

Source code and program

The provided source code constitutes a comprehensive Peer-to-Peer (P2P) file sharing system designed to facilitate efficient and reliable file distribution across a decentralized network. The system leverages Java Remote Method Invocation (RMI) to enable seamless communication between peers (LeafNodes) and superpeers (SuperPeers), ensuring robust file management and consistency throughout the network.

```
CS485_PA3/
├── .idea/
│   └── -Force/
├── bin/
├── cached_dir_P01/
├── cached_dir_P02/
├── cached_dir_P03/
├── cached_dir_P04/
├── cached_dir_P05/
├── cached_dir_P06/
├── cached_dir_P07/
├── cached_dir_P08/
├── cached_dir_P09/
├── cached_dir_P10/
├── config/
│   ├── consistency_config.txt
│   ├── peer_config.txt
│   ├── superpeer_config.txt
│   ├── topology.txt
│   └── topology_config.txt
├── lib/
├── master_dir_P01/
├── master_dir_P02/
├── master_dir_P03/
├── master_dir_P04/
├── master_dir_P05/
├── master_dir_P06/
├── master_dir_P07/
├── master_dir_P08/
└── master_dir_P09/
```

- master_dir_P10/
- out/
- output/
- Performance Results/
- src/
 - com.gfiletransfer/
 - AvgRespFileSearch
 - GetPeerDetails
 - GetPeerDetails.class
 - GetSuperPeerDetails
 - GetSuperPeerDetails.class
 - GetTopologyDetails
 - GetTopologyDetails.class
 - LeafNode
 - LeafNodeImpl
 - LeafNodeInterface
 - MultiClient
 - SetupConfig
 - SetupConfig.class
 - SuperPeer
 - SuperPeerImpl
 - SuperPeerImpl.class
 - SuperPeerInterface
- Test_Pull_Mechanism/
- Test_Push_Mechanism/
- .gitignore
- CS485_PA3.iml
- Design Document.pdf
- GnutellaP2P.jar
- output_pull.txt
- output_push.txt
- run_peer.bat
- run_server.bat
- run_test.bat

GetPeerDetails.java

Description:

The `GetPeerDetails` class encapsulates the configuration details for a `LeafNode` (peer) in the P2P system. It includes attributes such as Peer ID, Peer Port, directories for master and cached files, and the associated SuperPeer ID. This class provides getter and setter methods for each attribute, enabling other system components to access and modify these configurations as needed.

```
package com.gfiletransfer;
```

```
/**
 * The GetPeerDetails class encapsulates the configuration details for a LeafNode (peer).
 * It includes peer ID, port number, directories for master and cached files, and the associated
 * SuperPeer.
 */
public class GetPeerDetails {
    private String Peer_ID = null;    // Unique identifier for the peer
    private String Peer_Port = null;  // Port number on which the peer listens
    private String Dir = null;        // Directory path for master files
    private String CacheDir = null;   // Directory path for cached files
    private String SuperPeer = null;  // Associated SuperPeer ID
    /**
     * Retrieves the Peer ID.
     * @return the Peer ID.
     */
    public String getPeer_ID() {
        return Peer_ID;
    }

    /**
     * Sets the Peer ID.
     * @param peer_ID the Peer ID to set.
     */
    public void setPeer_ID(String peer_ID) {
        Peer_ID = peer_ID;
    }

    /**
     * Retrieves the Peer Port number.
     * @return the Peer Port number.
     */
    public String getPeer_Port() {
        return Peer_Port;
    }

    /**
     * Sets the Peer Port number.
     * @param peer_Port the Peer Port number to set.
     */
    public void setPeer_Port(String peer_Port) {
```

```

    Peer_Port = peer_Port;
}

/**
 * Retrieves the Master Directory path.
 * @return the Master Directory path.
 */
public String getDir() {
    return Dir;
}

/**
 * Sets the Master Directory path.
 * @param dir the Master Directory path to set.
 */
public void setDir(String dir) {
    Dir = dir;
}

/**
 * Retrieves the Cache Directory path.
 * @return the Cache Directory path.
 */
public String getCacheDir() {
    return CacheDir;
}

/**
 * Sets the Cache Directory path.
 * @param cacheDir the Cache Directory path to set.
 */
public void setCacheDir(String cacheDir) {
    CacheDir = cacheDir;
}

/**
 * Retrieves the associated SuperPeer ID.
 * @return the SuperPeer ID.
 */
public String getSuperPeer() {
    return SuperPeer;
}

/**
 * Sets the associated SuperPeer ID.
 * @param superPeer the SuperPeer ID to set.
 */
public void setSuperPeer(String superPeer) {
    SuperPeer = superPeer;
}

```

```
}  
  
}
```

GetSuperPeerDetails.java

Description:

The GetSuperPeerDetails class encapsulates the configuration details for a SuperPeer in the P2P system. It includes attributes such as SuperPeer ID, SuperPeer Port, and the IDs of associated LeafNodes. This class provides getter and setter methods for each attribute, enabling other system components to access and modify these configurations as needed.

```
package com.gfiletransfer;
```

```
/**  
 * The GetSuperPeerDetails class encapsulates the configuration details for a SuperPeer.  
 * It includes SuperPeer ID, port number, and associated LeafNode IDs.  
 */  
public class GetSuperPeerDetails {  
  
    private String Peer_ID = null;    // Unique identifier for the SuperPeer  
    private String Peer_Port = null;  // Port number on which the SuperPeer listens  
    private String Leaf_ID = null;    // Comma-separated list of associated LeafNode IDs  
  
    /**  
     * Retrieves the SuperPeer ID.  
     * @return the SuperPeer ID.  
     */  
    public String getPeer_ID() {  
        return Peer_ID;  
    }  
  
    /**  
     * Sets the SuperPeer ID.  
     * @param peer_ID the SuperPeer ID to set.  
     */  
    public void setPeer_ID(String peer_ID) {  
        Peer_ID = peer_ID;  
    }  
  
    /**  
     * Retrieves the SuperPeer Port number.  
     * @return the SuperPeer Port number.  
     */  
    public String getPeer_Port() {  
        return Peer_Port;  
    }  
  
    /**  
     * Sets the SuperPeer Port number.  
     */  
}
```

```

    * @param peer_Port the SuperPeer Port number to set.
    */
    public void setPeer_Port(String peer_Port) {
        Peer_Port = peer_Port;
    }

    /**
     * Retrieves the associated LeafNode IDs.
     * @return the LeafNode IDs.
     */
    public String getLeaf_ID() {
        return Leaf_ID;
    }

    /**
     * Sets the associated LeafNode IDs.
     * @param leaf_ID the LeafNode IDs to set.
     */
    public void setLeaf_ID(String leaf_ID) {
        Leaf_ID = leaf_ID;
    }
}

```

GetTopologyDetails.java

Description:

The GetTopologyDetails class encapsulates the configuration details for network topology within the P2P system. It includes Peer ID, All Neighbors, and Linear Neighbor attributes. This class provides getter and setter methods for each attribute, enabling other system components to access and modify these configurations as needed.

```
package com.gfiletransfer;
```

```

/**
 * The GetTopologyDetails class encapsulates the topology configuration details for SuperPeers.
 * It includes the SuperPeer ID, its immediate linear neighbor, and all its connected neighbors
 * based on the configured topology (All-to-All or Linear).
 */
public class GetTopologyDetails {

```

```

private String Peer_ID = null;    // Unique identifier for the SuperPeer

private String All_Neighbour = null; // Comma-separated list of all connected neighbors in
All-to-All Topology

private String Linear_Neighbour = null; // Immediate neighbor in Linear Topology


/**
 * Retrieves the list of all neighbors for All-to-All Topology.
 * @return the All Neighbour list.
 */
public String getAll_Neighbour() {
    return All_Neighbour;
}


/**
 * Sets the list of all neighbors for All-to-All Topology.
 * @param all_Neighbour the All Neighbour list to set.
 */
public void setAll_Neighbour(String all_Neighbour) {
    All_Neighbour = all_Neighbour;
}


/**
 * Retrieves the immediate linear neighbor for Linear Topology.
 * @return the Linear Neighbour.
 */

```

```
public String getLinear_Neighbour() {  
    return Linear_Neighbour;  
}
```

```
/**
```

```
 * Sets the immediate linear neighbor for Linear Topology.
```

```
 * @param linear_Neighbour the Linear Neighbour to set.
```

```
 */
```

```
public void setLinear_Neighbour(String linear_Neighbour) {  
    Linear_Neighbour = linear_Neighbour;  
}
```

```
/**
```

```
 * Retrieves the SuperPeer ID.
```

```
 * @return the Peer ID.
```

```
 */
```

```
public String getPeer_ID() {  
    return Peer_ID;  
}
```

```
/**
```

```
 * Sets the SuperPeer ID.
```

```
 * @param peer_ID the Peer ID to set.
```

```
 */
```

```
public void setPeer_ID(String peer_ID) {  
    Peer_ID = peer_ID;
```



```
}
```

```
}
```

SetupConfig.java

Description:

The `SetupConfig` class is responsible for loading and parsing all necessary configuration files required to initialize the P2P system. It reads configurations related to peers, super peers, network topology, and consistency mechanisms. By parsing these configurations, the class populates corresponding data structures (`ArrayList<GetPeerDetails>`, `ArrayList<GetSuperPeerDetails>`, and `ArrayList<GetTopologyDetails>`) that are utilized by other components of the system to establish connections and manage file consistency.

```
package com.gfiletransfer;
```

```
import java.io.File;
```

```
import java.io.IOException;
```

```
import java.util.ArrayList;
```

```
import java.util.Arrays;
```

```
import java.util.List;
```

```
import java.util.Scanner;
```

```
/**
```

```
 * The SetupConfig class is responsible for loading and parsing all configuration files
```

```
 * required to initialize the P2P system. It reads configurations related to peers,
```

```
 * super peers, network topology, and consistency mechanisms.
```

```
 */
```

```
public class SetupConfig {
```

```

// Lists to store configuration details

ArrayList<GetPeerDetails> arrPD = new ArrayList<>();    // List of peer configurations

ArrayList<GetSuperPeerDetails> arrSPD = new ArrayList<>(); // List of super peer
configurations

ArrayList<GetTopologyDetails> arrTD = new ArrayList<>(); // List of topology
configurations

String topology = null;                                // Type of network topology (e.g., all, linear)

String consisApp = null;                                // Consistency mechanism (e.g., PUSH, PULL1,
PULL2)

/**
 * Constructs a SetupConfig object and loads all configuration files.
 *
 * @throws IOException if any configuration file cannot be read.
 */
public SetupConfig() throws IOException {

    // Peer configuration

    File peerConfig = new File("config/peer_config.txt");
    loadPeerConfig(peerConfig);

    // SuperPeer configuration

    File superPeerConfig = new File("config/superpeer_config.txt");
    loadSuperPeerConfig(superPeerConfig);

    // Topology details configuration

    File topologyDetailsConfig = new File("config/topology_config.txt");
    loadTopologyDetailsConfig(topologyDetailsConfig);

```

```

// Topology type
File topologyFile = new File("config/topology.txt");
loadTopology(topologyFile);

// Consistency mechanism
File consistencyConfigFile = new File("config/consistency_config.txt");
loadConsistencyMechanism(consistencyConfigFile);
}

/**
 * Loads the peer configuration file.
 *
 * @param file The peer configuration file.
 * @throws IOException if the file cannot be read.
 */
private void loadPeerConfig(File file) throws IOException {
    try (Scanner scanner = new Scanner(file)) {
        while (scanner.hasNextLine()) {
            String line = scanner.nextLine();

            if (line.trim().isEmpty() || line.startsWith("#")) continue; // Skip empty or comment
lines
            List<String> tokens = Arrays.asList(line.split("\\s*,\\s*")); // Split by comma and trim
spaces
            if (tokens.size() != 5) {
                System.out.println("Invalid line in peer_config.txt: " + line);
            }
        }
    }
}

```

```

        continue; // Skip invalid lines
    }

    GetPeerDetails pd = new GetPeerDetails();
    pd.setPeer_ID(tokens.get(0));    // Set Peer ID
    pd.setPeer_Port(tokens.get(1));  // Set Peer Port
    pd.setDir(tokens.get(2));        // Set Master Directory
    pd.setCacheDir(tokens.get(3));   // Set Cached Directory
    pd.setSuperPeer(tokens.get(4));  // Set Associated SuperPeer
    arrPD.add(pd);                   // Add to peer configurations list
    }
}

System.out.println("Loaded " + arrPD.size() + " peer configurations.");
}

/**
 * Loads the superpeer configuration file.
 *
 * @param file The superpeer configuration file.
 * @throws IOException if the file cannot be read.
 */
private void loadSuperPeerConfig(File file) throws IOException {
    try (Scanner scanner = new Scanner(file)) {
        while (scanner.hasNextLine()) {
            String line = scanner.nextLine();

```

```
lines    if (line.trim().isEmpty() || line.startsWith("#")) continue; // Skip empty or comment
```

```
spaces   List<String> tokens = Arrays.asList(line.split("\\s*,\\s*")); // Split by comma and trim
```

```
    if (tokens.size() < 3) {  
        System.out.println("Invalid line in superpeer_config.txt: " + line);  
        continue; // Skip invalid lines  
    }
```

```
    GetSuperPeerDetails spd = new GetSuperPeerDetails();  
  
    spd.setPeer_ID(tokens.get(0));           // Set SuperPeer ID  
  
    spd.setPeer_Port(tokens.get(1));         // Set SuperPeer Port  
  
    spd.setLeaf_ID(String.join(",", tokens.subList(2, tokens.size()))); // Set Associated  
LeafNode IDs
```

```
    arrSPD.add(spd);                         // Add to superpeer configurations list  
}
```

```
}
```

```
System.out.println("Loaded " + arrSPD.size() + " superpeer configurations.");
```

```
}
```

```
/**
```

```
* Loads the topology details configuration file.
```

```
*
```

```
* @param file The topology details configuration file.
```

```
* @throws IOException if the file cannot be read.
```

```
*/
```

```

private void loadTopologyDetailsConfig(File file) throws IOException {
    try (Scanner scanner = new Scanner(file)) {
        while (scanner.hasNextLine()) {
            String line = scanner.nextLine().trim(); // Trim to handle leading/trailing spaces
            System.out.println("Raw Line: " + line); // Print out raw line for debugging

            if (line.isEmpty() || line.startsWith("#")) continue; // Skip empty or comment lines

            List<String> tokens = Arrays.asList(line.split("\\s*;\\s*")); // Split by semicolon and trim
spaces
            System.out.println("Tokens: " + tokens); // Print tokens for debugging

            if (tokens.size() != 3) {
                System.out.println("Invalid line in topology_config.txt: " + line);
                continue; // Skip invalid lines
            }

            GetTopologyDetails td = new GetTopologyDetails();
            td.setPeer_ID(tokens.get(0)); // Set SuperPeer ID
            td.setLinear_Neighbour(tokens.get(1)); // Set Linear Neighbor
            td.setAll_Neighbour(tokens.get(2)); // Set All Neighbors

            arrTD.add(td); // Add to topology configurations list
        }
    }

    System.out.println("Loaded " + arrTD.size() + " topology configurations.");
}

```

```

}

/**
 * Loads the topology type configuration file.
 *
 * @param file The topology type configuration file.
 * @throws IOException if the file cannot be read.
 */
private void loadTopology(File file) throws IOException {
    try (Scanner scanner = new Scanner(file)) {
        while (scanner.hasNextLine()) {
            String line = scanner.nextLine();

            if (line.trim().isEmpty() || line.startsWith("#")) continue; // Skip empty or comment
lines
            topology = line.trim(); // Set topology type
        }
    }

    System.out.println("Topology type: " + topology);
}

/**
 * Loads the consistency mechanism configuration file.
 *
 * @param file The consistency mechanism configuration file.
 * @throws IOException if the file cannot be read.

