Project: Hybrid Peer-to-Peer File Sharing

Name, Net Id: Laksh Kotian LK596

Resources: https://github.com/LakshK98/hybrid-p2p-file-sharing

1 Introduction

Unified endpoint management(UEM) solutions like Zenworks and BigFix allow admin machines to monitor all the machines within the org in which they are deployed. They have very detailed knowledge of the machines in the org and are responsible for many maintenance activities like inventory management and patch/update deployment. The servers in these UEM products send the same updates or bundles to hundreds of machines in the organization. This takes up a lot of server resources and time. Horizontal scaling of servers does fasten the process but is can be costlier as more servers are provisioned. We are proposing a hybrid p2p approach to provide these updates so that clients within a network can share updates among peers thus taking some load off the server and effectively replicating patches/updates/bundles on other machines. The UEM servers can also use the awareness of client capabilities to their advantage in setting up peers that perform the transfer.

2 Existing Solution

Some of the most widely used P2P file-sharing solutions are: Pure p2p protocols like bitorrent(bep 0003.rst post) are built to handle highly un- reliable peers which is why they send a lot of intermediate messages to ensure synchronization between peers, in a UEM setting the clients are usually reliable org devices and thus we can avoid most of this overhead.

Hybrid p2p approaches to file transfer do exist as seen in (https://link.springer.com/chapter/10.1007/978-3-540-70621-2_18 https://www.researchgate.net/figure/A-hybrid-P2P-file-sharing-network_fig1_31398827). However, these approaches assume the server as the only secure endpoint and do not give any of the peer's responsibility of controlling the peers. The decision of which peer transfers is made solely by the server. In UEM settings the server knows which servers are in secure locations, (like within restricted access zones) thus we can encourage peers to take up the responsibility of the server in case of server outage or maintenance.

3 Description

Consider a server that wants to push an update to a pool of 100 clients. The server initially sends the data to k clients where k stands for the spread factor (Refer to Figure 1(left), k here is set to 1. When the transfer is complete the client sends an ack that signifies that this client is capable of becoming a transmitter for the data. The server receives this ack and sets this client as the transmitter for another client say clientX

ClientX is removed from the transfer pool and put on a watch pool. The server can now select another client from the pool to send data to since the number of clients it is sending data to has dropped below k. When clientX has the update it sends an ack to the server which can then remove it from the watch pool and set clientX as a transmitter. In this way, multiple machines are transmitting at the same time leading to faster transmission.

We also plan to support the promotion of one of the peers as the server, this can help in two scenarios. One is when there is a server outage or if the server goes for maintenance. Other

