Program Listing Source Code

```
P2P-FILE-SHARING/
├── shared_files/
|   └── [contains shared files for the peers to discover and share]
├── aggregate_data.py
├── automatedclient.py
├── automatedclient_linear.py
├── average_response_time_plot.png
├── boxplot_response_time.png
├── compute_Average.py
├── compute_avg_linear.py
├── compute_category_averages.py
├── config.json
├── config_linear.json
├── consolidated_results.csv
├── Design Document.pdf
├── go.mod
├── go.sum
├── heatmap_response_time.png
├── leafnode.py
├── linear_topology_results.csv
├── main.go
├── Output_LN1.txt
├── Output_SN1.txt
├── p2p_app/
|   ├── plot_average.py
```

```
│   ├── plot_box_response_time.py
│   ├── plot_comparison.py
│   ├── plot_heatmap_response_time.py
│   ├── plot_response_time_by_category.py
│   ├── plot_scatter_response_time.py
│   └── plot_violin_response_time.py
├── response_time_by_category_plot.png
├── results_1_clients.csv
├── results_2_clients.csv
├── results_3_clients.csv
├── results_4_clients.csv
├── results_5_clients.csv
├── results_6_clients.csv
├── results_7_clients.csv
├── results_8_clients.csv
├── results_9_clients.csv
├── results_10_clients.csv
├── scatterplot_response_time.png
├── summary_diff_sp.csv
├── summary_response_times.csv
├── summary_same_sp.csv
├── super_peer.py
├── updated_results_1_clients.csv
├── updated_results_2_clients.csv
├── updated_results_3_clients.csv
├── updated_results_4_clients.csv
├── updated_results_5_clients.csv
├── updated_results_6_clients.csv
├── updated_results_7_clients.csv
├── updated_results_8_clients.csv
├── updated_results_9_clients.csv
├── updated_results_10_clients.csv
└── violinplot_response_time.png
```

Configuration File (config.json)

Before diving into the source code, ensure you have a config.json file that defines the network topology, including Super-Peers and Leaf-Nodes configurations.

Explanation:

- **super_peers: An array of Super-Peer configurations. Each Super-Peer has:**
  - **id: Unique identifier.**
  - **address: IP address.**
  - **port: Port number to listen on.**
  - **neighbors: Array of neighboring Super-Peer IDs for establishing connections.**
  - **leaf_nodes: Array of associated Leaf-Node IDs.**
- **leaf_nodes: An array of Leaf-Node configurations. Each Leaf-Node has:**
  - **id: Unique identifier.**
  - **address: IP address.**
  - **port: Port number to host the file server.**
  - **super_peer: The Super-Peer ID it connects to.**

```
{
  "super_peers": [
    {
```

```
  "id": "SP1",
  "address": "127.0.0.1",
  "port": 8000,
  "neighbors": [
    "SP2",
    "SP3",
    "SP4",
    "SP5",
    "SP6",
    "SP7",
    "SP8",
    "SP9",
    "SP10"
  ],
  "leaf_nodes": ["LN1", "LN2"]
},
{
  "id": "SP2",
  "address": "127.0.0.1",
  "port": 8001,
  "neighbors": [
    "SP1",
    "SP3",
    "SP4",
    "SP5",
    "SP6",
        "SP7",
        "SP8",
        "SP9",
        "SP10"
      ],
      "leaf_nodes": ["LN3", "LN4"]
    },
    {
      "id": "SP3",
      "address": "127.0.0.1",
      "port": 8002,
      "neighbors": [
        "SP1",
        "SP2",
        "SP4",
        "SP5",
        "SP6",
        "SP7",
        "SP8",
        "SP9",
        "SP10"
      ],
      "leaf_nodes": ["LN5", "LN6"]
    },
    {
      "id": "SP4",
      "address": "127.0.0.1",
      "port": 8003,
      "neighbors": [
        "SP1",
        "SP2",
        "SP3",
        "SP5",
        "SP6",
        "SP7",
        "SP8",
        "SP9",
        "SP10"
      ],
      "leaf_nodes": ["LN7"]
    },
    {
      "id": "SP5",
      "address": "127.0.0.1",
      "port": 8004,
      "neighbors": [
        "SP1",
        "SP2",
        "SP3",
        "SP4",
        "SP6",
        "SP7",
        "SP8",
        "SP9",
        "SP10"
      ],
      "leaf_nodes": ["LN8"]
    },
    {
      "id": "SP6",
      "address": "127.0.0.1",
      "port": 8005,
      "neighbors": [
        "SP1",
        "SP2",
        "SP3",
        "SP4",
        "SP5",
        "SP7",
        "SP8",
        "SP9",
        "SP10"
      ],
      "leaf_nodes": ["LN9"]
    },
    {
      "id": "SP7",
      "address": "127.0.0.1",
      "port": 8006,
      "neighbors": [
        "SP1",
        "SP2",
        "SP3",
        "SP4",
        "SP5",
        "SP6",
        "SP8",
        "SP9",
        "SP10"
      ],
      "leaf_nodes": ["LN10"]
    },
    {
      "id": "SP8",
      "address": "127.0.0.1",
      "port": 8007,
      "neighbors": [
        "SP1",
        "SP2",
        "SP3",
        "SP4",
        "SP5",
        "SP6",
        "SP7",
        "SP9",
        "SP10"
      ],
      "leaf_nodes": []
    },
    {
      "id": "SP9",
      "address": "127.0.0.1",
      "port": 8008,
      "neighbors": [
        "SP1",
        "SP2",
        "SP3",
        "SP4",
        "SP5",
        "SP6",
        "SP7",
        "SP8",
        "SP10"
      ],
      "leaf_nodes": []
    },
    {
      "id": "SP10",
      "address": "127.0.0.1",
      "port": 8009,
    "neighbors": [
      "SP1",
      "SP2",
      "SP3",
      "SP4",
      "SP5",
      "SP6",
      "SP7",
      "SP8",
      "SP9"
    ],
    "leaf_nodes": []
  }
],
"leaf_nodes": [
  { "id": "LN1", "address": "127.0.0.1", "port": 9000, "super_peer": "SP1" },
  { "id": "LN2", "address": "127.0.0.1", "port": 9001, "super_peer": "SP1" },
  { "id": "LN3", "address": "127.0.0.1", "port": 9002, "super_peer": "SP2" },
  { "id": "LN4", "address": "127.0.0.1", "port": 9003, "super_peer": "SP2" },
  { "id": "LN5", "address": "127.0.0.1", "port": 9004, "super_peer": "SP3" },
  { "id": "LN6", "address": "127.0.0.1", "port": 9005, "super_peer": "SP3" },
  { "id": "LN7", "address": "127.0.0.1", "port": 9006, "super_peer": "SP4" },
  { "id": "LN8", "address": "127.0.0.1", "port": 9007, "super_peer": "SP5" },
  { "id": "LN9", "address": "127.0.0.1", "port": 9008, "super_peer": "SP6" },
  { "id": "LN10", "address": "127.0.0.1", "port": 9009, "super_peer": "SP7" }
]
```

```
}
```

