November 17, 2024

P2P File Exchange

Computing Engineering

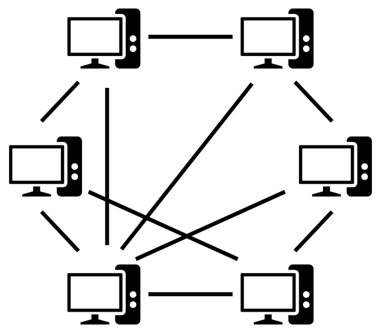
Operating Systems

Authors:
Desireé Avilés
Danilo Duque
Emmanuel Rojas

Professor:
Eddy Ramírez Jiménez

1 Executive Summary

The Peer-to-Peer (P2P) File Exchange system represents a decentralized approach to file sharing that addresses the limitations of traditional centralized methods. This executive summary outlines a comprehensive project aimed at developing a P2P system to enable efficient, scalable, and resilient file transfers across multiple peers. The system is implemented using C++ and leverages multithreading and TCP communication to enhance performance and ensure data reliability.



P2P networks operate without a central server, allowing each user, or peer, to act as both a client and a server. This distributed architecture eliminates the single point of failure inherent in centralized models and allows for simultaneous downloads from multiple sources. This project aims to demonstrate the advantages of P2P systems in terms of scalability, fault tolerance, and performance, especially in scenarios where large files are shared across a network of peers.

Given the challenge of distributing large files efficiently in a decentralized manner, the project implements segmented file transfer. Each file is divided into smaller chunks, which can be downloaded concurrently from multiple peers. This approach not only optimizes bandwidth utilization but also reduces the overall time required for file transfers.

Centralized systems can become bottlenecks due to high server load, while P2P networks distribute this load across active peers, enhancing resilience. If some peers go offline, others can continue file exchanges, ensuring robustness. An acknowledgment mechanism maintains synchronized communication and data integrity during transfers.

This project explores decentralization as a flexible and robust alternative to traditional file-sharing platforms, emphasizing multithreading for concurrent requests and improved performance in distributed environments.

2 Introduction

Traditional file-sharing systems that rely on centralized servers have long been the standard in networked communication. These systems offer several advantages, such as simplicity, centralized control, and ease of management. In centralized networks, the server is the hub that stores and distributes files to clients, ensuring a controlled and consistent flow of data. This setup works well in smaller networks with a limited number of users and low traffic.

However, as the number of users increases, centralized systems begin to face significant challenges. The server becomes a bottleneck, as it must handle all requests from clients. This leads to performance degradation, slower file transfers, and higher latency. Moreover, the reliance on a single server makes the system vulnerable to failure. If the server goes offline, the entire network is disrupted. The scalability of centralized systems is thus limited, and as more users join, the experience for each user worsens.

Peer-to-Peer (P2P) networks offer a compelling alternative. In a P2P network, there is no central server; instead, each participant, or peer, acts as both a client and a server. This decentralized architecture eliminates the single point of failure and distributes the load across all active peers. The more users that are connected to the network, the better the system performs. As more peers join, they contribute resources (such as bandwidth and storage), which improves the overall performance and efficiency of the network. In contrast to centralized networks, where additional peers increase the burden on the server and slow down transfers, P2P systems become faster and more resilient as the number of users increases.

In this project, a partial Peer-to-Peer (P2P) system was implemented, which maintains certain centralized aspects, particularly the need for an indexer. This type of P2P architecture is commonly attributed to early file-sharing platforms like Napster. While the system does not rely on a single central server for file storage, it does require a central indexer to catalog and track available files across the network. The indexer helps peers find and connect to each other, enabling them to locate the files they want to download.

This hybrid approach strikes a balance between decentralization and efficiency. While it eliminates the bottlenecks associated with central server models, the indexer still plays a crucial role in facilitating peer discovery and maintaining an organized structure for file availability. The use of an indexer in this P2P model allows for faster searches and more reliable connections, making it suitable for scenarios where users need to quickly find and exchange large files, while still benefiting from the scalability and fault tolerance of P2P networks.

To further enhance the performance of the system, this project leverages segmented file transfer, a technique that divides large files into smaller chunks. These chunks can be distributed across multiple peers, enabling concurrent downloads. This not only optimizes the use of available bandwidth but also significantly reduces the overall time required for file transfers. By breaking up files into manageable pieces, the system is able to improve efficiency, especially in environments where large files need to be exchanged frequently.

Additionally, the system incorporates an acknowledgment mechanism to ensure data integrity and reliability during the transfer process. This mechanism helps maintain synchronization between peers, verifying that each chunk of data is correctly received before the next one is transmitted. This process mitigates issues related to data loss or corruption, ensuring a smooth and reliable transfer experience, even in a decentralized environment where network conditions may vary.