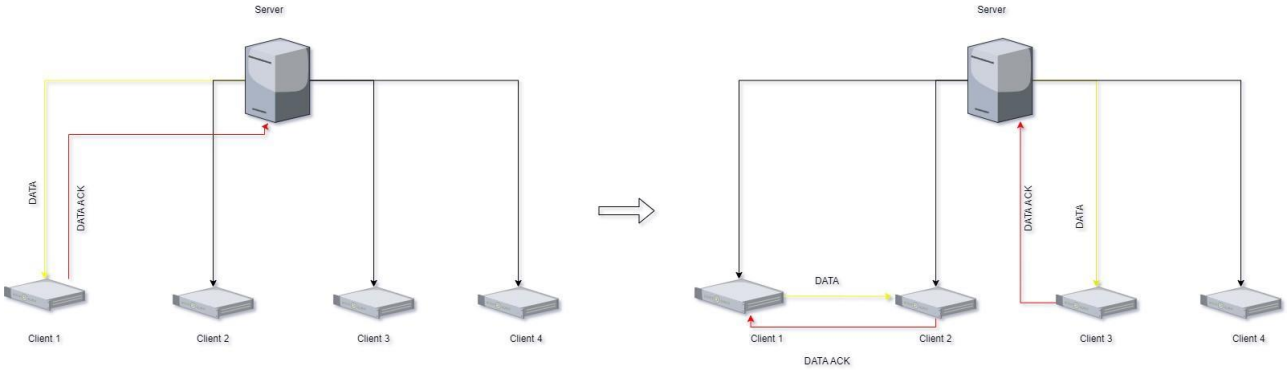


Figure 1: Topology

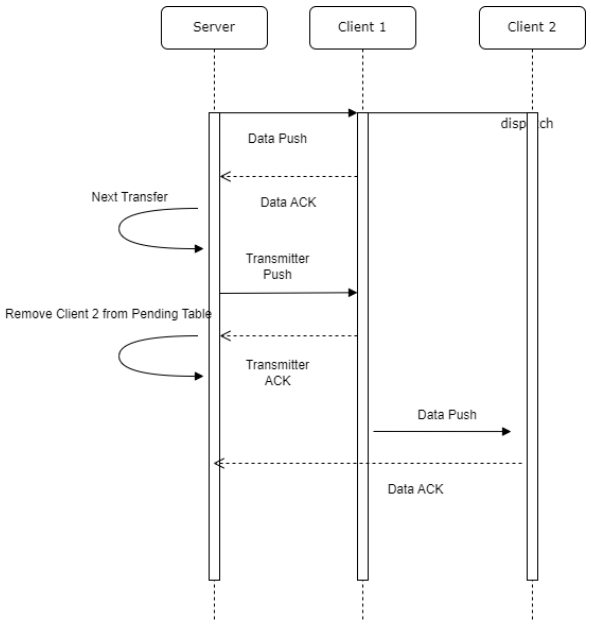


Figure 2: Data Transfer

is when the server is overloaded with other work, it can delegate the responsibility of performing file transfer to the peers. To show this possibility our server will mark the initial k client it sends data to as a candidate server and assign rank based on chronology (in the actual setting ranking will be based on computing power). When the server disconnects the client with the highest rank becomes the server.

An aside about possible future improvements. Since we are talking about smaller managed networks, the server has details regarding the capabilities of client machines. The server can make smarter decisions about assigning clients as transmitters. This could be done by selecting a different spread factor k' for the transmitter based on its availability and proximity to other clients. Thus every transmitter would be responsible for k' clients increasing overall propagation speed.(Refer Figure 1(Right))

4 Server-Client Architecture(Sequential-Share)

A server-to-client file-sharing application is a software tool that enables a server to share files with multiple clients over a network. The server-to-client file-sharing application using TCP sockets in Python utilizes a client-server architecture where the server stores the files and serves

them to clients upon request. The application uses TCP sockets to establish a reliable, two-way communication channel between the server and clients. Clients send requests to the server for specific files, and the server responds by sending the requested file over the established TCP connection(Refer to Figure 3).

Benefits: The server-to-client file-sharing application using TCP sockets in Python offers several benefits over other types of file-sharing solutions. One of the main advantages of this type of application is that it is highly reliable, as the use of TCP sockets ensures that data is transmitted reliably and in order. Another benefit is that the files being shared are stored on a central server, which ensures that the files are available at all times and can be easily managed.

Challenge: One of the main challenges is that the server may become overloaded if there are too many clients requesting files simultaneously, which can cause delays or even crashes

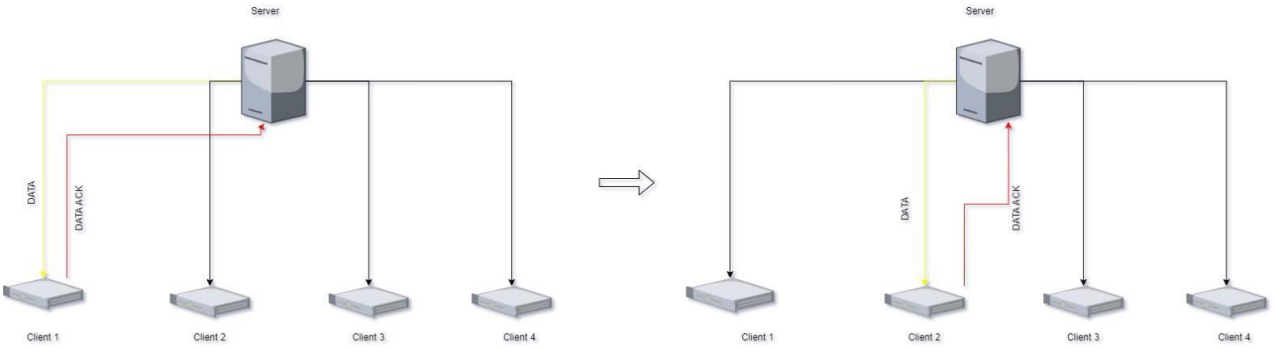


Figure 3 Data Transfer

Implementation:

