Apr 12, 18 3:44 **UserStatisticsViewer.java** Page 1/1

```java
import com.google.gson.Gson;
import com.rabbitmq.client.*;

import java.io.IOException;
import java.util.concurrent.TimeoutException;

public class UserStatisticsViewer extends RabbitMQProcess {

    public UserStatisticsViewer(String host) throws IOException, TimeoutException {
        super(host);

        // declare USERS_STATS exchange
        channel.exchangeDeclare(Configuration.UsersStatisticsExchange,
                BuiltinExchangeType.FANOUT);

        consumeStatistics();
    }

    private String consumeStatistics() throws IOException {
        String statisticsQueue = channel.queueDeclare().getQueue();
        channel.queueBind(statisticsQueue,
                Configuration.UsersStatisticsExchange, "");

        Consumer consumerStatistics = new DefaultConsumer(channel) {
            @Override
            public void handleDelivery(String consumerTag, Envelope envelope,
                                       AMQP.BasicProperties properties,
                                       byte[] body) throws IOException {

                String json = new String(body, "UTF-8");
                UsersSecondsListenedStatistics statistics = new Gson().fromJson
                        (json, UsersSecondsListenedStatistics.class);

                System.out.println(" [x] Showing connections per radio: ");
                for (UserSecondsListened userStats :
                        statistics.usersMostListenedSeconds) {
                    System.out.println(userStats.username + ":" +
                            userStats.secondsListened);
                }
            }
        };
        return channel.basicConsume(statisticsQueue, true, consumerStatistics);
    }

    public static void main(String[] argv) throws Exception {
        UserStatisticsViewer statisticsViewer =
                new UserStatisticsViewer(Configuration.RabbitMQHost);
    }
}
```

Apr 12, 18 3:51 **UsersSecondsListenedStatistics.java** Page 1/1

```java
import java.util.Collection;
import java.util.LinkedList;
import java.util.List;

public class UsersSecondsListenedStatistics {

    public List<UserSecondsListened> usersMostListenedSeconds;

    public UsersSecondsListenedStatistics(Collection<UserSecondsListened> stats)
 {
        this.usersMostListenedSeconds = new LinkedList<>(stats);
    }
}
```

Apr 12, 18 3:37 **UsersSecondsListenedComparator.java** Page 1/1

```java
import java.util.Comparator;


public class UsersSecondsListenedComparator
        implements Comparator<UserSecondsListened> {
    @Override
    public int compare(UserSecondsListened o1, UserSecondsListened o2) {

        if (o2.secondsListened < o1.secondsListened) {
            return -1;
        }
        if (o2.secondsListened > o1.secondsListened) {
            return 1;
        }
        return 0;
    }
}
```

Apr 12, 18 3:40 **UserSecondsListened.java** Page 1/1

```java
public class UserSecondsListened {

    public String username;
    public long secondsListened;

    public UserSecondsListened(String username, long secondsListened) {
        this.username = username;
        this.secondsListened = secondsListened;
    }
}
```

Apr 11, 18 21:38 **UsersDBRowRadioConnection.java** Page 1/1

```java
import java.util.Date;

public class UsersDBRowRadioConnection {

    public String radio;
    // para diferencias un usuario con varias conexiones a la misma radio
    public int connectionID;
    public Date keepAlive;

    public UsersDBRowRadioConnection(String radio, Date keepAlive,
                                     int connectionID){
        this.radio = radio;
        this.keepAlive = keepAlive;
        this.connectionID = connectionID;
    }
}
```

Apr 11, 18 23:20 **UsersDBRow.java** Page 1/1

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

public class UsersDBRow extends DatabaseRow {

    public String username;
    public List<UsersDBRowRadioConnection> connections = new LinkedList<>();
    public long secondsListening;
    public int connectionsLimit;

    public UsersDBRow(String username) {
        super(username);

        this.username = username;
        this.secondsListening = 0;
        this.connectionsLimit = Configuration.MaxConnectionsPerFreeUser;
    }
}
```

```java
import com.google.gson.Gson;
import com.rabbitmq.client.*;

import java.io.IOException;
import java.util.Date;
import java.util.LinkedList;
import java.util.List;
import java.util.ListIterator;
import java.util.concurrent.TimeoutException;

public class UsersDBHandler extends DBHandlerWithStatistics<UsersDBRow> {

    public UsersDBHandler(String host, Database<UsersDBRow> database) throws
            IOException,
            TimeoutException {
        super(host, database);

        // declare USERS_DB exchange
        channel.exchangeDeclare(Configuration.UsersDBExchange,
                BuiltinExchangeType.DIRECT);

        // declare USERS_STATS exchange
        channel.exchangeDeclare(Configuration.UsersStatisticsExchange,
                BuiltinExchangeType.FANOUT);

        // consumer methods
        consumeConnections();
        consumeDisconnections();
        consumeKeepAlive();
    }

    @Override
    protected List<Runnable> getStatisticsOperations() {
        List<Runnable> operations = new LinkedList<>();
        operations.add(new Runnable() {
            @Override
            public void run() {
                // get statistics
                LimitedSortedSet<UserSecondsListened> usersMostListened =
                        new LimitedSortedSet<>(Configuration.UserStatisticsN,
                                new UsersSecondsListenedComparator());
                for (UsersDBRow row : database.getRows()) {
                    UserSecondsListened userStats =
                            new UserSecondsListened(row.username,
                                    row.secondsListening);
                    usersMostListened.add(userStats);
                }

                UsersSecondsListenedStatistics stats =
                        new UsersSecondsListenedStatistics(usersMostListened);
                String jsonStats = new Gson().toJson(stats);

                try {
                    channel.basicPublish(Configuration.UsersStatisticsExchange,
                            "", null, jsonStats.getBytes());
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        });

        return operations;
    }

    @Override
    protected List<Integer> getStatisticsPeriodsSeconds() {
        List<Integer> operationsPeriods = new LinkedList<>();
        operationsPeriods.add(Configuration.UsersStatisticsPeriodSeconds);
        return operationsPeriods;
    }

    public String consumeConnections() throws IOException {
        // consume CONNECT_TO_RADIO requests
```