```

```

*/
private void loadConsistencyMechanism(File file) throws IOException {
    try (Scanner scanner = new Scanner(file)) {
        while (scanner.hasNextLine()) {
            String line = scanner.nextLine();
            if (line.trim().isEmpty() || line.startsWith("#")) continue; // Skip empty or comment
lines
            consisApp = line.trim(); // Set consistency mechanism
        }
    }
    System.out.println("Consistency mechanism: " + consisApp);
}
}

```

SuperPeer.java

Description:

The `SuperPeer` class serves as the entry point for initializing and running a SuperPeer in the P2P network. Upon execution, it prompts the user to input the SuperPeer ID, retrieves the corresponding port number from the configuration, sets up the RMI registry on the specified port, and binds the `SuperPeerImpl` implementation to the registry. This setup enables the SuperPeer to handle remote method invocations from LeafNodes and other SuperPeers, facilitating file indexing, query processing, and consistency management within the network.

```

package com.gfiletransfer;

import java.io.IOException;

import java.rmi.RemoteException;

import java.rmi.registry.LocateRegistry;

import java.rmi.registry.Registry;

```



```

import java.rmi.server.UnicastRemoteObject;

import java.util.Scanner;

/**
 * The SuperPeer class initializes and starts a SuperPeer in the P2P network.
 * It sets up the RMI registry, binds the SuperPeer implementation, and makes
 * the SuperPeer available for remote method invocations.
 */
public class SuperPeer {

    /**
     * The main method serves as the entry point for the SuperPeer application.
     *
     * @param args Command-line arguments (not used).
     * @throws RemoteException if there is an error during RMI operations.
     */
    public static void main(String[] args) throws RemoteException {

        Scanner sc = new Scanner(System.in); // Scanner for user input

        String peerID = null;

        String portNum = null;

        System.out.println("Enter the Super Peer ID.");

        peerID = sc.nextLine(); // Prompt user to enter SuperPeer ID

        // Reading Port details from property file for Instantiating Super Peer

        SetupConfig scg;

```

```

try {
    scg = new SetupConfig(); // Initialize SetupConfig to load configurations
    // Getting Calling Super Peer Port number
    for (GetSuperPeerDetails sp : scg.arrSPD) {
        if (sp.getPeer_ID().equalsIgnoreCase(peerID)) {
            portNum = sp.getPeer_Port(); // Retrieve port number for the given SuperPeer ID
            break;
        }
    }
} catch (IOException e1) {
    System.out.println("IOException occurred while reading the property file at SuperPeer
Initialization.");
    e1.printStackTrace();
    sc.close(); // Close the scanner before exiting
    return; // Exit the program if configuration loading fails
}

// Initialize RMI registry on the specified port
Registry registry = LocateRegistry.createRegistry(Integer.parseInt(portNum));

// Instantiate the SuperPeer implementation
SuperPeerImpl splImpl = new SuperPeerImpl();

// Export the SuperPeerImpl object to receive remote method invocations
SuperPeerInterface splInter = (SuperPeerInterface)
UnicastRemoteObject.exportObject(splImpl, 0);

```

```

// Bind the SuperPeer implementation to the RMI registry with a unique name
registry.rebind("root://SuperPeer/" + portNum, splInter);

System.out.println("SuperPeer Server is now up and running on port " + portNum + ".");

sc.close(); // Close the scanner as it's no longer needed
}

}

```

SuperPeerInterface.java

Description:

The `SuperPeerInterface` defines the remote methods that a `SuperPeer` must implement to facilitate communication within the P2P network. These methods include registering files, searching for files, handling queries, managing consistency mechanisms (both push and pull-based), broadcasting updates and invalidations, and polling for file version status. By extending `java.rmi.Remote`, this interface allows `SuperPeers` to expose these methods for remote invocation via Java RMI.

```

package com.gfiletransfer;

import java.rmi.NotBoundException;

import java.rmi.Remote;

import java.rmi.RemoteException;

import java.text.DateFormat;

import java.util.ArrayList;

import java.util.Collection;

/**

```

* The SuperPeerInterface defines the remote methods that a SuperPeer must implement.

* These methods facilitate file registration, searching, query handling, and consistency management

* using both push and pull mechanisms within the P2P network.

*/

public interface SuperPeerInterface extends Remote {

/**

* Registers a file with the SuperPeer. Handles actions such as "new", "delete", and "update".

*

* @param rd The action to perform ("new", "del", or "upd").

* @param filename The name of the file.

* @param peerid The ID of the peer registering the file.

* @param port_num The port number of the peer.

* @param directory The directory where the file is stored.

* @param sPeer The ID of the associated SuperPeer.

* @param copyType The type of copy ("Master" or "Replica").

* @param versionNum The version number of the file.

* @param status The status of the file ("valid" or "invalid").

* @param lastModTime The last modified time of the file.

* @param TTR Time-To-Refresh value for pull-based consistency.

* @param ogPeerId The original peer ID where the file originated.

* @throws RemoteException if a remote communication error occurs.

*/

public void registryFiles(

 String rd,

```

        String filename,
        String peerid,
        String port_num,
        String directory,
        String sPeer,
        String copyType,
        String versionNum,
        String status,
        String lastModTime,
        String TTR,
        String ogPeerId
    ) throws RemoteException;

/**
 * Searches for a file by its name within the SuperPeer's index.
 *
 * @param filename The name of the file to search for.
 * @return A collection of file details matching the search criteria.
 * @throws RemoteException if a remote communication error occurs.
 */
public Collection<ArrayList<String>> searchFile(String filename) throws RemoteException;

/**
 * Sends a query to the network to search for a specific file.
 *
 * @param msgId The unique message ID for the query.

```

- * @param TTL Time-To-Live for the query message.
- * @param filename The name of the file being searched.
- * @param reqPeerId The ID of the requesting peer.
- * @param reqPortNum The port number of the requesting peer.
- * @throws RemoteException if a remote communication error occurs.
- */

```
public void query(
    String msgId,
    int TTL,
    String filename,
    String reqPeerId,
    String reqPortNum
) throws RemoteException;
```

```
/**
 * Returns the result of a query, providing file information to the requesting peer.
 *
 * @param msgId      The unique message ID for the query.
 * @param TTL      Time-To-Live after the query.
 * @param filename   The name of the file searched for.
 * @param resultArr   The search results containing file details.
 * @throws RemoteException if a remote communication error occurs.
 */
```

```
public void queryHit(
    String msgId,
    int TTL,
```

```

        String filename,

        Collection<ArrayList<String>> resultArr

    ) throws RemoteException;

    /*----- start change -----*/

    /**
     * Broadcasts the status of a file update to other SuperPeers in the network.
     *
     * @param msgId      The unique message ID for the broadcast.
     * @param filename   The name of the file being updated.
     * @param originLNServer The originating LeafNode server ID.
     * @param verNum     The version number of the file after the update.
     * @throws RemoteException if a remote communication error occurs.
     */
    public void broadCastSP(

        String msgId,

        String filename,

        String originLNServer,

        String verNum

    ) throws RemoteException;

    /**
     * Broadcasts a file invalidation message to other SuperPeers in the network.
     *
     * @param msgId      The unique message ID for the broadcast.

```

- * @param filename The name of the file being invalidated.
- * @param originLNServer The originating LeafNode server ID.
- * @param verNum The version number of the file being invalidated.
- * @throws RemoteException if a remote communication error occurs.
- */

```
public void broadCastSS(
    String msgId,
    String filename,
    String originLNServer,
    String verNum
) throws RemoteException;
```

```
/**
 * Polls the SuperPeer to check the status and version number of a specific file.
 *
 * @param filename The name of the file to check.
 * @param verNum The version number of the cached file.
 * @param ogPeerId The original peer ID where the file originated.
 * @return A string indicating the status of the file ("Proper version" or "File out of Date").
 * @throws RemoteException if a remote communication error occurs.
 */
```

```
public String poll(
    String filename,
    String verNum,
    String ogPeerId
) throws RemoteException;
```



```
/**
```

```
* Retrieves the version number of a file from a specific peer and port.
```

```
*
```

```
* @param filename The name of the file.
```

```
* @param peerid The ID of the peer holding the file.
```

```
* @param port_num The port number of the peer.
```

```
* @param copyType The type of copy ("Master" or "Replica").
```

```
* @param ogPeerId The original peer ID where the file originated.
```

```
* @return The version number of the specified file.
```

```
* @throws RemoteException if a remote communication error occurs.
```

```
*/
```

```
public String getVersionNum(
```

```
    String filename,
```

```
    String peerid,
```

```
    String port_num,
```

```
    String copyType,
```

```
    String ogPeerId
```

```
) throws RemoteException;
```

```
/**
```

```
* Notifies the SuperPeer of a polling request and handles version checking.
```

```
*
```

```
* @param filename The name of the file being polled.
```

```
* @param reqPeerid The ID of the requesting peer.
```

```
* @param reqPort_num The port number of the requesting peer.
```

- * @param copyType The type of copy ("Master" or "Replica").
- * @param verNum The version number of the cached file.
- * @param ogPeerId The original peer ID where the file originated.
- * @param TTR Time-To-Refresh value for pull-based consistency.
- * @param lastModTime The last modified time of the file.
- * @throws RemoteException if a remote communication error occurs.
- * @throws NotBoundException if a remote object is not bound in the registry.
- */

```
public void notifyPoll(
    String filename,
    String reqPeerid,
    String reqPort_num,
    String copyType,
    String verNum,
    String ogPeerId,
    String TTR,
    String lastModTime
) throws RemoteException, NotBoundException;
```

```
/*----- end change -----*/
```

```
}
```

SuperPeerImpl.java

Description:

The `SuperPeerImpl` class implements the `SuperPeerInterface` and provides the core functionalities of a SuperPeer in the P2P network. It manages file indexing, handles search queries, processes file registrations (addition, deletion, updates), and ensures file consistency using both push-based and pull-based mechanisms. The class utilizes Java RMI for remote method invocations, enabling seamless communication between SuperPeers and LeafNodes. Additionally, it incorporates scheduled tasks to handle polling for file version updates and manages broadcasting of file status changes across the network.

```
package com.gfiletransfer;

import java.io.IOException;

import java.rmi.NotBoundException;

import java.rmi.RemoteException;

import java.rmi.registry.LocateRegistry;

import java.rmi.registry.Registry;

import java.text.DateFormat;

import java.text.SimpleDateFormat;

import javax.ws.rs.core.MultivaluedHashMap;

import javax.ws.rs.core.MultivaluedMap;

import java.util.*;

import java.util.Map.Entry;

import java.util.concurrent.Executors;

import java.util.concurrent.ScheduledExecutorService;

import java.util.concurrent.ScheduledFuture;

import java.util.concurrent.ScheduledThreadPoolExecutor;

import java.util.concurrent.TimeUnit;

/**
```

- * The SuperPeerImpl class implements the SuperPeerInterface and provides the core functionalities
- * of a SuperPeer in the P2P network. It manages file indexing, handles search queries, processes
- * file registrations (addition, deletion, updates), and ensures file consistency using both
- * push-based and pull-based mechanisms.
- */

```
public class SuperPeerImpl implements SuperPeerInterface {

    DateFormat dateFormat = new SimpleDateFormat("yyyy/MM/dd HH:mm:ss");

    // Get current date time with Date()

    Date date = new Date();

    // Defining a multivalued hash map for indexing the file details

    // Using MultivaluedHashMap to store multiple entries for a single filename.

    // For each filename (Key), there will be a collection of entries (Value).

    private MultivaluedMap<String, ArrayList<String>> fileDictionary = new
MultivaluedHashMap<>();

    // Buffer for storing the requests

    private Map<String, ArrayList<String>> myMap = new HashMap<String, ArrayList<String>>();

    private String supPeerId = null;

    /*----- start change -----*/

    // Static maps to manage flags and statuses for file consistency

    public static Map<String, String> flagTable = new HashMap<String, String>();

    public static Map<String, String> status1Table = new HashMap<String, String>();

    // ScheduledExecutorService for handling scheduled tasks

    final static ScheduledExecutorService scheduler = Executors.newScheduledThreadPool(1);
```

```
static ScheduledThreadPoolExecutor executor = new ScheduledThreadPoolExecutor(15); //
Thread pool for scheduled tasks
```

```
static ScheduledFuture<?> t; // Reference to a scheduled task
```

```
/*----- end change -----*/
```

```
/**
```

```
 * Registers, deletes, or updates files in the SuperPeer's index based on the action specified.
```

```
 *
```

```
 * @param rd      The action to perform ("new", "del", or "upd").
```

```
 * @param filename The name of the file.
```

```
 * @param peerid   The ID of the peer performing the action.
```

```
 * @param port_num The port number of the peer.
```

```
 * @param directory The directory where the file is stored.
```

```
 * @param sPeer    The SuperPeer ID associated with the peer.
```

```
 * @param copyType The type of copy ("Master" or "Replica").
```

```
 * @param versionNum The version number of the file.
```

```
 * @param status    The status of the file ("valid" or "invalid").
```

```
 * @param lastModTime The last modified time of the file.
```

```
 * @param TTR       Time-To-Refresh value for pull-based consistency.
```

```
 * @param ogPeerId  The original peer ID where the file originated.
```

```
 * @throws RemoteException if a remote communication error occurs.
```

```
 */
```

```
/*----- start change -----*/
```

```
@Override
```

```
public void registryFiles(String rd, String filename, String peerid, String port_num, String
directory,
```

String sPeer, String copyType, String versionNum, String status, String lastModTime, String TTR, String ogPeerId)

throws RemoteException {

/*----- end change -----*/

// Handle "new" file registration

if (rd.equalsIgnoreCase("new")) {

// Check for duplicate record in index

if (this.fileDictionary.containsKey(filename)) {

Collection<ArrayList<String>> existingFiles = this.fileDictionary.get(filename);

for (ArrayList<String> entry : existingFiles) {

if (entry.get(1).equalsIgnoreCase(peerid)) {

// Duplicate record found; reject registration

return;

}

}

}

// If no duplicate, add new file entry

ArrayList<String> arrFileDtl = new ArrayList<String>();

arrFileDtl.add(filename); // 0 - filename

arrFileDtl.add(peerid); // 1 - peer ID

arrFileDtl.add(port_num); // 2 - port number

arrFileDtl.add(directory); // 3 - directory

arrFileDtl.add(sPeer); // 4 - SuperPeer ID

arrFileDtl.add(copyType); // 5 - copy type

arrFileDtl.add(versionNum); // 6 - version number

arrFileDtl.add(status); // 7 - status

```

arrFileDtl.add(lastModTime); // 8 - last modified time

arrFileDtl.add(TTR); // 9 - Time-To-Refresh

arrFileDtl.add(ogPeerId); // 10 - original peer ID

this.fileDictionary.add(filename, arrFileDtl);

this.supPeerId = sPeer;
}

// Handle "del" (delete) file action
else if (rd.equalsIgnoreCase("del")) {

    Collection<ArrayList<String>> delArrFile = new ArrayList<ArrayList<String>>();

    Collection<ArrayList<String>> updatedEntries = new ArrayList<ArrayList<String>>();

    // Check if the file exists in the index
    if (this.fileDictionary.containsKey(filename)) {

        delArrFile = this.fileDictionary.get(filename);

        for (ArrayList<String> entry : delArrFile) {

            // Remove the specific peer's entry
            if (entry.get(1).equalsIgnoreCase(peerid)) {

                updatedEntries = this.fileDictionary.remove(filename);

                updatedEntries.remove(entry);

                for (ArrayList<String> remainingEntry : updatedEntries) {

                    this.fileDictionary.add(filename, remainingEntry);

                }

                System.out.println("Index Server Updated & Specified Record Deleted");

            }

        }

    } else {

```

```

        System.out.println("Delete Request: No entry detected for filename");
    }
}

/*----- start change -----*/

// Handle "upd" (update) file action
else if (rd.equalsIgnoreCase("upd")) {

    Collection<ArrayList<String>> updArrFile = new ArrayList<ArrayList<String>>();
    Collection<ArrayList<String>> updatedEntries = new ArrayList<ArrayList<String>>();

    // Check if the file exists in the index
    if (this.fileDictionary.containsKey(filename)) {
        updArrFile = this.fileDictionary.get(filename);
        for (ArrayList<String> entry : updArrFile) {
            // Update the specific peer's entry
            if (entry.get(1).equalsIgnoreCase(peerid)) {
                ArrayList<String> updatedEntry = entry;
                updatedEntry.set(6, versionNum);    // Update version number
                updatedEntry.set(8, lastModTime);    // Update last modified time
                updatedEntry.set(7, status);        // Update status

                updatedEntries = this.fileDictionary.remove(filename);
                updatedEntries.remove(entry);
                updatedEntries.add(updatedEntry);

                for (ArrayList<String> remainingEntry : updatedEntries) {
                    this.fileDictionary.add(filename, remainingEntry);
                }
            }
        }
    }
}

```



```

        }
        break;
    }
}

System.out.println("Index Server Updated & Specified Record Updated");
} else {
    System.out.println("Update Request: No entry detected for filename under requested
Peer");
}
}

/*----- end change -----*/
else {
    System.out.println("Invalid Request.");
}

// Display the updated index after every addition or removal
System.out.println("#####");
System.out.println("THE UPDATED INDEX at " + dateFormat.format(date));
for (Entry<String, List<ArrayList<String>>> entry : this.fileDictionary.entrySet()) {
    System.out.println(entry.getKey() + " => " + entry.getValue());
}
System.out.println("#####");
}

/**
 * Searches for a specified filename in the SuperPeer's index.

```

```

*

* @param filename The name of the file to search for.

* @return A collection of ArrayLists containing file details if found; otherwise, an empty
collection.

* @throws RemoteException if a remote communication error occurs.

*/

@Override

public synchronized Collection<ArrayList<String>> searchFile(String filename) throws
RemoteException {

    Collection<ArrayList<String>> resultArrFile = new ArrayList<ArrayList<String>>();

    if (this.fileDictionary.containsKey(filename)) {

        resultArrFile = this.fileDictionary.get(filename);

    }

    return resultArrFile;

}

/**

* Handles incoming search queries from peers. It decrements the TTL, stores the request,

* searches for the file locally, and forwards the query to neighboring SuperPeers based on
the topology.

*

* @param msgId    The unique message ID for the query.

* @param TTL      Time-To-Live for the query message.

* @param filename The name of the file being searched.

* @param reqPeerId The ID of the requesting peer.

* @param reqPortNum The port number of the requesting peer.

* @throws RemoteException if a remote communication error occurs.