Overall, the project aims to showcase the advantages of P2P systems, such as scalability, fault tolerance, and improved performance in distributed environments. Through the implementation of multithreading, segmented file transfer, and an efficient indexing system, this project provides a robust alternative to traditional centralized file-sharing methods.

3 Theoretical Framework

3.1 Knuth-Morris-Pratt (KMP) Algorithm

The Knuth-Morris-Pratt (KMP) algorithm is an efficient string-searching algorithm that searches for occurrences of a substring (or pattern) within a larger string (or text). It significantly reduces the time complexity compared to naïve string-searching methods by utilizing the information gained from previous character matches. Specifically, KMP avoids redundant comparisons by preprocessing the pattern and using this preprocessed information to skip over portions of the text that have already been matched.

The main advantage of the KMP algorithm lies in its ability to achieve a linear time complexity of $O(n + m)$, where n is the length of the text and m is the length of the pattern. This is achieved by preprocessing the pattern into an array called the partial match table or prefix function, which tells the algorithm how much the pattern can be shifted when a mismatch occurs.

Application in the P2P File Exchange System
In the context of the Peer-to-Peer (P2P) File Exchange system, the KMP algorithm plays a crucial role in optimizing the file search process across the distributed network. When a peer searches for a file, the system needs to match the search term (usually a substring representing the file name or part of it) against the file names indexed by the system. This is where KMP comes in.

Instead of scanning each file name character by character for every peer in the network, which could be computationally expensive, the KMP algorithm allows the system to efficiently search for substrings in the file names stored in the index. The search operation can be performed quickly by leveraging the preprocessed pattern, ensuring that file searches are fast even when the number of files in the index is large.

This optimization is particularly important in a decentralized P2P system where peers may be constantly joining or leaving the network. The KMP algorithm ensures that even with frequent updates to the file index, searches remain efficient and scalable, improving the overall user experience.

Thus, the KMP algorithm enhances the search functionality within the indexer of the P2P system, making it feasible for peers to search for files based on partial matches of their names without the need for excessive computational resources. This enables peers to find files more quickly and efficiently, thereby improving the overall performance of the file-sharing system.

Example Implementation

In the ICPC team notebook El Vasito the KMP algorithm is implemented based on the approach described. The algorithm leverages a preprocessing function `kmppre`, which computes the longest borders (prefix-suffix matches) for the pattern, and a search function `kmp` that finds matches in the target text. The code is as follows:

```cpp
vector<int> kmppre(string& t){ // r[i]: longest border
   of t[0,i)
    vector<int> r(t.size()+1); r[0] = -1;
    int j = -1;
    for(int i = 0; i < t.size(); i++) {
        while(j >= 0 && t[i] != t[j]) j = r[j];
        r[i+1] = ++j;
    }
    return r;
}

void kmp(string& s, string& t){ // find t in s
    int j = 0;
    vector<int> b = kmppre(t);
    for(int i = 0; i < s.size(); i++) {
        while(j >= 0 && s[i] != t[j]) j = b[j];
        if(++j == t.size()) {
            printf("Match at %d\n", i - j + 1);
            j = b[j];
        }
    }
}
```

Listing 1: KMP Algorithm Implementation from the ICPC Team Notebook by El Vasito

3.2 Use of C++ in the P2P Project

In the development of this distributed file-sharing system for peers, several C++ features were used to efficiently manage connections, memory, and data structures. Below are the key components used in the project.

3.2.1 Data Structures: Maps and Sets

To store and organize information in the system, two of the most common data structures in C++ were employed: `std::map` and `std::set`. These structures allow data to be stored efficiently and accessed quickly.

- **std::map**: `std::map` was used to associate unique keys with values, enabling efficient management of information. For example, it can be used to store metadata for files shared among peers, where the key could be a unique file identifier and the value contains the file details.
- **std::set**: `std::set` was useful for storing collections of unique elements, such as the IP addresses of connected peers, avoiding duplicates and enabling efficient searching.

3.2.2 Memory Management: shared_ptr

In the project, `std::shared_ptr` was used to efficiently manage the memory of objects shared between multiple threads. This smart pointer structure helps avoid memory leaks by maintaining a reference count for objects, ensuring that they are automatically released when no longer needed.

3.2.3 Peer Communication: send and recv

For communication between peers in the P2P network, `send` and `recv` were used, which are low-level functions for sending and receiving data over sockets. These functions allow for the efficient transmission of file fragments between peers, keeping data transfers synchronized.