Main.go

```
// main.go - Peer-to-Peer File-Sharing System Implementation

package main

import (
    "bufio"
    "encoding/json"
    "fmt"
    "github.com/google/uuid" // External package for generating unique IDs. Install using: go get github.com/google/uuid
    "io"
    "io/ioutil"
    "log"
    "net"
    "net/http"
    "os"
    "path/filepath"
    "strings"
    "sync"
    "time"
)
```

```
// -----------------------------
// Constants and Types
// -----------------------------

// Constants for Message Types
const (
    MsgTypePeerID           = "peer_id"           // Identifies the type of peer (Super-Peer or Leaf-Node)
    MsgTypeFileRegistration = "file_registration" // Message type for file registration
    MsgTypeFileQuery        = "file_query"        // Message type for file querying
    MsgTypeQueryHit         = "query_hit"         // Message type for query responses
)

// Peer Types
const (
    SuperPeerType = "super_peer" // Type identifier for Super-Peers
    LeafNodeType  = "leaf_node"  // Type identifier for Leaf-Nodes
)

// PeerIDMessage is used by peers to identify themselves upon connection
type PeerIDMessage struct {
    MessageType string `json:"message_type"` // Type of the message
    PeerType    string `json:"peer_type"`    // Type of the peer (Super-Peer or Leaf-Node)
    PeerID      string `json:"peer_id"`      // Unique identifier of the peer
}
```

```
// FileMetadata contains information about a shared file
type FileMetadata struct {
    FileName string `json:"file_name"` // Name of the file
    FileSize int64  `json:"file_size"` // Size of the file in bytes
}

// FileRegistrationMessage is sent by Leaf-Nodes to register their files with a Super-Peer
type FileRegistrationMessage struct {
    MessageType string         `json:"message_type"` // Type of the message
    LeafNodeID  string         `json:"leaf_node_id"` // ID of the Leaf-Node registering the files
    Files       []FileMetadata `json:"files"`        // List of files being registered
}

// FileQueryMessage is used by Leaf-Nodes to query for a specific file
type FileQueryMessage struct {
    MessageType string `json:"message_type"` // Type of the message
    MessageID   string `json:"message_id"`   // Unique identifier for the query
    OriginID    string `json:"origin_id"`    // ID of the Leaf-Node originating the query
    FileName    string `json:"file_name"`    // Name of the file being queried
    TTL         int    `json:"ttl"`          // Time-To-Live for the query (prevents infinite propagation)
}

// QueryHitMessage is sent in response to a FileQueryMessage, indicating the availability of the file
type QueryHitMessage struct {
    MessageType  string `json:"message_type"`  // Type of the message
    MessageID    string `json:"message_id"`    // ID of the original query
    TTL          int    `json:"ttl"`           // Time-To-Live (decremented)
    RespondingID string `json:"responding_id"` // ID of the Leaf-Node responding with the file
    FileName     string `json:"file_name"`     // Name of the file available
    Address      string `json:"address"`       // IP address of the responding Leaf-Node
    Port         int    `json:"port"`          // Port number of the responding Leaf-Node
}

// SuperPeerConfig contains configuration details for a Super-Peer
type SuperPeerConfig struct {
    ID        string   `json:"id"`          // Unique identifier for the Super-Peer
    Address   string   `json:"address"`     // IP address of the Super-Peer
    Port      int      `json:"port"`        // Port number the Super-Peer listens on
    Neighbors []string `json:"neighbors"`   // List of neighboring Super-Peer IDs
    LeafNodes []string `json:"leaf_nodes"`  // List of associated Leaf-Node IDs
}

// LeafNodeConfig contains configuration details for a Leaf-Node
type LeafNodeConfig struct {
    ID        string `json:"id"`          // Unique identifier for the Leaf-Node
    Address   string `json:"address"`     // IP address of the Leaf-Node
    Port      int    `json:"port"`        // Port number the Leaf-Node hosts its file server on
    SuperPeer string `json:"super_peer"`  // ID of the Super-Peer the Leaf-Node connects to
}

// Config represents the overall network configuration, including all Super-Peers and Leaf-Nodes
```

```go
type Config struct {
    SuperPeers []SuperPeerConfig `json:"super_peers"` // Array of Super-Peer configurations
    LeafNodes  []LeafNodeConfig  `json:"leaf_nodes"`  // Array of Leaf-Node configurations
}

// DownloadPrompt is used to prompt the user for downloading a file upon receiving a QueryHitMessage
type DownloadPrompt struct {
    QueryHitMessage QueryHitMessage // The QueryHitMessage received
    ResponseChan    chan bool       // Channel to receive the user's response (yes/no)
}

// CacheEntry stores information about forwarded messages to prevent duplicate processing
type CacheEntry struct {
    OriginID     string    // ID of the originating Leaf-Node
    UpstreamConn net.Conn  // Connection to the upstream peer
    Timestamp    time.Time // Time when the message was received
}

// -----------------------------
// Configuration Loading
// -----------------------------

// LoadConfig reads and parses the configuration from a JSON file
func LoadConfig(filename string) (*Config, error) {
    data, err := ioutil.ReadFile(filename)
    if err != nil {
        return nil, err
    }
    var config Config
    err = json.Unmarshal(data, &config)
    if err != nil {
        return nil, err
    }
    return &config, nil
}

// ----------------------------
// SuperPeer Implementation
// ----------------------------

// SuperPeer represents a Super-Peer in the network
type SuperPeer struct {
    Config          SuperPeerConfig             // Configuration details of the Super-Peer
    GlobalConfig    *Config                     // Reference to the global network configuration
    NeighborConns   map[string]net.Conn         // Active connections to neighboring Super-Peers
    NeighborConfigs map[string]SuperPeerConfig  // Configurations of neighboring Super-Peers
    LeafNodeConns   map[string]net.Conn         // Active connections to Leaf-Nodes
    LeafNodeConfigs map[string]LeafNodeConfig   // Configurations of associated Leaf-Nodes
    FileIndex       map[string]map[string]struct{} // FileIndex maps file names to a set of Leaf-Node IDs that have the file
    MessageCache    map[string]CacheEntry       // Cache to store processed MessageIDs to prevent duplication
    mu          sync.Mutex                  // Mutex to protect shared resources
    connMu        sync.Mutex                  // Mutex to protect connection writes
}

// NewSuperPeer initializes a new Super-Peer with the given configuration
func NewSuperPeer(config SuperPeerConfig, globalConfig *Config) *SuperPeer {
    neighborConfigs := make(map[string]SuperPeerConfig)
    // Populate NeighborConfigs with configurations of neighbors
    for _, neighborID := range config.Neighbors {
        for _, spConfig := range globalConfig.SuperPeers {
            if spConfig.ID == neighborID {
                neighborConfigs[neighborID] = spConfig
                break
            }
        }
    }

    leafNodeConfigs := make(map[string]LeafNodeConfig)
    // Populate LeafNodeConfigs with configurations of associated Leaf-Nodes
    for _, lnID := range config.LeafNodes {
        for _, lnConfig := range globalConfig.LeafNodes {
            if lnConfig.ID == lnID {
                leafNodeConfigs[lnID] = lnConfig
                break
            }
        }
```

```go
    }

    // Initialize FileIndex with empty maps for each file
    fileIndex := make(map[string]map[string]struct{})

    return &SuperPeer{
        Config:          config,
        GlobalConfig:    globalConfig,
        NeighborConns:   make(map[string]net.Conn),
        NeighborConfigs: neighborConfigs,
        LeafNodeConns:   make(map[string]net.Conn),
        LeafNodeConfigs: leafNodeConfigs,
        FileIndex:       fileIndex,
        MessageCache:    make(map[string]CacheEntry),
    }
}

// Start initializes the Super-Peer, sets up connections, and starts listening for incoming connections
func (sp *SuperPeer) Start() {
    // Start listening for incoming connections from Super-Peers and Leaf-Nodes
    address := fmt.Sprintf("%s:%d", sp.Config.Address, sp.Config.Port)
    listener, err := net.Listen("tcp", address)
    if err != nil {
        log.Fatalf("Super-Peer %s failed to listen on %s: %v", sp.Config.ID, address, err)
    }
    log.Printf("Super-Peer %s listening on %s", sp.Config.ID, address)

    // Initiate connections to neighboring Super-Peers
    for neighborID, neighborConfig := range sp.NeighborConfigs {
        go sp.connectToNeighbor(neighborID, neighborConfig)
    }

    // Start accepting incoming connections
    go sp.acceptConnections(listener)

    // Start periodic cleanup of the MessageCache to remove old entries
    go sp.cleanupMessageCache()

    // Start periodic logging of the FileIndex for debugging purposes
    go sp.logFileIndex()

    // Keep the Super-Peer running indefinitely
    select {}
}

// connectToNeighbor establishes a connection to a neighboring Super-Peer with retry logic
