CSE-3111: Computer Networking Lab

Adaptive, Fault-Tolerant P2P File Transfer System

Submitted by:

Tazkia Malik
Roll: 07

Taif Ahmed Turjo
Roll: 45

Submitted to:

Dr. Ismat Rahman
Associate Professor, Dept. of CSE, University of Dhaka

Mr. Palash Roy
Lecturer, Dept. of CSE, University of Dhaka

Mr. Jargis Ahmed
Lecturer, Dept. of CSE, University of Dhaka

Department of Computer Science & Engineering
University of Dhaka

Friday 18th July, 2025

Contents

Introduction

The project presents a **peer-to-peer file transfer system** designed for direct, one-to-one file sharing between users, ensuring fault-tolerance, high throughput, and end-to-end reliability. Users browse a central repository and connect directly to peers hosting the requested files.

File transfers are managed by a custom protocol named **AFT-FTS (Adaptive, Fault-Tolerant File Transfer System)**. This protocol intelligently selects between TCP and a custom Reliable-UDP transport, dynamically adapts congestion control strategies (Tahoe, Reno, or delay-based), supports crash recovery with resumable transfers, and ensures exactly-once delivery of each file chunk.

In contrast to BitTorrent-style swarm downloads, this system prioritizes **smart, simplified one-to-one P2P transfers**, making it ideal for challenging environments such as lossy Wi-Fi, VPNs, or high-latency links. It serves as a practical, hands-on application of key networking principles including socket programming, congestion control, protocol design, and secure file integrity.

Objectives

- **Build a Peer-to-Peer Transfer Protocol:** Design a one-to-one smart file transfer system that adapts to real-world network conditions.
- **Ensure Reliability and Resume Support:** Implement mechanisms to guarantee exactly-once delivery and crash-resilient transfer resume.
- **Apply Congestion Control Dynamically:** Integrate and switch between Tahoe, Reno, and delay-based algorithms based on observed throughput and network health.

Problem Statement

Traditional file transfer methods, such as FTP and centralized cloud-based solutions, face several limitations in terms of performance, security, and reliability—especially under constrained network conditions. This project aims to overcome these challenges by designing a direct peer-to-peer (P2P) file transfer system using a custom protocol.

- **Eliminating centralized bottlenecks:** Avoids the need for central servers, reducing latency and removing single points of failure.
- **Improved reliability:** Supports fault tolerance and crash recovery with resumable transfers.
- **Adaptive protocol selection:** Dynamically chooses between TCP and custom Reliable-UDP based on network conditions.
- **Intelligent congestion control:** Automatically switches between congestion control strategies (Tahoe, Reno, or delay-based) for optimal performance.
- **One-to-one optimized transfer:** Focuses on direct, private transfers instead of swarm-based models like BitTorrent.
- **Secure data transmission:** Ensures exactly-once delivery and data integrity using cryptographic techniques.
- **Real-world applicability:** Designed to perform reliably in lossy, high-latency, or VPN-based environments.

Objectives

- **Develop an Intelligent Peer-to-Peer Transfer Protocol:** Create a direct, one-to-one file transfer system that intelligently adapts to varying network conditions, ensuring optimal performance in both stable and lossy environments.
- **Guarantee Reliable Delivery with Crash Recovery:** Implement a robust mechanism to ensure exactly-once delivery of file chunks, along with support for seamless transfer resumption in the event of system or network failures.
- **Incorporate Adaptive Congestion Control Strategies:** Dynamically apply and switch between congestion control algorithms such as TCP Tahoe, Reno, and delay-based techniques based on real-time analysis of network throughput, delay, and congestion status.
- **Enable Protocol-Level Transport Selection:** Design the system to choose between TCP and a custom Reliable-UDP transport layer depending on current network characteristics like packet loss and latency.
- **Ensure Data Security and Integrity:** Integrate cryptographic techniques to verify file integrity and protect data during transmission against tampering or interception.
- **Design for Real-World Network Conditions:** Optimize the protocol for performance over challenging networks, such as Wi-Fi, VPNs, and high-latency links, ensuring robustness and fault tolerance.
- **Provide a User-Centric Interface (Optional):** Optionally include a simple interface or CLI for users to easily select files, view transfer progress, and handle errors or resumes.