- **send**: `send` was used to transmit data from the server to the client or between peers. It enables the transfer of binary data, such as file fragments, ensuring that data is correctly sent over the network.

- **recv**: `recv` was used to receive data from other peers. It was employed to receive file requests or fragments and also to receive transfer confirmations.

3.2.4 Synchronization: mutex

Concurrency of multiple threads managing simultaneous client connections required the use of `std::mutex` to ensure synchronization of shared resources. `Mutexes` are essential to prevent race conditions and ensure that only one thread can access a shared resource at a time.

3.2.5 Thread Management: threads

The use of threads was crucial to allow concurrent execution of tasks, such as managing multiple peer connections at the same time. C++ provides a standard library for handling threads, which was used to create and manage threads responsible for receiving requests, sending files, and handling connections with other peers.

- **std::thread**: `std::thread` was used to create threads responsible for tasks such as receiving files from other peers or transmitting file fragments.

- **Thread Synchronization**: To coordinate access to shared resources, synchronization mechanisms such as `mutexes` were used to ensure that threads would not interfere with each other when accessing the same resources.

These C++ tools were essential in ensuring the efficiency, safety, and performance of the system, enabling smooth communication and effective management of network operations and shared resources between peers.

3.3 Transmission Control Protocol (TCP)

In the context of the P2P file-sharing system, the Transmission Control Protocol (TCP) plays a crucial role in ensuring reliable communication between peers. TCP is a connection-oriented protocol that guarantees the orderly and error-free delivery of data packets across a network. This section discusses the key features of TCP and its application in the system.

3.3.1 Reliability and Connection-Oriented Communication

One of the fundamental aspects of TCP is its reliability. Unlike User Datagram Protocol (UDP), which is connectionless, TCP establishes a connection between the sender and receiver before transmitting data. This connection setup ensures that data is delivered in sequence and without loss. If any packet is lost or corrupted, TCP handles retransmission of the lost packet, ensuring that the communication remains reliable even in the case of network issues.

In the P2P file-sharing system, this feature of TCP is essential for the correct and reliable transmission of file fragments between peers. Each peer maintains a persistent connection with others, and the protocol ensures that the file transfer process is not interrupted by lost packets or out-of-order delivery.

3.3.2 Data Segmentation and Flow Control

TCP breaks down large data into smaller packets, which are sent across the network and reassembled at the receiving end. This process, called segmentation, allows TCP to handle large files efficiently by dividing them into manageable chunks. Each packet contains a sequence number, enabling the receiver to correctly reassemble the data in the proper order.

Additionally, TCP employs flow control mechanisms to prevent network congestion. The receiver advertises a window size, indicating how much data it can buffer before sending an acknowledgment. If the sender exceeds this window size, it will wait for the receiver to process the data and send an acknowledgment before continuing.

In the P2P file-sharing system, this feature ensures that file transfers are not overloaded with too much data at once, preventing potential bottlenecks or slowdowns in the network.

3.3.3 Error Detection and Retransmission

Another important feature of TCP is its ability to detect errors in transmitted data using checksums. When a peer sends data, it includes a checksum value that is used by the receiver to verify the integrity of the data. If any errors are detected, the receiver requests the retransmission of the affected packet.

In the context of file sharing, this ensures that the file fragments received by the peer are identical to those sent by the sender. If any corruption occurs during transmission, TCP guarantees that the affected fragments are retransmitted, ensuring the integrity of the transferred files.

3.3.4 TCP in the P2P System

In the P2P file-sharing system, the use of TCP guarantees that the communication between peers is robust and reliable. For example, when a peer requests a file or a fragment of a file from another peer, TCP ensures that the request is received correctly and that the file is transmitted without errors or loss. The protocol handles issues like packet loss, reordering, and congestion, providing a stable foundation for file sharing over the network.

Moreover, the continuous connection between peers ensures that the system can handle multiple concurrent file transfers without interference, and the built-in flow control and error handling prevent the system from being overwhelmed by data or disrupted by network problems.

Overall, TCP's reliability, error detection, flow control, and connection-oriented nature make it an ideal protocol for the P2P file-sharing system, where the successful transmission of large files between peers is a primary goal.

4 Solution Description

In the development of the P2P file-sharing system, three core components were essential to ensure the complete functionality of the network. As previously mentioned, the first key component was the index server, responsible for maintaining a searchable index of available files across the network. The second component, the peer server, handled incoming file requests from other peers, facilitating the sharing of files. Lastly, the file requester component allowed users to initiate file downloads by sending requests to the network, ensuring they could retrieve the files they needed. These components work together to form a seamless and efficient file-sharing experience in a decentralized P2P environment.