```java
        String connectUsersQueue = channel.queueDeclare().getQueue();
        channel.queueBind(connectUsersQueue, Configuration.UsersDBExchange,
                Configuration.UsersDBConnectTag);
        Consumer connectConsumer = new DefaultConsumer(channel) {
            @Override
            public void handleDelivery(String consumerTag, Envelope envelope,
                                       AMQP.BasicProperties properties,
                                       byte[] body) throws IOException {

                // parse request
                String jsonRequest = new String(body, "UTF-8");

                UserConnectRequest request = new Gson().fromJson(jsonRequest,
                        UserConnectRequest.class);
                UserConnectResponse response = new UserConnectResponse(request);

                // get user record from DB
                UsersDBRow user = database.getRow(request.username);
                if (user ≡ null) {
                    System.out.println("User: " + request.username + " not " +
                            "found, creating new user");
                    user = new UsersDBRow(request.username);
                }
                // first check if any connection is not active and remove it
                int connectionId = 0;
                Date now = new Date();
                ListIterator<UsersDBRowRadioConnection> connIter =
                        user.connections.listIterator();
                while(connIter.hasNext()){
                    UsersDBRowRadioConnection connection = connIter.next();

                    Date then = connection.keepAlive;
                    if (now.getTime() - then.getTime() >
                            Configuration.SecondsUntilDropConnection * 1000) {
                        UserDisconnectRequest disconnectRequest = new
                                UserDisconnectRequest(request.username,
                                connection.radio,
                                connection.connectionID);
                        response.closedConnections.add(disconnectRequest);
                        connIter.remove();

                        System.out.println(" [x] Closing old connection to: " +
                                connection.radio);
                    } else {
                        if (connection.connectionID > connectionId) {
                            connectionId = connection.connectionID;
                        }
                    }
                }
                connectionId = (connectionId + 1) %
                        Configuration.MaxConnectionsPerUnlimitedUser;
                // then check if user can connect to radio
                if (user.connections.size() < user.connectionsLimit) {
                    UsersDBRowRadioConnection connection =
                            new UsersDBRowRadioConnection(response.radio,
                                    new Date(), connectionId);
                    user.connections.add(connection);
                    response.couldConnect = true;
                    response.connectionId = connectionId;

                    System.out.println(" [x] User: " + user.username +
                            " connected to: " + request.radio);
                } else {
                    response.couldConnect = false;

                    System.out.println(" [x] User: " + user.username +
                            " not connected to: " + request.radio);
                }
                // update user
                database.updateRow(user);

                String jsonResponse = new Gson().toJson(response);
                channel.basicPublish("",
                        Configuration.ConnMgrUsersDBResponseQueue, null,
                        jsonResponse.getBytes());
            }
        };
        return channel.basicConsume(connectUsersQueue, true, connectConsumer);
    }

    public String consumeDisconnections() throws IOException {
        // consume DISCONNECT_FROM_RADIO requests
        String disconnectUsersQueue = channel.queueDeclare().getQueue();
        channel.queueBind(disconnectUsersQueue, Configuration.UsersDBExchange,
                Configuration.UsersDBDisconnectTag);
        Consumer disconnectConsumer = new DefaultConsumer(channel) {
            @Override
            public void handleDelivery(String consumerTag, Envelope envelope,
                                       AMQP.BasicProperties properties,
                                       byte[] body) throws IOException {

                // parse request
                String jsonRequest = new String(body, "UTF−8");

                UserDisconnectRequest request = new Gson().fromJson(jsonRequest,
                        UserDisconnectRequest.class);

                // get user record from DB
                UsersDBRow user = database.getRow(request.username);
                if (user ≠ null) {
                    ListIterator<UsersDBRowRadioConnection> connIter =
                            user.connections.listIterator();
                    while(connIter.hasNext()){
                        UsersDBRowRadioConnection connection = connIter.next();
                        if (connection.connectionID ≡ request.connectionId ∧
                                connection.radio.equals(request.radio)) {
                            connIter.remove();
                            System.out.println(" [x] Removing connection " +
                                    "from: " + user.username + " to radio: "
                                    + connection.radio);
                            break;
                        }
                    }
                }
                // update user
                database.updateRow(user);
            }
        };
        return channel.basicConsume(disconnectUsersQueue, true,
                disconnectConsumer);
    }

    public String consumeKeepAlive() throws IOException {
        // consume KEEP_ALIVE requests
        String keepaliveUsersQueue = channel.queueDeclare().getQueue();
        channel.queueBind(keepaliveUsersQueue, Configuration.UsersDBExchange,
                Configuration.UsersDBKeepAliveTag);
        Consumer keepaliveConsumer = new DefaultConsumer(channel) {
            @Override
            public void handleDelivery(String consumerTag, Envelope envelope,
                                       AMQP.BasicProperties properties,
                                       byte[] body) throws IOException {

                // parse request
                String jsonRequest = new String(body, "UTF−8");

                KeepAliveRequest request = new Gson().fromJson(jsonRequest,
                        KeepAliveRequest.class);

                // get user record from DB
                UsersDBRow user = database.getRow(request.username);
                if (user ≡ null) {
                    System.out.println(" [x] Error: user who sent keep alive " +
                            "does not exist");
                    return;
                }
```

```java

                // refresh keep alive
                ListIterator<UsersDBRowRadioConnection> connIter =
                        user.connections.listIterator();
                while(connIter.hasNext()){
                    UsersDBRowRadioConnection connection = connIter.next();
                    if (connection.connectionID ≡ request.connectionId ∧
                            connection.radio.equals(request.radio)) {
                        connection.keepAlive = new Date();
                        connIter.set(connection);
                        System.out.println(" [x] Refreshing keepalive " +
                                "from: " + user.username + " to radio: "
                                + connection.radio + " id: " +
                                connection.connectionID);
                        break;
                    }
                }

                // add to user total listened minutes
                user.secondsListening += Configuration.KeepAlivePeriodSeconds;

                // update user
                database.updateRow(user);
            }
        };
        return channel.basicConsume(keepaliveUsersQueue, true,
                keepaliveConsumer);
    }

    public static void main(String[] argv) throws Exception {

        // define database
        Database<UsersDBRow> database = new DatabaseRAM();

        // start database handler
        UsersDBHandler handler = new UsersDBHandler(Configuration.RabbitMQHost,
                database);
    }
}
```