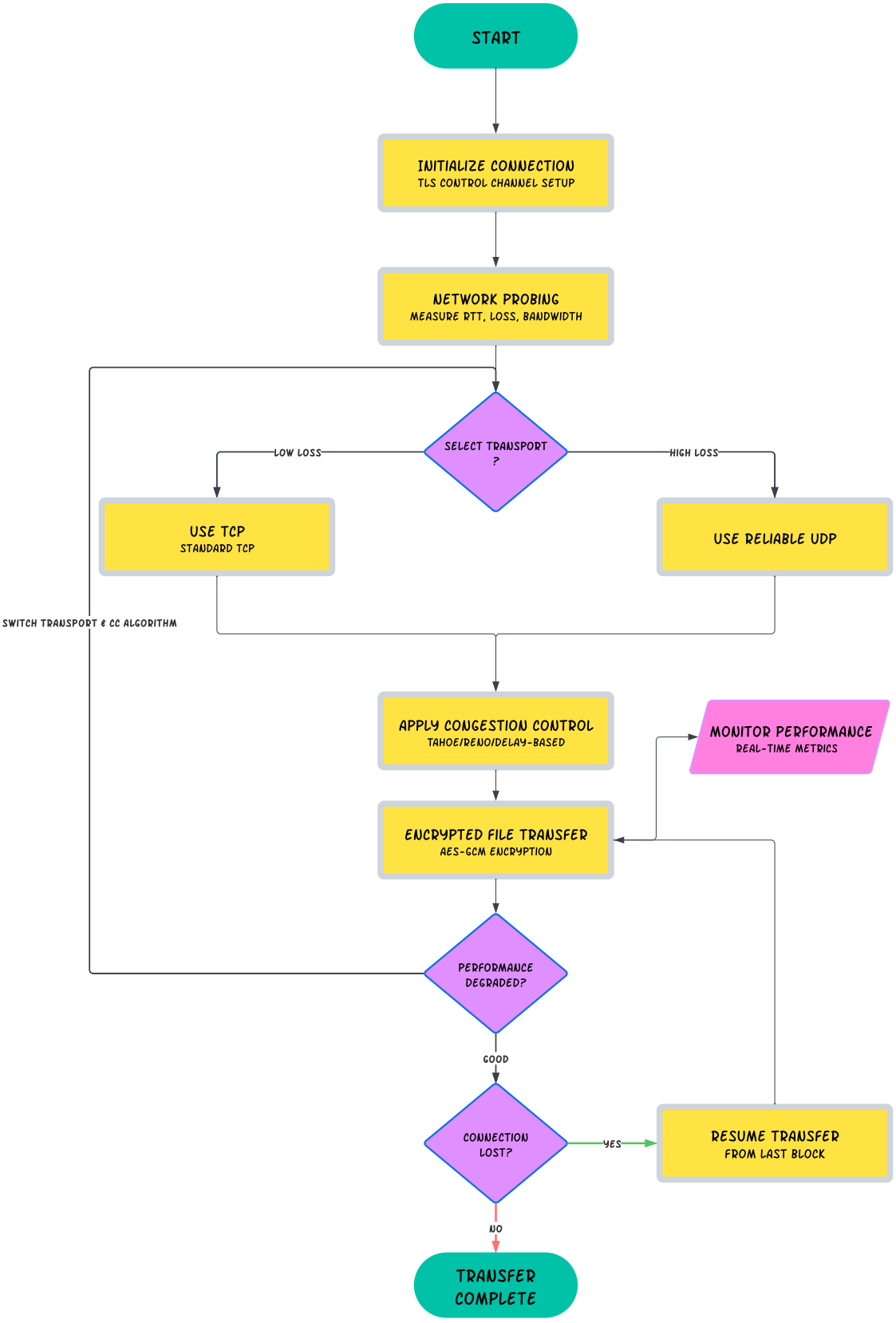
Project Features

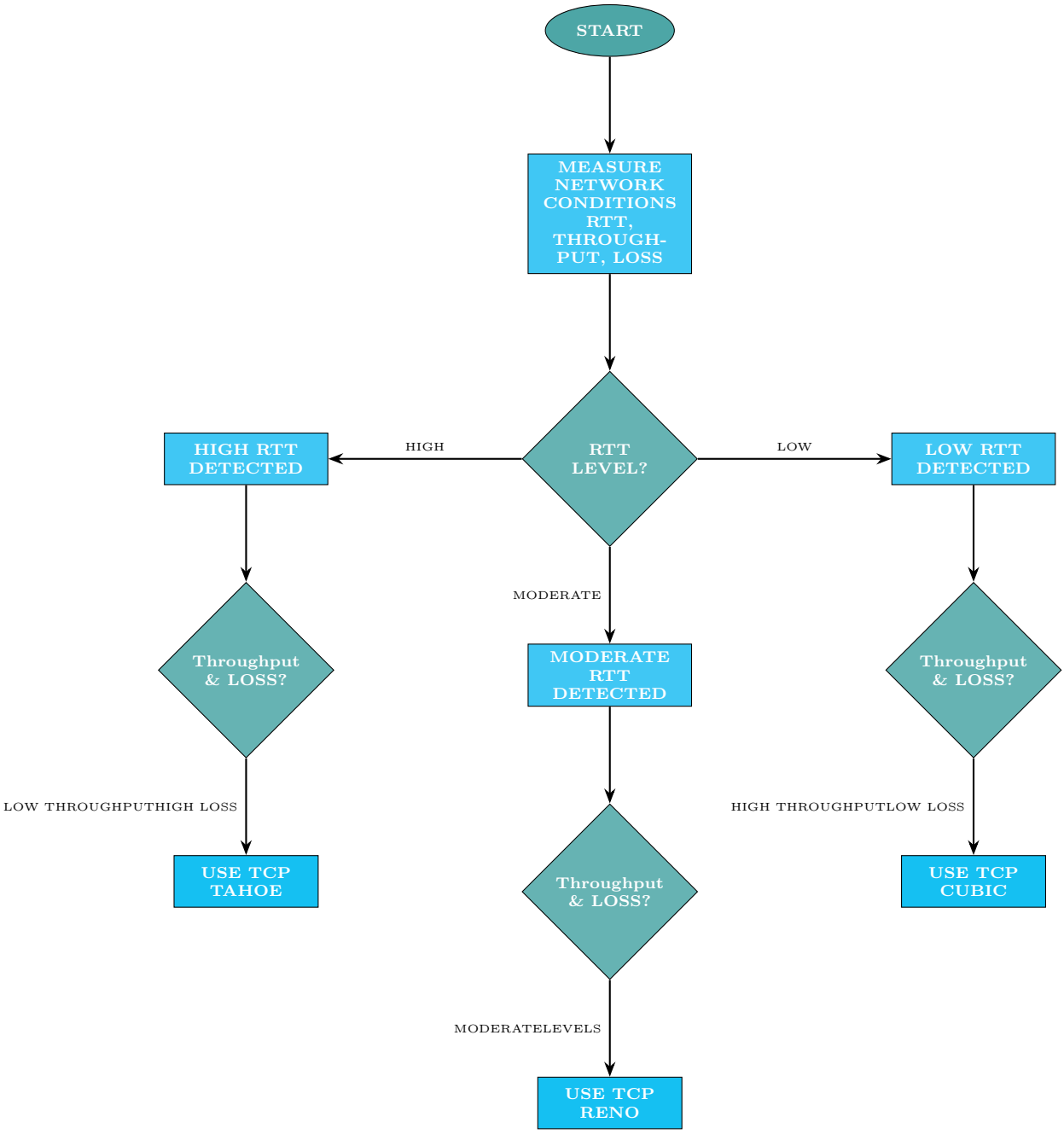
- **Interactive File Repository:** A user-friendly, searchable repository where users can browse and search for available files (e.g., movies, datasets). The repository dynamically shows file availability and hosting peers.
- **Visualized File Transfer Process:** File transfers are graphically represented in real-time, showing progress, transfer rates, and connection status to enhance user experience and transparency.
- **Direct Peer-to-Peer File Transfer:** The system establishes a direct connection between the requesting peer and a hosting peer, bypassing server bottlenecks. Transfers are one-to-one rather than swarm-based.
- **Adaptive Transport Protocol Selection:** Before each transfer, the system probes network conditions such as latency, packet loss, and UDP reachability to select the optimal protocol:
  - **TCP:** Reliable, stable network delivery.
  - **Reliable-UDP:** Faster delivery on lossy or high-latency networks.
- **Pluggable Congestion Control:** Supports multiple congestion control algorithms:
  - **TCP Tahoe**
  - **TCP Reno**
  - **Delay-Based Algorithm**

  The system monitors throughput and can switch algorithms mid-transfer to maintain optimal efficiency.
- **Exactly-Once Block Transmission:** Files are divided into fixed-size blocks with unique IDs and checksums. Duplicate or corrupted blocks are detected and filtered using persistent bitmaps to ensure data integrity.

- **Crash Recovery and Resume Support:** Transfers can resume seamlessly after crashes, with both peers maintaining bitmap logs to retransmit only missing blocks.
- **End-to-End Security:**
  - **TLS-based Control Channel** for secure connection setup.
  - **AES-GCM or ChaCha20-Poly1305** encryption for data blocks.
  - Guarantees confidentiality, integrity, and protection against replay and tampering attacks.
- **Real-Time Performance Monitoring:** Visual tracking of throughput, RTT, block resend rate, and congestion window size. Supports export in JSON/CSV formats for further analysis.
- **Dynamic Protocol Switching:** Transfers can switch between TCP and UDP modes dynamically during operation based on real-time network performance.
- **User-Friendly Interface with Search and Visualization:** Provides an interactive interface allowing users to search and filter files within the repository and chatting capabilities with peers, with intuitive visualization of ongoing transfers. Future enhancements may include a full-featured GUI or web-based client.

Block Diagram / Work Flow Diagram




START
MEASURE NETWORK CONDITIONS RTT, THROUGH-PUT, LOSS
RTT LEVEL?
HIGH
LOW
MODERATE
HIGH RTT DETECTED
LOW RTT DETECTED
MODERATE RTT DETECTED
Throughput & LOSS?
Throughput & LOSS?
Throughput & LOSS?
LOW THROUGHPUTHIGH LOSS
HIGH THROUGHPUTLOW LOSS
MODERATELEVELS
USE TCP TAHOE
USE TCP CUBIC
USE TCP RENO

Tools & Technologies

Tool	Purpose
Spring Boot	A Java-based framework used to build the backend RESTful API, providing rapid development, embedded servers, and production-ready features. Handles business logic, file transfer protocols, and peer-to-peer communication.
Java	The programming language used for backend development due to its portability, performance, and rich ecosystem. Implements core logic, network protocols, and security features, including WebSocket support to enable real-time, bidirectional communication between peers during file transfers.
Next.js	React-based frontend framework used for building the interactive, server-side rendered user interface. Enables fast, SEO-friendly search and browsing of the file repository with dynamic visualization of file transfers.

Table 1: Tools and technologies used in the project with their purposes

Networking Concepts

This section outlines the networking concepts implemented in the backend of our project, **AFT-FTS: Adaptive, Fault-Tolerant File Transfer System (TCP Only)**. The project leverages core principles from the networking course to build a secure, efficient, and reliable peer-to-peer file transfer system.

8.1 Socket Programming and Peer-to-Peer Architecture

We used **Java Sockets** to establish direct TCP connections between peers without relying on centralized servers or swarming techniques. This peer-to-peer architecture allows for decentralized communication and more efficient resource usage.

8.2 Round-Trip Time (RTT) and Throughput

RTT and throughput measurements were utilized to evaluate and dynamically adapt the network performance. These metrics guided the selection of congestion control algorithms and helped monitor the health of ongoing transfers.

8.3 Congestion Control

To handle varying network conditions, we implemented multiple **congestion control algorithms**:

- **Tahoe and Reno:** Classic loss-based algorithms for conservative congestion handling.
- **Cubic:** A modern TCP algorithm optimized for high-speed, long-distance networks.
- **Delay-based Control:** Adjusts sending rate based on RTT spikes to avoid bufferbloat.

These algorithms were selected and switched adaptively based on real-time network feedback.

8.4 Reliable Data Transfer and Crash Recovery

The system ensures reliable delivery using acknowledgments and retransmissions. To enhance robustness, we implemented the **Go-Back-N ARQ protocol**, enabling crash recovery by resending lost or unacknowledged segments without complex state synchronization.

8.5 Secure Transmission

Data confidentiality and integrity were prioritized using **AES encryption**. Files are encrypted before transmission and decrypted on the receiving side, ensuring secure end-to-end delivery over unsecured networks.

8.6 Flow Control

TCP's built-in **flow control mechanisms** (e.g., window size) were leveraged to prevent the sender from overwhelming the receiver. This ensures smooth data delivery even in resource-constrained environments.

8.7 FTP Protocol Usage

The project employs FTP-like semantics for file transfer, such as file listing, initiation, and completion signaling. However, the transmission is carried over secure TCP sockets with custom logic for adaptive and fault-tolerant behavior.

Implementation Details

9.1 AFT-FTS Protocol Implementation

The `AftFtsProtocol` class provides the core logic for the Adaptive Fault-Tolerant File Transfer System (AFT-FTS). It orchestrates peer-to-peer (P2P) TCP connections for efficient file transfers between backend instances, ensuring reliability, performance monitoring, and transfer state management.