4.1 Index Server

Imagine you need a book, but there isn't a single library in your city. Fortunately, there's a solution: you know someone who has a comprehensive list of every book that everyone in the city owns. So, you go to this person and ask, "Where can I find the book I need?" Once they respond, you know exactly which person to approach in the city to borrow the book. This scenario mirrors the concept of Peer-to-Peer (P2P) networks. Instead of relying on a central library that stores all the books, you have to ask the people in the city directly.

In this analogy, the index server represents the person who holds the list of where each book can be found. Whenever someone in the city is looking for a book, they first ask the index server. The server provides the location, and the person can then approach the specific individual who owns the book. Similarly, in a P2P system, the index server serves as the lookup point for finding resources within the network, enabling peers to locate and request files from one another.

To design and implement an effective index server for a Peer-to-Peer file-sharing system, there are three critical factors to consider: efficient memory management, fast lookups, and high availability.

To address memory management efficiency within the index server, the key strategy was to avoid duplicating `FileInfo` objects. Instead, each file is associated with a list of the peers where the file exists. This is achieved through the use of **shared references** and **efficient memory management**, which improves resource usage and ensures that CPU cycles and memory are not wasted unnecessarily.

Memory Management with Shared Pointers

One of the primary strategies adopted to enhance memory efficiency was the use of **smart pointers**, specifically `std::shared_ptr`. These pointers automatically manage the memory of the objects they point to. By using `shared_ptr`, the system ensures that memory occupied by `FileInfo` objects is automatically freed when no longer needed, eliminating the need for manual intervention. This helps prevent memory leaks and simplifies the management of the objects' lifetimes.

The use of shared pointers is particularly beneficial in a distributed system like a P2P server, where multiple components might have access to the same file. Instead of copying each file for every peer that owns it, only a single instance of `FileInfo` is kept in memory, and different peers simply hold references to this single instance. As a result, the same `FileInfo` object is shared among all references, eliminating the unnecessary duplication of data.

No Duplication of Files

In a typical P2P system, each file may be present on multiple peers, which would normally mean that if the data is duplicated for each peer, the memory usage would grow polynomially. Rather than maintaining a copy of the `FileInfo` object for each peer, the approach taken was to maintain a single `FileInfo` instance for each file, and associate that instance with a list of peers who possess the file.

This approach offers several advantages:

- **Reduced memory usage**: By not duplicating `FileInfo` objects, the system significantly reduces the amount of memory required to store information about files, especially when a file is available across many peers.

- **Improved scalability**: As the number of files and peers increases, this approach ensures memory is used efficiently, allowing the system to handle more files without running into memory bottlenecks.

- **Efficient updates**: When a new peer joins the system and shares a file, the server simply adds this information to the corresponding file in the index, rather than creating a new copy of `FileInfo`. This allows for a more efficient and faster update of the file database.

Associating Peers with Files

To associate peers with files, each `FileInfo` object contains a `fileInfo` set, which stores the information about the peers (such as IP address, port, and file name) that have a copy of the file. This way, multiple instances of the same file are not stored. Instead, only the locations (peers) where the file is available are stored.

This model is more efficient because each time a new peer joins the system, instead of creating a new `FileInfo` object for that peer, the system simply adds the peer's reference to the existing file and updates the list of peers that hold it. The file index then maintains a unique reference for each file and a dynamic set of peers associated with it.

Second, fast lookups are crucial for delivering a seamless user experience. The index server needs to quickly respond to queries from peers asking for specific files. To achieve this, the server employs efficient algorithms such as the Knuth-Morris-Pratt (KMP) algorithm, which is particularly useful for substring matching. When a user searches for a file by its name, the server must identify all file names in the index that contain the search term as a substring. The KMP algorithm allows the server to efficiently determine if the search term is a substring of any of the "known as" file names (aliases) in the index.

In this system, the index server returns all file names that match the search term, enabling the user to choose which file to download. The search process involves iterating over each file in the index, and for each file, checking all its "known as" file names (aliases). For each alias, the KMP algorithm is applied to check if the search term is a substring of the alias. If a match is found, the file is added to the list of potential matches.

The time complexity of this search process can be broken down as follows: - For each file in the index, we check all its aliases. - For each alias, the KMP algorithm performs substring matching, which has a linear time complexity, i.e., $O(m)$, where m is the length of the alias. - Let n be the number of files in the index and k be the average number of aliases per file.

Thus, the total time complexity of the search operation is $O(n \cdot k \cdot m)$, where: - n is the number of files in the index, - k is the average number of aliases per file, and - m is the length of the search term (or the alias).