```java
public class UserDisconnectRequest {

    // request
    public String username;
    public String radio;
    public int connectionId;

    public UserDisconnectRequest(String username, String radio, int
            connectionId) {
        this.username = username;
        this.radio = radio;
        this.connectionId = connectionId;
    }

    public String toLogLine() {
        return Configuration.LogsDisconnectionTag + " " + username + " " +
                radio + " " + connectionId;
    }
}
```

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

public class UserConnectResponse {

    // request
    public String username;
    public String radio;

    // id
    public String returnQueueName;

    // response
    public boolean couldConnect = false;
    public int connectionId;
    public List<UserDisconnectRequest> closedConnections = new LinkedList<>();

    public UserConnectResponse(UserConnectRequest request) {
        this.username = request.username;
        this.radio = request.radio;
        this.returnQueueName = request.returnQueueName;
    }

    public String toLogLine() {
        return Configuration.LogsConnectionTag + " " + username + " " +
                radio + " " + connectionId;
    }
}
```

```java

public class UserConnectRequest {

    // request
    public String username;
    public String radio;

    // id
    public String returnQueueName;

    public UserConnectRequest(String username, String radio,
                              String returnQueueName) {
        this.username = username;
        this.radio = radio;
        this.returnQueueName = returnQueueName;
    }
}
```

```java
public class RadiosUpdateRequest {

    public String radio;
    public String username;

    public RadiosUpdateRequest(String radio, String username) {
        this.radio = radio;
        this.username = username;
    }
}
```

Apr 12, 18 2:53 **RadioStatisticsViewer.java** Page 1/1

```java
import com.google.gson.Gson;
import com.rabbitmq.client.*;

import java.io.IOException;
import java.util.concurrent.TimeoutException;

public class RadioStatisticsViewer extends RabbitMQProcess {

    public RadioStatisticsViewer(String host) throws IOException,
            TimeoutException {
        super(host);

        // declare RADIOS_STATS exchange
        channel.exchangeDeclare(Configuration.RadiosStatisticsExchange,
                BuiltinExchangeType.FANOUT);

        consumeStatistics();
    }

    private String consumeStatistics() throws IOException {
        String statisticsQueue = channel.queueDeclare().getQueue();
        channel.queueBind(statisticsQueue,
                Configuration.RadiosStatisticsExchange, "");

        Consumer consumerStatistics = new DefaultConsumer(channel) {
            @Override
            public void handleDelivery(String consumerTag, Envelope envelope,
                                       AMQP.BasicProperties properties,
                                       byte[] body) throws IOException {

                String json = new String(body, "UTF-8");
                RadiosConnectionsStatistics statistics = new Gson().fromJson
                        (json, RadiosConnectionsStatistics.class);

                System.out.println(" [x] Showing connections per radio: ");
                for (String radio : statistics.radioConnections.keySet()) {
                    System.out.println(radio + ": " +
                            statistics.radioConnections.get(radio));
                }
            }
        };
        return channel.basicConsume(statisticsQueue, true, consumerStatistics);
    }

    public static void main(String[] argv) throws Exception {
        RadioStatisticsViewer statisticsViewer =
                new RadioStatisticsViewer(Configuration.RabbitMQHost);
    }
}
```

Apr 12, 18 13:55 **RadioSourceRandomNumbers.java** Page 1/1

```java
import java.util.concurrent.ThreadLocalRandom;

public class RadioSourceRandomNumbers implements RadioSource {


    @Override
    public void init() {

    }

    @Override
    public byte[] getNextByteBlock() {
        int randomNum = ThreadLocalRandom.current().nextInt(0, 100 + 1);
        System.out.println(" [x] Sent: " + randomNum);
        return Integer.toString(randomNum).getBytes();
    }

    @Override
    public void close() {

    }
}
```

Apr 12, 18 14:12 **RadioSource.java** Page 1/1

```java
import java.io.FileNotFoundException;

public interface RadioSource {

    void init() throws FileNotFoundException;

    byte[] getNextByteBlock();

    void close();
}
```

Apr 12, 18 14:29 **RadioSourceFile.java** Page 1/1

```java
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.util.Base64;

public class RadioSourceFile implements RadioSource {

    String filename;
    FileInputStream audio;
    int bytesPerRead;
    byte[] buffer;

    public RadioSourceFile(String filename, int bitrate) {
        this.filename = filename;
        this.bytesPerRead = 1000 * Configuration.RadioSendPeriodSeconds;
        buffer = new byte[this.bytesPerRead];
    }

    @Override
    public void init() throws FileNotFoundException {
        audio = new FileInputStream(filename);
    }

    @Override
    public byte[] getNextByteBlock() {
        try {
            audio.read(buffer);
            return Base64.getEncoder().encode(buffer);
        } catch (IOException e) {
            e.printStackTrace();
        }
        return "STATIC".getBytes();
    }

    @Override
    public void close() {

    }
}
```

Apr 11, 18 19:58 **RadiosDBRow.java** Page 1/1

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

public class RadiosDBRow extends DatabaseRow {
    public String name;
    public int connectedUsers;

    public RadiosDBRow(String name) {
        super(name);

        this.name = name;
        this.connectedUsers = 0;
    }
}
```

Apr 12, 18 3:24 **RadiosDBHandler.java** Page 1/2

```java
import com.google.gson.Gson;
import com.rabbitmq.client.*;

import java.io.IOException;
import java.util.LinkedList;
import java.util.List;
import java.util.concurrent.*;

public class RadiosDBHandler extends DBHandlerWithStatistics<RadiosDBRow> {

    public RadiosDBHandler(String host, Database<RadiosDBRow> database) throws
            IOException,
            TimeoutException {
        super(host, database);
        this.database = database;

        // declare RADIOS_DB exchange
        channel.exchangeDeclare(Configuration.RadiosDBExchange,
                BuiltinExchangeType.DIRECT);