Server: The code defines a class called ”Server” that implements the server side. The class has several methods to handle incoming connections from peers and upload data to those peers. The code uses Python’s built-in socket module to create a socket and bind it to a specific IP address and port number. The constructor takes a message (in bytes) as input, which will be sent to the connected peers upon their request. The code also creates two lists to keep track of connections and peers

The ”run” method is the main loop that listens for incoming connections. Upon accepting a new connection, it adds the peer to the list of peers and sends the updated list of peers to all connected peers. It then creates a thread to handle the connection and adds the connection to the list of connections. It also updates a plot of the time elapsed since the start of the server and the number of connected peers.

The ”handler” method is responsible for handling incoming requests from peers. It pops a connection from the ”connections2” list and receives data from the peer. If the received data is a disconnect signal, it removes the connection from the list of connections. Otherwise, if the received data is a request for the uploaded message, it sends the message back to the peer.

Client: The code defines a class called ”Client” that implements the Client-side. The purpose of this client is to download a file from the network. The constants module contains values that are used throughout the program, such as the port number to use.

The Client class contains an ”init” method that sets up a socket connection to the specified address and port number. It also starts a thread to send a message and waits for a response from the server. When a response is received, it updates a list of peers and creates a file on the client side.

The ” receive-message” method handles the receiving of messages from the server. It decodes the

received data and prints it to the console. If the data has changed since the last received message, it creates a new file on the client side using the fileIO module.
The ”send-message” method sends a message to the server. In this case, it sends a request to download the file.

5 Peer-Peer Architecture

The P2P file transfer application using TCP sockets in Python utilizes a decentralized architecture where each device acts as a client and a server. The application uses TCP sockets to establish a reliable, two-way communication channel between devices.The result is a highly efficient and scalable system where files can be shared quickly and easily. For our specific implementation,

Modified the producer-consumer problem
For our approach we need the server and the client both to be able to send files. This makes the problem at hand a modified version of the producer-consumer problem where the consumer becomes a producer after it consumes data. To tackle this we maintain a queue of producers (machines that have received the file) at the server. Whenever a client receives a file, it sends an ack to the server who then marks the client as a producer by putting it into the producer queue. When a new client enters the system the server gets a producer from the queue and assigns it as the producer for this particular client. If the producer queue is empty the client thread is blocked till the next producer becomes available.

Pseudo code:

```
# Shared queue
producer_queue = queue.Queue()

#initially server is the producer
producer_queue.put(server)

# Consumer function
def handle_incoming_consumer(consumer):
    producer = producer_queue.get(True)
    while True:
        consume_item(item)

    producer.send(consumer)

    # wait for acknowledgment
    producer.recv(1)

    producer_queue.put(consumer)
    producer_queue.put(producer)


# Create and start the threads
producer_thread = threading.Thread(target=handle_incoming_consumer)
```

Python queues are thread-safe which means that one machine will get assigned as the sender for one and only one client. The True parameter to the get() function ensures that the thread is blocked till a producer is available to serve a client.

Detailed Explanation

Server:
The Server is responsible for sending files to certain clients and also for assigning clients that would act as servers. The server creates a producer queue and adds itself to it , it then creates a socket that listens for incoming client connections. When a client makes a connection to the server the server accepts it and the connection socket is then sent as an argument to the consumer thread. On making the connection with the server the client sends an init message that contains the ip and port of the socket on which the client is listening for incoming connections.
The consumer_thread stays active till a client has received the file. To do this the server first queries the producer queue. If it is empty, this queue will get blocked till one of the other clients has the file or the server itself is free to make the transfer.
If the next producer is the server, it will make the transfer directly to the client and after the transfer is complete, both the server and the client will get added to the producer queue.
If one of the clients is retrieved from the producer pool, the server sends the details of the client's listening socket so that it can make a p2p connection with the client and share the file with it. The thread is blocked then till the client that is

supposed to receive the file sends an ack to the server. Just as before both clients get added to the producer pool after the transfer is complete.