Key Functionalities:

- **Block-based Transfer:** Files are divided into blocks (default 1MB) and transferred individually. This block granularity enables resumption, error recovery, and parallelization.
- **P2P TCP Connections:** The protocol supports direct TCP connections between backend peers, facilitating efficient data transfers without intermediary servers. For browser requests, it intelligently redirects to suitable backend peers hosting the requested files.
- **Concurrent Transfers:** It limits the number of simultaneous transfers to prevent resource exhaustion, using a configurable `maxConcurrentTransfers` parameter.
- **Transfer State Management:** Active transfers are tracked using a thread-safe `ConcurrentHashMap`. Each transfer session stores metadata including progress, status, start time, and performance metrics.
- **Progress Monitoring and Callbacks:** The protocol uses callback interfaces to provide real-time feedback on transfer progress, completion, and error events, enabling UI updates or logging.
- **Retry and Error Handling:** Transfers support retries with a configurable maximum number of attempts. Errors are caught and update the session status accordingly.
- **Background Tasks:** Scheduled background jobs periodically update transfer performance statistics (such as transfer rate) and clean up stalled or timed-out transfers.
- **Pause, Resume, and Cancel:** The protocol provides control methods to pause, resume, or cancel active transfers, adjusting their session status appropriately.
- **Crash Recovery Support:** By maintaining up-to-date session states and transfer progress, the system can resume interrupted transfers seamlessly.

Design Considerations:

- **Thread Safety:** Use of `ConcurrentHashMap` and asynchronous operations with `CompletableFuture` ensures safe concurrent access and non-blocking behavior.
- **Scalability:** The system is designed to handle multiple concurrent transfers and peers, with efficient filtering and selection logic for file hosting peers.
- **Flexibility:** The protocol can serve both backend-to-backend transfers and browser-triggered downloads, adapting to different client scenarios.

Overall, the `AftFtsProtocol` class encapsulates the core mechanisms needed for a resilient, performant, and adaptive file transfer system suitable for distributed backend environments.

9.2 Initialization

- **Service startup:**
  - Spring Boot component ‘P2pTcpServer‘ binds a ‘ServerSocket‘ on configured TCP port (default 9000).
  - A fixed thread pool (size 10) is created for handling concurrent peer connections.
  - The server begins accepting connections on a background thread when annotated ‘@PostConstruct‘.
  - Shutdown is managed in a ‘@PreDestroy‘ method that stops listening and cleans up threads.
- **Directory and security setup:**
  - ‘P2pTcpClient‘ ensures a local ‘./receivedfiles/‘ directory exists.
  - It also initializes encryption support for AES-based session keys.
- **PeerService background jobs:**
  - A scheduled executor runs two maintenance tasks:
    - *Offline cleanup*: removes peers not seen in the last 10 minutes (every 2 min).
    - *Session cleanup*: evicts file transfer sessions older than 1 hour, or those marked completed/-failed (every 5 min).

9.3 Peer Management

- **Registering peers:**
  - ‘registerPeer(ip, tcpPort, udpPort, publicKey)‘:
    - Generates a unique ‘peerId‘ via a UUID from IP:port:timestamp.
    - Performs network capability detection: TCP/UDP support, bandwidth, cipher preferences.
    - Runs a connectivity probe to estimate latency, jitter, packet loss, and bandwidth.
    - Stores the peer in ‘activePeers‘ map with timestamp and quality metrics.
- **Updating peer liveness:**
  - ‘updatePeerStatus(peerId)‘:
    - Refreshes ‘lastSeen‘ to now and retains peer as online.
    - Periodically re-probes network quality if last probe was older than 5 minutes.
- **Unregistering peers:**

  - 'unregisterPeer(peerId)':
    * Removes peer from active list and flags as offline.

- **Querying peers:**
  - 'getActivePeers()':
    * Returns a sorted, live peer list ordered by 'lastSeen' descending.
  - 'findPeersByCapability(...)':
    * Filters peers by TCP/UDP support, minimum bandwidth, and maximum latency.
    * Sorts result by increasing latency for optimal transfer selection.
- **Choosing best peer for transfer:**
  - 'getBestPeerForTransfer(...)':
    * Calculates a composite score: 40
    * Selects the peer with the lowest score.
- **Protocol and congestion control selection:**
  - 'selectOptimalProtocol(peer)' chooses between TCP and Reliable UDP based on thresholds: latency ¡100 ms,packet loss ¡1
  - 'selectCongestionControl(source, target)':
    * If latency ¿200 ms → delay-based CC.
    * Else if packet loss ¿2
    * Else defaults to Reno for general conditions.
- **Transfer session lifecycle:**
  - 'createTransferSession(fileId, source, target, protocol)':
    * Generates 'sessionId', sets transfer status to INITIALIZING.
    * Applies congestion control algorithm and stores session in 'activeSessions'.
  - 'updateTransferSession(...)', 'removeTransferSession(...)', 'getActiveTransferSessions()' manage session status and cleanup.
- **Connectivity probing and capability estimation:**
  - 'testTcpConnectivity(...)' attempts a TCP handshake using a 'Socket' with timeout.
  - 'testUdpConnectivity(...)' sends a probe and awaits a dummy UDP response.
  - 'measureLatency(...)' uses ICMP-like reachability and timestamps to estimate RTT.
  - 'estimateBandwidth(...)' uses IP ranges to guess LAN (100 Mbps) vs WAN (10 Mbps).
  - 'estimatePacketLoss(...)' maps latency ranges to empirical loss estimates.
  - Combined results populate a 'PeerInfo.ConnectionQuality' object used in peer selection.

```
1
2
3 private void handlePeerConnection(Socket clientSocket) {
4         try (
5             DataInputStream inputStream = new
                  DataInputStream(clientSocket.getInputStream());
```

```
            DataOutputStream outputStream = new
                DataOutputStream(clientSocket.getOutputStream())
        ) {
            // Read handshake
            String handshake = inputStream.readUTF();
            System.out.println("Received handshake: " + handshake);

            if (handshake.startsWith("AFT-FTS-REQUEST:")) {
                String fileId = handshake.substring("AFT-FTS-REQUEST:".length());
                handleFileRequest(fileId, inputStream, outputStream,
                    clientSocket.getInetAddress().getHostAddress());
            } else {
                System.err.println("Invalid handshake: " + handshake);
                outputStream.writeUTF("ERROR:Invalid handshake");
            }

        } catch (IOException e) {
            System.err.println("Error handling peer connection: " + e.getMessage());
        } finally {
            try {
                clientSocket.close();
            } catch (IOException e) {
                System.err.println("Error closing client socket: " + e.getMessage());
            }
        }
    }
```

Listing 1: Peer Handling

9.4 Data Transfer Protocol Implementation

The data transfer protocol within the AFT-FTS (Adaptive File Transfer and Transport System) is implemented using a modular interface-based design that supports multiple transport mechanisms. Two key implementations are provided: TCP-based and Reliable UDP-based transport protocols, each encapsulated under the TransportProtocol interface. This abstraction enables easy extension and testing of different transport modes while preserving consistency across the transfer stack.

The TCP transport leverages Java's Socket API and is optimized for low-latency communication using features like disabled Nagle's algorithm (setTcpNoDelay(true)) and keep-alive packets. Data blocks are encapsulated with a custom 16-byte header before transmission. The sender awaits explicit acknowledgments (ACKs) for every block, ensuring in-order and reliable delivery.

File blocks are fragmented to respect maximum payload size (1400 bytes).

Each fragment is prepended with metadata such as sequence number, block index, fragment index, and total fragments.

A receiver confirms block reception using an acknowledgment packet with a specific block index and success code.

A connection handshake (AFT-FTS-CONNECT and ACK) and graceful disconnection are both handled over TCP.

This protocol is dynamically selected and instantiated based on session parameters, enabling adaptive file transmission strategies. Integration with session tracking (TransferSession) ensures that transport protocols remain stateless beyond the scope of an active session, promoting clean separation of concerns.

```
class TcpTransport implements TransportProtocol {

    private final TransferSession session;
    private Socket socket;

    public TcpTransport(TransferSession session) {
        this.session = session;
    }
```

```
9
10      @Override
11      public void connect(String address, int port) throws IOException {
12          socket = new Socket();
13          socket.connect(new InetSocketAddress(address, port), 30000); // 30 second timeout
14          socket.setSoTimeout(10000); // 10 second read timeout
15          socket.setTcpNoDelay(true); // Disable Nagle's algorithm for lower latency
16          socket.setKeepAlive(true);
17      }
18
19      @Override
20      public boolean sendBlock(FileService.FileBlock block) throws IOException {
21          if (socket == null || socket.isClosed()) {
22              return false;
23          }
24
25          try {
26              // Create block header
27              ByteBuffer header = ByteBuffer.allocate(16);
28              header.putInt(block.getIndex());
29              header.putInt(block.getSize());
30              header.putLong(block.getChecksum().hashCode()); // Simplified checksum
31
32              // Send header
33              socket.getOutputStream().write(header.array());
34
35              // Send block data
36              socket.getOutputStream().write(block.getData());
37              socket.getOutputStream().flush();
38
39              // Wait for acknowledgment
40              return receiveAck();
41
42          } catch (SocketTimeoutException e) {
43              return false; // Timeout treated as failure
44          }
45      }
46
47      @Override
48      public boolean receiveAck() throws IOException {
49          if (socket == null || socket.isClosed()) {
50              return false;
51          }
52
53          try {
54              byte[] ackBuffer = new byte[4];
55              int bytesRead = socket.getInputStream().read(ackBuffer);
56
57              if (bytesRead == 4) {
58                  ByteBuffer ackBuf = ByteBuffer.wrap(ackBuffer);
59                  int ackStatus = ackBuf.getInt();
60                  return ackStatus == 1; // 1 = ACK, 0 = NAK
61              }
62
63              return false;
64          } catch (SocketTimeoutException e) {
65              return false;
66          }
67      }
68
69      @Override
70      public void disconnect() throws IOException {
71          if (socket != null && !socket.isClosed()) {
72              socket.close();
73          }
74      }
75
76      @Override
```

```
77      public boolean isConnected() {
78          return socket != null && socket.isConnected() && !socket.isClosed();
79      }
80  }
```

Listing 2: TCP Protocol

9.5 Network Probe

The AFT-FTS framework includes an intelligent and reactive network probing subsystem to continuously monitor and adapt to changing network conditions. This system is encapsulated primarily within the PerformanceMetrics and PerformanceMonitor classes, which together form a runtime telemetry engine that guides adaptive transfer behavior.