        // declare RADIOS_STATS exchange
        channel.exchangeDeclare(Configuration.RadiosStatisticsExchange,
                BuiltinExchangeType.FANOUT);

        consumeConnections();
        consumeDisconnections();
    }

    @Override
    protected List<Runnable> getStatisticsOperations() {
        List<Runnable> operations = new LinkedList<>();
        operations.add(new Runnable() {
            @Override
            public void run() {
                // get statistics
                RadiosConnectionsStatistics stats = new RadiosConnectionsStatist
ics();
                for (RadiosDBRow row : database.getRows()) {
                    stats.radioConnections.put(row.name, row.connectedUsers);
                }
                String jsonStats = new Gson().toJson(stats);

                try {
                    channel.basicPublish(Configuration.RadiosStatisticsExchange,
 "",
                            null, jsonStats.getBytes());
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        });

        return operations;
    }

    @Override
    protected List<Integer> getStatisticsPeriodsSeconds() {
        List<Integer> operationsPeriods = new LinkedList<>();
        operationsPeriods.add(Configuration.RadioStatisticsPeriodSeconds);
        return operationsPeriods;
    }

    private String consumeConnections() throws IOException {
        // consume CONNECT_TO_RADIO requests
        String connectUsersQueue = channel.queueDeclare().getQueue();
        channel.queueBind(connectUsersQueue, Configuration.RadiosDBExchange,
                Configuration.RadiosDBConnectTag);
        Consumer connectConsumer = new DefaultConsumer(channel) {
            @Override
            public void handleDelivery(String consumerTag, Envelope envelope,
                                       AMQP.BasicProperties properties,
                                       byte[] body) throws IOException {

```

```java
                // parse request
                String jsonRequest = new String(body, "UTF-8");

                RadiosUpdateRequest request = new Gson().fromJson(jsonRequest,
                        RadiosUpdateRequest.class);

                // get radio record from DB
                RadiosDBRow radio = database.getRow(request.radio);
                if (radio ≡ null) {
                    radio = new RadiosDBRow(request.radio);
                }

                // add one connection to counter
                radio.connectedUsers++;
                System.out.println(" [x] Adding one connection to radio: " +
                        radio.name);

                // save changes to db
                database.updateRow(radio);
            }
        };
        return channel.basicConsume(connectUsersQueue, true, connectConsumer);
    }

    private String consumeDisconnections() throws IOException {
        // consume DISCONNECT_FROM_RADIO requests
        String disconnectUsersQueue = channel.queueDeclare().getQueue();
        channel.queueBind(disconnectUsersQueue, Configuration.RadiosDBExchange,
                Configuration.RadiosDBDisconnectTag);
        Consumer disconnectConsumer = new DefaultConsumer(channel) {
            @Override
            public void handleDelivery(String consumerTag, Envelope envelope,
                                       AMQP.BasicProperties properties,
                                       byte[] body) throws IOException {

                // parse request
                String jsonRequest = new String(body, "UTF-8");

                RadiosUpdateRequest request = new Gson().fromJson(jsonRequest,
                        RadiosUpdateRequest.class);

                // get radio record from DB
                RadiosDBRow radio = database.getRow(request.radio);
                if (radio ≡ null) {
                    radio = new RadiosDBRow(request.radio);
                }

                // add one connection to counter
                radio.connectedUsers--;
                System.out.println(" [x] Removing one connection from " +
                        "radio: " + radio.name);

                // save changes to db
                database.updateRow(radio);

            }
        };
        return channel.basicConsume(disconnectUsersQueue, true,
                disconnectConsumer);
    }

    public static void main(String[] argv) throws Exception {
        // define database
        Database<RadiosDBRow> database = new DatabaseRAM();

