NETWORKS

MultiServer Downloader

PROJECT REPORT

SYED AFRAZ HASAN KAZMI (241847)

MUHAMMAD ABDULLAH (241548)

INTRODUCTION

This report extends and improves on the performance measures of single server downloader by inculcating multi-server capabilities where a user gains advantage of downloading a desired file from different hosts that have that file. This provides significant boost in download time by adapting **concurrent behavior**.

SCOPE

Limiting the scope of the project is an integral aspect of product building. A few tough decisions had to be taken primitively, such as, limiting our project to host **exactly 4 Servers capable of file transferring**. A user will be able to **manually shut down or wake up** the server during application, but however, there can at max be only 4 servers running at a given time.

LIMITATION

The only limitation of the project is that it requires a **Linux terminal** to execute, because it makes use of the **os module** of python, which is better equipped for work with Linux than Windows. However, this program is capable of running on Windows if an official tool called **Windows Subsystem for Linux(WSL)** from Microsoft Store is used. A Virtual Machine is equally applicable.

HOW TO RUN THE PROGRAM?

All we need to run the program is a **Linux terminal**. (Note: Windows Subsystem for Linux (WSL) also works)

- **Move into the directory** containing project files. (Use cd command in terminal).
- Moreover, **use different terminal windows** for server and client.

To run server.py, use

```
python3 server.py -i <STATUS INTERVAL> -n 4 -f <MP4 FILENAME/LOCATION> -a <SERVER_IP> -p 21018 27215 35416 19222 39223
```

where

- -i flag has <STATUS_INTERVAL> is an integer signifying how **frequently** server status is checked (in seconds)
- -n flag is supposed to signify the number of servers and it **should be kept as 4**. This number **cannot** be changed because the scope of the project does not permit it.
- -f flag has <MP4 FILENAME/LOCATION>. This should contain the **filename with its extension** if file exists in the project directory. If not, the **absolute path with filename and extension** should be provided. (Note: This is the file that will be segmented)
- -a flag has <SERVER_IP>, the **IP Address** of our current machine
- -p flag has **5 port numbers**, one for each server. (Note: Server 5 is not a file transfer server, it is actually a server needed to send status updates of file transfer servers to client)

Example:

```
/Afraz ➤ cd PycharmProjects
/Afraz/PycharmProjects ➤ python3 server.py -i 2 -n 4 -f Star.mp4 -a 127.0.0.1 -p 21018 27215 35416 19222 39223
```

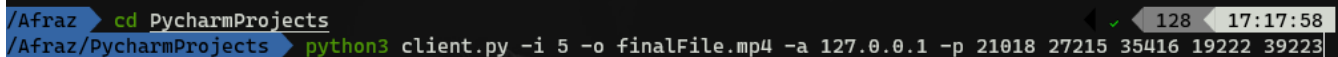
To run client.py, use

```
python3 client.py -i <METRIC_INTERVAL> -o <MP4 FILENAME/LOCATION> -a <SERVER_IP> -p 21018 27215 35416 19222 39223
```

where

- -i flag has <METRIC_INTERVAL> which is an integer signifying how **frequently** metric reporting is displayed (in seconds)
- -o flag has <MP4 FILENAME/LOCATION>. This should contain the **filename with its extension of output file** if file is to be created in the project directory. If not, the **absolute path with filename and extension** should be provided. (Note: This is the file that will be formed after recombination)
- -a flag has <SERVER_IP>, the **IP Address of our Server** machine. (Same IP if both client and server running on the same machine)
- -p flag has **5 port numbers**, one for each server we want to connect to. (Note: Server 5 is not a file transfer server, it is actually a server needed to send status updates of file transfer servers to the client)

Example:

A terminal window screenshot showing the command to run client.py. The prompt is /Afrax and the current directory is PycharmProjects. The command is python3 client.py -i 5 -o finalFile.mp4 -a 127.0.0.1 -p 21018 27215 35416 19222 39223. The terminal shows a green checkmark and the command executed successfully. The terminal also shows the file name 128 and the time 17:17:58.

```
/Afrax ➤ cd PycharmProjects  
/Afrax/PycharmProjects ➤ python3 client.py -i 5 -o finalFile.mp4 -a 127.0.0.1 -p 21018 27215 35416 19222 39223
```

IMPLEMENTATION DETAILS

The **key features** of this project are as follows,

- 1) **Setting up multi-servers**
- 2) **Segmentation**
- 3) **Recombination**
- 4) **Handling Server Failure**
- 5) **Load Balancing**
- 6) **Download resuming**
- 7) **Server-side input/output**
- 8) **Client-side output/metric reporting**
- 9) **Safely Exiting program**

Setting up multi-servers

The **Multiprocessing** python module is used to concurrently run different instances of the Server object. Each processor core runs a separate process where in no condition does a server communicate or even know about the existence of the other servers. There are **4 Servers** dedicated to **file transferring** and there is a **master server** called the **Check server (Server 5)** that monitors the status of all other servers. The client side simply connects to ports of these servers and the program communication begins.