The PerformanceMetrics class tracks real-time and historical performance of individual transfers, capturing:

Throughput (both overall and recent),

Average latency and recent latency trends,

Packet loss rate, computed from failed versus successful block transmissions,

Retransmission count, as an indicator of reliability challenges.

Metrics are computed both globally (since transfer start) and over a configurable sliding window of recent samples, supporting both long-term trends and short-term fluctuations. Each data point is encapsulated as a BlockTransfer record with metadata such as block size, latency, success status, and timestamp.

At a higher abstraction, PerformanceMonitor tracks aggregate metrics for each session using TransferHistory, which logs sequences of DataPoints. These are periodically updated via updateMetrics() and analyzed to detect performance degradation. For example, if the recent throughput drops below 80

This network probing and monitoring layer acts as the decision-making core for congestion control and protocol switching mechanisms. By isolating the probing logic from the transport logic, the design maintains modularity, enabling independent enhancements to the adaptation strategies.

```
1
2   public double getCurrentThroughput() {
3           synchronized (this) {
4               long elapsedTime = lastUpdateTime - transferStartTime;
5               if (elapsedTime > 0) {
6                   return (totalBytesTransferred * 1000.0) / elapsedTime;
7               }
8               return 0;
9           }
10      }
11
12
13      public double getRecentThroughput() {
14          synchronized (this) {
15              if (recentTransfers.isEmpty()) {
16                  return 0;
17              }
18
19              long recentBytes = 0;
20              long minTime = Long.MAX_VALUE;
21              long maxTime = Long.MIN_VALUE;
22
23              for (BlockTransfer transfer : recentTransfers) {
24                  if (transfer.successful) {
25                      recentBytes += transfer.blockSize;
26                      minTime = Math.min(minTime, transfer.timestamp);
27                      maxTime = Math.max(maxTime, transfer.timestamp);
28                  }
29              }
30
```

```
            long timeSpan = maxTime - minTime;
            if (timeSpan > 0) {
                return (recentBytes * 1000.0) / timeSpan;
            }
            return 0;
        }
    }


    public double getAverageThroughput() {
        return getCurrentThroughput();
    }


    public double getAverageLatency() {
        synchronized (this) {
            if (successfulBlocks > 0) {
                return totalLatency / successfulBlocks;
            }
            return 0;
        }
    }


    public double getRecentAverageLatency() {
        synchronized (this) {
            double totalRecentLatency = 0;
            int successfulRecentBlocks = 0;

            for (BlockTransfer transfer : recentTransfers) {
                if (transfer.successful) {
                    totalRecentLatency += transfer.latency;
                    successfulRecentBlocks++;
                }
            }

            if (successfulRecentBlocks > 0) {
                return totalRecentLatency / successfulRecentBlocks;
            }
            return 0;
        }
    }


    public double getPacketLossRate() {
        synchronized (this) {
            if (totalBlocks > 0) {
                return (double) failedBlocks / totalBlocks;
            }
            return 0;
        }
    }
```

Listing 3: Network Probe

9.6 Congestion Control Method Selection

Algorithm

1. Defined The methods for 3 congestion control mechanism - Reno,Tahoe and CUBIC,and created interface for each of them.
2. Fetched results from the network probe.(packet loss rate,latency,RTT)
3. Set a congestion control mechanism based on-

   1.TCP Reno: Used TCP Reno as default when:

- RTT is moderate, indicating a balanced network where latency isn't excessive.
- Throughput is moderate, meaning network resources are being utilized efficiently but congestion might occur.
- Packet loss is moderate, but we still need to handle retransmissions and recover quickly without too much data loss.

2. TCP Tahoe: Used TCP Tahoe when packet loss is greater than 2%.

- Used when RTT is high, indicating high latency.
- Throughput is low, or network conditions are heavily congested.
- Packet loss is high or frequent, especially in unreliable or fluctuating networks.
- Ideal for environments where quick recovery after packet loss is necessary but fast retransmit is not as critical.

3. TCP Cubic: Used TCP Cubic when latency is greater than 200ms and packet loss is low(less than 2%).

- Use when RTT is low, indicating a low-latency network.
- Throughput is high, indicating that the network has a lot of bandwidth available.
- Packet loss is low, ensuring that the congestion window can grow aggressively without packet loss due to buffer overflows or congestion.
- Most efficient in environments with stable, high-speed networks and low packet loss.

```

public class CongestionControlManager {

    /**
     * Initialize congestion control based on algorithm type
     */
    public CongestionControl initialize(TransferSession.CongestionControlAlgorithm
        algorithm,
                                        PeerInfo sourcePeer, PeerInfo targetPeer) {
        switch (algorithm) {
            case TAHOE:
                return new TahoeCongestionControl();
            case RENO:
                return new RenoCongestionControl();
            case DELAY_BASED:
                return new DelayBasedCongestionControl();
            default:
                return new RenoCongestionControl(); // Default
        }
    }

    /**
     * Select a better algorithm based on current performance
     */
    public TransferSession.CongestionControlAlgorithm selectBetterAlgorithm(
            TransferSession.CongestionControlAlgorithm current, PerformanceMetrics metrics)
                {

        double packetLoss = metrics.getPacketLossRate();
        double latency = metrics.getAverageLatency();
        double throughput = metrics.getCurrentThroughput();

        // High packet loss - use more conservative Tahoe
        if (packetLoss > 0.02) {
            return TransferSession.CongestionControlAlgorithm.TAHOE;
        }
```

```
36
37          // High latency - use delay-based control
38          if (latency > 200) {
39              return TransferSession.CongestionControlAlgorithm.DELAY_BASED;
40          }
41
42          // Good conditions - use Reno
43          return TransferSession.CongestionControlAlgorithm.RENO;
44      }
45  }
```

Listing 4: Congestion Control Selection

9.7 BlockManager: Managing File Blocks with Bitmap Tracking

The `BlockManager` class is designed to efficiently manage the transfer of large files by dividing them into fixed-size blocks and tracking their transfer status using a bitmap. This approach supports robust crash recovery and resumption of file transfers without redundant retransmission of already transferred data.

Key features of the `BlockManager` include:

- **Block Size Management:** The class operates on a configurable fixed block size, allowing flexible adaptation to network conditions and file sizes.
- **Bitmap Tracking:** Each bit in a `BitSet` corresponds to a block, where a set bit indicates a successfully transferred block. This compact representation is efficient in memory usage and allows quick querying and updates.
- **Transfer State Operations:**
  - Mark individual blocks as transferred.
  - Check the transfer status of specific blocks.
  - Retrieve the count of transferred blocks and overall transfer progress as a percentage.
  - Determine the next block that needs to be transferred, enabling sequential or prioritized block transfer strategies.
- **Persistence Support:** The class provides serialization and deserialization methods to convert the bitmap to and from a string representation. This feature enables the saving of transfer progress to persistent storage, facilitating recovery after application or system crashes.
- **Crash Recovery and Efficiency:** By using bitmap tracking, the system avoids retransmitting already completed blocks, saving bandwidth and time during interruptions.

Overall, `BlockManager` serves as a fundamental utility in file transfer protocols requiring reliable, resumable, and efficient block-level data management.

9.8 Encryption

Each data block is encrypted using AES-GCM (Advanced Encryption Standard in Galois/Counter Mode) with a 256-bit key generated from a cryptographically secure random number generator. For every block, a unique 96-bit IV (Initialization Vector) is also generated.

AES-GCM provides both confidentiality and integrity: encryption transforms the plaintext block into ciphertext, while a 128-bit authentication tag ensures the data has not been tampered with. This tag is transmitted alongside the encrypted block.

The encrypted output is serialized in the format: `[algorithm][IV][tag][ciphertext]` for compactness and compatibility. On the receiving end, the block is deserialized and decrypted using the same key and IV. If the tag verification fails, the block is discarded.

