CSC 573 - PROJECT 1

Team Member	Unity ID	Percentage Contribution
Akul Gopal Devali	adevali	33.34%
Tanay Taralbhai Shah	tshah6	33.33%
Umang Sureshbhai Diyora	udiyora	33.33%

TABLE I
TEAM MEMBER CONTRIBUTION

Sub-Tasks	Akul	Tanay	Umang
Research, identifying, and understanding the operation of the packages that implement the protocol	30%	35%	35%
Client implementation	40%	30%	30%
Server implementation	35%	35%	30%
Extend implementation to perform computations on both machines A and B	30%	30%	40%
Perform the experiments to derive the results in Results_file.xlsx	35%	35%	30%
Average contribution	**34%**	**33%**	**33%**

TABLE II
HTTP 1.1

Sub-Tasks	Akul	Tanay	Umang
Research, identifying, and understanding the operation of the packages that implement the protocol	30%	40%	30%
Client implementation	35%	30%	35%
Server implementation	30%	35%	35%
Extend implementation to perform computations on both machines A and B	40%	25%	35%
Perform the experiments to derive the results in Results_file.xlsx	30%	35%	35%
Average contribution	**33%**	**33%**	**34%**

TABLE III
HTTP 2

Sub-Tasks	Akul	Tanay	Umang
Research, identifying, and understanding the operation of the packages that implement the protocol	35%	35%	30%
Seeder implementation	30%	35%	35%
Leecher implementation	35%	30%	35%
Perform the experiments to derive the results in Results_file.xlsx	30%	35%	35%
Average contribution	**32.5%**	**33.75%**	**33.75%**

TABLE IV
BITTORRENT

]

Comparative Analysis of Network Protocols for File Transfer

I. HTTP 1.1 Protocol

I-A Library

For the HTTP 1.1 protocol testing, we implemented file transfer functionality by utilizing the built-in http package that is included in the standard Python 3 library distribution. This native module provided all the necessary components to establish and manage HTTP connections for our file transfer experiments.

https://docs.python.org/3/library/http.html

I-B Implementation Details

Our HTTP/1.1 implementation followed a classic client-server architecture. The server component was developed using Flask, with a dedicated endpoint (/download/¡filename¿) for file transfer operations. To optimize performance, we:

- Implemented proper error handling for missing files
- Used secure filename validation to prevent path traversal attacks
- Added basic logging functionality to track file transfers and server activities
- Configured the server with threading to handle multiple concurrent requests

The client component utilized the requests library to download files from the server. We implemented detailed metrics collection to measure:

- Throughput (kilobits per second)
- Overhead ratio (total data transferred / file size)
- Statistical analysis (mean and standard deviation) for multiple transfers

I-C Observations from experiment

Our experiments demonstrated that HTTP 1.1 offers a straightforward and efficient method for transferring files between server and client systems. Analysis of our results revealed that for smaller files, the protocol exhibits relatively poor throughput performance. This occurs because while the actual transmission time for small files is minimal, the propagation delay becomes disproportionately significant in the overall transfer process.

The data shows a clear pattern where throughput improves as file size increases. However, this improvement doesn't follow a linear progression. This non-linearity stems from the fact that propagation time generally remains consistent regardless of file size, while transmission time grows proportionally with larger files. The overall throughput is determined by the combined effect of these two timing factors.

Our implementation included the ability to test various file sizes (10kB, 100kB, 1MB, 10MB) with different repetition counts to ensure statistical significance. The client automatically calculates headers size and request overhead, providing accurate measurements of the protocol's efficiency.

One notable advantage of HTTP/1.1 is its simplicity and wide support across platforms, making implementation straightforward compared to newer protocols.

II. HTTP 2 Protocol

II-A Library

For our HTTP 2.0 protocol testing, we employed the Python package "h2" to implement the file transfer functionality. This specialized library provides low-level HTTP/2 protocol implementation and requires socket-level programming for complete functionality.

https://pypi.org/project/h2/

II-B Implementation Details

Our HTTP/2 implementation involved more complex socket-level programming compared to HTTP/1.1. The server component utilized:

- Socket handling with proper flow control
- H2Protocol class to manage HTTP/2 connections
- Custom stream management for concurrent transfers
- Detailed performance metrics and logging
- Optimized buffer sizes (1MB) for better throughput

The client component included:

- Direct socket connection with optimized TCP settings (TCP_NODELAY)
- Proper HTTP/2 upgrade handling via h2c (HTTP/2 cleartext)
- Advanced metrics collection including header size, framing overhead, and timing data
- Command-line interface for flexible testing configurations

Both components were designed to work with the same file formats as the HTTP/1.1 implementation, allowing direct performance comparisons. The implementation includes optimizations like:

- Larger flow control windows (1MB)
- Buffer size optimization
- Header table size adjustments
- Concurrent stream support

II-C Observations from experiment

Our experience with HTTP 2.0 showed that it requires more complex implementation compared to HTTP/1.1, especially due to the need for socket-level programming. The h2 library necessitated custom code for handling connection upgrades, stream management, and flow control windows.

The HTTP/2 implementation faced several challenges:

1) Flow control window management required careful handling to prevent stalls
2) Header compression added implementation complexity
3) Stream multiplexing required more sophisticated state management
4) Binary framing added overhead for small transfers but benefitted larger files

This additional implementation complexity appears to have impacted performance in our testing environment. While HTTP/2 theoretically offers benefits like header compression and multiplexing, our experimental results showed that throughput values were sometimes lower across different file sizes compared to HTTP/1.1. This was particularly evident in smaller file transfers, where the protocol overhead became more significant.

Our implementation included proper resource cleanup and connection handling, with comprehensive logging of system resources (CPU and memory usage) during transfers. The client supported testing with multiple servers simultaneously, allowing for more complex testing scenarios.