Segmentation

Segmentation is done by **one** server at the start of the program and all the segments are stored in the project directory where **every server** will be given **read-access** to these segments.

The **total number** of segments to be formed was **chosen** to be an optimum value of **12** (number found by applying maximum flow algorithm)

Methodology – The size of the main file is divided by 12 to give the number of bytes each segment is going to be. This number is rounded up to ensure data integrity. Next, these segments are one by one created and stored in the main project directory. Note: The last segment or the 12th segment is shorter in size compared to other segments because segment 1 to 11 were rounded up in size.

Example:

Suppose a file has a size of 245 bytes. This divided by 12 segments gives us the number 20.416, but since it is unsafe to store that many bytes in a segment, we round up the value to be 21 instead. Now every segment carries a size of 21 bytes(except the last segment).

So, 21 bytes x 11 segments form 231 bytes of data and therefore the 12th segment has to be of 14 bytes to complete the file which was of 245 bytes. (Note: This is just an unreal example to model the segmentation, only for explanatory purposes)

Recombination

Recombination is done at the very end on the client-side when it is **certain that all segments have been received correctly**. The recombination function is integrated with the metric reporting function as it has to continuously check whether all segments are received before triggering the recombination.

Methodology – Simply a new mp4 file is created and opened in **append** mode. Next, we traverse over all the received segments and incrementally append their data to our final file in **chronical order**.

Handling Server Failure

As mentioned earlier, we set up 4 servers capable of file transferring and behind the scenes we deploy a 5th server which continuously runs and is responsible of monitoring the status of the other 4 servers and sending regular status updates to client side. Client takes note of these updates and if in case a server fails or is shut down on the server-side, client is **immediately** notified, after which it does **load balancing** so that the communication is not broken at any cost.

Load balancing

Client has **two types** of lists it maintains in real-time for **each** server.

X – Lists that store the ID number of confirmed received segments.

Y – Lists that store the ID number of the required segments(the ones yet to be received)

Also, there are **4 X lists** and **4 Y lists** due to **4 servers**.

Initially Y may contain IDs of all segments it requires from the particular server and accordingly **X will be empty at the start**. Just as the segments are received, the IDs start **filling up in the X lists** and **Y starts reducing in size**. Subsequently, **X becomes full and Y becomes empty**.

Now when a **server fails or shuts down** and the client is successfully notified by the help of our Check Server (Server 5), client **instantly runs the load balancing** function where it first creates new Y lists for the alive servers and it discards the old ones. Then, in order to **avoid requesting duplicates**, we **subtract** the IDs in X from the new Y lists. Hence, our new servers only request the segments not yet received.

Download resuming

When all servers are shut down manually, the **client does not abort operations**, rather it continues waiting for any server to send any more segments given it has not yet received all of them. When the download is resumed, the first thing client does is, it does load balancing to update the Y lists which are then used to receive the segments.

Server-side input/output

The server-side continuously displays its output and provides options to manually shut down or wake up any server. To obtain this functionality, **threading module** has been used. So basically, the **output is running on a thread**. The reason we **did not** simply use the multiprocessing module to run the output function is because we did not want to **overwhelm the processor cores** which would lead to **degradation in overall performance**. Likewise, Server 5 is also running on a separate thread. The output looks as follows,

```
-----  
MultiServer Downloader  
-----  
Server 1 at Port: 21018 Status: Dead To shut down/wake up enter 1  
Server 2 at Port: 27215 Status: Alive To shut down/wake up enter 2  
Server 3 at Port: 35416 Status: Alive To shut down/wake up enter 3  
Server 4 at Port: 19222 Status: Alive To shut down/wake up enter 4  
To quit, enter -1:  
  
Enter your Choice: |
```

Client-side output/metric reporting

The client-side displays its output at **regular intervals** specified by the user **using the -i flag**. Metric reporting takes help of the **X Lists** to know how much data has been received. **Speed of each server** is found using the **time module** to calculate **how much time it takes to receive a segment**. Using this information, we calculate **data received per second (Kb/s)**, the **total speed** is the **average of individual server speeds**.

```
Server1: 0.0/0.0, download speed: 0kb/s  
Server2: 2880.91/2880.91, download speed: 0kb/s  
Server3: 2880.91/2880.91, download speed: 18005.67kb/s  
Server4: 2880.91/2880.91, download speed: 36011.35kb/s  
Total: 8642.72, download speed: 36011.35kb/s
```

Safely exiting program

In order to be **memory efficient**, we had to **ensure** that the program had a safe exiting routine as it is a **crucial** step of every software.

On **Server-side**, this is achieved by using **Exception Handling techniques of Python** and again by taking help of our Check Server (Server 5) which is, this time notified by the client that all segments are received and communication can finally halt.

Upon this alert, the **Server 5 raises an exception** which is caught at an outer point where the threading function is. Upon **catching the exception**, our program knows its time to shut down all services and abort the program and hence it calls the final function **exitProgram()** which is accountable for **safely terminating** all the on-going multi-processes.

Program can also **exit** if the user wishes to **enter -1** at the server-side.

As for the **Client-side**, once the recombination stage has occurred, the **os module** is used to **safely terminate** the program.