This block-level encryption ensures:

- Independent encryption of each block
- Resistance to tampering and replay attacks
- Seamless crash recovery and resumption of partial transfers

```
 public EncryptedData encryptAESGCM(byte[] plaintext, SecretKey key) throws Exception {
        Cipher cipher = Cipher.getInstance(AES_GCM_ALGORITHM);

        // Generate random IV
        byte[] iv = new byte[GCM_IV_LENGTH];
        secureRandom.nextBytes(iv);

        GCMParameterSpec gcmSpec = new GCMParameterSpec(GCM_TAG_LENGTH * 8, iv);
        cipher.init(Cipher.ENCRYPT_MODE, key, gcmSpec);

        byte[] encryptedWithTag = cipher.doFinal(plaintext);

        // Extract encrypted data and authentication tag
        int encryptedLength = encryptedWithTag.length - GCM_TAG_LENGTH;
        byte[] encrypted = new byte[encryptedLength];
        byte[] tag = new byte[GCM_TAG_LENGTH];

        System.arraycopy(encryptedWithTag, 0, encrypted, 0, encryptedLength);
        System.arraycopy(encryptedWithTag, encryptedLength, tag, 0, GCM_TAG_LENGTH);

        return new EncryptedData(encrypted, iv, tag, Algorithm.AES_GCM);
    }

public byte[] decryptAESGCM(EncryptedData encryptedData, SecretKey key) throws Exception {
        if (encryptedData.getAlgorithm() != Algorithm.AES_GCM) {
            throw new IllegalArgumentException("Expected AES-GCM encrypted data");
        }

        Cipher cipher = Cipher.getInstance(AES_GCM_ALGORITHM);
        GCMParameterSpec gcmSpec = new GCMParameterSpec(GCM_TAG_LENGTH * 8,
            encryptedData.getIv());
        cipher.init(Cipher.DECRYPT_MODE, key, gcmSpec);

        // Combine encrypted data and tag for decryption
        byte[] encryptedWithTag = new byte[encryptedData.getData().length +
            encryptedData.getTag().length];
        System.arraycopy(encryptedData.getData(), 0, encryptedWithTag, 0,
            encryptedData.getData().length);
        System.arraycopy(encryptedData.getTag(), 0, encryptedWithTag,
            encryptedData.getData().length, encryptedData.getTag().length);

        return cipher.doFinal(encryptedWithTag);
    }
```

Listing 5: Encryption and Decryption

9.9 Crash Recovery

Crash recovery is achieved by tracking the transfer status of each file block using a bitmap, where each bit represents whether a specific block has been successfully transferred. During the transfer process, this bitmap is continuously updated and can be serialized to persistent storage (e.g., disk or database).

In the event of an application or system crash, the saved bitmap can be deserialized to restore the exact state of the transfer. This allows the transfer to resume from the last successfully transferred block instead of restarting from the beginning. By querying the bitmap for the next unset bit (i.e., a block not yet transferred), the system efficiently identifies the next block to send, minimizing redundant data transfer and reducing overall recovery time.

This method ensures that partial transfers are preserved, improving reliability and user experience in unstable network environments.

```
1
2
3  public boolean resumeTransfer(String sessionId) {
4          ActiveTransfer transfer = activeTransfers.get(sessionId);
5          if (transfer != null && transfer.getSession().getStatus() ==
               TransferSession.TransferStatus.PAUSED) {
6              transfer.getSession().setStatus(TransferSession.TransferStatus.RESUMING);
7              return true;
8          }
9          return false;
10     }
```

Listing 6: Crash Recovery

9.10 File Handling

The AFT-FTS system implements a robust and extensible file handling mechanism that supports secure storage, efficient retrieval, file metadata management, and adaptive transfer preparation. The FileService class in the backend acts as the central service responsible for orchestrating all file-related operations, such as uploading, indexing, searching, and block-level segmentation.

Key functionalities include:

- **Storage Initialization:** At application startup, the system creates and prepares dedicated directories for file storage, temporary data, and metadata (repositoryPath, tempPath, and metadataPath) using Java NIO. This ensures clean separation of concerns and promotes scalability.
- **Upload and Metadata Generation:** During upload via HTTP multipart form, each file is stored on disk with dynamic conflict resolution for filenames. Metadata including size, checksum (SHA-256), MIME type, description, and category is automatically extracted and stored. A unique file ID is generated using a UUID derived from the filename and checksum to avoid duplication.
- **Checksum Calculation:** To ensure data integrity, the system computes cryptographic checksums for each uploaded file and its block fragments. This is implemented using Java's MessageDigest API, allowing dynamic configuration of the hashing algorithm (default: SHA-256).
- **Block-Based Transfer Preparation:** For network transmission, each file is segmented into fixed-size blocks (default 1 MB). The system reads the file in chunks, computes a checksum for each block, and wraps the content in a FileBlock object containing index, data, and hash. This design supports resumable transfers and integrity validation on the receiver side.
- **File Search and Categorization:** Users can search the repository by query string, category, or tags. The system categorizes files automatically based on common extensions (e.g., .jpg as "image", .mp4 as "video") and stores them with semantic metadata, enhancing user experience and enabling peer discovery.
- **Peer Tracking:** The service maintains a dynamic registry of peers hosting a particular file. Peers can be registered or unregistered at runtime, allowing decentralized peer-to-peer file sharing and distributed load handling.
- **Resilience and Extensibility:** The file registry is maintained in-memory for demo purposes but is designed to support future integration with persistent databases. The modular architecture ensures future scalability, including distributed file systems or cloud backends.

This subsystem forms the backbone of the P2P file exchange process, tightly integrated with the adaptive transport and probing layers to ensure consistency, resilience, and security in every file transfer.

9.11 GUI Implementation

The Graphical User Interface (GUI) of AFT-FTS is implemented using React.js, a popular JavaScript library for building interactive and component-based user interfaces. The interface is lightweight, responsive, and user-friendly, providing real-time access to file transfer functionality and monitoring.

9.11.1 Architecture Overview

The GUI is built with a modular component-based architecture. The central ‘App.tsx‘ file orchestrates the rendering of components based on the currently active tab using React’s ‘useState‘ hook. Each feature—uploading files, viewing network-wide files, monitoring performance, etc.—is encapsulated in its own component for clarity and maintainability.

9.11.2 Navigation and State Management

Navigation is handled through a visually distinct tabbed interface rendered in the ‘¡nav¿‘ element. Each tab is styled dynamically with conditionals using the ‘tabStyle‘ function, which updates the visual feedback depending on whether it is active. React’s ‘useState‘ manages the current tab (‘activeTab‘), allowing seamless conditional rendering of content.

- Tabs include: **Network Files**, **File Requests**, **Upload Files**, **File Repository**, **Active Transfers**, and **Monitor**.
- The currently active tab’s component is shown in the ‘¡main¿‘ area using conditional rendering.

9.11.3 Component Functionality

Each tab is mapped to a dedicated component with clearly defined roles:

- **NetworkFiles:** Displays a list of files shared across the P2P network.
- **FileRequests:** Allows the user to initiate a file request from another peer.
- **SimpleUpload:** Provides an interface to upload files to the local peer and make them available on the network.
- **FilesView (Inline Component):** Acts as a placeholder UI to display already-uploaded files in a user-friendly panel.
- **ActiveTransfers:** Monitors ongoing file transfers, showing live progress or retry states.
- **MonitorView:** A real-time dashboard visualizing throughput, round-trip time (RTT), and congestion algorithm data.

9.11.4 Styling and Layout

The GUI leverages inline React styling for fine-grained layout control:

- Tabs use soft colors and transitions for better UX, with rounded corners and hover highlights.
- Main content is centrally aligned, with consistent padding and responsive behavior.
- UI cues such as emojis (, ,) enhance clarity and user engagement.
- A global light background (‘f5f5f5‘) is used to differentiate between content and control areas.

9.11.5 Integration with Backend

The frontend communicates with the backend REST API hosted at `http://localhost:8080/api`. Each component is responsible for triggering its corresponding endpoint:

- Uploads are POSTed using ‘SimpleUpload‘.

- Network files and metadata are fetched via ‘GET‘ requests.
- File transfers trigger backend coordination before peer-to-peer transmission.
- ‘MonitorView‘ periodically polls statistics to reflect dynamic changes in network performance.