func (sp *SuperPeer) connectToNeighbor(neighborID string, neighborConfig SuperPeerConfig) {
    address := fmt.Sprintf("%s:%d", neighborConfig.Address, neighborConfig.Port)
    for {
        conn, err := net.Dial("tcp", address)
        if err != nil {
            log.Printf("Super-Peer %s failed to connect to neighbor Super-Peer %s at %s: %v", sp.Config.ID, neighborID, address, err)
            time.Sleep(5 * time.Second) // Retry after a delay
            continue
        }

        // Send identification message to the neighbor Super-Peer
        peerIDMsg := PeerIDMessage{
            MessageType: MsgTypePeerID,
            PeerType:    SuperPeerType,
            PeerID:      sp.Config.ID,
        }
        encoder := json.NewEncoder(conn)
        err = encoder.Encode(peerIDMsg)
        if err != nil {
            log.Printf("Super-Peer %s failed to send ID to neighbor Super-Peer %s: %v", sp.Config.ID, neighborID, err)
            conn.Close()
            time.Sleep(5 * time.Second) // Retry after a delay
            continue
        }

        // Store the connection to the neighbor
        sp.mu.Lock()
        sp.NeighborConns[neighborID] = conn
        sp.mu.Unlock()
        log.Printf("Super-Peer %s connected to neighbor Super-Peer %s at %s", sp.Config.ID, neighborID, address)

        // Handle communication with the neighbor Super-Peer
        go sp.handleNeighborConnection(conn, neighborID)
        break // Exit the loop upon successful connection
    }
}

// acceptConnections continuously accepts incoming connections to the Super-Peer
func (sp *SuperPeer) acceptConnections(listener net.Listener) {
    for {
        conn, err := listener.Accept()
        if err != nil {
            log.Printf("Super-Peer %s failed to accept connection: %v", sp.Config.ID, err)
            continue
        }

        // Handle the incoming connection in a separate goroutine
        go sp.handleIncomingConnection(conn)
    }
}

// handleIncomingConnection processes an incoming connection, determining if it's from a Super-Peer or Leaf-Node
func (sp *SuperPeer) handleIncomingConnection(conn net.Conn) {
    decoder := json.NewDecoder(conn)
    var msg PeerIDMessage

    // Decode the initial PeerIDMessage to identify the type of peer
    err := decoder.Decode(&msg)
    if err != nil {
        log.Printf("Super-Peer %s failed to decode message from %s: %v",
            sp.Config.ID, conn.RemoteAddr().String(), err)
        conn.Close()
        return
    }

    if msg.MessageType == MsgTypePeerID {
        if msg.PeerType == LeafNodeType {
            log.Printf("Super-Peer %s accepted connection from Leaf-Node %s", sp.Config.ID, msg.PeerID)
            // Handle Leaf-Node connection
            go sp.handleLeafNodeConnection(conn, msg.PeerID)
        } else if msg.PeerType == SuperPeerType {
            log.Printf("Super-Peer %s accepted connection from Super-Peer %s", sp.Config.ID, msg.PeerID)
            // Store the connection to the neighbor Super-Peer
            sp.mu.Lock()
            sp.NeighborConns[msg.PeerID] = conn
            sp.mu.Unlock()
            // Handle communication with the neighbor Super-Peer
            go sp.handleNeighborConnection(conn, msg.PeerID)
        } else {
            log.Printf("Super-Peer %s received unknown peer type from %s", sp.Config.ID, conn.RemoteAddr().String())
            conn.Close()
        }
    } else {
        log.Printf("Super-Peer %s received unknown message type from %s", sp.Config.ID, conn.RemoteAddr().String())
        conn.Close()
    }
}

// handleNeighborConnection manages communication with a neighbor Super-Peer
func (sp *SuperPeer) handleNeighborConnection(conn net.Conn, neighborID string) {
    log.Printf("Super-Peer %s handling neighbor connection with Super-Peer %s", sp.Config.ID, neighborID)

    decoder := json.NewDecoder(conn)
    for {
        var msg map[string]interface{}
        err := decoder.Decode(&msg)
        if err != nil {
            if err == io.EOF {
                log.Printf("Neighbor Super-Peer %s disconnected", neighborID)
            } else {
                log.Printf("Error decoding message from Neighbor Super-Peer %s: %v", neighborID, err)
            }
            break
        }

        // Extract the message type
        messageType, ok := msg["message_type"].(string)
        if !ok {
          log.Printf("Invalid message from Neighbor Super-Peer %s: missing message_type", neighborID)
          continue
        }

        // Handle the message based on its type
        switch messageType {
        case MsgTypeFileQuery:
            var queryMsg FileQueryMessage
            err := mapToStruct(msg, &queryMsg)
            if err != nil {
              log.Printf("Error decoding FileQueryMessage from Neighbor Super-Peer %s: %v", neighborID, err)
              continue
            }
            log.Printf("Super-Peer %s received FileQueryMessage for '%s' from %s", sp.Config.ID, queryMsg.FileName, queryMsg.OriginID)
            sp.handleFileQuery(queryMsg, conn)
        case MsgTypeQueryHit:
            var queryHitMsg QueryHitMessage
            err := mapToStruct(msg, &queryHitMsg)
            if err != nil {
```

```
                log.Printf("Error decoding QueryHitMessage from Neighbor Super-Peer %s: %v", neighborID, err)
                continue
            }
            sp.forwardQueryHit(queryHitMsg)
        default:
            log.Printf("Unknown message type '%s' from Neighbor Super-Peer %s", messageType, neighborID)
        }
    }

    // Remove the connection from NeighborConns upon disconnection
    sp.mu.Lock()
    delete(sp.NeighborConns, neighborID)
    sp.mu.Unlock()
}

// handleLeafNodeConnection manages communication with a connected Leaf-Node
func (sp *SuperPeer) handleLeafNodeConnection(conn net.Conn, leafNodeID string) {
    log.Printf("Super-Peer %s handling connection with Leaf-Node %s", sp.Config.ID, leafNodeID)

    // Store the connection to the Leaf-Node
    sp.mu.Lock()
    sp.LeafNodeConns[leafNodeID] = conn
    sp.mu.Unlock()

    decoder := json.NewDecoder(conn)
```

```
for {
    var msg map[string]interface{}
    err := decoder.Decode(&msg)
    if err != nil {
        if err == io.EOF {
            log.Printf("Leaf-Node %s disconnected", leafNodeID)
        } else {
            log.Printf("Error decoding message from Leaf-Node %s: %v", leafNodeID, err)
        }
        break
    }

    // Extract the message type
    messageType, ok := msg["message_type"].(string)
    if !ok {
        log.Printf("Invalid message from Leaf-Node %s: missing message_type", leafNodeID)
        continue
    }

    // Handle the message based on its type
    switch messageType {
    case MsgTypeFileRegistration:
        var registrationMsg FileRegistrationMessage
        err := mapToStruct(msg, &registrationMsg)
        if err != nil {
                log.Printf("Error decoding FileRegistrationMessage from Leaf-Node %s: %v", leafNodeID, err)
                continue
            }
            sp.handleFileRegistration(registrationMsg)
        case MsgTypeFileQuery:
            var queryMsg FileQueryMessage
            err := mapToStruct(msg, &queryMsg)
            if err != nil {
                log.Printf("Error decoding FileQueryMessage from Leaf-Node %s: %v", leafNodeID, err)
                continue
            }
            log.Printf("Super-Peer %s received FileQueryMessage for '%s' from %s", sp.Config.ID, queryMsg.FileName, queryMsg.OriginID)
            sp.handleFileQuery(queryMsg, conn)
        default:
            log.Printf("Unknown message type '%s' from Leaf-Node %s", messageType, leafNodeID)
        }
    }

    // Remove the connection from LeafNodeConns upon disconnection
    sp.mu.Lock()
    delete(sp.LeafNodeConns, leafNodeID)
    sp.mu.Unlock()
}

// sendJSONMessage serializes and sends a JSON message over a given connection
```

```
func (sp *SuperPeer) sendJSONMessage(conn net.Conn, msg interface{}) error {
    sp.connMu.Lock()
    defer sp.connMu.Unlock()
    encoder := json.NewEncoder(conn)
    return encoder.Encode(msg)
}

// handleFileRegistration processes a FileRegistrationMessage from a Leaf-Node
func (sp *SuperPeer) handleFileRegistration(msg FileRegistrationMessage) {
    sp.mu.Lock()
    defer sp.mu.Unlock()

    for _, file := range msg.Files {
        if sp.FileIndex[file.FileName] == nil {
            sp.FileIndex[file.FileName] = make(map[string]struct{})
        }

        if _, exists := sp.FileIndex[file.FileName][msg.LeafNodeID]; !exists {
            sp.FileIndex[file.FileName][msg.LeafNodeID] = struct{}{}
            log.Printf("Super-Peer %s registered file '%s' from Leaf-Node %s", sp.Config.ID,
file.FileName, msg.LeafNodeID)
        } else {
            log.Printf("Super-Peer %s ignored duplicate registration of file '%s' from Leaf-Node %s",
sp.Config.ID, file.FileName, msg.LeafNodeID)
        }
    }
}
```

```go
// handleFileQuery processes a FileQueryMessage, responding if the file is found and forwarding the query to neighbors
func (sp *SuperPeer) handleFileQuery(msg FileQueryMessage, sourceConn net.Conn) {
    sp.mu.Lock()
    defer sp.mu.Unlock()

    // Check if the MessageID has already been processed to prevent duplication
    if _, exists := sp.MessageCache[msg.MessageID]; exists {