```

```
*/
```

```
@Override
```

```
public void query(String msgId, int TTL, String filename, String reqPeerId, String reqPortNum)
```

```
    throws RemoteException {
```

```
    if (TTL > 0 && TTL != 0) {
```

```
        TTL = TTL - 1;
```

```
        // Insert request details into the map
```

```
        ArrayList<String> upStreamDtl = new ArrayList<String>();
```

```
        upStreamDtl.add(msgId);
```

```
        upStreamDtl.add(Integer.toString(TTL));
```

```
        upStreamDtl.add(reqPeerId);
```

```
        upStreamDtl.add(reqPortNum);
```

```
        this.myMap.put(msgId, upStreamDtl);
```

```
        // Display all the request details
```

```
        for (Entry<String, ArrayList<String>> entry : this.myMap.entrySet()) {
```

```
            System.out.println(entry.getKey() + " => " + entry.getValue());
```

```
        }
```

```
        // Search for the requested file locally
```

```
        Collection<ArrayList<String>> resultLocal = this.searchFile(filename);
```

```
        if (!resultLocal.isEmpty()) {
```

```
            try {
```

```
                // Locate the registry of the requesting SuperPeer or LeafNode
```

```
                Registry regis = LocateRegistry.getRegistry("localhost",
```

```
                    Integer.parseInt(this.myMap.get(msgId).get(3)));
```

```

String ref = msgId.substring(0, msgId.indexOf(":")); // Extract PeerId from msgId

// If the caller is the originating peer
if (ref.equalsIgnoreCase(reqPeerId)) {
    // Lookup the LeafNode interface
    LeafNodeInterface pInter = (LeafNodeInterface) regis.lookup("root://LeafNode/" +
        this.myMap.get(msgId).get(3) + "/FS");

    // Send the query hit response to the LeafNode
    if (pInter.queryHit(msgId, TTL, filename, resultLocal)) {
        System.out.println("Output Sent to Leaf Node");
    } else {
        System.out.println("An exception might have occurred at Leaf Node or TTL
expired.");
    }
} else {
    System.out.println("Calling Super Peer Caller");

    // Lookup the SuperPeer interface
    SuperPeerInterface spInter = (SuperPeerInterface)
regis.lookup("root://SuperPeer/" +
        this.myMap.get(msgId).get(3));

    // Forward the query hit to the SuperPeer
    spInter.queryHit(msgId, TTL, filename, resultLocal);
}
} catch (Exception e) {
    System.out.println("Exception at its own SuperPeer query function: " +
e.getMessage());
}

```

```

    }
} else {
    System.out.println("FOUND NOTHING in this SuperPeer");
}

// Retrieve local and remote SuperPeer port numbers
String remoteSupPeerPortNum = null;
String localSupPeerPortNum = null;

// Load configuration
SetupConfig sc;
try {
    sc = new SetupConfig();
    // Get the local SuperPeer's port number
    for (GetSuperPeerDetails sp : sc.arrSPD) {
        if (sp.getPeer_ID().equalsIgnoreCase(this.supPeerId)) {
            localSupPeerPortNum = sp.getPeer_Port();
            break;
        }
    }
}

String callingLeafId = msgId.substring(0, msgId.indexOf(":"));

if (sc.topology.equalsIgnoreCase("ALL")) {
    System.out.println("WORKING IN ALL TO ALL TOPOLOGY");
    if (callingLeafId.equalsIgnoreCase(this.myMap.get(msgId).get(2))) {
        for (GetTopologyDetails topo : sc.arrTD) {

```

```

        if (topo.getPeer_ID().equalsIgnoreCase(this.supPeerId)) {

            List<String> neighbourArr =
Arrays.asList(topo.getAll_Neighbour().split("\\s*,\\s*"));

            System.out.println("Total Number of Neighbours in ALL TOPOLOGY: " +
neighbourArr.size());

            // Forward the query to all neighbors in All-to-All Topology
            for (String spName : neighbourArr) {

                // Get the port number of each neighboring SuperPeer
                for (GetSuperPeerDetails speer : sc.arrSPD) {

                    if (speer.getPeer_ID().equalsIgnoreCase(spName)) {

                        remoteSupPeerPortNum = speer.getPeer_Port();

                        break;

                    }

                }

                // Invoke the query method on the neighboring SuperPeer
                try {

                    Registry regis = LocateRegistry.getRegistry("localhost",

                        Integer.parseInt(remoteSupPeerPortNum));

                    SuperPeerInterface spInter = (SuperPeerInterface)
regis.lookup("root://SuperPeer/" +

                        remoteSupPeerPortNum);

                    System.out.println("Calling Neighbour " + spName + " query()");

                    spInter.query(msgId, TTL, filename, this.supPeerId,
localSupPeerPortNum);

                } catch (Exception e) {

                    System.out.println("Exception occurred while calling Neighbour Query.
Neighbour is: " + spName);

```

```

        }
    }
    break;
} else {
    System.out.println("Did not find SuperPeer info in Config file object.");
}
}
} else {
    System.out.println("No Need of broadcasting query messages to all Super Peers.");
}
} else {
    System.out.println("WORKING IN LINEAR TOPOLOGY");
    List<String> leafPeerIdArr = null;
    for (GetTopologyDetails topo : sc.arrTD) {
        if (topo.getPeer_ID().equalsIgnoreCase(this.supPeerId)) {
            String neighbour = topo.getLinear_Neighbour();
            System.out.println("SuperPeer " + this.supPeerId + " has Neighbour in Linear
TOPOLOGY: " + neighbour);

            // Get the port number of the neighboring SuperPeer
            String spName = neighbour;
            for (GetSuperPeerDetails speer : sc.arrSPD) {
                if (speer.getPeer_ID().equalsIgnoreCase(spName)) {
                    remoteSupPeerPortNum = speer.getPeer_Port();
                    leafPeerIdArr = Arrays.asList(speer.getLeaf_ID().split("\\s*,\\s*"));
                    break;
                }
            }
        }
    }
}
}

```

```

    }
}

// Forward the query only if the calling LeafNode is not managed by the
neighbor

if (!leafPeerIdArr.contains(callingLeafId)) {
    try {
        Registry regis = LocateRegistry.getRegistry("localhost",
            Integer.parseInt(remoteSupPeerPortNum));

        SuperPeerInterface spInter = (SuperPeerInterface)
regis.lookup("root://SuperPeer/" +
            remoteSupPeerPortNum);

        System.out.println("Calling Neighbour " + spName + " query()");
        spInter.query(msgId, TTL, filename, this.supPeerId, localSupPeerPortNum);
    } catch (Exception e) {
        System.out.println("Exception occurred while calling Neighbour Query.
Neighbour is: " + spName);
    }
} else {
    System.out.println("No Need of forwarding query messages to Super Peers.");
}

break;
} else {
    System.out.println("Did not find SuperPeer info in Config file object.");
}
}
}

```



```

        } catch (IOException e1) {

            System.out.println("IOException occurred while reading the property file in SuperPeer
Query.");

        }

    }

/**
 * Handles the response to a search query by sending the results back to the requesting peer.
 *
 * @param msgId    The unique message ID for the query.
 * @param TTL      Time-To-Live remaining for the query message.
 * @param filename The name of the file searched for.
 * @param resultArr The search results containing file details.
 * @throws RemoteException if a remote communication error occurs.
 */
@Override

    public synchronized void queryHit(String msgId, int TTL, String filename,
Collection<ArrayList<String>> resultArr)

        throws RemoteException {

        if (TTL > 0 && TTL != 0) {

            try {

                TTL = TTL - 1;

                Registry regis = LocateRegistry.getRegistry("localhost",

                    Integer.parseInt(this.myMap.get(msgId).get(3)));

                String ref = msgId.substring(0, msgId.indexOf(":"));

```

```

// Check if the originating peer has received the query hit
if (ref.equalsIgnoreCase(this.myMap.get(msgId).get(2))) {
    // Lookup the LeafNode interface
    LeafNodeInterface pInter = (LeafNodeInterface) regis.lookup("root://LeafNode/" +
        this.myMap.get(msgId).get(3) + "/FS");

    // Send the query hit response to the LeafNode
    if (pInter.queryHit(msgId, TTL, filename, resultArr)) {
        System.out.println("Output Sent to Leaf Node");
    } else {
        System.out.println("An exception might have occurred at Leaf Node or TTL
expired.");
    }
} else {
    // Lookup the SuperPeer interface and forward the query hit
    SuperPeerInterface spInter = (SuperPeerInterface) regis.lookup("root://SuperPeer/"
+
        this.myMap.get(msgId).get(3));
    spInter.queryHit(msgId, TTL, filename, resultArr);
}
} catch (Exception e) {
    System.out.println("Exception at Remote SuperPeer queryHit function: " +
e.getMessage());
}
} else {
    System.out.println("Time to Live of a Message has expired at remote SuperNode. This
Message is no longer valid.");
}
}

```

```

}

/*----- start change -----*/

/**
 * Broadcasts the status of a file update to other SuperPeers in the network.
 * This method invalidates the cached copies of the file in all LeafNodes except the origin
 * LeafNode.
 *
 * @param msgId      The unique message ID for the broadcast.
 * @param filename    The name of the file being updated.
 * @param originLNServer The originating LeafNode server ID.
 * @param verNum      The version number of the file after the update.
 * @throws RemoteException if a remote communication error occurs.
 */
@Override
public void broadCastSP(String msgId, String filename, String originLNServer, String verNum)
    throws RemoteException {

    String remoteSupPeerPortNum = null;

    // Load configuration
    SetupConfig sc;

    try {
        sc = new SetupConfig();

        // Retrieve the list of LeafNodes managed by this SuperPeer
        List<String> apd = new ArrayList<String>();

        for (GetSuperPeerDetails sp : sc.arrSPD) {

```

```

        if (sp.getPeer_ID().equalsIgnoreCase(this.supPeerId)) {

            String pd = sp.getLeaf_ID();

            apd = Arrays.asList(pd.split(", "));

            break;

        }

    }

```

```

    System.out.println("BEFORE Leaf nodes under this super peer are: " + apd + " ORIGIN
SERVER: " + originLNServer);

```

```

// Remove the origin LeafNode from the list to avoid invalidating it
List<String> apdUpdated = new ArrayList<String>();

for (String s : apd) {

    if (s.equalsIgnoreCase(originLNServer)) {

        continue;

    }

    apdUpdated.add(s);

}

apd = apdUpdated;

System.out.println("AFTER Leaf nodes under this super peer are: " + apd);

```

```

// Invalidate the specified file in all associated LeafNodes except the origin
for (String ls : apd) {

    for (GetPeerDetails p : sc.arrPD) {

        if (p.getPeer_ID().equalsIgnoreCase(ls)) {

            String Port_No = p.getPeer_Port();

            Registry regis = LocateRegistry.getRegistry("localhost", Integer.parseInt(Port_No));

```

```

LeafNodeInterface pInter = (LeafNodeInterface) regis.lookup("root://LeafNode/" +
    Integer.parseInt(Port_No) + "/FS");
pInter.invalidate(filename, originLNServer, verNum);
break;
}
}
}

```

```

// Broadcast the invalidation to all neighboring SuperPeers based on the topology
for (GetTopologyDetails topo : sc.arrTD) {
    if (topo.getPeer_ID().equalsIgnoreCase(this.supPeerId)) {
        List<String> neighbourArr = Arrays.asList(topo.getAll_Neighbour().split("\\s*,\\s*"));

        // Forward the invalidation to all neighbors in All-to-All Topology
        for (String spName : neighbourArr) {
            // Retrieve the port number of the neighboring SuperPeer
            for (GetSuperPeerDetails speer : sc.arrSPD) {
                if (speer.getPeer_ID().equalsIgnoreCase(spName)) {
                    remoteSupPeerPortNum = speer.getPeer_Port();
                    break;
                }
            }
        }

        // Invoke the broadCastSS method on the neighboring SuperPeer
        try {
            Registry regis = LocateRegistry.getRegistry("localhost",
                Integer.parseInt(remoteSupPeerPortNum));

```

```

        SuperPeerInterface spInter = (SuperPeerInterface)
regist.lookup("root://SuperPeer/" +

        remoteSupPeerPortNum);

        System.out.println("Calling Neighbour " + spName + " broadCastSS()");

        spInter.broadCastSS(msgId, filename, originLNServer, verNum);

    } catch (Exception e) {

        System.out.println("Exception occurred while calling Neighbour Query.
Neighbour is: " + spName);

    }

    }

    break;

    }

    }

    } catch (Exception e1) {

        System.out.println("IOException occurred while reading the property file in SuperPeer
BroadcastSP function.");

    }

}

```

/**

- * Broadcasts a file invalidation message to other SuperPeers in the network.
- * This method invalidates the cached copies of the specified file in all associated LeafNodes.
- *
- * @param msgId The unique message ID for the broadcast.
- * @param filename The name of the file being invalidated.
- * @param originLNServer The originating LeafNode server ID.

* @param verNum The version number of the file being invalidated.

* @throws RemoteException if a remote communication error occurs.

*/

@Override

```
public void broadCastSS(String msgId, String filename, String originLNServer, String verNum)
    throws RemoteException {
    // Create an instance to use the config file
    SetupConfig sc;
    try {
        sc = new SetupConfig();
        // Retrieve the list of LeafNodes managed by this SuperPeer
        List<String> apd = new ArrayList<String>();
        for (GetSuperPeerDetails sp : sc.arrSPD) {
            if (sp.getPeer_ID().equalsIgnoreCase(this.supPeerId)) {
                String pd = sp.getLeaf_ID();
                apd = Arrays.asList(pd.split(","));
                break;
            }
        }
        // Invalidate the specified file in all associated LeafNodes
        for (String ls : apd) {
            for (GetPeerDetails p : sc.arrPD) {
                if (p.getPeer_ID().equalsIgnoreCase(ls)) {
                    String Port_No = p.getPeer_Port();
                    Registry regis = LocateRegistry.getRegistry("localhost", Integer.parseInt(Port_No));
                    LeafNodeInterface pInter = (LeafNodeInterface) regis.lookup("root://LeafNode/" +
```

```

        Integer.parseInt(Port_No) + "/FS");

    // Call the invalidate method on the LeafNode
    pInter.invalidate(filename, originLNServer, verNum);

    break;
}
}
}

} catch (Exception e) {
    // Handle exceptions
    System.out.println("Exception occurred at method broadCastSS()");
}
}

/**
 * Polls the SuperPeer to check the status and version number of a specific file.
 * This method verifies if the cached copy of the file is up-to-date with the master copy.
 *
 * @param filename The name of the file to check.
 * @param verNum The version number of the cached file.
 * @param ogPeerId The original peer ID where the file originated.
 * @return A string indicating the status of the file ("Proper version" or "File out of Date").
 * @throws RemoteException if a remote communication error occurs.
 */
@Override
public String poll(String filename, String verNum, String ogPeerId) throws RemoteException {

```



```

// Search for the file in the SuperPeer's index

Collection<ArrayList<String>> resultArrFile = searchFile(filename);

// 0 -
filename,1-peerid,2-portno,3-direct,4-superpeerid,5-copytype,6-vernum,7-status,8-lmt,9-TTR,10-ogPeerId

String retValue = "File not found at Master Node";

if (!resultArrFile.isEmpty()) {
    for (ArrayList<String> asr : resultArrFile) {
        // Check if the entry matches the original peer and is a Master Copy
        if (asr.get(1).equalsIgnoreCase(ogPeerId) && asr.get(5).equalsIgnoreCase("MC")) {
            if (asr.get(6).equalsIgnoreCase(verNum)) {
                retValue = "Proper version";
            } else {
                retValue = "File out of Date";
            }
            break;
        }
    }
    return retValue;
} else {
    return retValue;
}
}

```

/**

* Retrieves the version number of a file from a specific peer and port.

*

* @param filename The name of the file.

* @param peerid The ID of the peer holding the file.

* @param port_num The port number of the peer.

* @param copyType The type of copy ("Master" or "Replica").

* @param ogPeerId The original peer ID where the file originated.

* @return The version number of the specified file, or "-1" if not found.

* @throws RemoteException if a remote communication error occurs.