Result Analysis

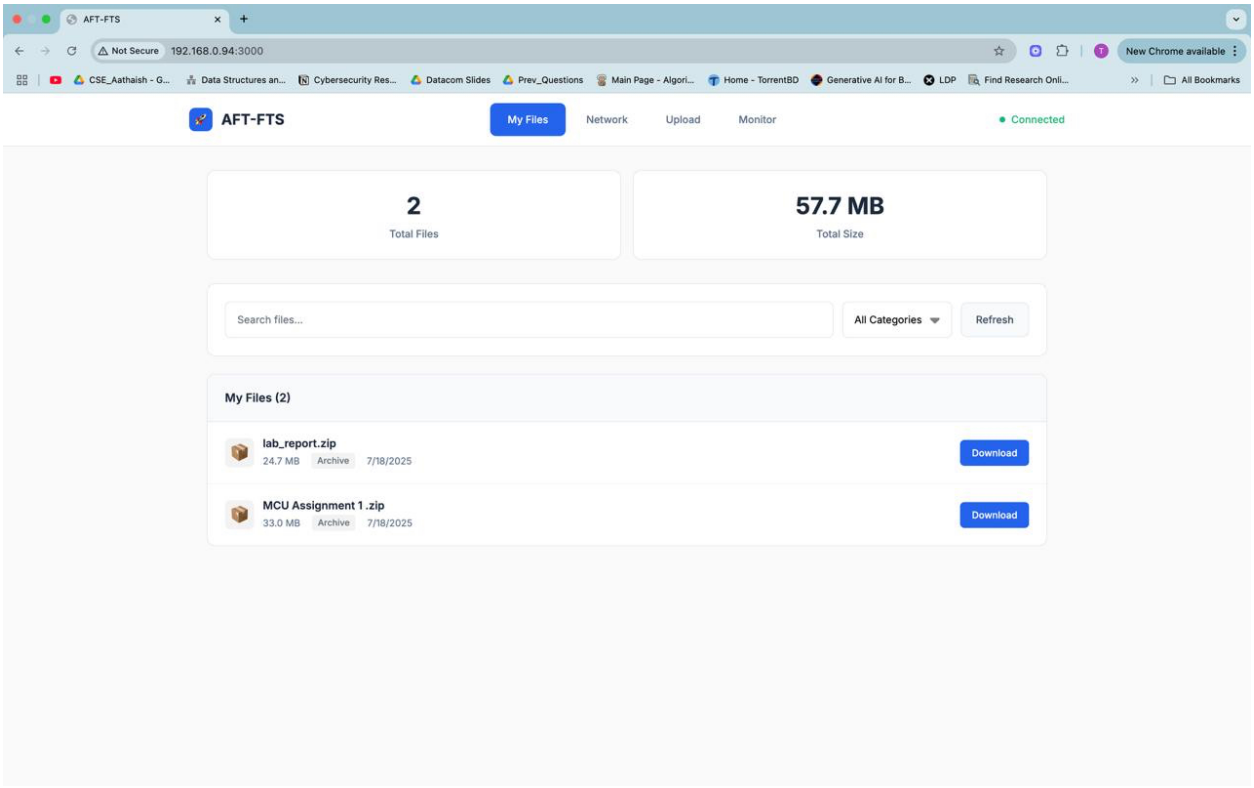


Figure 1: Shared files over the network

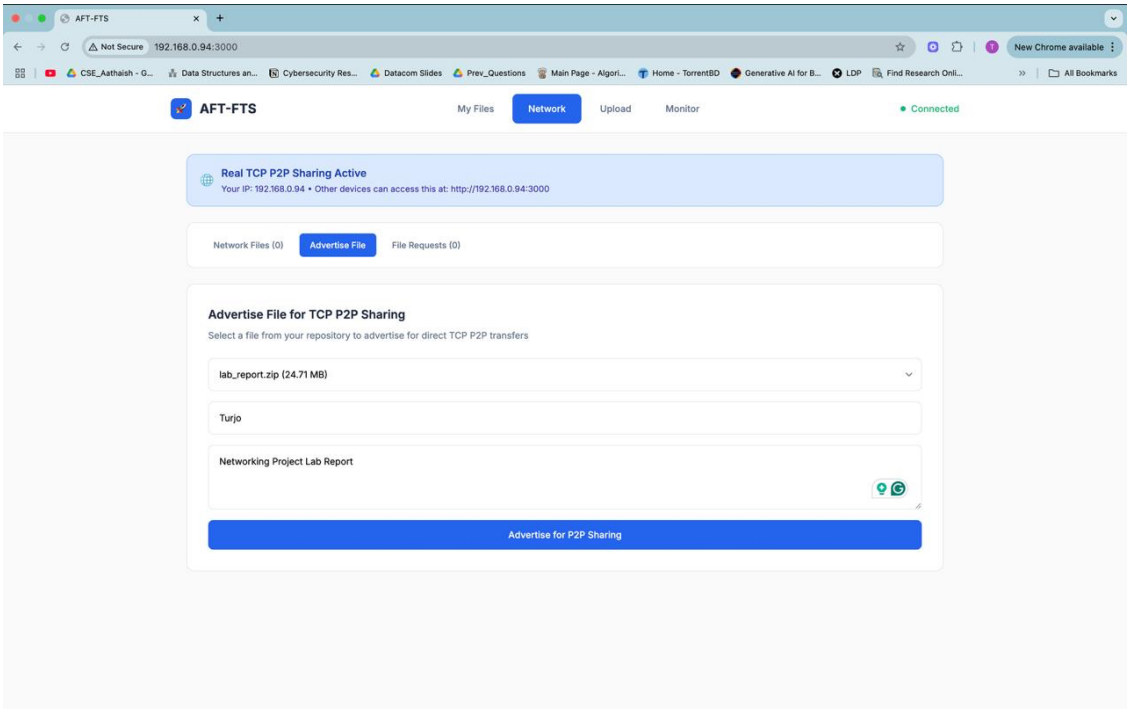


Figure 2: File Advertising Over the Network

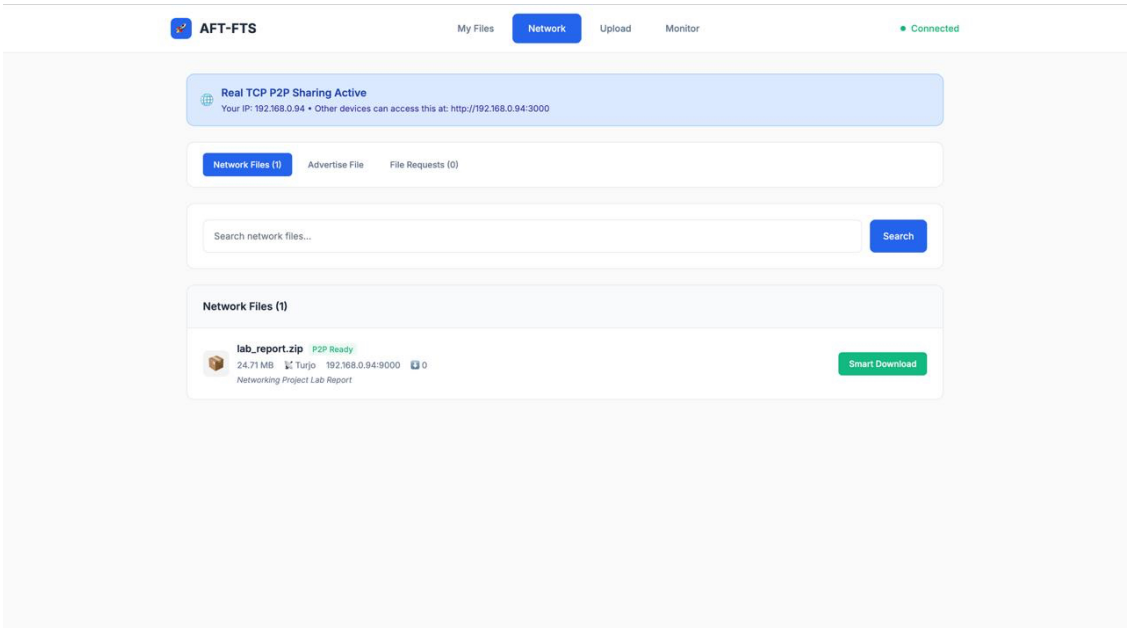


Figure 3: File Advertising Over the Network

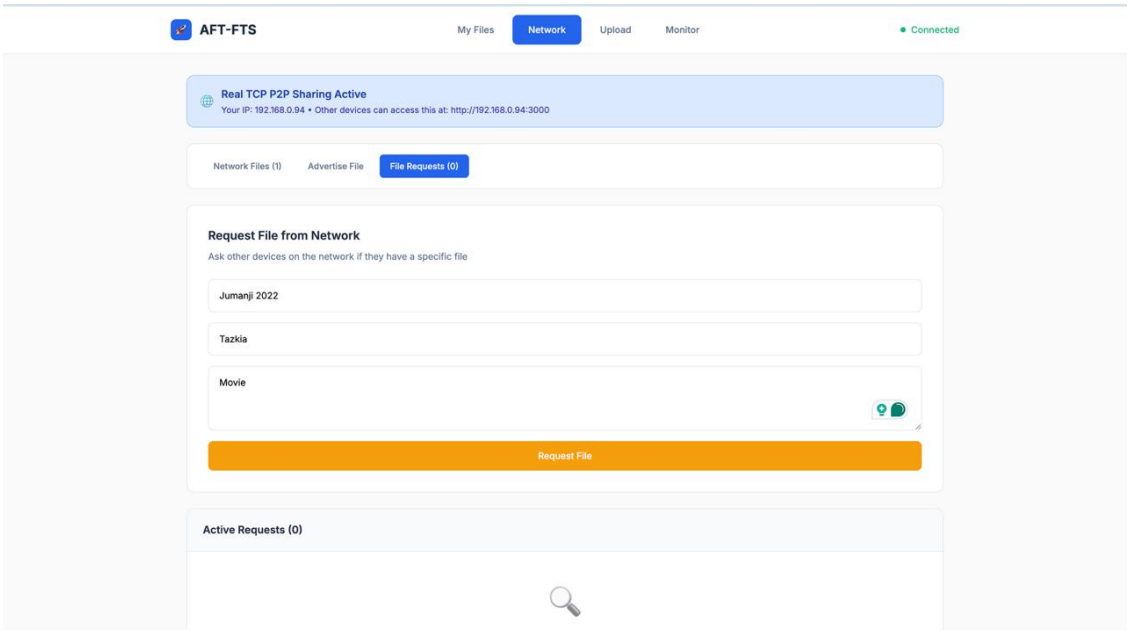


Figure 4: Requesting a File

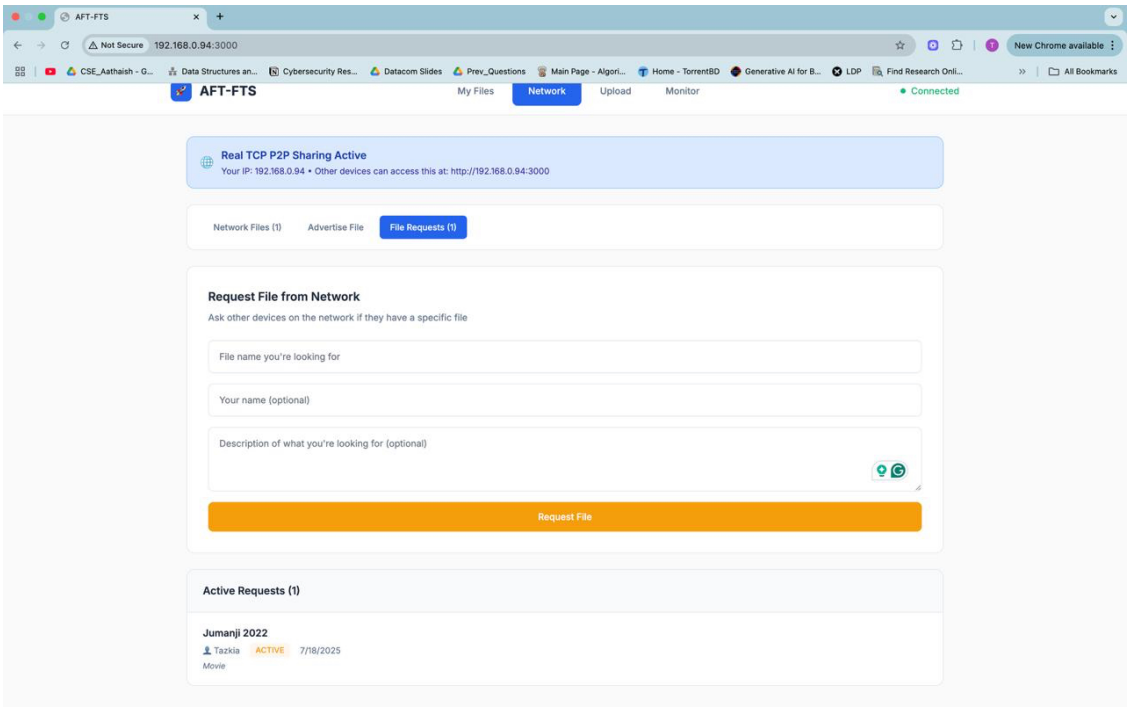


Figure 5: List of the Requested Files by the Peers

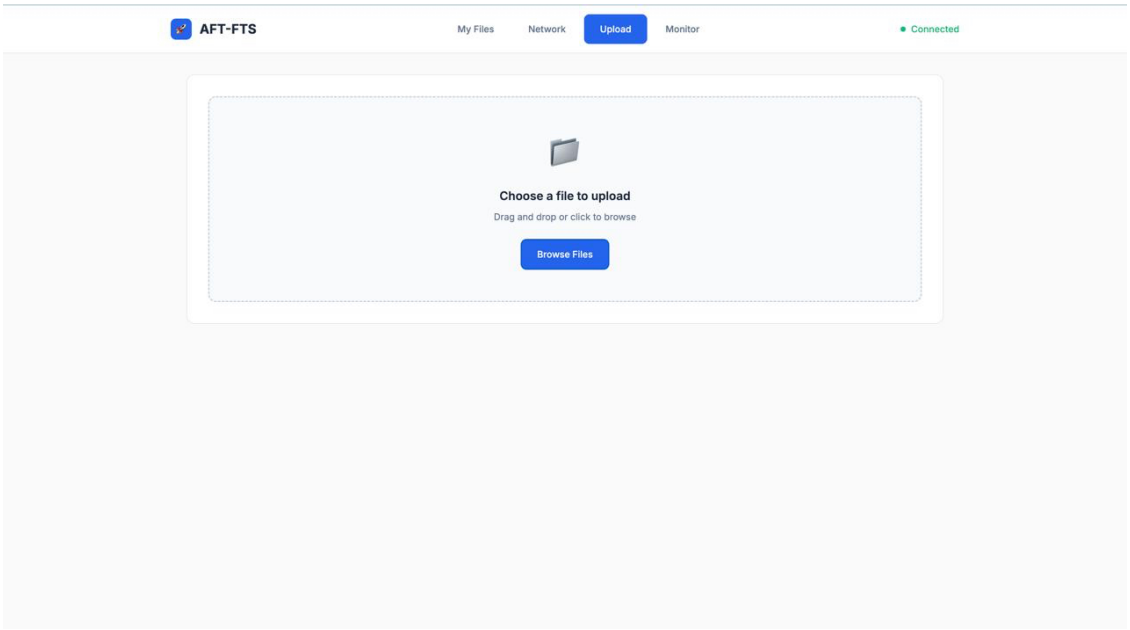


Figure 6: Uploading File to Share with the Network

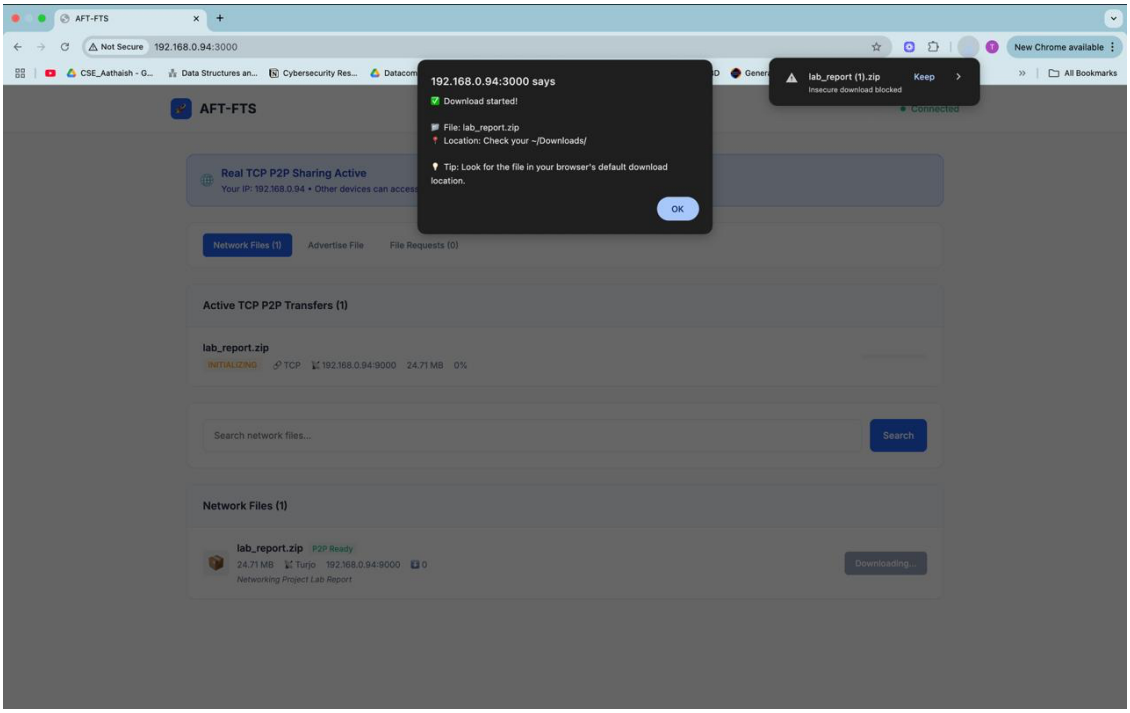


Figure 7: Downloading a File using Peer to Peer transfer protocol

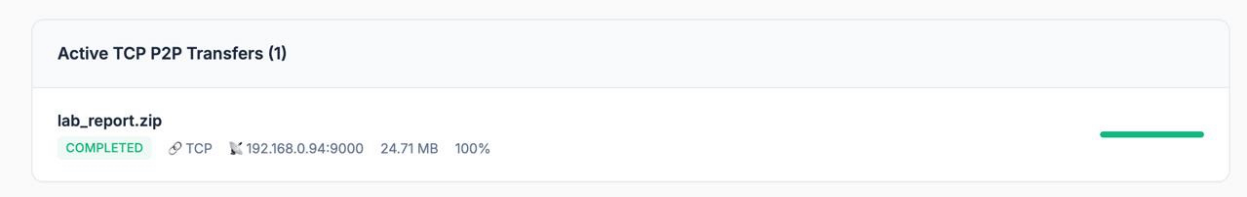


Figure 8: Progress Bar Showing Real-Time Transfer Process

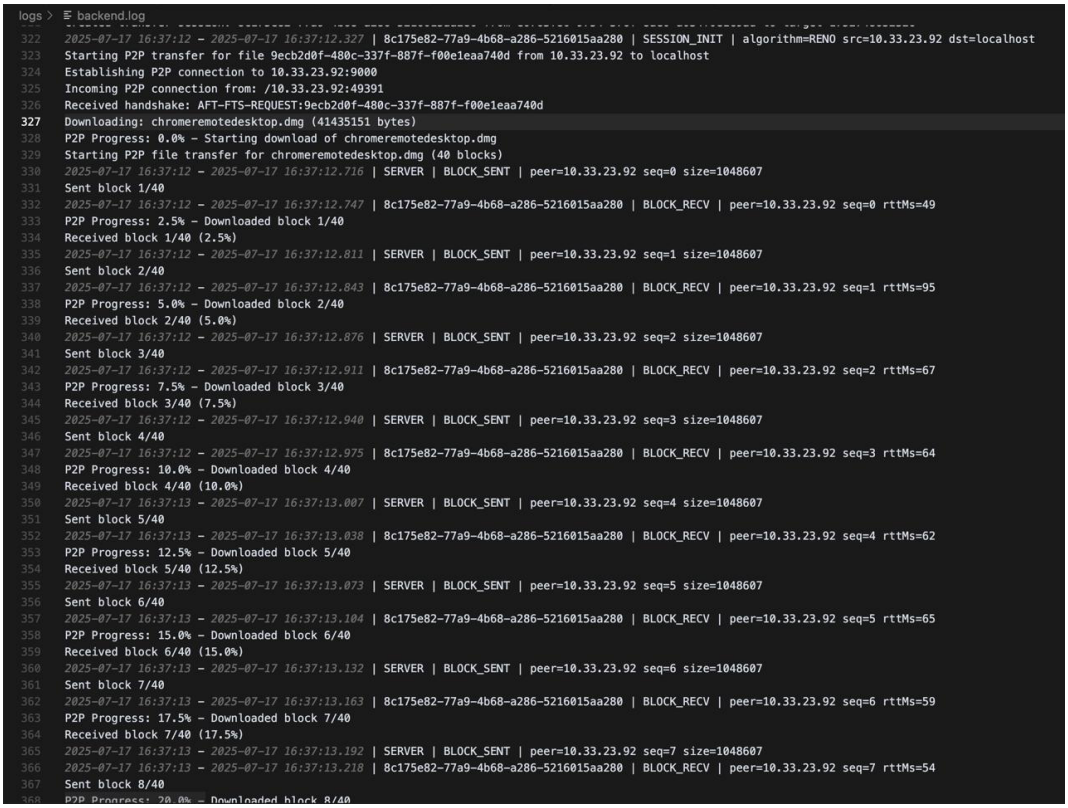


Figure 9: Network Conditions: Congestion selected-RENO

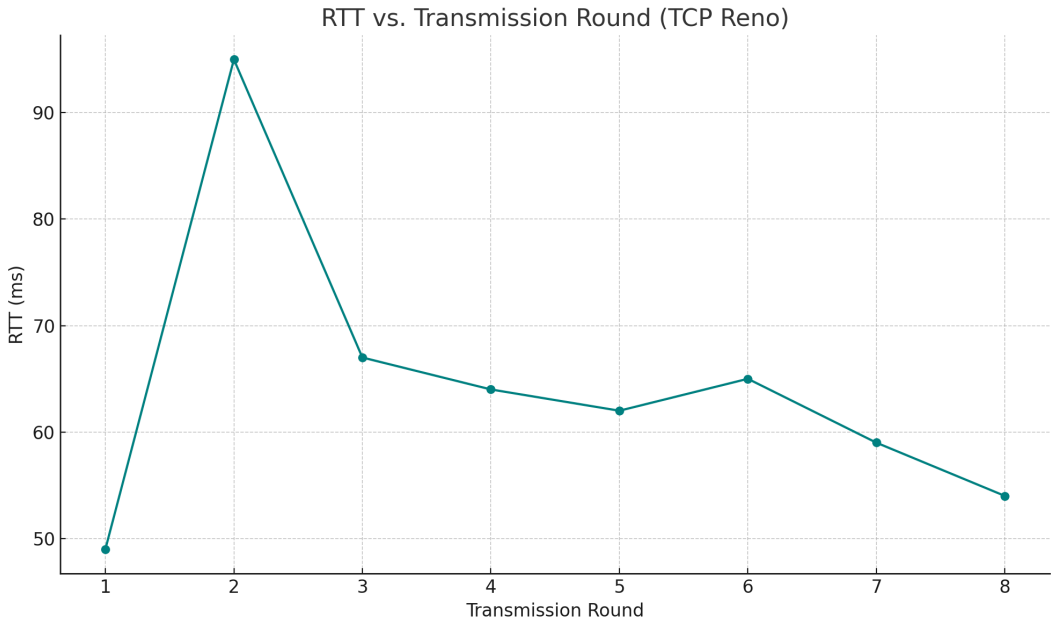


Figure 10: RTT vs Transmission Round

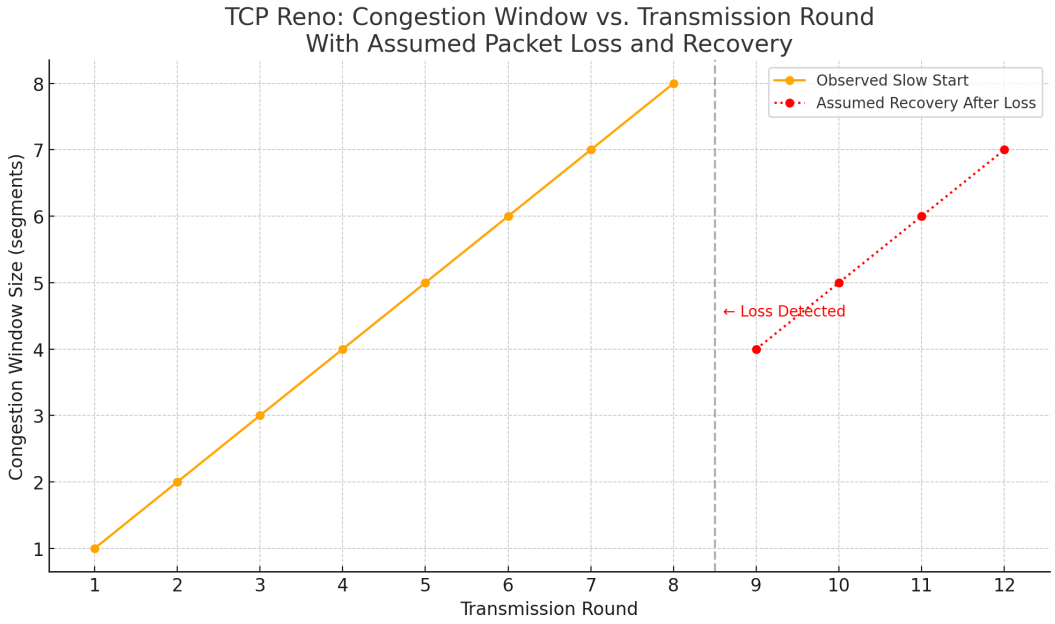