By using the KMP algorithm, the index server can perform these lookups efficiently, ensuring quick and responsive searches even as the number of files and aliases in the system grows. This optimization improves the overall performance of the file-sharing system.

Lastly, high availability guarantees that the index server remains operational at all times, even in the event of system failures or high demand. This can be achieved through strategies such as load balancing, fault tolerance, and redundancy, which allow the server to continue functioning smoothly even when certain components are temporarily unavailable. High availability is vital to ensure the system's reliability and prevent downtime, especially in a decentralized environment where users rely on the server to locate files across the network.

To further enhance availability, a TCP server architecture was implemented for handling requests from peers. For each incoming request, the server generates a separate thread to handle the interaction with the peer. This multithreaded approach allows the index server to manage multiple requests concurrently without blocking, ensuring that high traffic or large numbers of requests do not overload the system. Each thread operates independently, processing requests and responding to peers in parallel, which significantly improves the server's ability to handle high volumes of traffic and maintain continuous availability.

By using a combination of efficient multithreaded handling and reliable network communication through TCP, the index server can maintain high availability and provide timely responses to all peers in the system.

4.2 Peer Server

The peer server is responsible for managing file requests from other peers in the network. A critical aspect of its operation is the ability to find and transfer the requested file efficiently. When a peer requests a file, the request includes the hash of the file, which the peer obtained earlier from the index server. This hash serves as a unique identifier for the file, allowing the peer server to quickly locate it.

To ensure a reliable and efficient lookup mechanism, a double-hashing technique was implemented. The first hash function generates a polynomial hash of the file's content at the byte level. This polynomial hash uses a base of 257 and a large prime modulus to minimize the risk of collisions. The second hash function is the FNV-1a hash, a fast, non-cryptographic hash function that is particularly well-suited for small inputs. This double-hashing approach helps ensure the uniqueness of file identifiers and enhances the reliability of lookups.

The peer server maintains a hash map where each key is the result of this double hash, and the corresponding value is the file's location or metadata. This design allows for logarithmic time complexity $O(\log n)$ when searching for files, as the peer server only needs to compute the double hash from the request and query the hash map to retrieve the file.

Once the peer server locates the requested file using this double-hashing mechanism, it spawns a new thread to handle the file transfer to the requesting peer. By employing multithreading, the server can handle multiple requests in parallel, improving both performance and scalability in a decentralized environment.

4.3 File Requester

The file requester is a critical component in the distributed file-sharing system. Its primary responsibility is to request and download files from peers across the network. The process begins with the requester obtaining the file hash from the index server. This hash uniquely identifies the file and is used by the requester to locate peers that store different chunks of the requested file.

When a user initiates a request, the requester connects to the index server via a TCP socket, using the file name as input. The index server returns all known peers that store the file, along with the corresponding file metadata such as size, hash values, and available chunks. The requester uses this information to manage the download process efficiently by splitting the file into chunks and distributing the download across multiple peers.

The key mechanism that ensures the system's scalability is parallelism. Each file is divided into multiple chunks, and the file requester spawns a separate thread for each peer to download these chunks concurrently. By doing so, the requester maximizes the bandwidth utilization and minimizes the total download time.

To ensure data integrity, a double hash mechanism is applied to the file. The requester uses two types of hashes: a polynomial hash and an FNV-1a hash, both generated at the byte level of the file. The polynomial hash uses a base of 257 and a large prime modulus to reduce collisions, while the FNV-1a hash offers fast, non-cryptographic verification. Each file chunk is checked using these hash values to verify the integrity of the downloaded data.

Once the file requester retrieves all the chunks from the peers, it proceeds to assemble them back into a single file. The chunks are sorted and concatenated in the correct order, ensuring that the final file is an exact replica of the original. After the reconstruction, the temporary chunk files are deleted to free up space.

This modular and multithreaded approach ensures that the file requester can handle large files and multiple peers efficiently. By downloading different parts of the file in parallel, the requester improves the overall download speed while ensuring the file's integrity using the double-hashing mechanism.

This section described the design and functionality of the three main components of the P2P file-sharing system: the index server, the peer server, and the file requester. These components work together to ensure an efficient and decentralized file-sharing experience. The index server acts as the central search point for files, utilizing advanced memory management techniques and search algorithms to optimize performance. The peer server handles file requests, employing hashing mechanisms to ensure quick lookups and efficient file transfers. Lastly, the file requester coordinates the simultaneous download of multiple file fragments from various peers, maximizing speed and ensuring data integrity through hash verification. This modular and parallel approach allows the system to handle large volumes of data in a scalable and reliable manner.