*/

@Override

```
public String getVersionNum(String filename, String peerid, String port_num, String copyType,
String ogPeerId)
```

```
    throws RemoteException {
```

```
    // Search for the file in the SuperPeer's index
```

```
    Collection<ArrayList<String>> resultArrFile = new ArrayList<ArrayList<String>>();
```

```
    if (this.fileDictionary.containsKey(filename)) {
```

```
        resultArrFile = this.fileDictionary.get(filename);
```

```
    }
```

```
    for (ArrayList<String> as : resultArrFile) {
```

```
        if (as.get(0).equalsIgnoreCase(filename) && as.get(1).equalsIgnoreCase(peerid) &&
```

```
            as.get(2).equalsIgnoreCase(port_num) && as.get(5).equalsIgnoreCase(copyType) &&
```

```
            as.get(10).equalsIgnoreCase(ogPeerId)) {
```

```
            return as.get(6); // Return the version number
```

```
        }
```

```
    }
```

```
    // Master Copy Filename entry not found under requested peer ID
```

```

        return "-1";
    }

/**
 * Notifies the SuperPeer of a polling request and handles version checking.
 *
 * Initiates a consistent pull to verify the file version and schedules tasks to manage the
polling response.
 *
 * @param filename    The name of the file being polled.
 * @param reqPeerid   The ID of the requesting peer.
 * @param reqPort_num The port number of the requesting peer.
 * @param copyType    The type of copy ("Master" or "Replica").
 * @param verNum      The version number of the cached file.
 * @param ogPeerId    The original peer ID where the file originated.
 * @param TTR         Time-To-Refresh value for pull-based consistency.
 * @param lastModTime The last modified time of the file.
 * @throws RemoteException    if a remote communication error occurs.
 * @throws NotBoundException  if a remote object is not bound in the registry.
 */
@Override
public void notifyPoll(String filename, String reqPeerid, String reqPort_num, String copyType,
String verNum, String ogPeerId,
                        String TTR, String lastModTime) throws RemoteException, NotBoundException {

    // Find the master node's SuperPeer ID

    String masterSPID = null;

    SetupConfig scg;

```

```

try {
    scg = new SetupConfig();
    for (GetPeerDetails gpd : scg.arrPD) {
        if (gpd.getPeer_ID().equalsIgnoreCase(ogPeerId)) {
            masterSPID = gpd.getSuperPeer();
            break;
        }
    }
} catch (IOException e1) {
    System.out.println("IOException occurred while reading the property file at Polling
function Pull 1");
}

// Initiate consistent pull to verify file version
try {
    SuperPeerImpl.flagTable.put(filename, "true");
    consistentPull(masterSPID, filename, verNum, TTR, ogPeerId);
} catch (NotBoundException e) {
    System.out.println("NotBoundException occurred while calling consistentPull in
notifyPoll");
}

System.out.println("Line 5");

// Lookup the LeafNode interface to notify about the poll result
Registry regis = LocateRegistry.getRegistry("localhost", Integer.parseInt(reqPort_num));

LeafNodeInterface pInter = (LeafNodeInterface) regis.lookup("root://LeafNode/" +
reqPort_num + "/FS");

```

```

// Define a task to check the polling status periodically
class MyTask implements Runnable {
    public void run() {
        // Check if the flag has been updated to false, indicating the file is out of date
        if (flagTable.get(filename).equalsIgnoreCase("false")) {
            // Update the index to mark the file as invalid
            System.out.println("Consistency check completed. File is out of date.");
            try {
                // Update the file status in the index
                registryFiles("upd", filename, reqPeerid, "", "", "", "CC", verNum, "invalid",
lastModTime, TTR, ogPeerId);
            } catch (RemoteException e) {
                System.out.println("Exception occurred while updating the index in notifyPoll");
            }
            // Notify the LeafNode to invalidate its cached copy
            try {
                pInter.outOfDate(filename, "Out of Date", ogPeerId);
            } catch (RemoteException e) {
                System.out.println("Exception occurred while calling outOfDate function in
notifyPoll");
            }
            // Cancel the scheduled task as it's no longer needed
            t.cancel(false);
        }
    }
}

```

```

        System.out.println("Line 6");

        // Schedule the task to run every second

        t = executor.scheduleAtFixedRate(new MyTask(), 0, 1, TimeUnit.SECONDS);

        System.out.println("Line 7");
    }

    /**
     * Initiates a consistent pull to verify the file version from the master SuperPeer.
     *
     * @param masterPeerId The SuperPeer ID of the master node.
     * @param filename    The name of the file being polled.
     * @param verNum      The version number of the cached file.
     * @param TTR         Time-To-Refresh value for pull-based consistency.
     * @param ogPeerId    The original peer ID where the file originated.
     * @throws RemoteException if a remote communication error occurs.
     * @throws NotBoundException if a remote object is not bound in the registry.
     */
    // Leaf node

    public static void consistentPull(String masterPeerId, String filename, String verNum, String
TTR, String ogPeerId)

        throws RemoteException, NotBoundException {

        int timeToRef = Integer.parseInt(TTR);

        // Retrieve the port number of the master SuperPeer

        String masterPortNum = null;

        SetupConfig scg;

        try {

```

```

    scg = new SetupConfig();
    for (GetSuperPeerDetails gspd : scg.arrSPD) {
        if (gspd.getPeer_ID().equalsIgnoreCase(masterPeerId)) {
            masterPortNum = gspd.getPeer_Port();
            break;
        }
    }
} catch (IOException e1) {
    System.out.println("IOException occurred while reading the property file at Polling
function Pull 1");
}

// Lookup the master SuperPeer interface
Registry regis = LocateRegistry.getRegistry("localhost", Integer.parseInt(masterPortNum));

SuperPeerInterface splInter = (SuperPeerInterface) regis.lookup("root://SuperPeer/" +
masterPortNum);

// Define a task to poll the master SuperPeer for the file version
final Runnable leafNodePoll = new Runnable() {
    public void run() {
        try {
            // Poll the master SuperPeer for the file status
            status1Table.put(filename, splInter.poll(filename, verNum, ogPeerId));
        } catch (RemoteException e) {
            System.out.println("RemoteException occurred while polling the master
SuperPeer");
        }
    }
}

```

```
        System.out.println("STATUS from MASTER SUPER PEER for downloaded file " +  
filename + " : " + status1Table.get(filename));
```

```
    }
```

```
};
```

```
// Schedule the polling task to run at fixed intervals based on TTR
```

```
    final ScheduledFuture<?> leafNodePollHandle =  
scheduler.scheduleAtFixedRate(leafNodePoll, 1, timeToRef, TimeUnit.SECONDS);
```

```
// Executor to check and cancel the polling task if the file is out of date
```

```
    ScheduledExecutorService queueCancelCheckExecutor =  
Executors.newSingleThreadScheduledExecutor();
```

```
    final Runnable stopBeep = new Runnable() {
```

```
        public void run() {
```

```
            if (!status1Table.isEmpty()) {
```

```
                if (status1Table.get(filename).equalsIgnoreCase("File out of Date")) {
```

```
                    flagTable.put(filename, "false");
```

```
                    leafNodePollHandle.cancel(true); // Cancel the polling task
```

```
                }
```

```
            }
```

```
        }
```

```
};
```

```
// Schedule the stopBeep task to check the file status every 500 milliseconds
```

```
    queueCancelCheckExecutor.scheduleAtFixedRate(stopBeep, 1, 500,  
TimeUnit.MILLISECONDS);
```

```
}
```

```
/*----- end change -----*/
```

```
}
```


LeafNode.java

Description:

The `LeafNode` class serves as the entry point for initializing and running a `LeafNode` in the P2P network. Upon execution, it prompts the user to input the `LeafNode` (peer) ID, retrieves the corresponding port number, directories, and associated `SuperPeer` ID from the configuration, sets up the RMI registry on the specified port, and binds the `LeafNodeImpl` implementation to the registry. This setup enables the `LeafNode` to handle remote method invocations from `SuperPeers`, facilitating file operations such as searching, downloading, editing, and maintaining file consistency through pull-based mechanisms.

```
package com.gfiletransfer;
```

```
import java.io.IOException;
```

```
import java.rmi.RemoteException;
```

```
import java.rmi.registry.LocateRegistry;
```

```
import java.rmi.registry.Registry;
```

```
import java.rmi.server.UnicastRemoteObject;
```

```
import java.util.Scanner;
```

```
/**
```

```
 * The LeafNode class initializes and starts a LeafNode in the P2P network.
```

```
 * It sets up the RMI registry, binds the LeafNode implementation, and makes
```

```
 * the LeafNode available for remote method invocations by SuperPeers.
```

```
 */
```

```
public class LeafNode {
```

```
    /**
```

```
     * The main method serves as the entry point for the LeafNode application.
```

```
     *
```

```

* @param args Command-line arguments (not used).
* @throws RemoteException if there is an error during RMI operations.
*/
public static void main(String[] args) throws RemoteException {

    Scanner sc = new Scanner(System.in); // Scanner for user input

    String portno = null;

    String directoryName = null;

    String superPeerId = null;

    String cachedDirectoryName = null;


    System.out.println("Enter Peer ID ");

    String peerId = sc.nextLine(); // Prompt user to enter LeafNode ID


    // Reading configuration details from SetupConfig
    SetupConfig scg;

    try {

        scg = new SetupConfig(); // Initialize SetupConfig to load configurations

        // Retrieve LeafNode configuration based on Peer ID
        for (GetPeerDetails p : scg.arrPD) {

            if (p.getPeer_ID().equalsIgnoreCase(peerId)) {

                portno = p.getPeer_Port();           // LeafNode port number

                directoryName = p.getDir();           // Directory for master files

                cachedDirectoryName = p.getCacheDir(); // Directory for cached files

                superPeerId = p.getSuperPeer();       // Associated SuperPeer ID

                break;

            }

        }
    }

```

```

    }

    } catch (IOException e1) {

        System.out.println("IOException occurred while reading the property file at Leaf Node
Initialization.");

        e1.printStackTrace();

        sc.close(); // Close the scanner before exiting

        return;    // Exit the program if configuration loading fails

    }

// Initialize RMI registry on the specified port

Registry registry = LocateRegistry.createRegistry(Integer.parseInt(portno));

/*----- start change -----*/

// Instantiate the LeafNode implementation with configuration details

LeafNodeImpl lnImpl = new LeafNodeImpl(portno, directoryName, superPeerId, peerId,
cachedDirectoryName);

/*----- end change -----*/

// Export the LeafNodeImpl object to receive remote method invocations

LeafNodeInterface lnInter = (LeafNodeInterface)
UnicastRemoteObject.exportObject(lnImpl, 0);

// Bind the LeafNode implementation to the RMI registry with a unique name

registry.rebind("root://LeafNode/" + portno + "/FS", lnInter);

System.out.println("Peer is up and Running.");

try {

```

```

        InImpl.doWork(); // Start the LeafNode's main operations
    } catch (IOException e) {
        System.out.println("IOException at Leaf Node Main: " + e.getMessage());
        e.printStackTrace();
    }

    sc.close(); // Close the scanner as it's no longer needed
}
}

```

LeafNodeInterface.java

Description:

The `LeafNodeInterface` defines the remote methods that a `LeafNode` must implement in the P2P network. These methods facilitate file downloading, handling query responses from `SuperPeers`, and managing file consistency through polling and invalidation mechanisms. By extending `java.rmi.Remote`, this interface allows `LeafNodes` to expose these methods for remote invocation via Java RMI, enabling seamless interaction with `SuperPeers` and other network components.

```

package com.gfiletransfer;

import java.rmi.NotBoundException;

import java.rmi.Remote;

import java.rmi.RemoteException;

import java.util.ArrayList;

import java.util.Collection;

```

```
/**
```

- * The LeafNodeInterface defines the remote methods that a LeafNode must implement.
- * These methods facilitate file downloading, handling query responses, and maintaining
- * file consistency through polling and invalidation mechanisms.

```
*/
```

```
public interface LeafNodeInterface extends Remote {
```

```
/**
```

- * Downloads a file based on the provided directory paths.
- *
- * @param searchedDir A list of directory paths where the file is located.
- * @return A byte array representing the downloaded file.
- * @throws RemoteException if a remote communication error occurs.

```
*/
```

```
public byte[] fileDownload(ArrayList<String> searchedDir) throws RemoteException;
```

```
/**
```

- * Handles the response to a search query from a SuperPeer by returning the search results.
- *
- * @param msgId The unique message ID for the query.
- * @param TTL Time-To-Live remaining for the query message.
- * @param filename The name of the file being searched for.
- * @param resultArr A collection of search results containing file details.
- * @return A boolean indicating the success of handling the query hit.
- * @throws RemoteException if a remote communication error occurs.

```
*/
```

```
public boolean queryHit(String msgId, int TTL, String filename, Collection<ArrayList<String>>
resultArr) throws RemoteException;
```

```
/*----- start change -----*/
```

```
/**
```

```
 * Polls the SuperPeer to check if the file version is up to date.
```

```
 *
```

```
 * @param filename The name of the file to check.
```

```
 * @param verNum The version number of the cached file.
```

```
 * @return A string indicating the status of the file ("Proper version" or "File out of Date").
```

```
 * @throws RemoteException if a remote communication error occurs.
```

```
 * @throws NotBoundException if a remote object is not bound in the registry.
```

```
 */
```

```
public String poll(String filename, String verNum) throws RemoteException,
NotBoundException;
```

```
/**
```

```
 * Invalidates a specific file, marking it as outdated.
```

```
 *
```

```
 * @param filename The name of the file to invalidate.
```

```
 * @param originLNServer The originating LeafNode server ID.
```

```
 * @param verNum The version number of the file being invalidated.
```

```
 * @throws RemoteException if a remote communication error occurs.
```

```
 */
```

```
public void invalidate(String filename, String originLNServer, String verNum) throws
RemoteException;
```

```

/**
 * Marks a specific file as out of date, updating its status.
 *
 * @param filename    The name of the file to mark as out of date.
 * @param invalidStatus The status to set for the file ("Out of Date").
 * @param ogPeerId    The original peer ID where the file originated.
 * @throws RemoteException if a remote communication error occurs.
 */
public void outOfDate(String filename, String invalidStatus, String ogPeerId) throws
RemoteException;

/**
 * Retrieves the current status of a specific file.
 *
 * @param filename The name of the file whose status is to be retrieved.
 * @return A string representing the current status of the file.
 * @throws RemoteException if a remote communication error occurs.
 */
public String getStatus(String filename) throws RemoteException;

/*----- end change -----*/
}

```

LeafNodeImpl.java

Description:

The `LeafNodeImpl` class implements the `LeafNodeInterface` and provides the core functionalities of a `LeafNode` in the P2P network. It manages file operations such as downloading, searching, deleting, and editing files. Additionally, it handles file consistency through both push-based and pull-based mechanisms, ensuring that cached copies of files remain up-to-date with their master copies. The class utilizes Java RMI for remote method invocations, enabling seamless communication with `SuperPeers` and other `LeafNodes`. Scheduled tasks are employed to manage polling for file version updates and to handle invalidation processes when discrepancies are detected.

```
package com.gfiletransfer;
```

```
import java.io.File;
```

```
import java.io.FileOutputStream;
```

```
import java.io.FileWriter;
```

```
import java.io.IOException;
```

```
import java.nio.file.Files;
```

```
import java.rmi.NotBoundException;
```

```
import java.rmi.RemoteException;
```

```
import java.rmi.registry.LocateRegistry;
```

```
import java.rmi.registry.Registry;
```

```
import java.text.DecimalFormat;
```

```
import java.text.NumberFormat;
```

```
import java.text.SimpleDateFormat;
```

```
import java.util.ArrayList;
```

```
import java.util.Collection;
```

```
import java.util.Date;
```

```
import java.util.HashMap;
```



```
import java.util.Map;

import java.util.Scanner;

import java.util.concurrent.CountDownLatch;

import java.util.concurrent.ExecutionException;

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;

import java.util.concurrent.Future;

import java.util.concurrent.ScheduledExecutorService;

import java.util.concurrent.ScheduledFuture;

import java.util.concurrent.ScheduledThreadPoolExecutor;

import java.util.concurrent.TimeUnit;

import java.util.concurrent.TimeoutException;


import javax.ws.rs.core.MultivaluedHashMap;

import javax.ws.rs.core.MultivaluedMap;


import java.util.Timer;

import java.util.Map.Entry;
```

```
/**
```

```
 * The LeafNodeImpl class implements the LeafNodeInterface and provides the core functionalities
```

- * of a LeafNode in the P2P network. It manages file operations such as downloading, searching,
- * deleting, and editing files. Additionally, it handles file consistency through both push-based
- * and pull-based mechanisms, ensuring that cached copies of files remain up-to-date with their
- * master copies.

```

*/

public class LeafNodeImpl implements LeafNodeInterface {

    // Configuration and state variables

    String portNo = null; // Port number of the LeafNode

    String dirName = null; // Directory where master files are stored

    String cachedDirName = null; // Directory where downloaded/cached files are stored

    String fileName = null; // Name of the file to be searched

    String remotePeer = null; // Peer ID from whom the file is to be downloaded

    String superpeer = null; // SuperPeer ID associated with this LeafNode

    String peerID = null; // LeafNode ID

    int seqNum = -1; // Sequence number for queries

    int broadSeqNum = -1; // Sequence number for broadcasts

    int timeTL = 20; // Time-To-Live for queries (3 for All-to-All, 22 for Linear Topology)

    String msgId = null; // Message ID for queries

    String consistencyType = null; // Type of consistency mechanism (PUSH, PULL1, PULL2)

    String verString = null; // Version string for files

    /*----- start change -----*/

    // Scheduled executor services and flags for managing consistency

    final static ScheduledExecutorService scheduler = Executors.newScheduledThreadPool(1);

    static ScheduledThreadPoolExecutor executor = new ScheduledThreadPoolExecutor(15); //
    Thread pool for scheduled tasks

    static ScheduledFuture<?> t; // Reference to a scheduled task

    public static Map<String, String> flagTable = new HashMap<String, String>(); // Flags for
    consistency checks

```

```

    public static Map<String, String> status1Table = new HashMap<String, String>(); // Status
table for file versions

    static String status1 = null; // Temporary status holder

    private MultivaluedMap<String, ArrayList<String>> finalHM = new MultivaluedHashMap<>();
// HashMap to store query results

    private Map<String, ArrayList<String>> cachedTable = new HashMap<String,
ArrayList<String>>(); // Cached files table

    /*----- end change -----*/

/**
 * Constructor to initialize the LeafNodeImpl with necessary configuration details.
 *
 * @param portNo      The port number of the LeafNode.
 * @param dirName      The directory where master files are stored.
 * @param superpeer    The SuperPeer ID associated with this LeafNode.
 * @param peerID       The LeafNode ID.
 * @param cachedDirName The directory where downloaded/cached files are stored.
 */
    LeafNodeImpl(String portNo, String dirName, String superpeer, String peerID, String
cachedDirName) {

        this.portNo = portNo;

        this.dirName = dirName;

        this.superpeer = superpeer;

        this.peerID = peerID;

        this.cachedDirName = cachedDirName;

    }

/**

```

* The main operational method for the LeafNode. It handles user interactions for searching,
* deleting, and editing files, and manages file registration and consistency mechanisms.