        // start database handler
        RadiosDBHandler handler = new RadiosDBHandler(Configuration.RabbitMQHost
,
                database);
    }
}
```

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

public class RadiosConnectionsStatistics {

    public Map<String, Integer> radioConnections = new HashMap<>();
}
```

```java
import com.rabbitmq.client.*;

import java.io.IOException;
import java.util.concurrent.*;

public class Radio extends RabbitMQProcess {

    private String exchangeName;
    private ScheduledExecutorService transmissionScheduler;
    private ScheduledFuture<?> transmissionHandle;

    private RadioSource source;

    public Radio(String host, String radioName, RadioSource source) throws
            IOException, TimeoutException {
        super(host);

        this.source = source;
        source.init();

        // declare BROADCAST exchange
        exchangeName = Configuration.RadioExchangePrefix + radioName;
        channel.exchangeDeclare(exchangeName, BuiltinExchangeType.FANOUT);

        scheduleTransmission();
    }

    private void scheduleTransmission() {
        transmissionScheduler = Executors
                .newScheduledThreadPool(1);

        transmissionHandle =
                transmissionScheduler.scheduleAtFixedRate(new Runnable() {
            @Override
            public void run() {
                byte[] nextBlock = source.getNextByteBlock();

                try {
                    channel.basicPublish(exchangeName, "", null, nextBlock);
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }, Configuration.RadioSendPeriodSeconds,
                        Configuration.RadioSendPeriodSeconds,
                        TimeUnit.SECONDS);
    }

    @Override
    protected void close() throws IOException, TimeoutException {
        super.close();

        transmissionHandle.cancel(true);
        transmissionScheduler.shutdown();
        source.close();
    }

    public static void main(String[] argv) throws Exception {
        RadioSource source = argv.length ≡ 3 ?
                new RadioSourceFile(argv[1],
                        Integer.parseInt(argv[2]) * 1024) :
                new RadioSourceRandomNumbers();
        Radio radio = new Radio(Configuration.RabbitMQHost, argv[0], source);
    }

}
```

```java
import com.rabbitmq.client.Channel;
import com.rabbitmq.client.Connection;
import com.rabbitmq.client.ConnectionFactory;

import java.io.IOException;
import java.util.concurrent.TimeoutException;

public class RabbitMQProcess {

    protected Connection connection;
    protected Channel channel;

    public RabbitMQProcess(String host) throws IOException, TimeoutException {

        // load configuration
        Configuration.loadConfiguration("config");

        // init RabbitMQ connection and channel
        ConnectionFactory factory = new ConnectionFactory();
        factory.setHost(host);
        connection = factory.newConnection();
        channel = connection.createChannel();

        addShutdownHook();
    }

    public void addShutdownHook() {

        RabbitMQProcess instance = this;
        Thread mainThread = Thread.currentThread();
        Runtime.getRuntime().addShutdownHook(new Thread() {
            public void run() {

                System.out.println("Calling shutdown hook");

                try {
                    instance.close();
                } catch (IOException e) {
                    e.printStackTrace();
                } catch (TimeoutException e) {
                    e.printStackTrace();
                }
                try {
                    mainThread.join();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        });
    }

    protected void close() throws IOException, TimeoutException {
        channel.close();
        connection.close();
    }

}
```

```java
import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;
import java.text.SimpleDateFormat;
import java.util.Date;

public class Logger {

    public static logLevel currentLogLevel = logLevel.INFO;
    private static PrintWriter logWriter = null;

    public enum logLevel {
        ERROR,
        WARNING,
        INFO
    }

    public static logLevel intToLogLevel(int i) {
        switch (i) {
            case 0:
                return logLevel.ERROR;
            case 1:
                return logLevel.WARNING;
            case 2:
                return logLevel.INFO;
            default:
                return logLevel.INFO;
        }
    }

    private static String logLevelToString(logLevel level) {
        switch(level) {
            case ERROR:
                return "[ERROR]";
            case WARNING:
                return "[WARNING]";
            case INFO:
                return "[INFO]";
            default:
                return "[INVALID LOGLEVEL]";
        }
    }

    public static void init(String filename) {
        try {
            logWriter = new PrintWriter(new FileWriter(filename));

            String timeStamp = new SimpleDateFormat(
                    "yyyy/MM/dd/ HH:mm:ss").format(new Date());
            logWriter.println("***************" +
                    timeStamp + "***************");
        } catch (IOException e) {
            log("Logger", "Couldn't open logfile for writing",
                    logLevel.ERROR);
        }
    }

    public static void close() {
        if (logWriter ≠ null) {
            logWriter.close();
        }
    }

    public static void log(String name, String message, logLevel level) {

        String logLine = Thread.currentThread().getName() + "\t" +
                logLevelToString(level) + "\t" + name + ":" + message;

        // output to screen
        if (currentLogLevel.ordinal() ≥ level.ordinal()) {
            System.out.println(logLine);
        }
        // output to logfile
```

```java
        if (logWriter ≠ null) {
            logWriter.println(logLine);
        }
    }

    public static void output(String outString) {
        System.out.println(outString);
        logWriter.println(outString);
    }

}
```

```java
import java.util.Collection;
import java.util.Comparator;
import java.util.TreeSet;

// un sorted set que automaticamente borra elementos
// de si mismo si se pasa del maximo
class LimitedSortedSet<E> extends TreeSet<E> {

    private int maxSize;

    LimitedSortedSet( int maxSize ) {
        this.maxSize = maxSize;
    }

    LimitedSortedSet( int maxSize, Comparator<? super E> comparator ) {
        super(comparator);
        this.maxSize = maxSize;
    }

    @Override
    public boolean addAll( Collection<? extends E> c ) {
        boolean added = super.addAll( c );
        if( size() > maxSize ) {
            E firstToRemove = (E)toArray( )[maxSize];
            removeAll( tailSet( firstToRemove ) );
        }
        return added;
    }

    @Override
    public boolean add( E o ) {
        boolean added =  super.add( o );
        while (size() > maxSize) {
            remove(last());
        }
        return added;
    }


}
```

```java
public class KeepAliveRequest {

    public String username;
    public int connectionId;
    public String radio;

    public KeepAliveRequest(String username, int connectionId, String radio) {
        this.username = username;
        this.connectionId = connectionId;
        this.radio = radio;
    }
}
```

```java
import com.google.gson.Gson;
import com.rabbitmq.client.*;

import java.io.IOException;
import java.util.concurrent.TimeoutException;

public class KeepAliveManager extends RabbitMQProcess {

    public KeepAliveManager(String host) throws IOException, TimeoutException {
        super(host);

        // declare USERS_DB exchange
        channel.exchangeDeclare(Configuration.UsersDBExchange,
                BuiltinExchangeType.DIRECT);

        consumeKeepAlives();
    }

    private String consumeKeepAlives() throws IOException {
        // KEEP ALIVE consumer
        channel.queueDeclare(Configuration.KeepAliveQueue, true, false, false,
                null);
        Consumer consumer_keepalive = new DefaultConsumer(channel) {
            @Override
            public void handleDelivery(String consumerTag, Envelope envelope,
                                       AMQP.BasicProperties properties,
                                       byte[] body) throws IOException {

                String json = new String(body, "UTF-8");
                KeepAliveRequest request = new Gson().fromJson(json,
                        KeepAliveRequest.class);

                System.out.println(" [x] Received keep alive request from: "
                        + request.username + " to: " + request.radio + " id: " +
                        request.connectionId);

                // ask usersDB to register connection
                channel.basicPublish(Configuration.UsersDBExchange,
                        Configuration.UsersDBKeepAliveTag, null,
                        new Gson().toJson(request).getBytes());
            }
        };
        return channel.basicConsume(Configuration.KeepAliveQueue, true,
                consumer_keepalive);
    }

    public static void main(String[] argv) throws Exception {
        KeepAliveManager manager =
                new KeepAliveManager(Configuration.RabbitMQHost);
    }
}
```

```java
import com.google.gson.Gson;
import com.rabbitmq.client.*;

import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;
import java.util.Date;
import java.util.List;
import java.util.ListIterator;
import java.util.concurrent.TimeoutException;

public abstract class FileLogger extends RabbitMQProcess {

    PrintWriter logWriter;
    String logsQueue;

    public FileLogger(String host, String logFilename) throws
            IOException, TimeoutException {
        super(host);

        // declare LOGS exchange
        channel.exchangeDeclare(Configuration.LogsExchange,
                BuiltinExchangeType.DIRECT);

        logWriter = new PrintWriter(new FileWriter(logFilename, true));

        logsQueue = channel.queueDeclare().getQueue();
        for (String tag : getBindings()) {
            channel.queueBind(logsQueue, Configuration.LogsExchange, tag);
        }
        consumeLogs();
    }

    protected abstract List<String> getBindings();

    public String consumeLogs() throws IOException {
        // consume connection logs
        Consumer connectConsumer = new DefaultConsumer(channel) {
            @Override
            public void handleDelivery(String consumerTag, Envelope envelope,
                                       AMQP.BasicProperties properties,
                                       byte[] body) throws IOException {

                // write log to file
                String logLine = new String(body, "UTF-8");
                logWriter.println(logLine);

                System.out.println(" [x] Received: " + logLine);
            }
        };
        return channel.basicConsume(logsQueue, true, connectConsumer);
    }

    @Override
    protected void close() throws IOException, TimeoutException {
        super.close();
        logWriter.close();
    }
}
```