The HTTP/2 protocol demonstrated more consistent performance across variable network conditions, but the implementation complexity made it more challenging to optimize fully compared to HTTP/1.1.

III. Bittorrent Protocol

III-A Library

For our BitTorrent protocol implementation and testing, we utilized two specialized Python packages: libtorrent and py3createtorrent. These libraries provided the necessary functionality to create and manage the peer-to-peer file sharing system.

https://www.libtorrent.org/python_binding.html
https://py3createtorrent.readthedocs.io/en/latest/user.html

Since our implementation was conducted on a Linux environment, we installed the required libraries using the following system-level package management commands:

sudo apt install python-libtorrent

III-B Observations from experiment

Our peer-to-peer file transfer experiments, testing files ranging from 10kB to 10MB, demonstrated that BitTorrent achieved the highest performance metrics. For 10MB files specifically, BitTorrent delivered approximately twice the throughput compared to other protocols.

We identified several architectural advantages that contribute to BitTorrent's superior performance with larger files:

- **Adaptable Bandwidth:** First, BitTorrent employs dynamic bandwidth allocation. While HTTP1.1 and HTTP2 typically utilize fixed bandwidth allocations throughout the entire transfer process (potentially leading to inefficient resource utilization), BitTorrent allows individual peers to dynamically adjust upload and download rates based on current network conditions. This adaptive approach optimizes bandwidth usage according to real-time availability.
- **Robust:** Second, BitTorrent offers enhanced resilience. The client-server architectures of HTTP1.1 and HTTP2 create single points of failure that can completely interrupt file transfers during network disruptions. In contrast, BitTorrent's distributed approach enables peers to retrieve different portions of files from multiple sources simultaneously, significantly reducing vulnerability to individual connection failures.
- **Increase in throughput:** Third, BitTorrent achieves higher throughput through parallel processing. Traditional client-server models create bottlenecks where a single server must handle all file distribution, limiting scalability and creating performance constraints with larger files. BitTorrent's approach divides files into smaller segments that can be simultaneously downloaded from numerous peers. This parallelization dramatically accelerates transfer speeds, especially for larger files.

It's worth noting that our experimental measurements showed minimal variance (low standard deviation) for BitTorrent performance metrics since all peers were operating locally on the same machine, eliminating external network variables.

TABLE V
THROUGHPUT PERFORMANCE FOR SMALL FILES (IN KBPS)

Protocol	10 KB		100 KB	
	Average	Std. Dev.	Average	Std. Dev.
HTTP 1.1	27,133.58	6,053.04	276,264.52	54,729.89
HTTP 2	48,290,797.18	20,952,011.58	744,585.05	323,269.67
BitTorrent	71.31	22.53	658.21	263.07

TABLE VI
THROUGHPUT PERFORMANCE FOR LARGE FILES (IN KBPS)

Protocol	1 MB		10 MB	
	Average	Std. Dev.	Average	Std. Dev.
HTTP 1.1	1,400,089.25	484,029.83	1,329,799.69	0
HTTP 2	749,995.91	377,858.91	259,095.74	0
BitTorrent	6,263.89	3,611.37	27,878.07	0

IV. Experimental Results

Tables V and VI present the throughput measurements for the three protocols across different file sizes. The results are given in kilobits per second (kbps).

From the results in Tables V and VI, we can observe several interesting patterns:

1) **Small Files (Table V):** For 10KB files, HTTP 2 shows exceptionally high throughput, while HTTP 1.1 offers moderate performance and BitTorrent performs poorly. For 100KB files, HTTP 2 maintains its lead over HTTP 1.1, while BitTorrent still shows relatively low throughput.

2) **Large Files (Table VI):** For 1MB and 10MB files, HTTP 1.1 outperforms HTTP 2, showing that HTTP 2's advantage diminishes with larger file sizes. BitTorrent shows significant improvement as file size increases, particularly for 10MB files.
3) **Performance Trends:** HTTP 1.1 shows consistent performance across file sizes. HTTP 2 performs worse than HTTP 1.1 for larger files in our implementation. BitTorrent, while starting with poor performance for small files, improves dramatically with larger files.
4) **Variability:** The standard deviation values indicate that HTTP 2 has the highest variability in performance, especially for smaller files, suggesting that its performance is less predictable in our testing environment.

Note that the extremely high value for HTTP 2 with 10KB files (48,290,797.18 kbps) appears to be an outlier and may be due to implementation specificities or measurement methodology. This further supports our earlier observations about the implementation complexity of HTTP 2 using the h2 library.

V. Observations of all the protocols combined

Based on our comprehensive analysis, we can conclude that within the client-server architectural model, HTTP 1.1 demonstrated solid performance metrics. Theoretically, HTTP 2.0 should outperform HTTP 1.1, considering that HTTP 2.0 infrastructure inherently provides enhanced performance characteristics and reduced latency.

When examining specific use cases, HTTP/2 typically exhibits potential efficiency for transferring smaller data payloads. This advantage stems from HTTP/2's advanced capabilities, particularly multiplexing (handling multiple concurrent requests) and header compression. Conversely, HTTP/1.1 may demonstrate better performance when handling larger file transfers due to its more straightforward framing mechanisms.

However, our experimental results revealed underperformance in our HTTP 2.0 implementation. This performance deficit can be attributed to two primary factors: incomplete implementation of certain protocol features in our testing environment, and the additional overhead required to manually implement socket-level programming. These implementation requirements introduced extra processing time that negatively impacted overall performance metrics.

BitTorrent, with its peer-to-peer architecture, demonstrates impressive scalability for larger files. While its throughput for small files is significantly lower than the HTTP protocols, its performance for larger files suggests it is the ideal choice for distributing large content to multiple users.