*

* @throws IOException if an I/O error occurs during operations.

*/

```
public void doWork() throws IOException {  
    String superPeerPort = null;  
  
    // Reading SuperPeer Port details from configuration for connecting to the indexing server  
    (SuperPeer)  
    SetupConfig scg;  
  
    try {  
        scg = new SetupConfig(); // Initialize SetupConfig to load configurations  
        // Retrieve SuperPeer port number based on SuperPeer ID  
        for (GetSuperPeerDetails sp : scg.arrSPD) {  
            if (sp.getPeer_ID().equalsIgnoreCase(this.superpeer)) {  
                superPeerPort = sp.getPeer_Port();  
                break;  
            }  
        }  
  
        consistencyType = scg.consisisApp; // Retrieve consistency mechanism type  
    } catch (IOException e1) {  
        System.out.println("IOException occurred while reading the property file at connecting to  
        Super Peer.");  
        e1.printStackTrace();  
        return; // Exit if configuration loading fails  
    }  
}
```

```

try {

    // Locate the RMI registry of the SuperPeer

    Registry regis = LocateRegistry.getRegistry("localhost", Integer.parseInt(superPeerPort));

    SuperPeerInterface splInter = (SuperPeerInterface) regis.lookup("root://SuperPeer/" +
superPeerPort);

    Scanner sc = new Scanner(System.in); // Scanner for user input


    // Obtain list of files in the master directory

    File dirList = new File(dirName);

    String[] record = dirList.list();


    // Initialize versioning for master copies

    int versionNumEdit = 1;

    verString = "v" + String.format("%02d", versionNumEdit);


    // Registering all master files with the SuperPeer

    for (int c = 0; c < record.length; c++) {

        File currentFile = new File(record[c]);

        System.out.println("Registering details of File name " + currentFile.getName() + " in
Indexing Server");

        /*----- start change -----*/

        String timeStamp = new SimpleDateFormat("yyyy.MM.dd.HH.mm.ss").format(new
Date());

        splInter.registryFiles("new", currentFile.getName(), peerID, portNo, dirName,
superpeer, "MC", verString, "valid", timeStamp, "30", peerID);

        /*----- end change -----*/

    }
}

```

```

// User interaction loop for Search, Delete, Edit, or Exit

System.out.println("Do you want to Search a File, Delete File, Edit file or Exit?
(Search/Delete/Edit/Exit)");

String sd = sc.nextLine();

while (!sd.equalsIgnoreCase("Exit")) {

    if (sd.equalsIgnoreCase("Delete")) {

        // Deleting files from the local directory and updating the SuperPeer

        String wantToDel = "";

        while (!wantToDel.equalsIgnoreCase("No")) {

            System.out.println("Enter the file name which you want to delete");

            String fname = sc.nextLine();

            if (fname != null) {

                File fileToDel = new File(dirName + "\\" + fname);

                // Attempt to delete the specified file

                if (fileToDel.delete()) {

                    System.out.println("File deleted Successfully.");

                    // Notify the SuperPeer about the deletion

                    splInter.registryFiles("del", fname, peerID, portNo, dirName, superpeer, "", "",
"", "", peerID);

                } else {

                    System.out.println("Failed to delete the File");

                }

            } else {

                System.out.println("Please Enter a Filename");

            }

            System.out.println("Do you want to delete more files? (Yes/No)");

            wantToDel = sc.nextLine();

```

```

    }
}

/*----- start change -----*/

// Editing files in the local directory and updating the SuperPeer
if (sd.equalsIgnoreCase("Edit")) {
    String wantToEdit = "";
    while (!wantToEdit.equalsIgnoreCase("No")) {
        System.out.println("Enter the file name which you want to Edit(Append)");
        String fname = sc.nextLine();
        if (fname != null) {
            File fileToEdit = new File(dirName + "\\\" + fname);

            System.out.println("Enter anything you want to append in this file");
            String appendString = sc.nextLine();

            FileWriter fw = new FileWriter(dirName + "\\\" + fname, true);
            fw.write(appendString);
            fw.close();
            System.out.println("File edited Successfully.");

            // Retrieve the current version number from the SuperPeer
            verString = splInter.getVersionNum(fname, peerID, portNo, "MC", peerID);

            if (verString.equalsIgnoreCase("-1")) {
                System.out.println("File not found under Peer " + peerID + " in Registry
Index");
            }
        }
    }
}

```

```

    }

    // If the master copy is found, update the version number
    else {

        // Increment the version number

        int intvernum = Integer.parseInt(verString.substring(1)); // Extract numeric
part
        intvernum += 1;

        verString = "v" + String.format("%02d", intvernum);

        System.out.println("New Version Number for edited file: " + verString);

        try {

            String timeStamp = new
SimpleDateFormat("yyyy.MM.dd.HH.mm.ss").format(new Date());

            // Update the file details in the SuperPeer

            spInter.registryFiles("upd", fname, peerID, portNo, dirName, superpeer,
"MC", verString, "valid", timeStamp, "", peerID);

        } catch (Exception e) {

            System.out.println("Exception occurred: Updating the Registry Index at
Super Peer");

        }

        // Handle consistency based on the configured type
        if (consistencyType.equalsIgnoreCase("PUSH")) {

            System.out.println("Send invalidate request to all nodes (PUSH)");

            broadSeqNum = broadSeqNum + 1;

            String messId = peerID + ":" + superpeer + ":" + String.format("%02d",
broadSeqNum);

            spInter.broadCastSP(messId, fname, peerID, verString);

        }

```



```

    }
} else {
    System.out.println("Please Enter a Filename");
}

System.out.println("Do you want to edit more files? (Yes/No)");
wantToEdit = sc.nextLine();
}
}

// EDIT PART END

/*----- end change -----*/

else if (sd.equalsIgnoreCase("Search")) {

    // Searching and downloading a file from the network

    String ans = "";

    while (!ans.equalsIgnoreCase("No")) {

        // Initiate a new search query

        seqNum = seqNum + 1;

        System.out.println("Enter the file name which you want to search");

        fileName = sc.nextLine();

        if (fileName != null) {

            msgId = peerID + ":" + Integer.toString(seqNum);

            // Execute the query with a timeout

            ExecutorService service = Executors.newSingleThreadExecutor();

            try {

                Runnable r = new Runnable() {

```

```

@Override
public void run() {
    try {
        System.out.println("Now Started Calling the query() from Leaf
Node...");

        splInter.query(msgId, timeTL, fileName, peerID, portNo);
    } catch (RemoteException e) {
        System.out.println("TimeOut: It ran too long. Need to stop searching
and continue.");
    }
}
};

Future<?> f = service.submit(r);
f.get(4, TimeUnit.SECONDS); // Attempt the search for 4 seconds
} catch (final InterruptedException e) {
    // Handle thread interruption

    System.out.println("Interrupted Exception Occurred");
} catch (final TimeoutException e) {
    // Handle timeout

    System.out.println("Timeout Exception Occurred. It ran too long. Need to stop
searching and continue.");
} catch (final ExecutionException e) {
    // Handle other execution exceptions

    System.out.println("Execution Exception Occurred");
} finally {
    service.shutdownNow(); // Shutdown the executor service
}

```

```

// Retrieve the search results from finalHM
Collection<ArrayList<String>> finalRes = this.finalHM.get(msgId);

String refVerNum = null;

/*----- start change -----*/

if (!finalRes.isEmpty()) {

    // Determine the version number of the master copy
    for (ArrayList<String> as : finalRes) {

        if (as.get(0).equalsIgnoreCase(fileName)) {

            if (as.get(5).equalsIgnoreCase("MC")) {

                refVerNum = as.get(6);

                break;

            }

        }

    }

    // Display the list of peers providing the requested file

System.out.println("#####");

    System.out.println("\n");

    for (ArrayList<String> als : finalRes) {

        if (als.get(0).equalsIgnoreCase(fileName)) {

            if (als.get(5).equalsIgnoreCase("MC")) {

                System.out.println("VALID Peer providing the file with Peer ID is " +
als.get(1) + " under Super Peer: " + als.get(4) + " which is a Master Copy");

            } else {

                // Validate cached copies based on version number

```

```

        if (als.get(6).equalsIgnoreCase(refVerNum)) {
            System.out.println("VALID Peer providing the file with Peer ID is " +
als.get(1) + " under Super Peer: " + als.get(4) + " which is a Cached Copy");
        }
    }
}
}

System.out.println("\n");

```

```

System.out.println("#####");

```

```

/*----- end change -----*/

// Prompt user to select a peer to download the file from
System.out.println("Enter Peer ID you wish to take the file from");
remotePeer = sc.nextLine();

// Download the file from the selected peer
int co = finalRes.size();
if (remotePeer != null) {
    for (ArrayList<String> als : finalRes) {
        if (als.get(1).equalsIgnoreCase(remotePeer)) {
            // Locate the RMI registry of the selected peer
            Registry regis2 = LocateRegistry.getRegistry("localhost",
Integer.parseInt(als.get(2)));
            LeafNodeInterface lnInter = (LeafNodeInterface)
regis2.lookup("root://LeafNode/" + als.get(2) + "/FS");

/*----- start change -----*/

// If the selected peer holds the master copy, download directly

```

```

        if (als.get(5).equalsIgnoreCase("MC")) {
            byte[] output = InInter.fileDownload(als);
            System.out.println(output.length);

            // Convert the downloaded byte array into a file
            if (output.length != 0) {
                FileOutputStream ostream = null;
                try {
                    ostream = new FileOutputStream(cachedDirName + "\\\" +
fileName);

                    ostream.write(output);

                    System.out.println("File Downloading Successful.");
                    System.out.println("Display File " + fileName);

                    // Update the SuperPeer's index after downloading the file
                    splInter.registryFiles("new", fileName, peerID, portNo,
cachedDirName, superpeer, "CC", als.get(6), "valid", als.get(8), als.get(9), als.get(10));

                    // Add the downloaded file details to the cached table
                    // 0 - filename, 1 - copy_type, 2 - cachedDirectory, 3 -
versionNumber, 4 - OriginServerId, 5 - status, 6 - TTR

                    ArrayList<String> ct = new ArrayList<String>();
                    ct.add(fileName); // filename
                    ct.add("CC"); // copy_type
                    ct.add(cachedDirName); // cachedDirectory
                    ct.add(als.get(6)); // versionNumber
                    ct.add(als.get(10)); // OriginServerId

```

```

        ct.add("valid"); // status
        ct.add(als.get(9)); // TTR
        this.cachedTable.put(fileName, ct);
        // Display the updated cached table
        System.out.println("Updated Cached Table Entry after insertion
(File download)");

        for (Entry<String, ArrayList<String>> entry :
this.cachedTable.entrySet()) {
            System.out.println(entry.getKey() + " => " + entry.getValue());
        }

        // Initiate polling for file consistency if configured
        if (consistencyType.equalsIgnoreCase("PULL1")) {
            LeafNodeImpl.flagTable.put(fileName, "true");
            // Start the consistent pull process
            consistentPull(als.get(10), fileName, als.get(6), als.get(9));

            System.out.println("Line 5");
            // Define a task to check the polling status
            class MyTask implements Runnable {
                public void run() {
                    // Check if the polling flag has been set to false
                    if (flagTable.get(fileName).equalsIgnoreCase("false")) {
                        // Update the cached table and SuperPeer's index
                        System.out.println("Consistency check completed. File
is out of date.");
                        try {

```

```

        // Mark the file as invalid in the SuperPeer's index
        splInter.registryFiles("upd", fileName, peerID, portNo,
cachedDirName, superpeer, "CC", als.get(6), "invalid", als.get(8), als.get(9), als.get(10));

        } catch (RemoteException e) {

            System.out.println("Exception occurred while
updating the Index");

        }

        // Invalidate the cached file in the local cached table
        try {

            outOfDate(fileName, "Out of Date", als.get(10));

        } catch (RemoteException e) {

            System.out.println("Exception occurred while calling
outOfDate function in notifyPoll");

        }

        // Cancel the scheduled task as it's no longer needed
        t.cancel(false);

    }

}

System.out.println("Line 6");

// Schedule the task to run every second
t = executor.scheduleAtFixedRate(new MyTask(), 0, 1,
TimeUnit.SECONDS);

System.out.println("Line 7");

} else if (consistencyType.equalsIgnoreCase("PULL2")) {

    // Handle PULL2 consistency by notifying the SuperPeer
    if (!this.cachedTable.isEmpty()) {

        if (this.cachedTable.containsKey(fileName)) {

```

```

        ArrayList<String> ct1 = this.cachedTable.get(fileName);

        splInter.notifyPoll(fileName, peerID, portNo, ct1.get(1),
ct1.get(3), ct1.get(4), ct1.get(6), als.get(8));

    }

    }

    }

    } catch (Exception e) {

        System.out.println("Exception in bytearray to file conversion: " +
e.getMessage());

    } finally {

        if (ostream != null) {

            ostream.close(); // Ensure the output stream is closed

        }

    }

    } else {

        System.out.println("File is not present at Remote Location.");

    }

    break; // Exit the loop after successful download

}

// If the selected peer holds a cached copy, validate before
downloading

else {

    if (lnInter.getStatus(als.get(0)).equalsIgnoreCase("valid")) {

        // Download the cached file

        byte[] output = lnInter.fileDownload(als);

        System.out.println(output.length);

```



```

        // Convert the downloaded byte array into a file
        if (output.length != 0) {
            FileOutputStream ostream = null;
            try {
                ostream = new FileOutputStream(cachedDirName + "\\\" +
fileName);

                ostream.write(output);

                System.out.println("File Downloading Successful.");
                System.out.println("Display File " + fileName);

                // Update the SuperPeer's index after downloading the file
                splInter.registryFiles("new", fileName, peerID, portNo,
cachedDirName, superpeer, "CC", als.get(6), "valid", als.get(8), als.get(9), als.get(10));

                // Add the downloaded file details to the cached table
                ArrayList<String> ct = new ArrayList<String>();
                ct.add(fileName);
                ct.add("CC");
                ct.add(cachedDirName);
                ct.add(als.get(6));
                ct.add(als.get(10));
                ct.add("valid");
                ct.add(als.get(9));
                this.cachedTable.put(fileName, ct);

                // Display the updated cached table
                System.out.println("Updated Cached Table Entry after
insertion (File download)");

```

```

this.cachedTable.entrySet()) {
    for (Entry<String, ArrayList<String>> entry :
        System.out.println(entry.getKey() + " => " +
            entry.getValue());
    }

    // Initiate polling for file consistency if configured
    if (consistencyType.equalsIgnoreCase("PULL1")) {
        this.flagTable.put(fileName, "true");
        // Start the consistent pull process
        consistentPull(als.get(10), fileName, als.get(6), als.get(9));

        System.out.println("Line 5");
        // Define a task to check the polling status
        class MyTask implements Runnable {
            public void run() {
                // Check if the polling flag has been set to false
                if (flagTable.get(fileName).equalsIgnoreCase("false")) {
                    // Update the cached table and SuperPeer's index
                    System.out.println("Consistency check completed.
File is out of date.");

                    try {
                        // Mark the file as invalid in the SuperPeer's index
                        spInter.registryFiles("upd", fileName, peerID,
portNo, cachedDirName, superpeer, "CC", als.get(6), "invalid", als.get(8), als.get(9), als.get(10));
                    } catch (RemoteException e) {
                        System.out.println("Exception occurred while
updating the Index");

```

```

    }

    // Invalidate the cached file in the local cached table
    try {
        outOfDate(fileName, "Out of Date", als.get(10));
    } catch (RemoteException e) {
        System.out.println("Exception occurred while
calling outOfDate function in notifyPoll");
    }

    // Cancel the scheduled task as it's no longer needed
    t.cancel(false);
}
}
}

System.out.println("Line 6");

// Schedule the task to run every second
t = executor.scheduleAtFixedRate(new MyTask(), 0, 1,
TimeUnit.SECONDS);

System.out.println("Line 7");
} else if (consistencyType.equalsIgnoreCase("PULL2")) {
    // Handle PULL2 consistency by notifying the SuperPeer
    if (!this.cachedTable.isEmpty()) {
        if (this.cachedTable.containsKey(fileName)) {
            ArrayList<String> ct1 = this.cachedTable.get(fileName);
            splInter.notifyPoll(fileName, peerID, portNo, ct1.get(1),
ct1.get(3), ct1.get(4), ct1.get(6), als.get(8));
        }
    }
}
}

```

```

    }

    } catch (Exception e) {

        System.out.println("Exception in bytearray to file conversion: "
+ e.getMessage());

    } finally {

        if (ostream != null) {

            ostream.close(); // Ensure the output stream is closed

        }

    }

}

/*----- end change -----*/

else {

    System.out.println("File is not present at Remote Location.");

}

break; // Exit the loop after successful download

} else {

    System.out.println("The Peer which you had selected just got its
file invalidated. Please select Master Copy for guaranteed file download.");

}

}

} else {

    if (co == 1)

        System.out.println("Peer with that ID " + remotePeer + " does not
exist. Please choose a proper PeerId.");

}

co--;

```

```

        }
    } else {
        System.out.println("Please enter a proper Peer ID");
    }
} else {
    System.out.println("Sorry, File which you are searching doesn't exist in our
Server.");
}
System.out.println("Do you want to search again? (Yes/No)");
ans = sc.nextLine();
}
}
else{
    System.out.println("Please select an appropriate choice");
}
System.out.println("Do you want to Search a File, Delete File or Exit?
(Search/Delete/Exit)");
sd = sc.nextLine();
}
System.exit(0); // Terminate the application
} catch (Exception e) {
    System.out.println("Exception at Client Interface: " + e.getMessage());
    e.printStackTrace();
}
}
}

```

/**

```
* Downloads a file from the specified directory paths.  
*  
* @param searchedDir A list containing directory paths where the file is located.  
* @return A byte array representing the downloaded file.  
* @throws RemoteException if a remote communication error occurs.  
*/
```

```
@Override
```

```
public byte[] fileDownload(ArrayList<String> searchedDir) throws RemoteException {  
    // Extract filename and remote directory from the searchedDir list  
    String fname = searchedDir.get(0);  
    String remoteDir = searchedDir.get(3);  
    try {  
        File file = new File(remoteDir + "\\\" + fname);  
        if (file.exists()) {  
            byte buffer[] = Files.readAllBytes(file.toPath()); // Read file bytes  
            return buffer; // Return the file as a byte array  
        }  
    } catch (Exception e) {  
        System.out.println("Error in File download part: " + e.getMessage());  
        e.printStackTrace();  
        return new byte[0]; // Return an empty byte array in case of error  
    }  
    return new byte[0]; // Return an empty byte array if file does not exist  
}
```

```
/**
```

* Handles the response to a search query by storing the search results.