5 Performance Evaluation

The performance tests for the P2P file-sharing system were conducted using two distinct machines. The index server and all peer servers were executed concurrently on a computer equipped with an 11th-generation Intel i3 processor and 8GB of RAM. Meanwhile, the file request operations were initiated from a separate machine, a Mac with an M2 processor. This setup allowed us to assess the system's performance under realistic network conditions, simulating a scenario where different machines act as peers and requesters within the P2P network.

All the tests were conducted using a 50MB video file, which provided a consistent workload for evaluating the performance of the system.

During the performance evaluation, two key parameters were identified that could potentially affect the file download time:

- The first parameter was the buffer size used for the send and receive operations. Adjusting the buffer size could influence the rate of data transfer, with different buffer sizes potentially leading to variations in performance.

- The second parameter was the number of peers that had the requested file. When multiple peers stored the same file, the file requester would split the file into equal-sized chunks and download these parts concurrently from each peer, potentially impacting the overall download speed.

To better visualize the relationship between these parameters and the file download time, a three-dimensional graph was generated. In this graph, the x-axis represents the download time, the y-axis represents the buffer size, and the z-axis represents the number of peers that possess the requested file. This 3D graph helps illustrate how varying the buffer size and the number of peers influences the download time in the P2P system.

For the performance evaluation, a series of 30 tests were conducted, varying the number of peers from 1 to 6. For each peer configuration, buffer sizes corresponding to powers of two were selected: 2^{10}, 2^{12}, 2^{14}, 2^{16}, and 2^{18} KB. The download time for each test was recorded, and the results were plotted in the first 3D graph, which illustrates the relationship between the number of peers, buffer size, and download time.

This initial set of results provided valuable insight into the system's performance under different conditions. Subsequently, to approximate the behavior of the system and generate a predictive model, symbolic regression was employed. The goal of this regression was to derive a function that closely approximates the behavior of the observed data.

The resulting function is plotted in the second graph, which represents the model approximated by symbolic regression. This plot gives an analytical view of how download time scales with changes in the number of peers and buffer sizes. The derived function, based on the symbolic regression analysis, demonstrates a more general behavior of the system and allows for predictive analysis in different scenarios.

The symbolic regression process yielded the approximated function:

$$300 \cdot \ln(x^2 \cdot y)$$

Where: - x represents the number of peers,

- y represents the buffer size.

From this function, it can be concluded that increasing both the number of peers and the buffer size reduces the download time. However, there is an asymptotic point where increasing these parameters yields diminishing returns, making further increases less impactful on performance.

5.1 First Plot - Empirical Data

The first plot below represents the empirical data gathered during the tests:

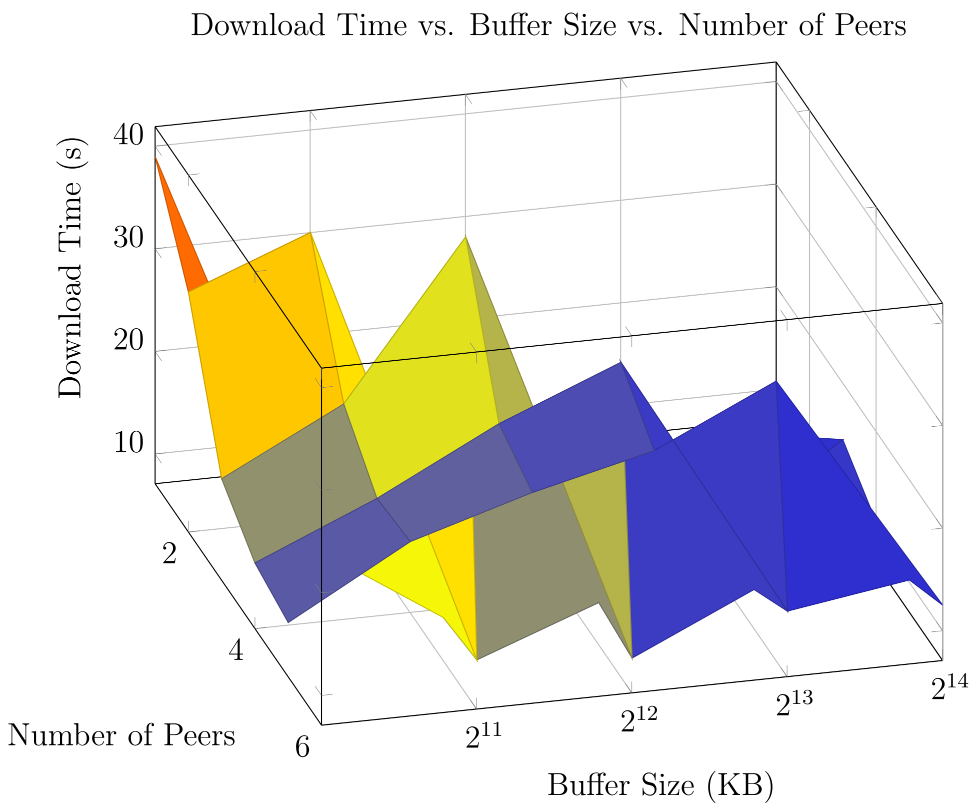


Figure 1: 3D Plot of Download Time, Buffer Size, and Number of Peers with Logarithmic Buffer Size.