        // Already processed, ignore the query
        return
    }

    // Store the MessageID in the cache
    sp.MessageCache[msg.MessageID] = CacheEntry{
        OriginID:     msg.OriginID,
        UpstreamConn: sourceConn,
        Timestamp:    time.Now(),
    }

    // Check if the file exists in the local FileIndex
    leafNodeIDs, found := sp.FileIndex[msg.FileName]

    if found {
        // Send QueryHitMessages back to the originator Leaf-Node for each Leaf-Node that has the file
        for leafNodeID := range leafNodeIDs {
```

```
if leafNodeID == msg.OriginID {
  // Skip sending to the origin Leaf-Node itself
  continue
}

leafConfig, exists := sp.LeafNodeConfigs[leafNodeID]
if !exists {
  continue
}

// Construct the QueryHitMessage
queryHitMsg := QueryHitMessage{
  MessageType:  MsgTypeQueryHit,
  MessageID:    msg.MessageID,
  TTL:         msg.TTL,
  RespondingID: leafNodeID,
  FileName:     msg.FileName,
  Address:      leafConfig.Address,
  Port:         leafConfig.Port,
}

// Send the QueryHitMessage to the originator's connection
err := sp.sendJSONMessage(sourceConn, queryHitMsg)
if err != nil {
  log.Printf("Error sending QueryHitMessage to originator: %v", err)
}
```

```
        }
    }

    // Forward the query to neighboring Super-Peers if TTL > 1
    if msg.TTL > 1 {
        msg.TTL--

        for neighborID, conn := range sp.NeighborConns {
            // Avoid sending back to the source if necessary
            if conn == sourceConn {
                continue
            }
            log.Printf("Super-Peer %s forwarding query for '%s' to neighbor Super-Peer %s",
sp.Config.ID, msg.FileName, neighborID)
            err := sp.sendJSONMessage(conn, msg)
            if err != nil {
                log.Printf("Error forwarding query to neighbor Super-Peer %s: %v", neighborID, err)
            }
        }
    }
}

// forwardQueryHit forwards a QueryHitMessage back to the originator Leaf-Node
func (sp *SuperPeer) forwardQueryHit(msg QueryHitMessage) {
    sp.mu.Lock()
    defer sp.mu.Unlock()
    // Retrieve the cache entry using MessageID
    entry, exists := sp.MessageCache[msg.MessageID]
    if !exists {
        log.Printf("No origin connection found for MessageID %s", msg.MessageID)
        return
    }

    // Decrement TTL if applicable
    if msg.TTL > 1 {
        msg.TTL--
    }

    log.Printf("Super-Peer %s forwarding QueryHitMessage for MessageID %s to originator", sp.Config.ID, msg.MessageID)
    err := sp.sendJSONMessage(entry.UpstreamConn, msg)
    if err != nil {
        log.Printf("Error forwarding QueryHitMessage: %v", err)
    }
}

// cleanupMessageCache periodically removes old entries from the MessageCache to free up resources
func (sp *SuperPeer) cleanupMessageCache() {
    ticker := time.NewTicker(10 * time.Minute) // Adjust the interval as needed
    defer ticker.Stop()
    for range ticker.C {
        sp.mu.Lock()
        for msgID, entry := range sp.MessageCache {
            if time.Since(entry.Timestamp) > 30*time.Minute { // Adjust the timeout as needed
                delete(sp.MessageCache, msgID)
                log.Printf("Super-Peer %s removed MessageID %s from MessageCache", sp.Config.ID,
msgID)
            }
        }
        sp.mu.Unlock()
    }
}

// logFileIndex periodically logs the current FileIndex for debugging and monitoring purposes
func (sp *SuperPeer) logFileIndex() {
    ticker := time.NewTicker(1 * time.Minute) // Adjust the interval as needed
    defer ticker.Stop()
    for range ticker.C {
        sp.mu.Lock()
        log.Printf("Super-Peer %s FileIndex Status:", sp.Config.ID)
        for file, leafNodes := range sp.FileIndex {
            leafList := []string{}
            for ln := range leafNodes {
                leafList = append(leafList, ln)
            }
            log.Printf("  File: '%s' -> Leaf-Nodes: %v", file, leafList)
        }
        sp.mu.Unlock()
    }
}

// -----------------------------
// LeafNode Implementation
// -----------------------------

// LeafNode represents a Leaf-Node in the network
type LeafNode struct {
    Config            LeafNodeConfig         // Configuration details of the Leaf-Node
    SuperPeerConfig    SuperPeerConfig        // Configuration details of the connected Super-Peer
    conn              net.Conn               // Active connection to the Super-Peer
    connMu            sync.Mutex             // Mutex to protect writes to the connection
    mu                sync.Mutex             // Mutex to protect shared resources
    responseTimes     []time.Duration        // Slice to record response times for queries
    startTimes        map[string]time.Time    // Map to store start times of queries
    downloadPromptChan chan DownloadPrompt     // Channel to handle download prompts
}

// NewLeafNode initializes a new Leaf-Node with the given configuration
func NewLeafNode(config LeafNodeConfig, globalConfig *Config) *LeafNode {
    var superPeerConfig SuperPeerConfig
    found := false
    // Find the Super-Peer configuration that the Leaf-Node connects to
    for _, spConfig := range globalConfig.SuperPeers {
        if spConfig.ID == config.SuperPeer {
            superPeerConfig = spConfig
            found = true
            break
        }
    }
    if !found {
        log.Fatalf("Super-Peer ID %s for Leaf-Node %s not found in configuration", config.SuperPeer, config.ID)
    }

    return &LeafNode{
        Config:             config,
        SuperPeerConfig:    superPeerConfig,
        responseTimes:      []time.Duration{},
        startTimes:         make(map[string]time.Time),
        downloadPromptChan: make(chan DownloadPrompt),
    }
}

// Start initializes the Leaf-Node, connects to its Super-Peer, and starts necessary services
func (ln *LeafNode) Start() {
    // Specify the shared directory based on the Leaf-Node ID
    sharedDir := "./shared_files/" + ln.Config.ID

    // Discover shared files in the designated directory
```

```go
    files, err := ln.discoverFiles(sharedDir)
    if err != nil {
        log.Fatalf("Leaf-Node %s failed to discover files: %v", ln.Config.ID, err)
    }

    log.Printf("Leaf-Node %s discovered %d files", ln.Config.ID, len(files))

    // Start the HTTP file server to serve shared files
    ln.startFileServer()

    // Attempt to connect to the Super-Peer with retry logic
    address := fmt.Sprintf("%s:%d", ln.SuperPeerConfig.Address, ln.SuperPeerConfig.Port)
    var conn net.Conn
    for {
        conn, err = net.Dial("tcp", address)
        if err != nil {
            log.Printf("Leaf-Node %s failed to connect to Super-Peer at %s: %v", ln.Config.ID, address, err)
            time.Sleep(5 * time.Second) // Retry after a delay
            continue
        }
        break // Exit the loop upon successful connection
    }

    // Store the active connection to the Super-Peer
    ln.conn = conn
```

```
// Send identification message to the Super-Peer
peerIDMsg := PeerIDMessage{
    MessageType: MsgTypePeerID,
    PeerType:    LeafNodeType,
    PeerID:      ln.Config.ID,
}
err = ln.sendJSONMessage(peerIDMsg)
if err != nil {
    log.Fatalf("Leaf-Node %s failed to send ID to Super-Peer: %v", ln.Config.ID, err)
}

log.Printf("Leaf-Node %s connected to Super-Peer at %s", ln.Config.ID, address)

// Send file registration message to the Super-Peer
registrationMsg := FileRegistrationMessage{
    MessageType: MsgTypeFileRegistration,
    LeafNodeID:  ln.Config.ID,
    Files:       files,
}
err = ln.sendJSONMessage(registrationMsg)
if err != nil {
    log.Fatalf("Leaf-Node %s failed to send file registration to Super-Peer: %v", ln.Config.ID, err)
}

// Handle incoming messages from the Super-Peer
```

```
    go ln.handleSuperPeerConnection(conn)

    // Start the user interface for handling user inputs and download prompts
    ln.startUserInterface()
}

// discoverFiles scans the shared directory and returns metadata of shared files
func (ln *LeafNode) discoverFiles(sharedDir string) ([]FileMetadata, error) {
    var files []FileMetadata

    err := filepath.Walk(sharedDir, func(path string, info os.FileInfo, err error) error {
        if err != nil {
            return err
        }
        if !info.IsDir() {
            files = append(files, FileMetadata{
                FileName: info.Name(),
                FileSize: info.Size(),
            })
        }
        return nil
    })

    if err != nil {
        return nil, err
    }
    return files, nil
}

// startFileServer starts an HTTP server to serve shared files
func (ln *LeafNode) startFileServer() {
    sharedDir := "./shared_files/" + ln.Config.ID
    fs := http.FileServer(http.Dir(sharedDir))
    http.Handle("/", fs)

    address := fmt.Sprintf("%s:%d", ln.Config.Address, ln.Config.Port)
    log.Printf("Leaf-Node %s starting file server at %s", ln.Config.ID, address)
    go func() {
        if err := http.ListenAndServe(address, nil); err != nil {
            log.Fatalf("Leaf-Node %s file server error: %v", ln.Config.ID, err)
        }
    }()
}

// handleSuperPeerConnection manages incoming messages from the Super-Peer
func (ln *LeafNode) handleSuperPeerConnection(conn net.Conn) {
    decoder := json.NewDecoder(conn)
    for {
        var msg map[string]interface{}