*

* @param msgId The unique message ID for the query.

* @param TTL Time-To-Live remaining for the query message.

* @param filename The name of the file being searched for.

* @param resultArr A collection of search results containing file details.

* @return A boolean indicating the success of handling the query hit.

* @throws RemoteException if a remote communication error occurs.

*/

@Override

```
public synchronized boolean queryHit(String msgId, int TTL, String filename,
Collection<ArrayList<String>> resultArr)
```

```
    throws RemoteException {
```

```
    if (TTL > 0 && TTL != 0) {
```

```
        try {
```

```
            // Add the search results to the finalHM HashMap
```

```
            for (ArrayList<String> arrFileDtl : resultArr) {
```

```
                this.finalHM.add(msgId, arrFileDtl);
```

```
            }
```

```
            return true; // Indicate successful handling
```

```
        } catch (Exception e) {
```

```
            System.out.println("Exception at Peer's Interface: " + e.getMessage());
```

```
            return false; // Indicate failure
```

```
        }
```

```
    } else {
```

```
        System.out.println("Time to Live of a Message has expired at Leaf Node. This Message is no longer valid.");
```

```

        return false; // Indicate failure due to TTL expiry
    }
}

/*----- start change -----*/
/**
 * Polls the SuperPeer to check if the cached file version is up-to-date.
 *
 * @param filename The name of the file to check.
 * @param verNum The version number of the cached file.
 * @return A string indicating the status of the file ("Proper version" or "File out of Date").
 * @throws RemoteException if a remote communication error occurs.
 * @throws NotBoundException if a remote object is not bound in the registry.
 */
@Override
public String poll(String filename, String verNum) throws RemoteException,
NotBoundException {
    // Retrieve the SuperPeer port number from configuration
    String spPort = null;
    SetupConfig scg;
    try {
        scg = new SetupConfig();
        // Find the SuperPeer port based on SuperPeer ID
        for (GetSuperPeerDetails sp : scg.arrSPD) {
            if (sp.getPeer_ID().equalsIgnoreCase(this.superpeer)) {
                spPort = sp.getPeer_Port();
            }
        }
    }
}

```



```

        break;
    }
}
} catch (IOException e1) {
    System.out.println("IOException occurred while reading the property file in Polling
function");
    e1.printStackTrace();
    return "Error: Unable to read configuration.";
}

// Locate the SuperPeer's RMI registry
Registry regis = LocateRegistry.getRegistry("localhost", Integer.parseInt(spPort));
SuperPeerInterface splInter = (SuperPeerInterface) regis.lookup("root://SuperPeer/" +
spPort);

// Search for the file in the SuperPeer's index
Collection<ArrayList<String>> resultArrFile = splInter.searchFile(filename);

// Initialize return value
String retValue = "File not found at Master Node";

if (!resultArrFile.isEmpty()) {
    for (ArrayList<String> asr : resultArrFile) {
        // Check if the entry corresponds to the master copy of the file
        if (asr.get(1).equalsIgnoreCase(this.peerID) && asr.get(5).equalsIgnoreCase("MC")) {
            if (asr.get(6).equalsIgnoreCase(verNum)) {
                retValue = "Proper version";
            } else {

```

```

        retValue = "File out of Date";
    }
    break;
}
}
return retValue; // Return the status of the file
} else {
    return retValue; // Return the status if the file was not found
}
}

/**
 * Invalidates a specific file by marking its status as "invalid" in the cached table.
 *
 * @param filename    The name of the file to invalidate.
 * @param originLNserver The originating LeafNode server ID.
 * @param verNum      The version number of the file being invalidated.
 * @throws RemoteException if a remote communication error occurs.
 */
@Override
public void invalidate(String filename, String originLNserver, String verNum) throws
RemoteException {
    // Check if the file exists in the cached table
    if (this.cachedTable.containsKey(filename)) {
        // Iterate over the cached table to find and invalidate the specific file entry
        for (String key : this.cachedTable.keySet()) {

```

```

        if (key.equalsIgnoreCase(filename)) {

            ArrayList<String> ls = this.cachedTable.get(key);

            // Compare the origin server ID and version number to determine if invalidation is
needed

            if (ls.get(4).equalsIgnoreCase(originLNserver) &&
!(ls.get(3).equalsIgnoreCase(verNum))) {

                ls.set(5, "invalid"); // Set the status to "invalid"

                this.cachedTable.remove(key);

                this.cachedTable.put(key, ls);

            }

            // If version numbers match, do nothing

            break;

        }

    }

    // Display the updated cached table

    System.out.println("Updated Cached Table Entry after updation");

    for (Entry<String, ArrayList<String>> entry : this.cachedTable.entrySet()) {

        System.out.println(entry.getKey() + " => " + entry.getValue());

    }

} else {

    // If the file is not found in the cached table

    System.out.println("Entry not found in cached table");

}

}

}

```

/**

* Marks a specific file as out of date by invalidating its cached copy.

*

* @param filename The name of the file to mark as out of date.

* @param invalidStatus The status to set for the file ("Out of Date").

* @param ogPeerId The original peer ID where the file originated.

* @throws RemoteException if a remote communication error occurs.

*/

@Override

```
public void outOfDate(String filename, String invalidStatus, String ogPeerId) throws
RemoteException {
```

```
    if (invalidStatus.equalsIgnoreCase("Out of Date")) {
```

```
        System.out.println("Received notification from Super Peer to invalidate the file: " +
filename);
```

```
        InInvalidate(filename, ogPeerId); // Invalidate the file in the cached table
```

```
    }
```

```
}
```

/**

* Retrieves the current status of a specific file from the cached table.

*

* @param filename The name of the file whose status is to be retrieved.

* @return A string representing the current status of the file ("valid", "invalid", or "-1" if not found).

* @throws RemoteException if a remote communication error occurs.

*/

@Override

```
public String getStatus(String filename) throws RemoteException {
```

```
    if (this.cachedTable.containsKey(filename)) {
```

```

    for (String key : this.cachedTable.keySet()) {
        if (key.equalsIgnoreCase(filename)) {
            ArrayList<String> ls = this.cachedTable.get(key);
            return ls.get(5); // Return the status of the file
        }
    }
}

return "-1"; // Return "-1" if the file is not found
}

/**
 * Invalidates a specific file in the cached table by setting its status to "invalid".
 *
 * @param filename    The name of the file to invalidate.
 * @param originLNserver The originating LeafNode server ID.
 */
public void InInvalidate(String filename, String originLNserver) {
    if (this.cachedTable.containsKey(filename)) {
        // Iterate over the cached table to find and invalidate the specific file entry
        for (String key : this.cachedTable.keySet()) {
            if (key.equalsIgnoreCase(filename)) {
                ArrayList<String> ls = this.cachedTable.get(key);
                // Compare the origin server ID to determine if invalidation is needed
                if (ls.get(4).equalsIgnoreCase(originLNserver)) {
                    ls.set(5, "invalid"); // Set the status to "invalid"
                    this.cachedTable.remove(key);
                }
            }
        }
    }
}

```

```

        this.cachedTable.put(key, ls);
    }

    // If the origin server ID does not match, do nothing
    break;
}
}

// Display the updated cached table
System.out.println("Updated Cached Table Entry after Invalidation from Pull");
for (Entry<String, ArrayList<String>> entry : this.cachedTable.entrySet()) {
    System.out.println(entry.getKey() + " => " + entry.getValue());
}
} else {
    // If the file is not found in the cached table
    System.out.println("Entry not found in cached table");
}
}

```

/**

* Initiates a consistent pull to verify the file version from the master SuperPeer.

*

* @param masterPeerId The SuperPeer ID of the master node.

* @param filename The name of the file being polled.

* @param verNum The version number of the cached file.

* @param TTR Time-To-Refresh value for pull-based consistency.

* @throws RemoteException if a remote communication error occurs.

* @throws NotBoundException if a remote object is not bound in the registry.

```

*/

public static void consistentPull(String masterPeerId, String filename, String verNum, String
TTR)

    throws RemoteException, NotBoundException {

    int timeToRef = Integer.parseInt(TTR); // Time-To-Refresh in seconds

    // Retrieve the port number of the master SuperPeer

    String masterPortNum = null;

    SetupConfig scg;

    try {

        scg = new SetupConfig();

        for (GetPeerDetails gpd : scg.arrPD) {

            if (gpd.getPeer_ID().equalsIgnoreCase(masterPeerId)) {

                masterPortNum = gpd.getPeer_Port();

                break;

            }

        }

    } catch (IOException e1) {

        System.out.println("IOException occurred while reading the property file at Polling
function Pull 1");

        e1.printStackTrace();

        return; // Exit if configuration loading fails

    }

    // Locate the RMI registry of the master SuperPeer

    Registry regis = LocateRegistry.getRegistry("localhost", Integer.parseInt(masterPortNum));

```

```

LeafNodeInterface pInter = (LeafNodeInterface) regis.lookup("root://LeafNode/" +
masterPortNum + "/FS");

// Define a task to poll the master SuperPeer for the file status

final Runnable leafNodePoll = new Runnable() {

    public void run() {

        try {

            // Poll the master SuperPeer for the file status

            status1Table.put(filename, pInter.poll(filename, verNum));

        } catch (RemoteException e) {

            System.out.println("Remote Exception occurred while Polling the master node");

        } catch (NotBoundException e) {

            System.out.println("NotBoundException occurred while Polling the master node");

        }

        System.out.println("STATUS from MASTER NODE for downloaded file " + filename + " :
" + status1Table.get(filename));

    }

};

// Schedule the polling task to run at fixed intervals based on TTR

final ScheduledFuture<?> leafNodePollHandle =
scheduler.scheduleAtFixedRate(leafNodePoll, 1, timeToRef, TimeUnit.SECONDS);

// Executor to check and cancel the polling task if the file is out of date

ScheduledExecutorService queueCancelCheckExecutor =
Executors.newSingleThreadScheduledExecutor();

final Runnable stopBeep = new Runnable() {

    public void run() {

        if (!status1Table.isEmpty()) {

```



```

        if (status1Table.get(filename).equalsIgnoreCase("File out of Date")) {
            flagTable.put(filename, "false");
            leafNodePollHandle.cancel(true); // Cancel the polling task
        }
    }
}

};

// Schedule the stopBeep task to check the file status every 500 milliseconds
queueCancelCheckExecutor.scheduleAtFixedRate(stopBeep, 1, 500,
TimeUnit.MILLISECONDS);

}

/*----- end change -----*/
}

```

MultiClient.java

Description:

The `MultiClient` class serves as the entry point for initializing and running a `LeafNode` in the P2P network, similar to the `LeafNode` class. However, instead of using the standard `LeafNodeImpl` implementation, it utilizes the `AvgRespFileSearch` class to handle file operations. This setup allows for potentially enhanced or specialized behaviors in file searching and downloading, such as optimizing for average response times or implementing alternative search strategies. Upon execution, the program prompts the user to input the `LeafNode` (peer) ID, retrieves the corresponding port number, directories, and associated `SuperPeer` ID from the configuration, sets up the RMI registry on the specified port, and binds the `AvgRespFileSearch` implementation to the registry. This configuration enables the `LeafNode` to handle remote method invocations from `SuperPeers`, facilitating file operations and maintaining file consistency through advanced mechanisms.

```
package com.gfiletransfer;
```

```
import java.io.IOException;
```

```
import java.rmi.RemoteException;
```

```
import java.rmi.registry.LocateRegistry;
```

```
import java.rmi.registry.Registry;
```

```
import java.rmi.server.UnicastRemoteObject;
```

```
import java.util.ArrayList;
```

```
import java.util.Scanner;
```

```
/**
```

```
 * The MultiClient class initializes and starts a LeafNode in the P2P network using the
```

```
 * AvgRespFileSearch implementation. It sets up the RMI registry, binds the LeafNode  
implementation,
```

```
 * and makes the LeafNode available for remote method invocations by SuperPeers.
```

```
 */
```

```
public class MultiClient {
```

```
    /**
```

* The main method serves as the entry point for the MultiClient application.

*

* @param args Command-line arguments (not used).

* @throws RemoteException if there is an error during RMI operations.

*/

```
public static void main(String[] args) throws RemoteException {

    Scanner sc = new Scanner(System.in); // Scanner for user input

    String portno = null;

    String directoryName = null;

    String superPeerId = null;

    String cachedDirectoryName = null;


    // Prompt user to enter Peer ID

    System.out.println("Enter Peer ID ");

    String peerId = sc.nextLine(); // Capture LeafNode ID


    // Reading configuration details from SetupConfig

    SetupConfig scg;

    try {

        scg = new SetupConfig(); // Initialize SetupConfig to load configurations

        // Retrieve LeafNode configuration based on Peer ID

        for (GetPeerDetails p : scg.arrPD) {

            if (p.getPeer_ID().equalsIgnoreCase(peerId)) {

                portno = p.getPeer_Port();          // LeafNode port number

                directoryName = p.getDir();          // Directory for master files

                cachedDirectoryName = p.getCacheDir(); // Directory for cached files
```

```

        superPeerId = p.getSuperPeer();    // Associated SuperPeer ID
        break;
    }
}
} catch (IOException e1) {
    System.out.println("IOException occurred while reading the property file at Leaf Node
Initialization.");
    e1.printStackTrace();
    sc.close(); // Close the scanner before exiting
    return;    // Exit the program if configuration loading fails
}

/*----- start change -----*/

// Registering the peer on the specified port & setting up the remote object
Registry registry = LocateRegistry.createRegistry(Integer.parseInt(portno));

// Create the LeafNode implementation object using AvgRespFileSearch
AvgRespFileSearch InImpl = new AvgRespFileSearch(portno, directoryName, superPeerId,
peerId, cachedDirectoryName);

/*----- end change -----*/

// Export the LeafNodeImpl object to receive remote method invocations
LeafNodeInterface InInter = (LeafNodeInterface)
UnicastRemoteObject.exportObject(InImpl, 0);

// Bind the LeafNode implementation to the RMI registry with a unique name
registry.rebind("root://LeafNode/" + portno + "/FS", InInter);

```

```

        System.out.println("Peer is up and Running.");

        // Start the LeafNode's main operations
        try {
            InImpl.doWork(); // Invoke the main operational method
        } catch (IOException e) {
            System.out.println("IO Exception at Leaf Node Main: " + e.getMessage());
            e.printStackTrace();
        }

        sc.close(); // Close the scanner as it's no longer needed
    }
}

```

AvgRespFileSearch.java

Description:

The AvgRespFileSearch class implements the LeafNodeInterface and extends the functionalities of a standard LeafNode by incorporating mechanisms to calculate and display the percentage of invalid query results. This enhancement aims to provide insights into the reliability and accuracy of search responses within the P2P network. The class manages file operations such as downloading, searching, deleting, and editing files. Additionally, it handles file consistency through both push-based and pull-based mechanisms, ensuring that cached copies of files remain synchronized with their master copies. The implementation utilizes Java RMI for remote method invocations, enabling seamless communication with SuperPeers and other LeafNodes. Scheduled tasks are employed to manage polling for file version updates and to handle invalidation processes when discrepancies are detected.

```

package com.gfiletransfer;

import java.io.File;

import java.io.FileOutputStream;

