### Internet Protocols – Programming Assignment 1

**Team members:** Swetha Sairamakrishnan (ssairam)**,** Naveen Jayanna (njayann)

|  |  |  |
| --- | --- | --- |
| **HTTP 1.1** | | |
| Tasks | Swetha Sairamakrishnan | Naveen Jayanna |
| Research, identifying, and understanding the operation of the packages that implement the protocol | 40% | 60% |
| Testing on different setups - WiFi, ethernet | 50% | 50% |
| Running tests and benchmarking for different scenarios | 50% | 50% |

|  |  |  |
| --- | --- | --- |
| **HTTP2.0** | | |
| Tasks | Swetha Sairamakrishnan | Naveen Jayanna |
| Research, identifying, and understanding the operation of the packages that implement the protocol | 60% | 40% |
| Measuring application layer data | 50% | 50% |
| Running tests and benchmarking for different scenarios | 50% | 50% |

|  |  |  |
| --- | --- | --- |
| **GRPC** | | |
| Tasks | Swetha Sairamakrishnan | Naveen Jayanna |
| Understanding the working of GRPC protocol and its implementation in python | 60% | 40% |
| Measuring application layer data | 50% | 50% |
| Running tests and benchmarking for different scenarios | 50% | 50% |

|  |  |  |
| --- | --- | --- |
| **BitTorrent** | | |
| Tasks | Swetha Sairamakrishnan | Naveen Jayanna |
| Research on operation of the packages that implement the protocol | 40% | 60% |
| Learning the difference between private and public trackers and their usage. | 50% | 50% |
| Running tests and benchmarking for different scenarios | 50% | 50% |
| Using different torrent clients to understand the creation of a .torrent file. | 50% | 50% |
| Measuring application layer data | 50% | 50% |

# Network environment setup:

All of the below experiments were conducted by connecting two systems using a Ethernet cable to bring them onto the same LAN. The server and client scripts were executed on both systems alternatively (A and B files) and the time was measured accordingly on the client side. For the bittorrent experiment we simulated 4 systems by running two peers on each system.

# HTTP 1.1

File transfer using HTTP 1.1 protocol was tested using the python packages http which comes as a part of python3.

[http — HTTP modules — Python 3.11.2 documentation](https://docs.python.org/3/library/http.html)

To ensure that the server serves HTTP 1.1, we set the handler protocol\_version to HTTP 1.1. Execution instructions can be found in HTTP1.1/README.txt.

**Observations from experiment**: Through our experiments of sending files of sizes 10kB upto 10MB between the client and server, we observed that HTTP 1.1 offered the lowest throughputs of them all. This is likely because HTTP1.1 allows only one outstanding request per TCP connection. Furthermore, HTTP1.1 uses plaintext format for all its request and responses which adds additional overhead when compared to more efficiently encoded schemes.

**Application Layer data:** The overhead due to application layer data calculated as –

Total Application Layer Data/ File Data was 1.00039 for 10 kB file and reduced down to 1.00000039 for the 10MB file. This is because the header content remains constant with only the body size changing respective to the file being sent.

# HTTP 2.0

File transfer using HTTP2.0 protocol was tested using the python packages the python package hyper-h2 which can be installed through these instructions –

[Installation — hyper-h2 4.1.0 documentation (python-hyper.org)](https://python-hyper.org/projects/hyper-h2/en/stable/installation.html) (For HTTP2 server)

For the client, we can use either python package httpx or python package sockets –

[socket — Low-level networking interface — Python 3.11.2 documentation](https://docs.python.org/3/library/socket.html)

[HTTP/2 Support - HTTPX (python-httpx.org)](https://www.python-httpx.org/http2/)

These can be installed using pip like this –

pip install h2 httpx[h2]

**Observations from experiment**: Through our experiments of sending files of sizes 10kB upto 10MB between the client and server, we observed that HTTP 2.0 offered higher throughput than HTTP 1.1 for all file sizes – roughly 5-6 times higher for file sizes upto 1MB. This is likely because HTTP2.0 allows multiple TCP connections parallelly, due to this multiplexing it utilizes higher bandwidth. Furthermore, HTTP2.0 uses binary framing as opposed to text which makes the payload smaller in size, ensuring higher throughput.