Apr 12, 18 12:19 **DisconnectionManager.java** Page 1/1

```java
import com.google.gson.Gson;
import com.rabbitmq.client.*;

import java.io.IOException;
import java.util.concurrent.TimeoutException;

public class DisconnectionManager extends RabbitMQProcess {

    public DisconnectionManager(String host) throws IOException,
            TimeoutException {
        super(host);

        // declare USERS_DB exchange
        channel.exchangeDeclare(Configuration.UsersDBExchange,
                BuiltinExchangeType.DIRECT);

        // declare RADIOS_DB exchange
        channel.exchangeDeclare(Configuration.RadiosDBExchange,
                BuiltinExchangeType.DIRECT);

        // declare LOGS exchange
        channel.exchangeDeclare(Configuration.LogsExchange,
                BuiltinExchangeType.DIRECT);

        consumeDisconnections();
    }

    private String consumeDisconnections() throws IOException {
        // DISCONNECTIONS consumer
        channel.queueDeclare(Configuration.DisconnectionsQueue, true, false, fal
se,
                null);
        Consumer consumer_disconnect = new DefaultConsumer(channel) {
            @Override
            public void handleDelivery(String consumerTag, Envelope envelope,
                                       AMQP.BasicProperties properties,
                                       byte[] body) throws IOException {

                String json = new String(body, "UTF-8");
                UserDisconnectRequest request = new Gson().fromJson(json,
                        UserDisconnectRequest.class);

                System.out.println(" [x] Received request to disconnect user:" +
                        " " + request.username + " from: " + request.radio);

                // ask usersDB to register disconnection
                channel.basicPublish(Configuration.UsersDBExchange,
                        Configuration.UsersDBDisconnectTag, null,
                        new Gson().toJson(request).getBytes());

                // ask radiosDB to register disconnection
                channel.basicPublish(Configuration.RadiosDBExchange,
                        Configuration.RadiosDBDisconnectTag, null,
                        new Gson().toJson(request).getBytes());

                // send disconnects to file logger
                channel.basicPublish(Configuration.LogsExchange,
                        Configuration.LogsDisconnectionTag, null,
                        request.toLogLine().getBytes());
            }
        };
        return channel.basicConsume(Configuration.DisconnectionsQueue, true,
                consumer_disconnect);
    }

    public static void main(String[] argv) throws Exception {
        DisconnectionManager manager =
                new DisconnectionManager(Configuration.RabbitMQHost);
    }
}
```

Apr 12, 18 13:55 **DBHandlerWithStatistics.java** Page 1/1

```java
import com.google.gson.Gson;

import java.io.IOException;
import java.util.LinkedList;
import java.util.List;
import java.util.concurrent.*;

public abstract class DBHandlerWithStatistics<T extends DatabaseRow>
        extends RabbitMQProcess {

    Database<T> database;
    private ScheduledExecutorService statisticsScheduler = null;
    private List<ScheduledFuture<?>> statisticsHandles = null;

    public DBHandlerWithStatistics(String host, Database database) throws
            IOException,
            TimeoutException {
        super(host);
        this.database = database;

        List<Runnable> statisticTasks = getStatisticsOperations();
        List<Integer> statisticTasksPeriods = getStatisticsPeriodsSeconds();
        if (statisticTasks.size() > 0) {
            statisticsScheduler = Executors
                    .newScheduledThreadPool(1);
            statisticsHandles = new LinkedList<>();

            for (int i = 0; i < statisticTasks.size(); ++i) {
                Runnable r = statisticTasks.get(i);
                int period = statisticTasksPeriods.get(i);
                ScheduledFuture<?> statisticsHandle =
                        statisticsScheduler.scheduleAtFixedRate(r, period,
                                period, TimeUnit.SECONDS);
                statisticsHandles.add(statisticsHandle);
            }
        }
    }

    @Override
    protected void close() throws IOException, TimeoutException {
        super.close();

        if (statisticsScheduler ≠ null) {
            for (ScheduledFuture<?> f : statisticsHandles) {
                f.cancel(true);
            }
            statisticsScheduler.shutdown();
        }
    }

    protected abstract List<Runnable> getStatisticsOperations();

    protected abstract List<Integer> getStatisticsPeriodsSeconds();
}
```

Apr 11, 18 19:12 **DatabaseRow.java** Page 1/1

```java
public abstract class DatabaseRow {

    public String primary_key;

    public DatabaseRow(String primary_key) {
        this.primary_key = primary_key;
    }
}
```

Apr 12, 18 2:38 **DatabaseRAM.java** Page 1/1

```java
import java.util.HashMap;
import java.util.LinkedList;
import java.util.List;
import java.util.Map;

public class DatabaseRAM<T extends DatabaseRow> implements Database<T> {

    private Map<String, T> database = new HashMap<>();

    public T getRow(String key) {
        T row = database.getOrDefault(key, null);
        return row;
    }

    @Override
    public List<T> getRows() {
        return new LinkedList<>(database.values());
    }

    @Override
    public boolean createRow(T row) {
        if (database.put(row.primary_key, row) ≡ null) {
            return true;
        }
        return false;
    }

    @Override
    public boolean updateRow(T row) {
        if (¬database.containsKey(row.primary_key)) {
            return createRow(row);
        }
        database.put(row.primary_key, row);
        return true;
    }

    @Override
    public boolean removeRow(String primary_key) {
        if (database.remove(primary_key) ≠ null) {
            return true;
        }
        return false;
    }
}
```

Apr 12, 18 2:38 **Database.java** Page 1/1

```java
import java.util.List;

public interface Database<T extends DatabaseRow> {

    T getRow(String key);

    List<T> getRows();

    boolean createRow(T row);

    boolean updateRow(T row);

    boolean removeRow(String key);
}
```