```

```
import java.io.FileWriter;

import java.io.IOException;

import java.nio.file.Files;

import java.rmi.NotBoundException;

import java.rmi.RemoteException;

import java.rmi.registry.LocateRegistry;

import java.rmi.registry.Registry;

import java.text.DateFormat;

import java.text.DecimalFormat;

import java.text.NumberFormat;

import java.text.SimpleDateFormat;

import java.time.format.DateTimeFormatter;

import java.util.ArrayList;

import java.util.Collection;

import java.util.Date;

import java.util.HashMap;

import java.util.Map;

import java.util.Scanner;

import java.util.concurrent.CountDownLatch;

import java.util.concurrent.ExecutionException;

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;

import java.util.concurrent.Future;

import java.util.concurrent.ScheduledExecutorService;

import java.util.concurrent.ScheduledFuture;

import java.util.concurrent.ScheduledThreadPoolExecutor;
```

```
import java.util.concurrent.TimeUnit;

import java.util.concurrent.TimeoutException;
```

```
import javax.ws.rs.core.MultivaluedHashMap;

import javax.ws.rs.core.MultivaluedMap;
```

```
import java.util.Timer;

import java.util.Map.Entry;
```

```
/**
```

```
 * The AvgRespFileSearch class implements the LeafNodeInterface and provides enhanced
 * functionalities for a LeafNode in the P2P network. It manages file operations such as
 * downloading, searching, deleting, and editing files. Additionally, it handles file
 * consistency through both push-based and pull-based mechanisms, ensuring that cached
 * copies of files remain up-to-date with their master copies. This implementation also
 * calculates and displays the percentage of invalid query results to assess the reliability
 * of search responses.
```

```
 */
```

```
public class AvgRespFileSearch implements LeafNodeInterface {
```

```
    // Configuration and state variables
```

```
    String portNo = null; // Port number of the LeafNode
```

```
    String dirName = null; // Directory where master files are stored
```

```
    String cachedDirName = null; // Directory where downloaded/cached files are stored
```

```
    String fileName = null; // Name of the file to be searched
```

```
    String remotePeer = null; // Peer ID from whom the file is to be downloaded
```

```

String superpeer = null; // SuperPeer ID associated with this LeafNode

String peerID = null; // LeafNode ID

int seqNum = -1; // Sequence number for queries

int broadSeqNum = -1; // Sequence number for broadcasts

int timeTL = 20; // Time-To-Live for queries (3 for All-to-All, 22 for Linear Topology)

String msgId = null; // Message ID for queries

String consistencyType = null; // Type of consistency mechanism (PUSH, PULL1, PULL2)

String verString = null; // Version string for files


// Scheduled executor services and flags for managing consistency

final static ScheduledExecutorService scheduler = Executors.newScheduledThreadPool(1);

static ScheduledThreadPoolExecutor executor = new ScheduledThreadPoolExecutor(15); //
Thread pool for scheduled tasks

static ScheduledFuture<?> t; // Reference to a scheduled task

public static Map<String, String> flagTable = new HashMap<String, String>(); // Flags for
consistency checks

public static Map<String, String> status1Table = new HashMap<String, String>(); // Status
table for file versions

static String status1 = null; // Temporary status holder


// Data structures for storing search results and cached files

private MultivaluedMap<String, ArrayList<String>> finalHM = new MultivaluedHashMap<>();
// HashMap to store query results

private Map<String, ArrayList<String>> cachedTable = new HashMap<String,
ArrayList<String>>(); // Cached files table


/**
 * Constructor to initialize the AvgRespFileSearch with necessary configuration details.

```


*

* @param portNo The port number of the LeafNode.

* @param dirName The directory where master files are stored.

* @param superpeer The SuperPeer ID associated with this LeafNode.

* @param peerID The LeafNode ID.

* @param cachedDirName The directory where downloaded/cached files are stored.

*/

```
AvgRespFileSearch(String portNo, String dirName, String superpeer, String peerID, String
cachedDirName){
```

```
    this.portNo = portNo;
```

```
    this.dirName = dirName;
```

```
    this.superpeer = superpeer;
```

```
    this.peerID = peerID;
```

```
    this.cachedDirName = cachedDirName;
```

```
}
```

```
// #####
```

```
/**
```

```
* The main operational method for the AvgRespFileSearch LeafNode. It handles user
* interactions for searching, deleting, and editing files, and manages file registration
* and consistency mechanisms.
```

```
*
```

```
* @throws IOException if an I/O error occurs during operations.
```

```
*/
```

```
public void doWork() throws IOException {
```

```

String superPeerPort = null;

// Reading SuperPeer Port details from configuration for connecting to the indexing server
(SuperPeer)

SetupConfig scg;

try {

    scg = new SetupConfig(); // Initialize SetupConfig to load configurations

    // Retrieve SuperPeer port number based on SuperPeer ID

    for (GetSuperPeerDetails sp : scg.arrSPD){

        if(sp.getPeer_ID().equalsIgnoreCase(this.superpeer)){

            superPeerPort = sp.getPeer_Port();

            break;

        }

    }

    consistencyType = scg.consisApp; // Retrieve consistency mechanism type

}

catch (IOException e1) {

    System.out.println("IOException occurred while reading the property file at connecting to
Super Peer.");

    e1.printStackTrace();

    return; // Exit if configuration loading fails

}

try{

    // Locate the RMI registry of the SuperPeer

    Registry regis = LocateRegistry.getRegistry("localhost", Integer.parseInt(superPeerPort));

    SuperPeerInterface splInter = (SuperPeerInterface) regis.lookup("root://SuperPeer/" +
superPeerPort);

```

```

Scanner sc = new Scanner(System.in); // Scanner for user input

// Obtain list of files in the master directory
File dirList = new File(dirName);
String[] record = dirList.list();

// Initialize versioning for master copies
int versionNumEdit = 1;
verString = "v" + String.format("%02d", versionNumEdit);

// Registering all master files with the SuperPeer
for(int c = 0; c < record.length; c++){
    File currentFile = new File(record[c]);

    System.out.println("Registering details of File name " + currentFile.getName() + " in
Indexing Server");

    String timeStamp = new SimpleDateFormat("yyyy.MM.dd.HH.mm.ss").format(new
Date());

    spInter.registryFiles("new", currentFile.getName(), peerID, portNo, dirName,
superpeer, "MC", verString, "valid", timeStamp, "30", peerID);
}

// User interaction loop for Search, Delete, Edit, or Exit

System.out.println("Do you want to Search a File, Delete File, Edit file or Exit?
(Search/Delete/Edit/Exit)");

String sd = sc.nextLine();

while(!sd.equalsIgnoreCase("Exit")){
    if(sd.equalsIgnoreCase("Delete")){

```

```

// Deleting files from the local directory and updating the SuperPeer
String wantToDel = "";
while(!wantToDel.equalsIgnoreCase("No")){
    System.out.println("Enter the file name which you want to delete");
    String fname = sc.nextLine();
    if(fname != null){
        File fileToDel = new File(dirName + "/" + fname);
        // Attempt to delete the specified file
        if(fileToDel.delete()){
            System.out.println("File deleted Successfully.");
            // Notify the SuperPeer about the deletion
            splInter.registryFiles("del", fname, peerID, portNo, dirName, superpeer, "", "",
"", "", peerID);
        }
        else{
            System.out.println("Failed to delete the File");
        }
    }
    else{
        System.out.println("Please Enter a Filename");
    }
    System.out.println("Do you want to delete more files? (Yes/No)");
    wantToDel = sc.nextLine();
}
}

```

```

//EDIT PART START
if(sd.equalsIgnoreCase("Edit")){
    // Editing files in the local directory and updating the SuperPeer
    String wantToEdit = "";
    while(!wantToEdit.equalsIgnoreCase("No")){
        System.out.println("Enter the file name which you want to Edit(Append)");
        String fname = sc.nextLine();
        if(fname != null){
            File fileToEdit = new File(dirName + "/" + fname);

            System.out.println("Enter anything you want to append in this file");
            String appendString = sc.nextLine();

            FileWriter fw = new FileWriter(dirName + "/" + fname, true);
            fw.write(appendString);
            fw.close();
            System.out.println("File edited Successfully.");

            // Retrieve the current version number from the SuperPeer
            verString = splInter.getVersionNum(fname, peerID, portNo, "MC", peerID);

            if(verString.equalsIgnoreCase("-1")){
                System.out.println("File not found under Peer "+ peerID + " in Registry
Index");
            }

            // If the master copy is found, update the version number

```

```

else{

    // Increment the version number

    int intvernum = Integer.parseInt(verString.substring(1)); // Extract numeric
part
    intvernum += 1;

    verString = "v" + String.format("%02d", intvernum);

    System.out.println("New Version Number for edited file : " + verString);

    try{

        String timeStamp = new
SimpleDateFormat("yyyy.MM.dd.HH.mm.ss").format(new Date());

        // Update the file details in the SuperPeer

        spInter.registryFiles("upd", fname, peerID, portNo, dirName, superpeer,
"MC", verString, "valid", timeStamp, "", peerID);

    }catch(Exception e){

        System.out.println("Exception occurred: Updating the Register Index at
Super Peer");

    }

    // Handle consistency based on the configured type

    // Send invalidate request to all nodes (PUSH)

    if(consistencyType.equalsIgnoreCase("PUSH")){

        System.out.println("Send invalidate request to all nodes (PUSH)");

        broadSeqNum = broadSeqNum + 1;

        String messId = peerID + ":" + superpeer + ":" + String.format("%02d",
broadSeqNum);

        spInter.broadCastSP(messId, fname, peerID, verString);

    }

}
}

```

```

    }
    else{
        System.out.println("Please Enter a Filename");
    }
    System.out.println("Do you want to edit more files? (Yes/No)");
    wantToEdit = sc.nextLine();
}
}
// EDIT PART END

else if(sd.equalsIgnoreCase("Search")){
    // Searching and downloading a file from the network
    String ans = "";

    while(!ans.equalsIgnoreCase("No")){
        // Initiate a new search query
        seqNum = seqNum + 1;
        System.out.println("Enter the file name which you want to search");
        fileName = sc.nextLine();
        if(fileName != null){
            msgId = peerID + ":" + Integer.toString(seqNum);

            //
            peerID, portNo);
            splInter.query(msgId, timeTL, fileName,

            // Execute the query with a timeout

```

```

ExecutorService service = Executors.newSingleThreadExecutor();

try {
    Runnable r = new Runnable() {
        @Override
        public void run() {
            try {
                System.out.println("Now Started Calling the query() from Leaf
Node...");

                splInter.query(msgId, timeTL, fileName, peerID, portNo);

                //System.out.println("Still running the run method");
            } catch (RemoteException e) {
                // Handle remote exception during query
                System.out.println("TimeOut: It ran too long. Need to stop searching
and continue.");
            }
        }
    };

    Future<?> f = service.submit(r);

    f.get(4, TimeUnit.SECONDS); // Attempt the search for 4 seconds
}

catch (final InterruptedException e) {
    // Handle thread interruption
    System.out.println("Interrupted Exception Occured");
}

catch (final TimeoutException e) {
    // Handle timeout

```



```
        System.out.println("TimeOut Exception Occured. It ran too long. Need to stop  
searching and continue.");
```

```
    }
```

```
    catch (final ExecutionException e) {
```

```
        // Handle other execution exceptions
```

```
        System.out.println("Execution Exception Occured");
```

```
    }
```

```
    finally {
```

```
        service.shutdownNow(); // Shutdown the executor service
```

```
    }
```

```
}
```

```
else{
```

```
    System.out.println("Please Enter a Filename");
```

```
}
```

```
// Retrieve the search results from finalHM
```

```
Collection<ArrayList<String>> finalRes = this.finalHM.get(msgId);
```

```
String refVerNum = null;
```

```
String lmTime = null;
```

```
String refValid = null;
```

```
// 0 -
```

```
filename,1-peerid,2-portno,3-direct,4-superpeerid,5-copytype,6-vernum,7-status,8-lmt,9-TTR,1  
0-ogPeerId
```

```
if(!finalRes.isEmpty()){
```

```
    // Taking Version Number of Master Copy to check validity of Cached copies
```

```

for(ArrayList<String> as : finalRes){
    if(as.get(0).equalsIgnoreCase(fileName)){
        if(as.get(5).equalsIgnoreCase("MC")){
            refVerNum = as.get(6);

            lmTime = as.get(8);

            refValid = as.get(7);

            break;
        }
    }
}

/*----- start change -----*/

// Compute and display the percentage of invalid query results
double invalid_count = 0.0;

for (ArrayList<String> as1 : finalRes){

    DateFormat dtf = new SimpleDateFormat("yyyy.MM.dd.HH.mm.ss");

    int val = dtf.parse(as1.get(8)).compareTo(dtf.parse(lmTime));

    //System.out.println("Value in percentage method is" + val +" VALUES ARE : "+
as1.get(8) + lmTime);

    if(!(as1.get(7).equalsIgnoreCase(refValid))){

        invalid_count++;

    }

}

double percentage = (invalid_count / (double)finalRes.size()) * 100.0;

System.out.println("Percentage of Invalid query results : "+ invalid_count + " and
% "+percentage);

/*----- end change -----*/

```

```

        // Display the list of peers providing the requested file

System.out.println("#####");

        System.out.println("\n");

        for(ArrayList<String> als : finalRes){

            if(als.get(0).equalsIgnoreCase(fileName)){

                if(als.get(5).equalsIgnoreCase("MC")){

                    System.out.println("VALID Peer providing the file with Peer ID is "+
als.get(1)+ " under Super Peer : " + als.get(4) + " which is a Master Copy");

                }

                else{

                    // Validate cached copies based on version number

                    if(als.get(6).equalsIgnoreCase(refVerNum)){

                        System.out.println("VALID Peer providing the file with Peer ID is "+
als.get(1)+ " under Super Peer : " + als.get(4) + " which is a Cached Copy");

                    }

                }

            }

        }

        System.out.println("\n");

System.out.println("#####");

        // Prompt user to select a peer to download the file from

        System.out.println("Enter Peer ID you wish to take the file from");

        remotePeer = sc.nextLine();

        // Download the file from the selected peer

```

```

int co = finalRes.size();

if(remotePeer != null){

    for(ArrayList<String> als : finalRes){

        if(als.get(1).equalsIgnoreCase(remotePeer)){

            // Locate the RMI registry of the selected peer

            Registry regis2 = LocateRegistry.getRegistry("localhost",
Integer.parseInt(als.get(2)));

            LeafNodeInterface lnInter = (LeafNodeInterface)
regis2.lookup("root://LeafNode/" + als.get(2) + "/FS");

            // If chosen node is Master copy node, then direct download

            if(als.get(5).equalsIgnoreCase("MC")){

                byte[] output = lnInter.fileDownload(als);

                System.out.println(output.length);

                // Convert the downloaded byte array into a file

                if(output.length != 0){

                    FileOutputStream ostream = null;

                    try {

                        ostream = new FileOutputStream(cachedDirName + "/" +
fileName);

                        ostream.write(output);

                        System.out.println("File Downloading Successful.");

                        System.out.println("Display File " + fileName);

                        // Update the SuperPeer's index after downloading the file

                        splInter.registryFiles("new", fileName, peerID, portNo,
cachedDirName, superpeer, "CC", als.get(6), "valid", als.get(8), als.get(9), als.get(10));

```

```

        // Add the downloaded file details to the cached table

        // 0-
filename,1-copy_type,2-cachedDirectory,3-versionNumber,4-OriginServerId,5-status,6-TTR

        ArrayList<String> ct = new ArrayList<String>();
        ct.add(fileName); // filename
        ct.add("CC"); // copy_type
        ct.add(cachedDirName); // cachedDirectory
        ct.add(als.get(6)); // versionNumber
        ct.add(als.get(10)); // OriginServerId
        ct.add("valid"); // status
        ct.add(als.get(9)); // TTR
        this.cachedTable.put(fileName, ct);

        // Display the updated cached table

        System.out.println("Updated Cached Table Entry after insertion
(File download)");

        for (Entry<String, ArrayList<String>> entry :
this.cachedTable.entrySet()) {

            System.out.println(entry.getKey() + " => " + entry.getValue());

        }

        // Polling implementation starts here

        if(consistencyType.equalsIgnoreCase("PULL1")){

            AvgRespFileSearch.flagTable.put(fileName, "true");

            // Start the consistent pull process

            consistentPull(als.get(10), fileName, als.get(6), als.get(9));

```

```

System.out.println("Line 5");

// Define a task to check the polling status
class MyTask implements Runnable {

    public void run() {

        //System.out.println(flagTable);          //DEBUG
        if (flagTable.get(fileName).equalsIgnoreCase("false")) {

            // Update the cached table and SuperPeer's index
            System.out.println("Consistency check completed. File is
out of date.");

            try {

                // Mark the file as invalid in the SuperPeer's index
                splInter.registryFiles("upd", fileName, peerID, portNo,
cachedDirName, superpeer, "CC", als.get(6), "invalid", als.get(8), als.get(9), als.get(10));

            } catch (RemoteException e) {

                System.out.println("Exception occurred while updating
the Index");

            }

            // Invalidate the cached file in the local cached table
            InInvalidate(fileName, als.get(10));

            t.cancel(false); // Cancel the scheduled task as it's no
longer needed

        }

        else{

            //      System.out.println("DEBUG OF ELSE"); //DEBUG

        }

    }

}

System.out.println("Line 6");