**Application Layer data:** The overhead due to application layer data calculated as –

Total Application Layer Data/ File Data which was 1.058 for all file sizes. This is because the header content remains constant, but the number of chunks being sent over the same connection increases in proportion to the size of the file being sent.

# gRPC

gRPC protocol was tested using the python packages grpcio and grpcio-tools mentioned below -

<https://pypi.org/project/grpcio/>

<https://pypi.org/project/grpcio-tools/>

These can be installed using pip like this –

pip install grpcio grpcio-tools.

**Observations from experiment**: Through our experiments of sending files of sizes 10kB up to 10MB between the client and server, we observed a much higher throughput using gRPC – roughly 8x higher than HTTP 1.1 for file sizes below 10 MB and nearly 80x higher than HTTP 1.1 for the 10MB file.

The increase in throughput using gRPC can be attributed to the following reasons.

* Using HTTP/2 instead of HTTP/1.1

While HTTP/1.1 allows for processing just one request at a time, HTTP/2 supports multiple calls via the same channel. HTTP/2 communication is divided into smaller messages and framed in binary format, which unlike text-based HTTP/1.1, makes sending and receiving messages compact and efficient.

* Using protocol buffers as an alternative to XML and JSON.

Parsing with Protocol Buffers is less CPU-intensive because data is represented in a binary format which minimizes the size of encoded messages. At runtime, messages are compressed and serialized in binary format.

**Application Layer data:** The overhead due to application layer data calculated as –

Total Application Layer Data/ File Data was 1.07519 for all file sizes. This is probably because the header content remains constant, but the number of chunks being sent over the same connection increases in proportion to the size of the file being sent.

BitTorrent

The BitTorrent protocol was implemented and tested using the python packages py3createtorrent, libtorrent, and torrentp mentioned below -

<https://www.libtorrent.org/python_binding.html>

<https://py3createtorrent.readthedocs.io/en/latest/user.html>

<https://pypi.org/project/torrentp/>

These can be installed using pip and apt-get like this –

* pip3 install py3createtorrent
* pip install torrentp
* sudo apt-get install python3-libtorrent

**Observations from experiment**: Through our experiments of sending files of sizes 10kB up to 10MB between the peers, we observed the highest throughput using BitTorrent– roughly 2x higher than gRPC for all the file sizes from 10kB to 10MB.

The increase in throughput using BitTorrent can be attributed to the following reasons:

Adaptable Bandwidth: A fixed bandwidth is often used by HTTP1.1, HTTP2, and gRPC for the entirety of the file transfer process, which might result in inefficient bandwidth use. In contrast, BitTorrent allows each peer to change the upload and download rates according to the network conditions, so it can adapt to the available bandwidth.

Robust: A single point of failure in the network can halt the entire file transfer process, making HTTP1.1, HTTP2, and gRPC vulnerable to network disruptions. BitTorrent, on the other hand, enables each peer to download portions of the file from numerous other peers, making it more robust to network disruptions.

Increase in throughput: HTTP1.1, HTTP2, and gRPC uses client-server model where the server sends the entire file. This acts as a bottleneck that prevents the server from handling multiple requests and can be slow for large files. In contrast, BitTorrent splits the file into smaller chunks and enables each peer to simultaneously download chunks from a number of other peers. Parallel downloads are made possible as a result, greatly accelerating file transfers.

**Application Layer data:** The overhead due to application layer data calculated as –

Total Application Layer Data/ File Data was 1.0319 for 10 kB file and reduced to 1.0008269 for the 10MB file.

We obtained this value through the total\_upload, total\_payload\_upload parameters exposed by libtorrent on the seeder’s side.