        err := decoder.Decode(&msg)
        if err != nil {
```

```
    if err == io.EOF {
        log.Printf("Super-Peer disconnected")
    } else {
        log.Printf("Error decoding message from Super-Peer: %v", err)
    }
    break
}

// Extract the message type
messageType, ok := msg["message_type"].(string)
if !ok {
    log.Printf("Invalid message from Super-Peer: missing message_type")
    continue
}

// Handle the message based on its type
switch messageType {
case MsgTypeQueryHit:
    // Decode the message as QueryHitMessage
    var queryHitMsg QueryHitMessage
    err := mapToStruct(msg, &queryHitMsg)
    if err != nil {
        log.Printf("Error decoding QueryHitMessage: %v", err)
        continue
    }
    go ln.handleQueryHit(queryHitMsg)
```

```
        default:
            log.Printf("Unknown message type '%s' from Super-Peer", messageType)
        }
    }
}

// startUserInterface handles user inputs and download prompts
func (ln *LeafNode) startUserInterface() {
    reader := bufio.NewReader(os.Stdin)
    for {
        select {
        case prompt := <-ln.downloadPromptChan:
            // Handle download prompt when a QueryHitMessage is received
            fmt.Printf("\nQuery Hit: File '%s' is available at Leaf-Node %s (%s:%d)\n",
                prompt.QueryHitMessage.FileName, prompt.QueryHitMessage.RespondingID,
                prompt.QueryHitMessage.Address, prompt.QueryHitMessage.Port)
            fmt.Printf("Do you want to download this file? (yes/no): ")
            response, _ := reader.ReadString('\n')
            response = strings.TrimSpace(strings.ToLower(response))
            if response == "yes" {
                ln.downloadFile(prompt.QueryHitMessage)
                prompt.ResponseChan <- true
            } else {
                prompt.ResponseChan <- false
            }
        default:
            // Prompt user for file search input
            fmt.Printf("\nEnter file name to search (or 'exit' to quit): ")
            fileName, _ := reader.ReadString('\n')
            fileName = strings.TrimSpace(fileName)

            if fileName == "exit" {
                fmt.Println("Exiting...")
                ln.conn.Close()
                os.Exit(0)
            }

            if fileName != "" {
                ln.sendFileQuery(fileName)
            }
        }
    }
}

// sendFileQuery sends a FileQueryMessage to the Super-Peer to search for a file
func (ln *LeafNode) sendFileQuery(fileName string) {
    messageID := ln.generateMessageID()
    queryMsg := FileQueryMessage{
        MessageType: MsgTypeFileQuery,
        MessageID:   messageID,
        OriginID:    ln.Config.ID,
        FileName:    fileName,
        TTL:       5, // Set an appropriate TTL value to limit query propagation
    }

    // Record the start time for response time measurement
    startTime := time.Now()

    // Store the start time associated with the MessageID
    ln.mu.Lock()
    ln.startTimes[messageID] = startTime
    ln.mu.Unlock()

    // Send the FileQueryMessage to the Super-Peer
    err := ln.sendJSONMessage(queryMsg)
    if err != nil {
        log.Printf("Leaf-Node %s failed to send file query: %v", ln.Config.ID, err)
        return
    }

    log.Printf("Leaf-Node %s sent file query for '%s' with MessageID %s", ln.Config.ID, fileName, messageID)

    // Optionally, print the query issued in a nicely formatted manner
    fmt.Printf("Issued Query: Looking for file '%s' with MessageID %s\n", fileName, messageID)
}

// sendJSONMessage serializes and sends a JSON message over the connection to the Super-Peer
```

```
func (ln *LeafNode) sendJSONMessage(msg interface{}) error {
    ln.connMu.Lock()
    defer ln.connMu.Unlock()
    encoder := json.NewEncoder(ln.conn)
    return encoder.Encode(msg)
}

// generateMessageID creates a unique MessageID using UUID and the Leaf-Node's ID
func (ln *LeafNode) generateMessageID() string {
    return fmt.Sprintf("%s-%s", ln.Config.ID, uuid.New().String())
}

// handleQueryHit processes a QueryHitMessage received from the Super-Peer
func (ln *LeafNode) handleQueryHit(msg QueryHitMessage) {
    // Record the response time
    endTime := time.Now()

    ln.mu.Lock()
    startTime, exists := ln.startTimes[msg.MessageID]
    if exists {
        responseTime := endTime.Sub(startTime)
        ln.responseTimes = append(ln.responseTimes, responseTime)
        // Remove the startTime as it's no longer needed
        delete(ln.startTimes, msg.MessageID)
        log.Printf("Response time for MessageID %s: %v", msg.MessageID, responseTime)
    }
    ln.mu.Unlock()

    // Prepare to prompt the user for downloading the file
    responseChan := make(chan bool)
    prompt := DownloadPrompt{
        QueryHitMessage: msg,
        ResponseChan:    responseChan,
    }

    // Send the prompt to the main input loop
    ln.downloadPromptChan <- prompt

    // Wait for the user's response (handled in startUserInterface)
    <-responseChan
}

// downloadFile downloads the specified file from the responding Leaf-Node
func (ln *LeafNode) downloadFile(msg QueryHitMessage) {
    url := fmt.Sprintf("http://%s:%d/%s", msg.Address, msg.Port, msg.FileName)
    log.Printf("Downloading file from %s", url)

    resp, err := http.Get(url)
    if err != nil {
        log.Printf("Error downloading file: %v", err)
        return
    }
```

```
defer resp.Body.Close()

if resp.StatusCode != http.StatusOK {
    log.Printf("Failed to download file: %s", resp.Status)
    return
}

// Save the file to the local shared directory
sharedDir := "./shared_files/" + ln.Config.ID
filePath := filepath.Join(sharedDir, msg.FileName)

// Check if the file already exists to prevent re-registration
if _, err := os.Stat(filePath); err == nil {
    log.Printf("File '%s' already exists. Skipping download.", msg.FileName)
    return
}

outFile, err := os.Create(filePath)
if err != nil {
    log.Printf("Error creating file: %v", err)
    return
}
defer outFile.Close()

_, err = io.Copy(outFile, resp.Body)
if err != nil {
        log.Printf("Error saving file: %v", err)
        return
    }

    // Display the downloaded file message as per requirement
    fmt.Printf("display file '%s'\n", msg.FileName)

    log.Printf("File '%s' downloaded successfully", msg.FileName)

    // Re-register the new file only if it's newly downloaded
    newFile := FileMetadata{
        FileName: msg.FileName,
        FileSize: getFileSize(filePath),
    }

    registrationMsg := FileRegistrationMessage{
        MessageType: MsgTypeFileRegistration,
        LeafNodeID:  ln.Config.ID,
        Files:       []FileMetadata{newFile},
    }

    err = ln.sendJSONMessage(registrationMsg)
    if err != nil {
        log.Printf("Error re-registering file: %v", err)
    }
}
```

```
// getFileSize returns the size of the file at the given path
func getFileSize(path string) int64 {
    info, err := os.Stat(path)
    if err != nil {
        return 0
    }
    return info.Size()
}

// ----------------------------
// Helper Functions
// ----------------------------

// mapToStruct converts a map to a struct using JSON marshalling and unmarshalling
func mapToStruct(m map[string]interface{}, result interface{}) error {
    data, err := json.Marshal(m)
    if err != nil {
        return err
    }
    return json.Unmarshal(data, result)
}

// ----------------------------
// Main Function
// ----------------------------
```

```
func main() {
    // Ensure that the config file path and peer ID are provided as command-line arguments
    if len(os.Args) < 3 {
        log.Fatal("Usage: go run main.go [config file] [peer ID]")
    }

    // Retrieve the config file path and peer ID from command-line arguments
    configFile := os.Args[1]
    peerID := os.Args[2]

    // Load the network configuration from the config file
    config, err := LoadConfig(configFile)
    if err != nil {
        log.Fatalf("Error loading config: %v", err)
    }

    // Determine if the provided peer ID corresponds to a Super-Peer or a Leaf-Node
    isSuperPeer := false
    var superPeerConfig SuperPeerConfig
    var leafNodeConfig LeafNodeConfig

    // Search for the peer ID in the Super-Peers list
    for _, sp := range config.SuperPeers {
        if sp.ID == peerID {
            isSuperPeer = true
            superPeerConfig = sp
            break
        }
    }

    if !isSuperPeer {
        // If not found in Super-Peers, search in the Leaf-Nodes list
        found := false
        for _, ln := range config.LeafNodes {
            if ln.ID == peerID {
                leafNodeConfig = ln
                found = true
                break
            }
        }
        if !found {
            log.Fatalf("Peer ID %s not found in configuration", peerID)
        }
    }

    // Initialize and start the peer based on its role
    if isSuperPeer {
        fmt.Printf("Starting Super-Peer %s\n", superPeerConfig.ID)
        // Create a new Super-Peer instance and start it
        sp := NewSuperPeer(superPeerConfig, config)
        sp.Start()
    } else {
        fmt.Printf("Starting Leaf-Node %s\n", leafNodeConfig.ID)
        // Create a new Leaf-Node instance and start it
        ln := NewLeafNode(leafNodeConfig, config)
        ln.Start()
    }}
```