Client:
The client initially receives the file from the server, then based on the server's input starts transferring the file to another client. To do this the client initially creates 2 sockets. One socket is used to listen for connections made by other clients (p2p socket) and one is used to establish a connection with the server.
Initially, after the connection is established with the server, the client sends a message in which the IP address and port number of the p2p socket are sent to the server. The server can then send this to the peer that it wants to assign as the sender to this client.
Next two threads are initialized, one to receive communication from the server (server_thread) and the other to receive connection from other peers (peer_thread). The peer_thread is responsible for receiving files from other peers
On the server_thread we handle two types of messages: UPLOAD_START and PEER_CONNECTION
UPLOAD_START:

- After receiving this message the next set of bytes is the length of the file being sent. We need to read exactly this many bytes so that we don't lose any further messages sent by the server.
- The bytes are written into a file in the client_data folder. After the entire file is written, a transfer_complete ack is sent to the server.

PEER_CONNECTION:

- In this message the server sends the connection details of a peer to which this client is supposed to send the file to.
- We create another socket and connect to the peer ip and send the file to it , which will be handled on the peer's side by its peer_thread

Benefits: One of the main advantages of this type of application is that it is highly scalable, as each device becomes a source for other devices to download. This means that as more users join the network, the download speed can actually increase.

6 Metrics

To ensure that our approach leads to faster propagation of files, we will conduct a test to compare the time it takes for a server to send a file to multiple clients individually, with the time taken by our approach. We will record the total time for different numbers of clients, aiming to determine if our approach leads to faster overall propagation of files.

We ran both Sequential Share and our hybrid Peer to Peer approach on ilab machines for 100MB, 500MB, and 1GB files
The server was hosted on **h257-1.cs.rutgers.edu.** The following machines were used as clients.

- kill.cs.rutgers.edu
- grep.cs.rutgers.edu
- cd.cs.rutgers.edu
- ls.cs.rutgers.edu
- man.cs.rutgers.edu
- pwd.cs.rutgers.edu
- rm.cs.rutgers.edu
- top.cs.rutgers.edu
- vi.cs.rutgers.edu
- less.cs.rutgers.edu

The Observed Elasped time in File Transfer among the clients in Sequential Share and Hybrid Peer-to-Peer File Transfer is as follows:

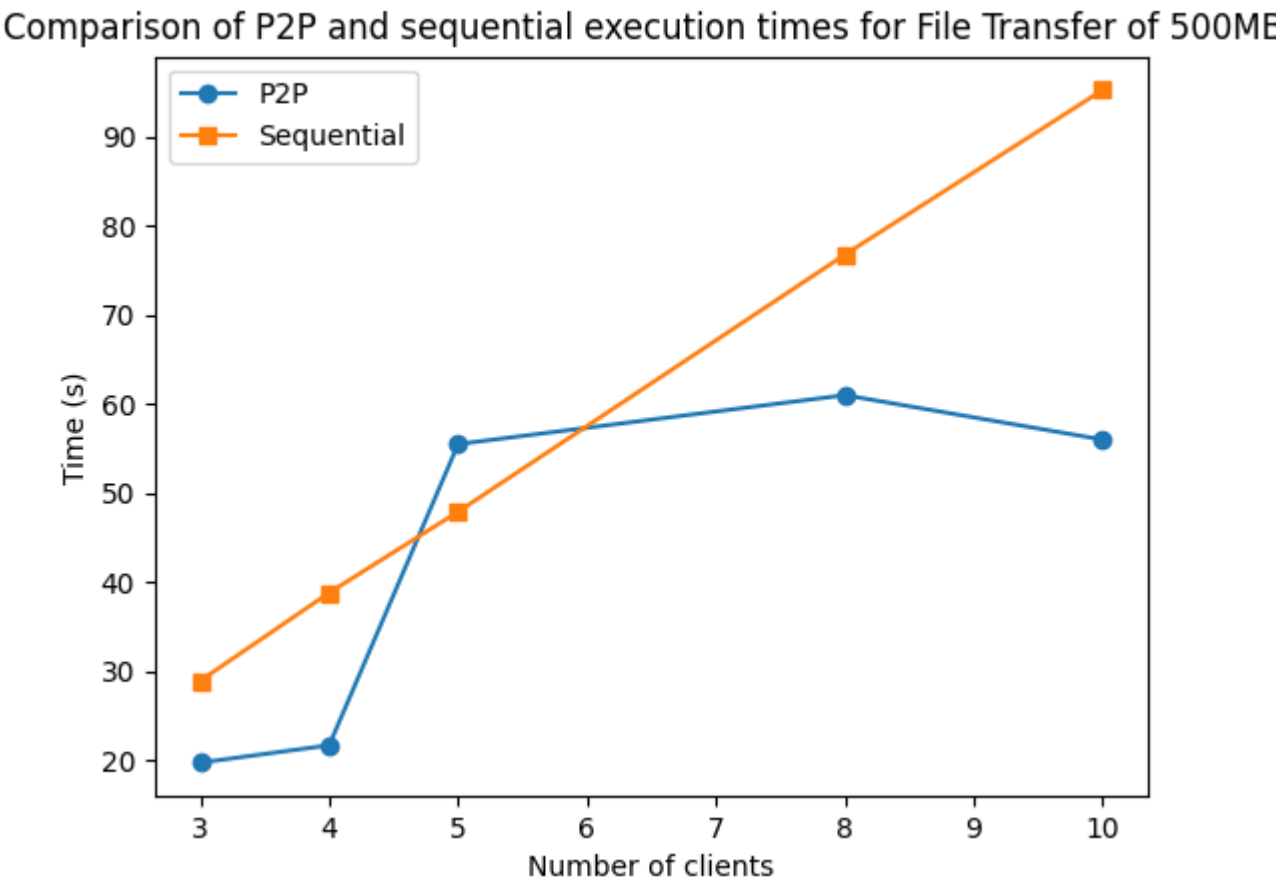
Case1: File of 100MBs.

Type / No. of clients	3	4	5	8	10
P2P	4.08s	6.14s	10.53s	13.33s	18.4s
SEQ	6.044s	7.53s	9.24s	14.8s	18.52

Observation: With a small number of clients (3-5), there is not much difference in transfer time between P2P and Client-Server approach.
However, as the number of clients increases, P2P outperforms the Client-Server approach in terms of transfer time. This is because P2P allows clients to download file from each other, reducing the load on the server and improving overall transfer speed. For 100MB files, client-server transfer is similar to P2P transfer as the overhead of establishing a P2P connection is not justified.

Case 2: File of 500MBs.