```

```

        // Schedule the task to run every second
        t = executor.scheduleAtFixedRate(new MyTask(), 0, 1,
TimeUnit.SECONDS);

        System.out.println("Line 7");
    }
    else if(consistencyType.equalsIgnoreCase("PULL2")){
        // Handle PULL2 consistency by notifying the SuperPeer
        if(!this.cachedTable.isEmpty()){
            if(this.cachedTable.containsKey(fileName)){
                ArrayList<String> ct1 = this.cachedTable.get(fileName);
                splInter.notifyPoll(fileName, peerID, portNo, ct1.get(1),
ct1.get(3), ct1.get(4), ct1.get(6), als.get(8));
            }
        }
    }
}
catch(Exception e){
    System.out.println("Exception in bytearray to file conversion: " +
e.getMessage());
}
finally {
    if(ostream != null){
        ostream.close(); // Ensure the output stream is closed
    }
}
}
else{

```

```

        System.out.println("File is not present at Remote Location.");
    }

    break; // Exit the loop after successful download
}

// If chosen node is cached copy, then validate before downloading
else{
    if(InInter.getStatus(als.get(0)).equalsIgnoreCase("valid")){
        // Download the cached file
        byte[] output = InInter.fileDownload(als);
        System.out.println(output.length);

        // Convert the downloaded byte array into a file
        if(output.length != 0){
            FileOutputStream ostream = null;
            try {
                ostream = new FileOutputStream(cachedDirName + "/" +
fileName);

                ostream.write(output);
                System.out.println("File Downloading Successful.");
                System.out.println("Display File " + fileName);

                // Update the SuperPeer's index after downloading the file
                splInter.registryFiles("new", fileName, peerID, portNo,
cachedDirName, superpeer, "CC", als.get(6), "valid", als.get(8), als.get(9), als.get(10));

                // Add the downloaded file details to the cached table
                ArrayList<String> ct = new ArrayList<String>();

```



```

        ct.add(fileName); // filename
        ct.add("CC"); // copy_type
        ct.add(cachedDirName); // cachedDirectory
        ct.add(als.get(6)); // versionNumber
        ct.add(als.get(10)); // OriginServerId
        ct.add("valid"); // status
        ct.add(als.get(9)); // TTR
        this.cachedTable.put(fileName, ct);
        // Display the updated cached table
        System.out.println("Updated Cached Table Entry after insertion
(File download)");

        for (Entry<String, ArrayList<String>> entry :
this.cachedTable.entrySet()) {
            System.out.println(entry.getKey() + " => " + entry.getValue());
        }

        // Polling implementation starts here
        if(consistencyType.equalsIgnoreCase("PULL1")){
            this.flagTable.put(fileName, "true");
            // Start the consistent pull process
            consistentPull(als.get(10), fileName, als.get(6), als.get(9));

            System.out.println("Line 5");
            // Define a task to check the polling status
            class MyTask implements Runnable {
                public void run() {
                    //System.out.println(flagTable); //////////// DEBUG

```

```

        if (flagTable.get(fileName).equalsIgnoreCase("false")) {
            // Update the cached table and SuperPeer's index
            System.out.println("Consistency check completed. File
is out of date.");

            try {
                // Mark the file as invalid in the SuperPeer's index
                splInter.registryFiles("upd", fileName, peerID, portNo,
cachedDirName, superpeer, "CC", als.get(6), "invalid", als.get(8), als.get(9), als.get(10));
            } catch (RemoteException e) {
                System.out.println("Exception occurred while
updating the Index");
            }

            // Invalidate the cached file in the local cached table
            lnInvalidate(fileName, als.get(10));

            t.cancel(false); // Cancel the scheduled task as it's no
longer needed
        }
    }
}

System.out.println("Line 6");

// Schedule the task to run every second
t = executor.scheduleAtFixedRate(new MyTask(), 0, 1,
TimeUnit.SECONDS);

System.out.println("Line 7");
}

else if(consistencyType.equalsIgnoreCase("PULL2")){
    // Handle PULL2 consistency by notifying the SuperPeer
    if(!this.cachedTable.isEmpty()){

```

```

        if(this.cachedTable.containsKey(fileName)){

            ArrayList<String> ct1 = this.cachedTable.get(fileName);

            splInter.notifyPoll(fileName, peerID, portNo, ct1.get(1),
ct1.get(3), ct1.get(4), ct1.get(6), als.get(8));

        }

    }

}

}

catch(Exception e){

    System.out.println("Exception in bytearray to file conversion: " +
e.getMessage());

}

finally {

    if(ostream != null){

        ostream.close(); // Ensure the output stream is closed

    }

}

}

else{

    System.out.println("File is not present at Remote Location.");

}

break; // Exit the loop after successful download

}

else{

    System.out.println("The Peer which you had selected just got its file
invalidated. Please select Master Copy for guaranteed file download.");

```

```

        }
    }
}
else{
    if(co == 1)
        System.out.println("Peer with that ID " + remotePeer + " does not exist.
Please choose proper PeerId.");
    }
    co--;
}
}
else{
    System.out.println("Please enter proper Peer ID");
}
}
else{
    System.out.println("Sorry, File which you are searching doesn't exist in our
Server.");
}
    System.out.println("Do you want to search again ? (Yes/No)");
    ans = sc.nextLine();
}
}
else{
    System.out.println("Please select appropriate choice");
}
}

```

```
        System.out.println("Do you want to Search a File, Delete File or Exit?  
(Search/Delete/Exit)");
```

```
        sd = sc.nextLine();
```

```
    }
```

```
    System.exit(0); // Terminate the application
```

```
} catch(Exception e) {
```

```
    System.out.println("Exception at Client Interface: " + e.getMessage());
```

```
    e.printStackTrace();
```

```
}
```

```
}
```

```
/**
```

```
 * Downloads a file from the specified directory paths.
```

```
 *
```

```
 * @param searchedDir A list containing directory paths where the file is located.
```

```
 * @return A byte array representing the downloaded file.
```

```
 * @throws RemoteException if a remote communication error occurs.
```

```
 */
```

```
@Override
```

```
public byte[] fileDownload(ArrayList<String> searchedDir) throws RemoteException{
```

```
    // Extract filename and remote directory from the searchedDir list
```

```
    String fname = searchedDir.get(0);
```

```
    String remoteDir = searchedDir.get(3);
```

```
    try {
```

```
        File file = new File(remoteDir + "/" + fname);
```

```
        // Check if the file exists
```

```

        if(file.exists()){
            byte buffer[] = Files.readAllBytes(file.toPath()); // Read file bytes
            return buffer; // Return the file as a byte array
        }
    }
    catch(Exception e){
        System.out.println("Error in File download part: " + e.getMessage());
        e.printStackTrace();
        return new byte[0]; // Return an empty byte array in case of error
    }
    return new byte[0]; // Return an empty byte array if file does not exist
}

/**
 * Handles the response to a search query by storing the search results.
 *
 * @param msgId    The unique message ID for the query.
 * @param TTL      Time-To-Live remaining for the query message.
 * @param filename The name of the file being searched for.
 * @param resultArr A collection of search results containing file details.
 * @return A boolean indicating the success of handling the query hit.
 * @throws RemoteException if a remote communication error occurs.
 */
@Override
public synchronized boolean queryHit(String msgId, int TTL, String filename,
Collection<ArrayList<String>> resultArr)

```

```

        throws RemoteException {
    if(TTL > 0 && TTL != 0){
        try{

            // Add the search results to the finalHM HashMap

            for(ArrayList<String> arrFileDtl : resultArr){

                this.finalHM.add(msgId, arrFileDtl);

            }

            return true; // Indicate successful handling

        }catch(Exception e){

            System.out.println("Exception at Peer's Interface: " + e.getMessage());

            return false; // Indicate failure

        }

    }

    else{

        System.out.println("Time to Live of a Message has expired at Leaf Node. This Message is
no longer valid.");

        return false; // Indicate failure due to TTL expiry

    }

}

/**
 * Polls the SuperPeer to check if the cached file version is up-to-date.
 *
 * @param filename The name of the file to check.
 * @param verNum The version number of the cached file.
 * @return A string indicating the status of the file ("Proper version" or "File out of Date").

```

```

* @throws RemoteException    if a remote communication error occurs.
* @throws NotBoundException  if a remote object is not bound in the registry.
*/
@Override

public String poll(String filename, String verNum) throws RemoteException,
NotBoundException {

    // Retrieve the SuperPeer port number from configuration

    String spPort = null;

    SetupConfig scg;

    try {

        scg = new SetupConfig();

        // Find the SuperPeer port based on SuperPeer ID

        for (GetSuperPeerDetails sp : scg.arrSPD){

            if(sp.getPeer_ID().equalsIgnoreCase(this.superpeer)){

                spPort = sp.getPeer_Port();

                break;

            }

        }

    }

    catch (IOException e1) {

        System.out.println("IOException occurred while reading the property file in Polling
function");

        e1.printStackTrace();

        return "Error: Unable to read configuration.";

    }

    // Locate the RMI registry of the SuperPeer

```



```

Registry regis = LocateRegistry.getRegistry("localhost", Integer.parseInt(spPort));

SuperPeerInterface splInter = (SuperPeerInterface) regis.lookup("root://SuperPeer/" +
spPort);

// Search for the file in the SuperPeer's index

Collection<ArrayList<String>> resultArrFile = splInter.searchFile(filename);

// 0 -
filename,1-peerid,2-portno,3-direct,4-superpeerid,5-copytype,6-vernum,7-status,8-lmt,9-TTR,1
0-ogPeerId

String retValue = "File not found at Master Node";

if(!resultArrFile.isEmpty()){
    for (ArrayList<String> asr: resultArrFile){
        // Check if the entry corresponds to the master copy of the file
        if(asr.get(1).equalsIgnoreCase(this.peerID) && asr.get(5).equalsIgnoreCase("MC")){
            if(asr.get(6).equalsIgnoreCase(verNum)){
                retValue = "Proper version";
            }
            else{
                retValue = "File out of Date";
            }
            break;
        }
    }
    return retValue; // Return the status of the file
}
else{

```

```

        return retValue; // Return the status if the file was not found
    }
}

/**
 * Invalidates a specific file by marking its status as "invalid" in the cached table.
 *
 * @param filename    The name of the file to invalidate.
 * @param originLNserver The originating LeafNode server ID.
 * @param verNum      The version number of the file being invalidated.
 * @throws RemoteException if a remote communication error occurs.
 */
@Override
// filename5
public void invalidate(String filename, String originLNserver, String verNum) throws
RemoteException {
    // Check if the file exists in the cached table
    if (this.cachedTable.containsKey(filename)) {
        // Iterate over the cached table to find and invalidate the specific file entry
        for(String key : this.cachedTable.keySet()) {
            if(key.equalsIgnoreCase(filename)) {
                ArrayList<String> ls = this.cachedTable.get(key);
                // Compare the origin server ID and version number to determine if invalidation is
needed
                if(ls.get(4).equalsIgnoreCase(originLNserver) &&
!(ls.get(3).equalsIgnoreCase(verNum))) {
                    ls.set(5, "invalid"); // Set the status to "invalid"

```

```

        this.cachedTable.remove(key);

        this.cachedTable.put(key, ls);
    }

    // If version numbers match, do nothing

    break;
}

}

// Display the updated cached table
System.out.println("Updated Cached Table Entry after updation");
for (Entry<String, ArrayList<String>> entry : this.cachedTable.entrySet()) {
    System.out.println(entry.getKey() + " => " + entry.getValue());
}
}

else {

    // If the file is not found in the cached table

    System.out.println("Entry not found in cached table");
}
}

/**
 * Marks a specific file as out of date by invalidating its cached copy.
 *
 * @param filename    The name of the file to mark as out of date.
 * @param invalidStatus The status to set for the file ("Out of Date").
 * @param ogPeerId    The original peer ID where the file originated.
 * @throws RemoteException if a remote communication error occurs.

```

```

*/

@Override

public void outOfDate(String filename, String invalidStatus, String ogPeerId) throws
RemoteException {

    if(invalidStatus.equalsIgnoreCase("Out of Date")){

        System.out.println("Received notification from Super Peer to invalidate the file: " +
filename);

        InInvalidate(filename, ogPeerId); // Invalidate the file in the cached table

    }

}

/**
 * Retrieves the current status of a specific file from the cached table.
 *
 * @param filename The name of the file whose status is to be retrieved.
 * @return A string representing the current status of the file ("valid", "invalid", or "-1" if not
found).
 * @throws RemoteException if a remote communication error occurs.
 */

@Override

public String getStatus(String filename) throws RemoteException {

    if (this.cachedTable.containsKey(filename)) {

        for(String key : this.cachedTable.keySet()) {

            if(key.equalsIgnoreCase(filename)) {

                ArrayList<String> ls = this.cachedTable.get(key);

                return ls.get(5); // Return the status of the file

            }

        }

    }

}

```

```

    }
}

return "-1"; // Return "-1" if the file is not found
}

/**
 * Invalidates a specific file in the cached table by setting its status to "invalid".
 *
 * @param filename    The name of the file to invalidate.
 * @param originLNserver The originating LeafNode server ID.
 */
public void InInvalidate(String filename, String originLNserver){
    if (this.cachedTable.containsKey(filename)) {
        // Iterate over the cached table to find and invalidate the specific file entry
        for(String key : this.cachedTable.keySet()) {
            if(key.equalsIgnoreCase(filename)) {
                ArrayList<String> ls = this.cachedTable.get(key);
                // Compare the origin server ID to determine if invalidation is needed
                if(ls.get(4).equalsIgnoreCase(originLNserver)) {
                    ls.set(5, "invalid"); // Set the status to "invalid"
                    this.cachedTable.remove(key);
                    this.cachedTable.put(key, ls);
                }
                // If origin server ID does not match, do nothing
                break;
            }
        }
    }
}

```

```

    }

    // Display the updated cached table

    System.out.println("Updated Cached Table Entry after Invalidation from Pull");

    for (Entry<String, ArrayList<String>> entry : this.cachedTable.entrySet()) {

        System.out.println(entry.getKey() + " => " + entry.getValue());

    }

}

else {

    // If the file is not found in the cached table

    System.out.println("Entry not found in cached table");

}

}

/**
 * Initiates a consistent pull to verify the file version from the master SuperPeer.
 *
 * @param masterPeerId The SuperPeer ID of the master node.
 * @param filename The name of the file being polled.
 * @param verNum The version number of the cached file.
 * @param TTR Time-To-Refresh value for pull-based consistency.
 * @throws RemoteException if a remote communication error occurs.
 * @throws NotBoundException if a remote object is not bound in the registry.
 */

public static void consistentPull(String masterPeerId, String filename, String verNum, String
TTR) throws RemoteException, NotBoundException {

```

```

int timeToRef = Integer.parseInt(TTR); // Time-To-Refresh in seconds

// Retrieve the port number of the master SuperPeer
String masterPortNum = null;
SetupConfig scg;
try {
    scg = new SetupConfig();
    for(GetPeerDetails gpd: scg.arrPD){
        if(gpd.getPeer_ID().equalsIgnoreCase(masterPeerId)){
            masterPortNum = gpd.getPeer_Port();
            break;
        }
    }
}
catch (IOException e1) {
    System.out.println("IOException occurred while reading the property file at Polling
function Pull 1");
    e1.printStackTrace();
    return; // Exit if configuration loading fails
}

// Locate the RMI registry of the master SuperPeer
Registry regis = LocateRegistry.getRegistry("localhost", Integer.parseInt(masterPortNum));

LeafNodeInterface plInter = (LeafNodeInterface) regis.lookup("root://LeafNode/" +
masterPortNum + "/FS");

// Define a task to poll the master SuperPeer for the file status
final Runnable leafNodePoll = new Runnable() {

```

```

public void run() {
    try {
        // Poll the master SuperPeer for the file status
        status1Table.put(filename, pInter.poll(filename, verNum));
    }
    catch (RemoteException e) {
        System.out.println("Remote Exception occurred while Polling the master node");
    } catch (NotBoundException e) {
        System.out.println("NotBoundException occurred while Polling the master node");
    }

    System.out.println("STATUS from MASTER NODE for downloaded file " + filename + " :
" + status1Table.get(filename));
}

};

// Schedule the polling task to run at fixed intervals based on TTR

final ScheduledFuture<?> leafNodePollHandle =
scheduler.scheduleAtFixedRate(leafNodePoll, 1, timeToRef, TimeUnit.SECONDS);

// Executor to check and cancel the polling task if the file is out of date

ScheduledExecutorService queueCancelCheckExecutor =
Executors.newSingleThreadScheduledExecutor();

final Runnable stopBeep = new Runnable() {
    public void run() {
        if(!status1Table.isEmpty()){
            if(status1Table.get(filename).equalsIgnoreCase("File out of Date")){
                flagTable.put(filename, "false");

                leafNodePollHandle.cancel(true); // Cancel the polling task
            }
        }
    }
}

```



```
        }  
    }  
}  
};  
  
// Schedule the stopBeep task to check the file status every 500 milliseconds  
queueCancelCheckExecutor.scheduleAtFixedRate(stopBeep,1, 500,  
TimeUnit.MILLISECONDS);  
  
}  
}
```