Performance test

automatedclient.py

```
import socket
import json
import threading
import time
import uuid
import sys
from queue import Queue
import csv
import os
import logging

# Configure logging
logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(levelname)s - %(message)s')

# Configuration Files
CONFIG_FILE = 'config.json'
```

```
# Constants
FILE_NAME = 'file12.txt'  # The file you want to query
NUM_QUERIES = 200         # Number of queries each client will send per repetition
CUT_OFF_TIME = 5          # Time in seconds to wait for responses after sending queries
REPEAT_EXPERIMENTS = 200  # Number of times to repeat the experiment

# Thread-safe queue to collect results
results_queue = Queue()

def load_config(config_file):
    """
    Load the configuration from the config.json file.
    Returns dictionaries mapping Super-Peers and Leaf-Nodes.
    """
    with open(config_file, 'r') as f:
        config = json.load(f)

    super_peers = config['super_peers']
    leaf_nodes = config['leaf_nodes']

    # Create a mapping from Leaf-Node ID to Super-Peer ID
    leaf_to_sp = {leaf['id']: leaf['super_peer'] for leaf in leaf_nodes}

    # Create a list of Super-Peers
    super_peer_list = []
    for sp in super_peers:
        super_peer_list.append({
            'id': sp['id'],
            'address': sp['address'],
            'port': sp['port'],
            'neighbors': sp['neighbors'],
            'leaf_nodes': sp['leaf_nodes']
        })

    return super_peer_list, leaf_to_sp

def client_thread(client_id, num_queries, super_peer_map, leaf_to_sp):
    """
    Function executed by each client thread.
    Connects to a Super-Peer, sends queries, and records response times.
    """
    try:
        # Determine which Super-Peer this client will connect to
        # For simplicity, distribute clients evenly across Super-Peers
        sp_index = (client_id - 1) % len(super_peer_map)
        super_peer = super_peer_map[sp_index]
        sp_address = super_peer['address']
        sp_port = super_peer['port']
        sp_id = super_peer['id']

        logging.info(f"[Client {client_id}] Connecting to Super-Peer {sp_id} at {sp_address}:{sp_port}...")
```

```
# Establish connection to Super-Peer
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.connect((sp_address, sp_port))
logging.info(f"[Client {client_id}] Connected to Super-Peer {sp_id}.")

# Send PeerIDMessage
peer_id = f"TestClient-{client_id}-{uuid.uuid4()}"
peer_id_msg = {
    "message_type": "peer_id",
    "peer_type": "leaf_node",
    "peer_id": peer_id
}
sock.sendall((json.dumps(peer_id_msg) + '\n').encode())
logging.info(f"[Client {client_id}] Sent PeerIDMessage.")

# Send FileRegistrationMessage
registration_msg = {
    "message_type": "file_registration",
    "leaf_node_id": peer_id,
    "files": [
        {"file_name": FILE_NAME, "file_size": 1024}  # Example file metadata
    ]
}
sock.sendall((json.dumps(registration_msg) + '\n').encode())
logging.info(f"[Client {client_id}] Sent FileRegistrationMessage.")
```

```
# Function to receive QueryHitMessages
def receive_messages(sock, message_id_map, client_id, response_category_map):
    logging.info(f"[Client {client_id}] Receiver thread started.")
    sock_file = sock.makefile('r')
    while True:
        line = sock_file.readline()
        if not line:
            logging.info(f"[Client {client_id}] Connection closed by Super-Peer.")
            break
        try:
            msg = json.loads(line.strip())
            if msg.get("message_type") == "query_hit":
                msg_id = msg.get("message_id")
                if msg_id in message_id_map:
                    response_time = (time.time() - message_id_map[msg_id]['start_time']) * 1000  # Convert to ms
                    leaf_node = msg.get("responding_id")
                    # Determine if Leaf-Node is within the same Super-Peer or different
                    category = "Same Super-Peer" if leaf_to_sp.get(leaf_node) == sp_id else "Different Super-Peer"
                    results_queue.put({
                        "client_id": client_id,
                        "message_id": msg_id,
                        "response_time_ms": response_time,
                        "leaf_node": leaf_node,
                        "category": category
                    })
                    logging.info(f"[Client {client_id}] Received QueryHitMessage from {leaf_node} ({category}) with response time {response_time:.2f} ms.")
        except json.JSONDecodeError as e:
            logging.error(f"[Client {client_id}] Failed to decode JSON message: {e}")

# Start receiver thread
message_id_map = {}
receiver = threading.Thread(target=receive_messages, args=(sock, message_id_map, client_id, {}))
receiver.daemon = True
receiver.start()
logging.info(f"[Client {client_id}] Started receiver thread.")

# Send queries
for i in range(1, num_queries + 1):
    message_id = f"{peer_id}-{uuid.uuid4()}"
    query_msg = {
        "message_type": "file_query",
        "message_id": message_id,
        "origin_id": peer_id,
        "file_name": FILE_NAME,
        "ttl": 5
    }
    message_id_map[message_id] = {'start_time': time.time()}
    sock.sendall((json.dumps(query_msg) + '\n').encode())
    logging.info(f"[Client {client_id}] Sent Query {i}/{num_queries}.")
```

```
            time.sleep(0.01)  # Small delay between queries to prevent overwhelming the network

        logging.info(f"[Client {client_id}] All queries sent. Waiting for responses...")
        # Wait for the cutoff time
        time.sleep(CUT_OFF_TIME)
        sock.close()
        logging.info(f"[Client {client_id}] Connection closed.")

    except Exception as e:
        logging.error(f"[Client {client_id}] An error occurred: {e}")
        # Optionally, handle specific exceptions or perform cleanup

def run_experiment(num_clients, super_peer_map, leaf_to_sp):
    """
    Runs the experiment for a given number of clients.
    """
    threads = []
    for client_id in range(1, num_clients + 1):
        t = threading.Thread(target=client_thread, args=(client_id, NUM_QUERIES,
super_peer_map, leaf_to_sp))
        t.start()
        threads.append(t)

    # Wait for all threads to finish
    for t in threads:
        t.join()
    # Collect and process results
    results = []
    while not results_queue.empty():
        results.append(results_queue.get())

    # Write results to CSV
    csv_filename = f'results_{num_clients}_clients.csv'
    file_exists = os.path.isfile(csv_filename)

    with open(csv_filename, 'a', newline='') as csvfile:
        fieldnames = ['client_id', 'message_id', 'response_time_ms', 'leaf_node', 'category']
        writer = csv.DictWriter(csvfile, fieldnames=fieldnames)
        # Write header only if file does not exist
        if not file_exists:
            writer.writeheader()
        for r in results:
            writer.writerow(r)

    # Compute average response time
    if results:
        total_response_time = sum(r['response_time_ms'] for r in results)
        average_response_time = total_response_time / len(results)
        logging.info(f"\nAverage Response Time for {num_clients} client(s):
{average_response_time:.2f} ms over {len(results)} hits\n")
    else:
        logging.info("\nNo QueryHitMessages received.\n")


def main():
    # Load configuration
    super_peer_map, leaf_to_sp = load_config(CONFIG_FILE)

    # Command-line arguments: number of clients and number of repetitions
    if len(sys.argv) == 3:
        try:
            num_clients = int(sys.argv[1])
            repeat_experiments = int(sys.argv[2])
        except ValueError:
            logging.error("Invalid arguments. Usage: python automatedclient.py <num_clients> <repeat_experiments>")
            sys.exit(1)
    elif len(sys.argv) == 2:
        try:
            num_clients = int(sys.argv[1])
            repeat_experiments = 200  # Default
        except ValueError:
            logging.error("Invalid number of clients. Usage: python automatedclient.py <num_clients> <repeat_experiments>")
            sys.exit(1)
    else:
        logging.error("Usage: python automatedclient.py <num_clients> <repeat_experiments>")
        logging.error("Example: python automatedclient.py 5 200")
        sys.exit(1)
    logging.info(f"Starting automated_client.py with {num_clients} client(s) for {repeat_experiments} repetitions...\n")

    # Initialize CSV files by writing headers if they don't exist
    for _ in range(1, num_clients + 1):
        csv_filename = f'results_{num_clients}_clients.csv'
        if not os.path.isfile(csv_filename):
            with open(csv_filename, 'w', newline='') as csvfile:
                fieldnames = ['client_id', 'message_id', 'response_time_ms', 'leaf_node', 'category']
                writer = csv.DictWriter(csvfile, fieldnames=fieldnames)
                writer.writeheader()

    # Run experiments
    for repetition in range(1, repeat_experiments + 1):
        logging.info(f"--- Experiment {repetition}/{repeat_experiments} for {num_clients} client(s) ---")
        run_experiment(num_clients, super_peer_map, leaf_to_sp)

if __name__ == "__main__":
    main()
```

Compute_Average.py

```
import pandas as pd

def compute_averages():
    # Load consolidated data
    df = pd.read_csv('consolidated_results.csv')

    # Calculate average response time per client count
    summary = df.groupby('Number of Clients')['response_time_ms'].mean().reset_index()
    summary.rename(columns={'response_time_ms': 'Average Response Time (ms)'}, inplace=True)

    # Save summary to CSV
    summary.to_csv('summary_response_times.csv', index=False)
    print("Summary of Average Response Times:")
    print(summary)

if __name__ == "__main__":
    compute_averages()
```

Linear Topology

Config_linear.json

```
{
  "super_peers": [
    {
      "id": "SP1",
      "address": "127.0.0.1",
      "port": 8000,
      "client_port": 9000,
      "neighbors": ["SP2"],
      "leaf_nodes": ["LN1", "LN2"]
```

```
},
{
  "id": "SP2",
  "address": "127.0.0.1",
  "port": 8001,
  "client_port": 9001,
  "neighbors": ["SP1", "SP3"],
  "leaf_nodes": ["LN3", "LN4"]
},
{
  "id": "SP3",
  "address": "127.0.0.1",
  "port": 8002,
  "client_port": 9002,
  "neighbors": ["SP2", "SP4"],
  "leaf_nodes": ["LN5", "LN6"]
},
{
  "id": "SP4",
  "address": "127.0.0.1",
  "port": 8003,
  "client_port": 9003,
  "neighbors": ["SP3", "SP5"],
  "leaf_nodes": ["LN7"]
},
{
      "id": "SP5",
      "address": "127.0.0.1",
      "port": 8004,
      "client_port": 9004,
      "neighbors": ["SP4", "SP6"],
      "leaf_nodes": ["LN8"]
    },
    {
      "id": "SP6",
      "address": "127.0.0.1",
      "port": 8005,
      "client_port": 9005,
      "neighbors": ["SP5", "SP7"],
      "leaf_nodes": ["LN9"]
    },
    {
      "id": "SP7",
      "address": "127.0.0.1",
      "port": 8006,
      "client_port": 9006,
      "neighbors": ["SP6", "SP8"],
      "leaf_nodes": ["LN10"]
    },
    {
      "id": "SP8",
      "address": "127.0.0.1",
      "port": 8007,
      "client_port": 9007,
      "neighbors": ["SP7", "SP9"],
      "leaf_nodes": []
    },
    {
      "id": "SP9",
      "address": "127.0.0.1",
      "port": 8008,
      "client_port": 9008,
      "neighbors": ["SP8", "SP10"],
      "leaf_nodes": []
    },
    {
      "id": "SP10",
      "address": "127.0.0.1",
      "port": 8009,
      "client_port": 9009,
      "neighbors": ["SP9"],
      "leaf_nodes": []
    }
  ],
  "leaf_nodes": [
    {
      "id": "LN1",
      "address": "127.0.0.1",
  "port": 57210,
  "super_peer": "SP1"
},
{
  "id": "LN2",
  "address": "127.0.0.1",
  "port": 57226,
  "super_peer": "SP1"
},
{
  "id": "LN3",
  "address": "127.0.0.1",
  "port": 57228,
  "super_peer": "SP2"
},
{
  "id": "LN4",
  "address": "127.0.0.1",
  "port": 57237,
  "super_peer": "SP2"
},
{
  "id": "LN5",
  "address": "127.0.0.1",
  "port": 57285,
  "super_peer": "SP3"
```

```
},
{
  "id": "LN6",
  "address": "127.0.0.1",
  "port": 57292,
  "super_peer": "SP3"
},
{
  "id": "LN7",
  "address": "127.0.0.1",
  "port": 57301,
  "super_peer": "SP4"
},
{
  "id": "LN8",
  "address": "127.0.0.1",
  "port": 57309,
  "super_peer": "SP5"
},
{
  "id": "LN9",
  "address": "127.0.0.1",
  "port": 57312,
  "super_peer": "SP6"
},
{
      "id": "LN10",
      "address": "127.0.0.1",
      "port": 57339,
      "super_peer": "SP7"
    }
  ]
}
```