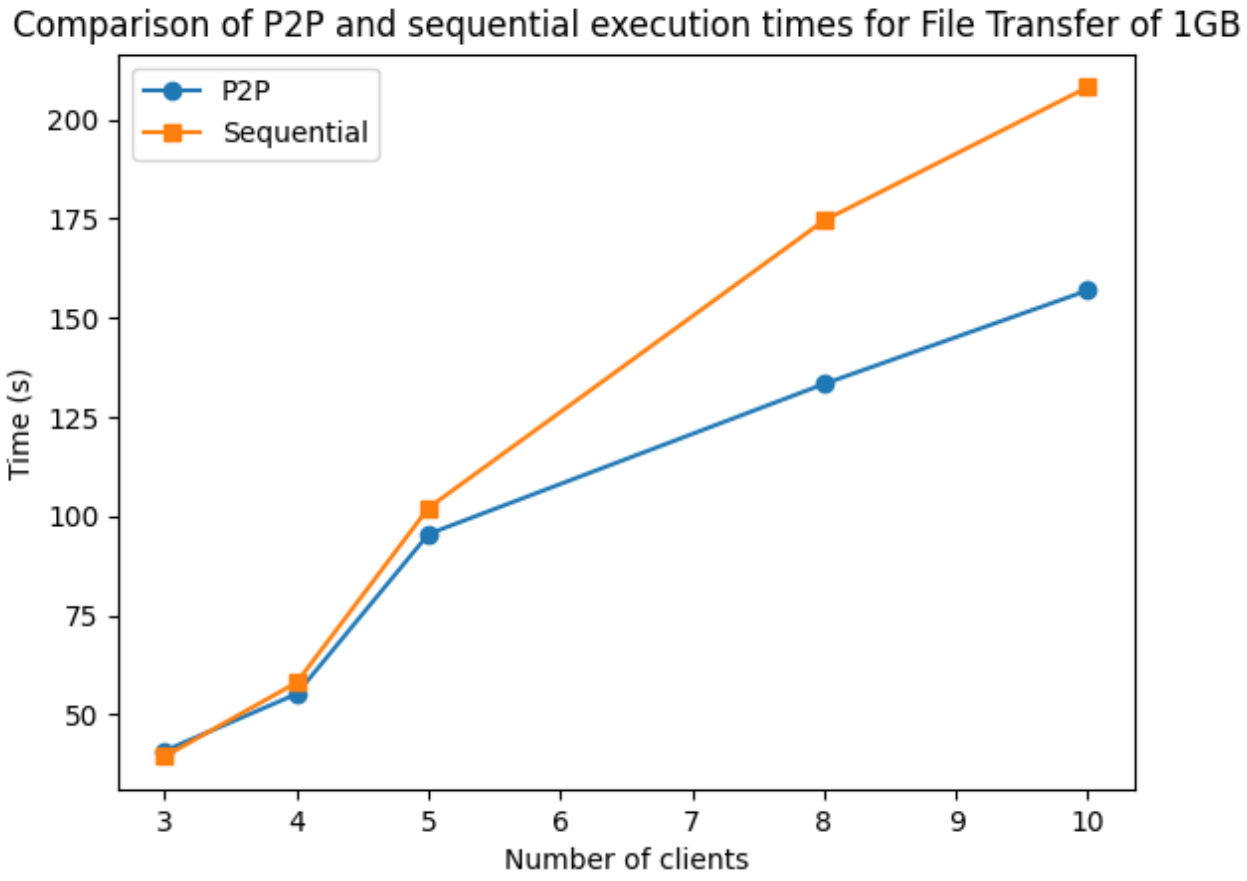
type / # of clients	3	4	5	8	10
P2P	19.7s	21.65s	55.5s	61s	56s
SEQ	28.9s	38.8s	47.9s	76.8s	95.3s



Observation: For 500MB files, both P2P and client-server transfer methods are comparable in terms of transfer time. However, P2P transfer may be slightly faster due to parallel downloads. Similar to 100 MB files, P2P outperforms the Client-Server approach as the number of clients increases. The difference in transfer time becomes more pronounced as the number of clients increases.
However, even with a small number of clients, P2P is generally faster than the Client-Server approach.

Case 3: File of 1GB

Type / Noof clients	3	4	5	8	10
P2P	40.927	55.24	95.51	133.3	156.9
SEQ	39.5	58.12	102.1	174.50	208.1



Observation: P2P significantly outperforms the Client-Server approach in all scenarios. Even with a small number of clients, P2P is much faster than the Client-Server approach.
As the number of clients increases, the difference in transfer time between P2P and Client-Server becomes even more pronounced.
For 1GB files, P2P transfer is significantly faster than client-server transfer. With multiple peers and seeds, P2P transfer allows for parallel downloads, reducing the overall transfer time.

Inference: Overall, it can be concluded that P2P is generally faster than the Client-Server approach for file transfer, particularly for larger files and as the number of clients increases while client-server transfer may be more efficient for smaller files. However, the performance of P2P can be affected by factors such as network bandwidth, latency, and the number of peers available for download Therefore, it is important to consider these factors when choosing an approach for file transfer.