5.2 Second Plot - Symbolic Regression Model

The second plot shows the function derived from symbolic regression, which approximates the system's performance based on the gathered data:

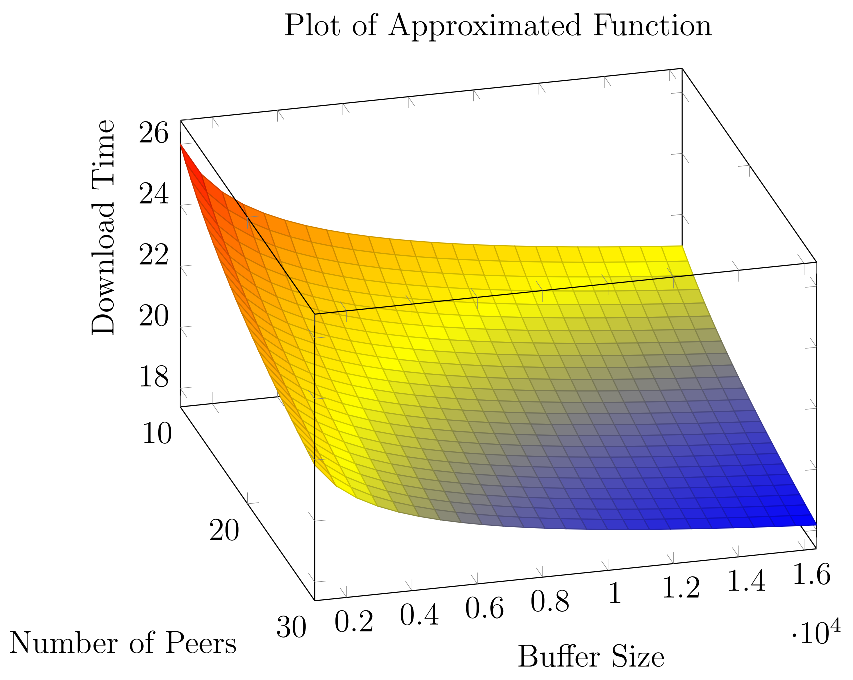


Figure 2: 3D Plot of the Approximated Function $300 \cdot \ln(x^2 \cdot y)$.

From the results, it can be concluded that increasing the number of peers reduces the download time significantly at first. However, as the number of peers continues to increase, the rate of improvement diminishes, indicating an asymptotic behavior where additional peers provide less benefit as their number grows.

Similarly, increasing the buffer size leads to a reduction in download time, especially at smaller buffer sizes. Beyond a certain point, however, further increases in buffer size yield diminishing returns, meaning that larger buffer sizes do not continue to improve performance at the same rate.

The function approximated by symbolic regression, $300 \cdot \ln(x^2 \cdot y)$, suggests that while increasing both the number of peers and the buffer size improves performance, there is a point of diminishing returns for both parameters. At larger buffer sizes and higher numbers of peers, the system reaches an asymptotic limit where further increases in these parameters provide little to no additional benefit.

6 Conclusions

The development and analysis of this Peer-to-Peer (P2P) File Exchange system has provided valuable insights into the performance and scalability of distributed file-sharing architectures. Through rigorous testing and performance evaluation, we observed that the system demonstrates strong scalability, especially in its initial scaling phase when increasing the number of peers. The use of concurrent downloads via multithreading was highly effective in reducing overall transfer times. However, performance gains diminished after reaching certain thresholds in both buffer size and peer count, as evidenced by our symbolic regression analysis. It's important to note that the optimal system performance is not easily defined, as it depends heavily on factors such as file size, network conditions, and the specific use case. Therefore, the conclusions drawn here are contingent upon the parameters tested, and further evaluation with larger files and different network configurations would be necessary to determine more precise optimal settings.

The hybrid approach used in this project—combining a central indexer with decentralized P2P file transfers—proved to be an effective compromise between fully centralized and purely P2P architectures. This approach, alongside the implementation of double-hashing for file verification, ensured strong data integrity throughout the file transfer process. The modular architecture, with separate components for the index server, peer server, and file requester, made the system easier to maintain, manage, and extend, ensuring flexibility for future improvements.