Superpeer.py

```
import socket
import json
import threading
import argparse


def parse_arguments():
    parser = argparse.ArgumentParser(description="Super-Peer")
    parser.add_argument('config_file', type=str, help='Path to configuration JSON file')
    parser.add_argument('super_peer_id', type=str, help='ID of the Super-Peer to start')
    return parser.parse_args()


def handle_leaf_node(conn, addr, super_peer):
    try:
        with conn:
            leaf_file = conn.makefile('r')
            # Expecting FileRegistrationMessage
            line = leaf_file.readline()
            if not line:
                print(f"[{super_peer['id']}] No data received from Leaf Node at {addr}. Closing connection.")
                return
            msg = json.loads(line.strip())
            if msg.get("message_type") != "file_registration":
                print(f"[{super_peer['id']}] Unknown initial message type from {addr}: {msg.get('message_type')}")
                return
            # Process file registration
            leaf_id = msg.get("leaf_node_id")
            files = msg.get("files", [])
            super_peer['leaf_nodes'][leaf_id] = {'files': files, 'address': addr}
            print(f"[{super_peer['id']}] Registered Leaf Node {leaf_id} with files: {files}")
            # Continue handling leaf node messages
            while True:
                line = leaf_file.readline()
                if not line:
                    print(f"[{super_peer['id']}] Leaf Node {leaf_id} disconnected.")
                    break
                # Handle further messages from leaf nodes if necessary
    except Exception as e:
        print(f"[{super_peer['id']}] Error handling Leaf Node at {addr}: {e}")


def handle_client(conn, addr, super_peer):
    try:
        with conn:
            client_file = conn.makefile('r')
            while True:
                line = client_file.readline()
                if not line:
                    print(f"[{super_peer['id']}] Client at {addr} disconnected.")
                    break
                msg = json.loads(line.strip())
                if msg.get("message_type") == "file_query":
                    # Process file query
                    query_id = msg.get("message_id")
                    origin_id = msg.get("origin_id")
                    file_name = msg.get("file_name")
                    ttl = msg.get("ttl", 5)
                    print(f"[{super_peer['id']}] Received file_query from {origin_id}: {msg}")
                    # Search for the file in registered leaf nodes
                    for leaf_id, leaf_info in super_peer['leaf_nodes'].items():
                        for file in leaf_info['files']:
                            if file['file_name'] == file_name:
                                # Send QueryHitMessage back to client
                                response_msg = {
                                    "message_type": "query_hit",
                                    "message_id": query_id,
                                    "responding_id": leaf_id
                                }
                                conn.sendall((json.dumps(response_msg) + '\n').encode())
                                print(f"[{super_peer['id']}] Sent QueryHitMessage to Client {origin_id} for {file_name} from {leaf_id}")
                else:
                    print(f"[{super_peer['id']}] Unknown message type from Client at {addr}: {msg.get('message_type')}")
    except Exception as e:
        print(f"[{super_peer['id']}] Error handling Client at {addr}: {e}")


def start_super_peer(super_peer):
    # Start Leaf Node listener
    leaf_listener = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    leaf_listener.bind((super_peer['address'], super_peer['port']))
    leaf_listener.listen()
    print(f"[{super_peer['id']}] Listening for Leaf Nodes on {super_peer['address']}:{super_peer['port']}")

    # Start Client listener
    client_listener = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    client_listener.bind((super_peer['address'], super_peer['client_port']))
    client_listener.listen()
    print(f"[{super_peer['id']}] Listening for Clients on {super_peer['address']}:{super_peer['client_port']}")

    def accept_leaf_nodes():
        while True:
            conn, addr = leaf_listener.accept()
            print(f"[{super_peer['id']}] Connected to Leaf Node at {addr}")
            threading.Thread(target=handle_leaf_node, args=(conn, addr, super_peer), daemon=True).start()
    def accept_clients():
        while True:
            conn, addr = client_listener.accept()
            print(f"[{super_peer['id']}] Connected to Client at {addr}")
            threading.Thread(target=handle_client, args=(conn, addr, super_peer),
daemon=True).start()

    # Start threads to accept connections
    threading.Thread(target=accept_leaf_nodes, daemon=True).start()
    threading.Thread(target=accept_clients, daemon=True).start()

    # Keep the main thread alive
    while True:
        try:
            threading.Event().wait(1)
        except KeyboardInterrupt:
            print(f"[{super_peer['id']}] Shutting down.")
            break

def main():
    args = parse_arguments()
    config_file = args.config_file
    super_peer_id = args.super_peer_id

    # Load configuration
    with open(config_file, 'r') as f:
        config = json.load(f)

    # Find the super_peer in the config
    super_peers = config.get('super_peers', [])
    super_peer = next((sp for sp in super_peers if sp['id'] == super_peer_id), None)
    if not super_peer:
        print(f"Super-Peer ID {super_peer_id} not found in configuration.")
        return

    # Initialize leaf_nodes dictionary
    super_peer['leaf_nodes'] = {}

    # Start the Super-Peer
    start_super_peer(super_peer)

if __name__ == "__main__":
    main()
```

Leafnode.py

```
import socket
import json
import threading
import time
import uuid
import argparse
```

```
def parse_arguments():
    parser = argparse.ArgumentParser(description="Leaf Node")
    parser.add_argument('--id', type=str, required=True, help='Leaf Node ID')
    parser.add_argument('--address', type=str, default='127.0.0.1', help='Leaf Node IP Address')
    parser.add_argument('--port', type=int, required=True, help='Leaf Node Port')
    parser.add_argument('--super_peer_id', type=str, required=True, help='Assigned Super-Peer ID')
    parser.add_argument('--super_peer_address', type=str, default='127.0.0.1', help='Super-Peer IP Address')
    parser.add_argument('--super_peer_port', type=int, required=True, help='Super-Peer Port')
    parser.add_argument('--file', type=str, required=True, help='File to register')
    return parser.parse_args()


def send_file_registration(sock, leaf_node_id, files):
    registration_msg = {
        "message_type": "file_registration",
        "leaf_node_id": leaf_node_id,
        "files": files
    }
    sock.sendall((json.dumps(registration_msg) + '\n').encode())
    print(f"[{leaf_node_id}] Sent FileRegistrationMessage: {registration_msg}")


def handle_queries(sock, leaf_node_id, available_files):
    sock_file = sock.makefile('r')
    while True:
        line = sock_file.readline()
        if not line:
            print(f"[{leaf_node_id}] Connection closed by Super-Peer.")
            break
        try:
            msg = json.loads(line.strip())
            if msg.get("message_type") == "file_query":
                query_id = msg.get("message_id")
                requested_file = msg.get("file_name")
                origin_id = msg.get("origin_id")
                ttl = msg.get("ttl", 5)
                print(f"[{leaf_node_id}] Received file_query: {msg}")

                # Check if the requested file exists
                if requested_file in available_files:
                    response_msg = {
                        "message_type": "query_hit",
                        "message_id": query_id,
                        "responding_id": leaf_node_id
                    }
                    sock.sendall((json.dumps(response_msg) + '\n').encode())
                    print(f"[{leaf_node_id}] Sent QueryHitMessage for MessageID {query_id} to Super-Peer.")
                else:
                    # Optionally send a QueryMissMessage or ignore
                    print(f"[{leaf_node_id}] File {requested_file} not found. Ignoring query.")
        except json.JSONDecodeError as e:
            print(f"[{leaf_node_id}] Failed to decode JSON message: {e}")

def main():
    args = parse_arguments()

    leaf_node_id = args.id
    leaf_node_address = args.address
    leaf_node_port = args.port
    super_peer_id = args.super_peer_id
    super_peer_address = args.super_peer_address
    super_peer_port = args.super_peer_port
    file_to_register = args.file

    available_files = [file_to_register]  # List of files this leaf node has

    try:
        # Establish connection to Super-Peer
        print(f"[{leaf_node_id}] Connecting to Super-Peer {super_peer_id} at {super_peer_address}:{super_peer_port}...")
        sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        sock.connect((super_peer_address, super_peer_port))
        print(f"[{leaf_node_id}] Connected to Super-Peer {super_peer_id}.")

        # Send FileRegistrationMessage
        send_file_registration(sock, leaf_node_id, [{"file_name": f, "file_size": 1024} for f in available_files])
        # Start thread to handle incoming queries
        query_handler = threading.Thread(target=handle_queries, args=(sock, leaf_node_id, available_files))
        query_handler.daemon = True
        query_handler.start()
        print(f"[{leaf_node_id}] Started query handler thread.")

        # Keep the main thread alive
        while True:
            time.sleep(1)

    except Exception as e:
        print(f"[{leaf_node_id}] Encountered an error: {e}")
    finally:
        sock.close()
        print(f"[{leaf_node_id}] Connection closed.")

if __name__ == "__main__":
    main()
```