Figure 11: Reno Behaviour in the project

Limitations

- **Limited Peer Scalability:** The current design supports only one-to-one transfers, not multi-peer or swarm-based sharing.
- **Basic UI/UX:** The system currently lacks a user-friendly graphical interface.
- **No NAT Traversal:** Peer discovery and connection may fail behind strict NATs or firewalls without manual configuration.
- **Partial Resume Support:** Resume logic may not cover all edge cases, especially during mid-transfer network changes.

Future Plan

- **Add NAT Traversal Techniques:** Integrate STUN/TURN or similar methods to improve connectivity in restricted networks.
- **Multi-peer Support:** Expand the protocol to allow parallel downloads from multiple peers (BitTorrent-style hybrid).
- **Graphical Interface:** Develop a cross-platform GUI for easier file selection, progress tracking, and error recovery.
- **Mobile and Cross-Platform Support:** Extend compatibility to Android/iOS and low-resource devices.
- **Enhanced Resume Logic:** Implement more robust, checksum-verified chunk tracking and mid-transfer recovery.
- **Advanced Security Features:** Add support for end-to-end encryption key negotiation and forward secrecy.

Conclusion

This project presents a robust peer-to-peer file transfer system featuring an interactive repository with advanced search capabilities and real-time visualization of file transfers. Leveraging technologies such as Spring Boot, Java, Firebase, and Next.js, the system ensures efficient, secure, and adaptive data exchange tailored to varying network conditions. The implementation of pluggable congestion control algorithms, dynamic protocol switching, and end-to-end security mechanisms guarantees reliable and resilient transfers even in challenging environments. Overall, this project lays a strong foundation for scalable and user-friendly distributed file sharing with potential for future enhancements including richer user interfaces and expanded protocol support.