From a technical perspective, modern C++ features such as smart pointers and STL containers facilitated efficient memory management, while TCP as the transport protocol ensured data integrity and correct sequencing. The KMP algorithm for file searching enhanced the system's string-matching capabilities, contributing to its overall performance.

Regarding performance optimization, buffer size showed improvements up to a point, specifically around 8192 KB, after which further increases in buffer size resulted in diminishing returns. Similarly, the ideal number of peers for optimal file sharing appeared to be between 4-5 peers, with additional peers contributing less to overall performance beyond this range. The symbolic regression model $300 \cdot \ln(x^2 \cdot y)$ provided a useful approximation for predicting system performance based on varying peer count and buffer size.

While the system demonstrated practical real-world potential for distributed file sharing and was resilient to typical network conditions, the results of our tests highlight the need for further experiments, particularly with larger networks and more diverse file sizes. Additional testing with larger files and in environments with more peers could provide more insight into the scalability limits and optimization potential of the system.

Several areas for future improvement have been identified. These include optimizing peer selection algorithms to further improve download speeds, investigating faster methods for searching indices on the index server, such as algorithms like Aho-Corasick or Ukkonen, and implementing more granular chunk verification to increase transfer reliability. Furthermore, exploring encryption mechanisms for secure file transfers could enhance data security.

In conclusion, this project has successfully created a functional and efficient P2P file-sharing system, demonstrating the importance of performance analysis and optimization in distributed systems. The experience gained from this project has provided invaluable knowledge on network programming, concurrent programming, and the challenges involved in designing and implementing distributed systems. The mathematical model derived through symbolic regression offers a valuable tool for system optimization, and the lessons learned here will guide future developments in distributed systems and peer-to-peer networking technologies.

7 Experience Description

Through this project, a key lesson learned was the importance of avoiding a central server for many types of applications, such as in the case of Spotify. Centralizing the system would create a bottleneck, hindering scalability and performance. This highlighted the value of decentralized architectures, particularly in systems requiring high availability and distributed resource management.

In terms of technical challenges, initially working with TCP was somewhat uncomfortable, as it required managing low-level details of communication and connection handling. However, by modeling the servers as classes, we were able to abstract the underlying complexities. This allowed the team members to focus on the high-level functionality of the servers, such as the available methods and their behaviors, without needing to understand the intricate workings behind them. This approach not only made the codebase more modular and maintainable but also streamlined the development process by simplifying the tasks at hand. With the framework in place, the focus could shift to higher-level features, improving the efficiency and productivity of the development process.

8 Learnings

8.1 Danilo Duque

This project helped me realize the importance of Peer-to-Peer (P2P) systems in real-world applications, where avoiding central servers prevents bottlenecks and enhances scalability. I also gained a deeper appreciation for pattern matching algorithms like KMP and Aho-Corasick, which are crucial for efficient data handling and search operations in large systems.

8.2 Desireé Avilés

Throughout this project, I learned the significance of modular system design and how clear separation of concerns can simplify complex systems. The experience of working with multiple components in a distributed environment taught me how to ensure that each piece works harmoniously while allowing flexibility for future improvements. Additionally, I gained hands-on experience in implementing data integrity checks, which are essential for maintaining system reliability.

8.3 Emanuel Rojas

This project allowed me to understand the challenges of optimizing file transfer speeds in a distributed system. I realized how critical it is to manage network resources effectively to minimize delays and maximize throughput. I also learned the importance of optimizing algorithms and system parameters, as small changes can significantly impact overall performance. Furthermore, I gained practical experience working with multithreading, which provided a solid foundation for managing concurrent tasks in real-time systems.

9 Bibliography

References

[1] D. Duque, *Evolutionary Computation*, GitHub repository, 2024. [Online]. Available: `https://github.com/DaniloDuque/Evolutionary-Computation`. [Accessed: Nov. 2024].

[2] M. Hunicken, *ICPC Team Notebook - El Vasito*, GitHub repository, 2024. [Online]. Available: `https://github.com/mhunicken/icpc-team-notebook-el-vasito`. [Accessed: Nov. 2024].

[3] *C++ Reference*, 2024. [Online]. Available: `https://en.cppreference.com/w/`. [Accessed: Nov. 2024].

[4] IBM, *Transmission Control Protocol (TCP)*, IBM Documentation, 2024. [Online]. Available: `https://www.ibm.com/docs/es/aix/7.1?topic=protocols-transmission-control-protocol`. [Accessed: Nov. 2024].