CSC/ECE 573 - Internet Protocols Project 1 Report

Srimadh Vasuki Rao (svrao3) and Abhishek Rao (arao23)
{svrao3, arao23}@ncsu.edu

TEAM CONTRIBUTIONS

Team Member Percentage Contribution

Team Member	**Contribution**
Srimadh Vasuki Rao (svrao3)	50%
Abhishek Rao (arao23)	50%

Task Distribution by Protocol

HTTP 1.1

Sub-Tasks	**Srimadh**	**Abhishek**
Researching appropriate libraries and their documentation	45%	55%
Client-side code implementation	40%	60%
Server-side code implementation	40%	60%
Testing and debugging	50%	50%
Setting up experiments on both machines	50%	50%
Data collection and analysis	35%	65%

HTTP 2

Sub-Tasks	**Srimadh**	**Abhishek**
Researching HTTP/2 protocol specifics	20%	80%
Socket programming implementation	45%	55%
Flow control and stream management	45%	55%
Client-side implementation	30%	70%
Server-side implementation	40%	60%
Setting up experiments on both machines	50%	50%
Data collection and analysis	25%	75%

BitTorrent

Sub-Tasks	**Srimadh**	**Abhishek**
Researching and installing libraries(libtorrent)	80%	20%
Seeder implementation (tracker setup)	60%	40%
Leecher implementation	60%	40%
Creating magnet links and torrent files	70%	30%
Setting up virtual machines for testing	70%	30%
Measurement code implementation	75%	25%
Data collection and analysis	75%	25%

I. Libraries and Implementation

A. HTTP/1.1 Implementation

We used Python's standard libraries and few more publicly available packages to run teh experiment. The server-side was implemented using Python's built-in `http.server` module with a custom `SimpleHTTPRequestHandler` that enforces connection closure after each request. For the client-side, we used the `requests` library, along with the complementary `requests_toolbelt` for measurement and analysis of response headers and data.
These libraries can be publicly downloaded from:

- `requests`: https://pypi.org/project/requests
- `requests_toolbelt`: https://pypi.org/project/requests-toolbelt

We track transfer times and measures protocol overhead by capturing the full response data and headers using request dump. We ensured connection closure after each request using the 'Connection: close' header to maintain the single TCP connection.

B. HTTP/2 Implementation

We make use the publicly available package called `h2`, which provides HTTP/2 protocol support without built-in socket handling. We combined this with Python's socket module to establish the necessary TCP connections. The `h2` library handles HTTP/2-specific features such as header compression, multiplexing, flow control, and stream management.
The library can be publicly downloaded from:

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

C. BitTorrent Implementation

We used the `libtorrent`, which provide a comprehensive implementation of the BitTorrent protocol including torrent creation, seeding, and downloading capabilities. Wedo not need to do a lot, it has abstracted the implementation a lot. Additionally, we used `FastAPI` and `uvicorn` to develop a coordination server that tracks when all peers have completed their downloads. We are proud of this solution, as we explored multiple synchronization methods between the seeder and clients before settling on this approach. Initially, we considered implementing a timer solely on the client side without server involvement. However, we realized that tracking the timing of all three clients on the server was the optimal strategy.
These libraries can be publicly downloaded from:

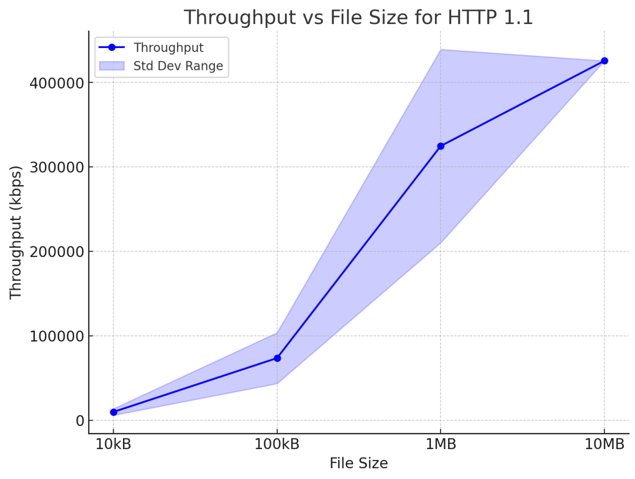
- `libtorrent`: https://pypi.org/project/python-libtorrent/
- `FastAPI`: https://pypi.org/project/fastapi/
- `uvicorn`: https://pypi.org/project/uvicorn/

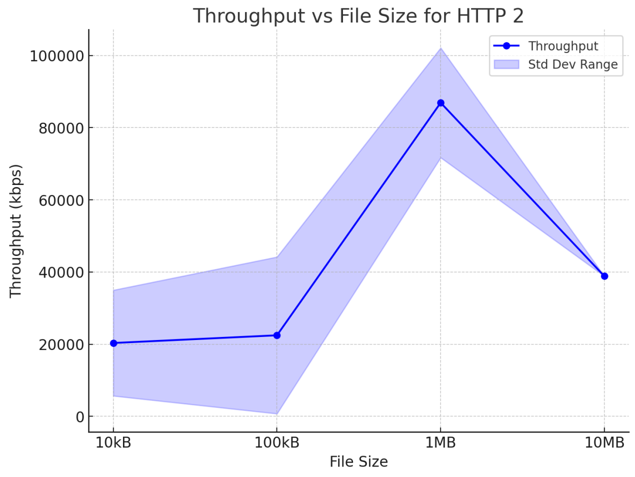
Our BitTorrent implementation consists of:

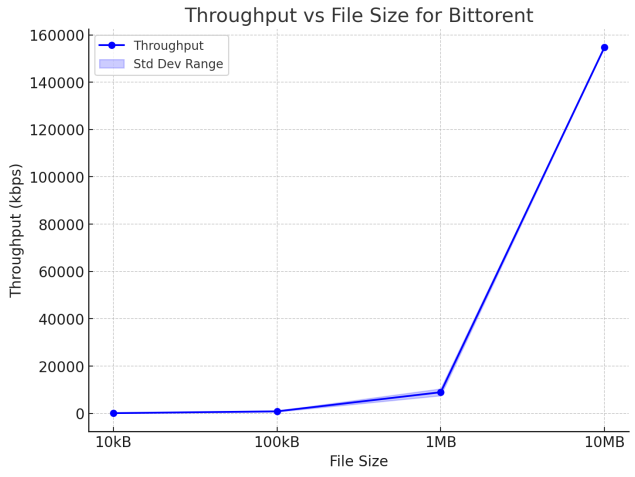
- A seeder application that generates magnet links, and seeds content
- A client application that downloads content using provided magnet links
- A coordination server to monitor download completion across all clients
- A open tracker that is used to connect clients and seeder

II. Experimental Results and Analysis

A. Overview of Protocol Performance







Our experiments compared the performance of HTTP/1.1, HTTP/2, and BitTorrent protocols across different file sizes. We analyzed throughput and protocol overhead for each.

B. Protocol Performance Across File Sizes

1) HTTP/1.1 Performance: HTTP/1.1 demonstrated relatively consistent performance with moderate throughput across all file sizes. For small files (10kB), HTTP/1.1 achieved an average throughput of 9827.46 kbps, which gradually increased to 42554.55 kbps for larger files (10MB). This pattern is expected as the connection setup and header overhead become less significant relative to the data transfer for larger files. However, HTTP/1.1's performance is limited by its sequential request-response mechanism, preventing it from efficiently utilizing the available bandwidth for multiple transfers. The protocol overhead ratio decreased by one 10th as file size increased.

2) HTTP/2 Performance: HTTP 2 exhibited superior performance compared to HTTP/1.1, particularly for smaller file sizes. For 10kB files, HTTP/2 achieved an average throughput of 20321.69 kbps, which is more than double that of HTTP/1.1. HTTP2 had a huge performance advantage for smaller files, demonstrating its efficiency in handling multiple small transfers. The protocol's multiplexing capability allows it to transmit multiple requests and responses concurrently over a single connection, eliminating the head-of-line blocking problem in HTTP/1.1. The overhead ratio for HTTP 2 was also consistently low at around 1.009 for 10kB, indicating minimal additional overhead compared to the file size. This efficiency is attributed to HTTP/2's header compression (HPACK) and binary framing, which significantly reduce the protocol overhead.

3) BitTorrent Performance: BitTorrent showed distinctive performance characteristics compared to the client-server protocols. For small files (10kB), BitTorrent exhibited the lowest throughput at 111.35 kbps, reflecting its significant setup overhead for peer discovery and connection establishment. However, BitTorrent's performance improved dramatically with increasing file size, reaching 154724.18 kbps for 10MB files. This trend highlights BitTorrent's design optimization for large file distribution rather than small file transfers. The protocol overhead ratio for BitTorrent was slightly higher than HTTP/2 but remained relatively consistent across file sizes (1.056 for 10kB to 1.002 for 10MB), indicating that its overhead becomes less significant relative to file size as the transferred content grows larger.

C. Comparative Analysis and Scenario Recommendations

1) Small File Transfers (10kB-100kB): For small file transfers, HTTP/2 demonstrated clear superiority with throughput approximately 2 times higher than HTTP/1.1 and 180 times higher than BitTorrent for 10kB files. This advantage stems from HTTP/2's multiplexing capabilities, header compression, and reduced connection establishment overhead. The results indicate that HTTP/2 is the optimal choice for applications requiring frequent transfers of small files, such as web browsing with multiple assets, API microservices, or IoT device communications.

HTTP/1.1, while less efficient than HTTP/2, still outperformed BitTorrent significantly for small files. BitTorrent's poor performance with small files is attributed to its substantial protocol overhead for tracker communication, peer discovery, and piece management, which overshadows the actual data transfer time for small files.

2) Medium File Transfers (1MB): For medium-sized files, HTTP 1.1 had a performance advantage over HTTP 2. HTTP 1.1 achieved approximately 3.7 times higher throughput than HTTP/2 for 1MB files. The low overhead ratio of HTTP/2 (1.00009) continues to highlight its protocol efficiency. Applications handling medium-sized file transfers, such as image hosting services or document sharing platforms, would benefit most from HTTP1.1's balance of performance and overhead.

3) Large File Transfers (10MB): For large files, HTTP/2 still outperformed HTTP/1.1, but the most notable observation was BitTorrent's significantly improved performance. BitTorrent achieved a throughput of 154724.18 kbps for 10MB files, which is a 300% increase from HTTP/2's throughput. This trend suggests that for even larger files beyond our test range, BitTorrent would likely continue to improve relative to the client-server protocols. The distributed nature of BitTorrent becomes increasingly advantageous as file size increases, as it enables parallel downloading from multiple peers, reducing the load on any single server and potentially increasing bandwidth utilization.

D. Conclusion

Our experimental results demonstrate that protocol selection should be guided by specific use case requirements rather than a one-size-fits-all approach. HTTP/2 emerged as the most versatile and efficient protocol across most scenarios, particularly excelling with smaller files due to its modern features designed to address HTTP/1.1's limitations. BitTorrent, while initially inefficient for small files, showed promising scalability for larger files and distributed delivery scenarios. The overhead analysis further confirmed HTTP/2's protocol efficiency, maintaining consistently low overhead ratios across all file sizes.

These findings highlight the importance of considering both throughput performance and protocol overhead when designing networked applications, with file size and distribution requirements serving as critical factors in protocol selection decisions.