automatedclient_linear.py

```
import socket
import json
import threading
import time
import uuid
import sys
```

```
from queue import Queue
import csv
import argparse

# Thread-safe queue to collect results
results_queue = Queue()

def parse_arguments():
    parser = argparse.ArgumentParser(description="Automated P2P Client")
    parser.add_argument('--config', type=str, required=True, help='Path to configuration JSON file')
    parser.add_argument('--clients', type=int, default=1, help='Number of concurrent clients to simulate')
    parser.add_argument('--queries', type=int, default=200, help='Number of queries each client will send')
    return parser.parse_args()

def load_config(config_file):
    try:
        with open(config_file, 'r') as f:
            config = json.load(f)
        return config
    except Exception as e:
        print(f"Error loading configuration file: {e}")
        sys.exit(1)

# Configuration
```

```
FILE_NAME = 'file12.txt'          # The file you want to query
CUT_OFF_TIME = 30                 # Increased time in seconds to wait for responses after sending queries


def client_thread(client_id, super_peer, num_queries):
    try:
        super_peer_address = super_peer['address']
        super_peer_port = super_peer['client_port']  # Connect to client port
        super_peer_id = super_peer.get('id', f"SP{super_peer_port}")  # Default ID if not provided
        print(f"[Client {client_id}] Establishing connection to Super-Peer {super_peer_id} at {super_peer_address}:{super_peer_port}...")

        # Establish connection to Super-Peer
        sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        sock.connect((super_peer_address, super_peer_port))
        print(f"[Client {client_id}] Connected to Super-Peer {super_peer_id}.")

        # Function to receive QueryHitMessages
        def receive_messages(sock, message_id_map):
            print(f"[Client {client_id}] Receiver thread started and listening for messages.")
            sock_file = sock.makefile('r')
            while True:
                line = sock_file.readline()
                if not line:
                    print(f"[Client {client_id}] No more messages. Connection closed by Super-Peer.")
                    break
                try:
```

```
                msg = json.loads(line.strip())
                if msg.get("message_type") == "query_hit":
                    msg_id = msg.get("message_id")
                    if msg_id in message_id_map:
                        response_time = (time.perf_counter() - message_id_map[msg_id]['start_time'])
* 1000  # Convert to ms
                        results_queue.put({
                            "client_id": client_id,
                            "message_id": msg_id,
                            "response_time_ms": response_time,
                            "leaf_node": msg.get("responding_id")
                        })
                        print(f"[Client {client_id}] Received QueryHitMessage for MessageID {msg_id}
from Leaf-Node {msg.get('responding_id')} with response time {response_time:.2f} ms.")
            except json.JSONDecodeError as e:
                print(f"[Client {client_id}] Failed to decode JSON message: {e}")

    # Start receiver thread
    message_id_map = {}
    receiver = threading.Thread(target=receive_messages, args=(sock, message_id_map))
    receiver.daemon = True
    receiver.start()
    print(f"[Client {client_id}] Started receiver thread.")

    # Send queries
    for i in range(1, num_queries + 1):
        message_id = f"Client-{client_id}-{uuid.uuid4()}"
```

```
            query_msg = {
                "message_type": "file_query",
                "message_id": message_id,
                "origin_id": f"Client-{client_id}",
                "file_name": FILE_NAME,
                "ttl": 5
            }
            message_id_map[message_id] = {'start_time': time.perf_counter()}
            sock.sendall((json.dumps(query_msg) + '\n').encode())
            print(f"[Client {client_id}] Sent Query {i}/{num_queries}: {query_msg}")
            time.sleep(0.01)  # Small delay between queries to prevent overwhelming the network

        print(f"[Client {client_id}] All queries sent. Waiting for responses...")
        # Wait for the cutoff time
        time.sleep(CUT_OFF_TIME)
        sock.close()
        print(f"[Client {client_id}] Connection closed.")

    except Exception as e:
        print(f"[Client {client_id}] Encountered an error: {e}")


def main():
    args = parse_arguments()

    config_file = args.config
    num_clients = args.clients
    num_queries = args.queries

    print(f"Loading configuration from {config_file}...")
    config = load_config(config_file)

    super_peers = config.get('super_peers', [])
    if not super_peers:
        print("No super-peers found in configuration.")
        sys.exit(1)

    print(f"Starting automated_client.py with {num_clients} client(s) and {num_queries} queries each...")
    threads = []
    for client_id in range(1, num_clients + 1):
        # Assign super-peers in a round-robin fashion
        super_peer = super_peers[(client_id - 1) % len(super_peers)]
        t = threading.Thread(target=client_thread, args=(client_id, super_peer, num_queries))
        t.start()
        threads.append(t)

    # Wait for all threads to finish
    for t in threads:
        t.join()

    # Collect and process results
    results = []
    while not results_queue.empty():
        results.append(results_queue.get())

    # Write results to CSV
    csv_filename = f'results_{num_clients}_clients.csv'
    try:
        with open(csv_filename, 'w', newline='') as csvfile:
            fieldnames = ['client_id', 'message_id', 'response_time_ms', 'leaf_node']
            writer = csv.DictWriter(csvfile, fieldnames=fieldnames)
            writer.writeheader()
            for r in results:
                writer.writerow(r)
        print(f"\nResults have been written to {csv_filename}")
    except Exception as e:
        print(f"Error writing to CSV: {e}")

    # Compute average response time
    if results:
        total_response_time = sum(r['response_time_ms'] for r in results)
        average_response_time = total_response_time / len(results)
        print(f"\nAverage Response Time: {average_response_time:.2f} ms over {len(results)} hits")
    else:
        print("\nNo QueryHitMessages received.")

if __name__ == "__main__":
    main()
```

compute_avg_linear.py

```
import pandas as pd


# Define the data for Linear Topology
data = {
    'Client Number': [
        'Client 1', 'Client 2', 'Client 3', 'Client 4', 'Client 5',
        'Client 6', 'Client 7', 'Client 8', 'Client 9', 'Client 10'
    ],
    'Average Response Time (ms)': [0.73, 0.50, 0.64, 0.43, 0.39, 0.48, 0.46, 0.48, 0.54, 0.41],
    'Total Hits': [400, 800, 1200, 1400, 1600, 1800, 2000, 2000, 2000, 2000]
}


# Create a DataFrame from the data
df = pd.DataFrame(data)


# Calculate Overall Average Response Time and Total Hits
overall_average = df['Average Response Time (ms)'].mean()
overall_hits = df['Total Hits'].sum()


# Create a summary row as a DataFrame
summary = pd.DataFrame({
    'Client Number': ['Overall Average'],
    'Average Response Time (ms)': [round(overall_average, 2)],
    'Total Hits': [overall_hits]
})

# Concatenate the summary row to the original DataFrame using pd.concat
df = pd.concat([df, summary], ignore_index=True)

# Specify the CSV file name
csv_file = 'linear_topology_results.csv'

# Write the DataFrame to a CSV file
df.to_csv(csv_file, index=False)

print(f"CSV file '{csv_file}' has been created successfully.")
```

ALL TO ALL Topology vs Linear Topology

```
import matplotlib.pyplot as plt

# Data for All-to-All Topology
all_to_all_clients = list(range(1, 11))
all_to_all_hits = [400, 800, 1200, 1600, 2000, 2400, 2800, 3200, 3187, 4000]
all_to_all_response_times = [1.24, 1.19, 3.59, 151.96, 511.72, 879.73, 1650.13, 1535.96, 1812.33, 2146.98]

# Data for Linear Topology
linear_clients = list(range(1, 11))
linear_hits = [400, 800, 1200, 1400, 1600, 1800, 2000, 2000, 2000, 2000]
linear_response_times = [0.73, 0.50, 0.64, 0.43, 0.39, 0.48, 0.46, 0.48, 0.54, 0.41]
```

```
# Plot Average Response Time for Linear Topology
plt.figure(figsize=(12, 6))
plt.plot(linear_hits, linear_response_times, marker='s', color='blue', label='Linear Topology')
plt.xlabel('Number of Hits')
plt.ylabel('Average Response Time (ms)')
plt.title('Average Response Time vs. Number of Hits for Linear Topology')
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()

# Plot Comparison Between All-to-All and Linear Topologies
plt.figure(figsize=(12, 6))
plt.plot(all_to_all_hits, all_to_all_response_times, marker='o', color='red', label='All-to-All Topology')
plt.plot(linear_hits, linear_response_times, marker='s', color='blue', label='Linear Topology')
plt.xlabel('Number of Hits')
plt.ylabel('Average Response Time (ms)')
plt.title('Performance Comparison: All-to-All vs. Linear Topology')
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
```