Apr 12, 18 13:34 **ConnectionManager.java** Page 1/2

```java
import com.google.gson.Gson;
import com.rabbitmq.client.*;

import java.io.IOException;
import java.util.concurrent.TimeoutException;

public class ConnectionManager extends RabbitMQProcess {

    public ConnectionManager(String host) throws IOException, TimeoutException {
        super(host);

        // declare USERS_DB exchange
        channel.exchangeDeclare(Configuration.UsersDBExchange,
                BuiltinExchangeType.DIRECT);

        // declare usersDB responses queue
        channel.queueDeclare(Configuration.ConnMgrUsersDBResponseQueue,
                true, false, false, null);

        // declare RADIOS_DB exchange
        channel.exchangeDeclare(Configuration.RadiosDBExchange,
                BuiltinExchangeType.DIRECT);

        // declare LOGS exchange
        channel.exchangeDeclare(Configuration.LogsExchange,
                BuiltinExchangeType.DIRECT);

        consumeConnections();
        consumeUsersDB();
    }

    private String consumeConnections() throws IOException {
        // CONNECTIONS consumer
        channel.queueDeclare(Configuration.ConnectionsQueue,
                true, false, false, null);
        Consumer consumer_connect = new DefaultConsumer(channel) {
            @Override
            public void handleDelivery(String consumerTag, Envelope envelope,
                                       AMQP.BasicProperties properties,
                                       byte[] body) throws IOException {

                String json = new String(body, "UTF-8");
                UserConnectRequest request = new Gson().fromJson(json,
                        UserConnectRequest.class);

                System.out.println(" [x] Received connection request from: "
                        + request.username + " to: " + request.radio);

                // ask usersDB to register connection
                channel.basicPublish(Configuration.UsersDBExchange,
                        Configuration.UsersDBConnectTag, null,
                        new Gson().toJson(request).getBytes());
            }
        };
        return channel.basicConsume(Configuration.ConnectionsQueue,
                true, consumer_connect);
    }

    private String consumeUsersDB() throws IOException {
        // usersDB consume
        Consumer consumer_usersdb = new DefaultConsumer(channel) {
            @Override
            public void handleDelivery(String consumerTag, Envelope envelope,
                                       AMQP.BasicProperties properties,
                                       byte[] body) throws IOException {

                String json = new String(body, "UTF-8");
                UserConnectResponse response = new Gson().fromJson(json,
                        UserConnectResponse.class);

                for (UserDisconnectRequest disconn :
                        response.closedConnections) {

```

```java
                    // register closed connections in radios DB
                    RadiosUpdateRequest radiosRequest =
                            new RadiosUpdateRequest(disconn.radio,
                                    disconn.username);
                    channel.basicPublish(Configuration.RadiosDBExchange,
                            Configuration.RadiosDBDisconnectTag, null,
                            new Gson().toJson(radiosRequest).getBytes());

                    // send disconnects to file logger
                    channel.basicPublish(Configuration.LogsExchange,
                            Configuration.LogsDisconnectionTag, null,
                            disconn.toLogLine().getBytes());
                }

                if (response.couldConnect) {
                    // send connect to file logger
                    channel.basicPublish(Configuration.LogsExchange,
                            Configuration.LogsConnectionTag, null,
                            response.toLogLine().getBytes());

                    // register connection in radios DB
                    RadiosUpdateRequest radioConnectRequest =
                            new RadiosUpdateRequest(response.radio,
                                    response.username);
                    channel.basicPublish(Configuration.RadiosDBExchange,
                            Configuration.RadiosDBConnectTag, null,
                            new Gson().toJson(radioConnectRequest).getBytes());

                    System.out.println(" [X] User: " + response.username +
                            " connected to radio: " + response.radio);
                } else {
                    System.out.println(" [X] User: " + response.username +
                            " denied connection to radio: " + response.radio);
                }

                String jsonResponse = new Gson().toJson(response);
                channel.basicPublish("", response.returnQueueName, null,
                        jsonResponse.getBytes());
            }
        };
        return channel.basicConsume(Configuration.ConnMgrUsersDBResponseQueue,
                true, consumer_usersdb);
    }

    public static void main(String[] argv) throws Exception {

        ConnectionManager manager =
                new ConnectionManager(Configuration.RabbitMQHost);
    }

}
```

```java
import java.io.IOException;
import java.util.LinkedList;
import java.util.List;
import java.util.concurrent.TimeoutException;

public class ConnDisconnFileLogger extends FileLogger {

    public ConnDisconnFileLogger(String host, String logFilename) throws
            IOException, TimeoutException {
        super(host, logFilename);
    }

    @Override
    protected List<String> getBindings() {
        List<String> bindings = new LinkedList<>();
        bindings.add(Configuration.LogsConnectionTag);
        bindings.add(Configuration.LogsDisconnectionTag);
        return bindings;
    }

    public static void main(String[] argv) throws Exception {

        ConnDisconnFileLogger fileLogger =
                new ConnDisconnFileLogger(Configuration.RabbitMQHost,
                        argv[0]);
    }
}
```

```java
import com.google.gson.Gson;
import com.google.gson.GsonBuilder;

import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;

public class Configuration {

    public static int RadioSendPeriodSeconds = 2;
    public static int KeepAlivePeriodSeconds = 5;
    public static int SecondsUntilDropConnection = 10;
    public static int MaxConnectionsPerFreeUser = 3;
    public static int MaxConnectionsPerUnlimitedUser = 999;

    public static String RabbitMQHost = "localhost";

    public static String UsersDBExchange = "USERS_DB";
    public static String UsersDBConnectTag = "connect";
    public static String UsersDBDisconnectTag = "disconnect";
    public static String UsersDBKeepAliveTag = "keepalive";
    public static String UsersStatisticsExchange = "USERS_STATS";
    public static int UsersStatisticsPeriodSeconds = 10;
    public static int UserStatisticsN = 100;

    public static String RadiosDBExchange = "RADIOS_DB";
    public static String RadiosDBConnectTag = "connect";
    public static String RadiosDBDisconnectTag = "disconnect";
    public static String RadiosStatisticsExchange = "RADIOS_STATS";
    public static int RadioStatisticsPeriodSeconds = 10;

    public static String ConnMgrUsersDBResponseQueue =
            "usersDBResponseQueueName";

    public static String ConnectionsQueue = "CONNECTIONS";
    public static String DisconnectionsQueue = "DISCONNECTIONS";
    public static String KeepAliveQueue = "KEEP_ALIVE";

    public static String RadioExchangePrefix = "BROADCAST-";

    public static String LogsExchange = "LOGS";
    public static String LogsConnectionTag = "connect";
    public static String LogsDisconnectionTag = "disconnect";

    public static boolean loadConfiguration(String configFilename) {

        try {
            // read json config
            BufferedReader br = new BufferedReader(
                    new FileReader( configFilename));
            String jsonString = "";
            String s;
            while ((s = br.readLine()) ≠ null) {
                jsonString += s;
            }

            // esto es para que gson serialize variables estaticas
            GsonBuilder gsonBuilder  = new GsonBuilder();
            gsonBuilder.excludeFieldsWithModifiers(
                    java.lang.reflect.Modifier.TRANSIENT);

            Gson gson = gsonBuilder.create();
            // load to object
            Configuration config = gson.fromJson(jsonString,
                    Configuration.class);

            return true;
        }
        catch (FileNotFoundException e) {
            /*Logger.log("Monitor", "config file: " + configFilename +
                    " not found", Logger.logLevel.ERROR);*/
        } catch (IOException e) {
```

```java
            /*Logger.log("Monitor", "config file: " + configFilename +
                    " could not be read", Logger.logLevel.ERROR);*/
        }

        return false;
    }
}
```

```java
import com.google.gson.Gson;
import com.rabbitmq.client.*;

import java.io.FileOutputStream;
import java.io.FileWriter;
import java.io.IOException;
import java.text.SimpleDateFormat;
import java.util.Base64;
import java.util.Date;
import java.util.NoSuchElementException;
import java.util.Scanner;
import java.util.concurrent.*;

public class Client extends  RabbitMQProcess {

    private String radioExchange = "";

    private String username = "";
    private String radio;
    private int connectionId;
    private String radioConsumeTag = "";
    FileOutputStream transmissionWriter = null;

    private ScheduledExecutorService keepAliveScheduler =
            Executors.newScheduledThreadPool(1);
    private ScheduledFuture<?> keepAliveHandle;

    public Client(String host) throws IOException, TimeoutException {
        super(host);
    }

    public void setUsername(String username) {
        this.username = username;
    }

    public void setRadio(String radio) {
        this.radio = radio;
    }

    public boolean requestConnectionToRadio() throws IOException,
            InterruptedException {

        if (username.equals("")) {
            System.out.println("ERROR: Did you specify a username?");
            return false;
        }

        // define callback queue
        String callbackQueueName = channel.queueDeclare().getQueue();

        // create request
        UserConnectRequest request = new UserConnectRequest(username, radio,
                callbackQueueName);
        String requestJson = new Gson().toJson(request);

        // publish to CONNECTIONS queue
        channel.basicPublish("", Configuration.ConnectionsQueue, null,
                requestJson.getBytes());

        final BlockingQueue<String> responseQueue =
                new ArrayBlockingQueue<String>(1);

        String callbackTag = channel.basicConsume(callbackQueueName,true,
                new DefaultConsumer(channel) {
            @Override
            public void handleDelivery(String consumerTag, Envelope envelope,
                                       AMQP.BasicProperties properties,
                                       byte[] body) throws IOException {
                responseQueue.offer(new String(body, "UTF-8"));
            }
        });

        String jsonResponse = responseQueue.take();
```

```java
        channel.basicCancel(callbackTag);
        UserConnectResponse response = new Gson().fromJson(jsonResponse,
                UserConnectResponse.class);
        if (¬response.couldConnect) {
            System.out.println("ERROR: Connection refused, are " +
                    "you already connected on 3 devices?");
            return false;
        }

        connectionId = response.connectionId;
        radioExchange = Configuration.RadioExchangePrefix + response.radio;
        return true;
    }

    public boolean listenToRadio() throws IOException {

        if (radioExchange.equals("")) {
            return false;
        }

        // declare radio broadcast exchange
        channel.exchangeDeclare(radioExchange, BuiltinExchangeType.FANOUT);

        // declare temporary queue and bind
        String queueName = channel.queueDeclare().getQueue();
        channel.queueBind(queueName, radioExchange, "");
        System.out.println("Creating queue: " + queueName);

        // open new file for transmission
        SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd-HH-mm-ss");
        String transmissionName = "client" + "-" + username + "-" + radio +
                "-" + connectionId + "-" + sdf.format(new Date()) + ".wav";
        transmissionWriter = new FileOutputStream(transmissionName);

        Consumer consumer = new DefaultConsumer(channel) {
            @Override
            public void handleDelivery(String consumerTag, Envelope envelope,
                                       AMQP.BasicProperties properties,
                                       byte[] body) throws IOException {
                String message = new String(body, "UTF-8");
                System.out.println(" [x] Received '" + message + "'");
                transmissionWriter.write(body);
            }
        };
        radioConsumeTag = channel.basicConsume(queueName, true, consumer);
        return true;
    }

    public void scheduleKeepAlive() {
        final Runnable sendKeepAlive = new Runnable() {
            @Override
            public void run() {
                KeepAliveRequest request = new KeepAliveRequest(username,
                        connectionId, radio);
                String requestJson = new Gson().toJson(request);
                try {
                    channel.basicPublish("", Configuration.KeepAliveQueue,
                            null, requestJson.getBytes());
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        };
        keepAliveHandle = keepAliveScheduler.scheduleAtFixedRate
                (sendKeepAlive, 5,5, TimeUnit.SECONDS);
    }

    public void stopKeepAlive() {
        keepAliveScheduler.schedule(new Runnable() {
            @Override
            public void run() {
                keepAliveHandle.cancel(true);
            }
        }, 0, TimeUnit.SECONDS);
    }

    public void stopListeningToRadio() throws IOException {
        if (radioConsumeTag.equals("")) {
            System.out.println("ERROR: not listening to radio");
        } else {
            // create request
            UserDisconnectRequest request = new UserDisconnectRequest(username,
                    radio, connectionId);
            String requestJson = new Gson().toJson(request);

            // publish to DISCONNECTIONS queue
            channel.basicPublish("", Configuration.DisconnectionsQueue, null,
                    requestJson.getBytes());

            // stop receiving transmission
            channel.basicCancel(radioConsumeTag);
            radioConsumeTag = "";

            // close transmission file
            transmissionWriter.close();
            transmissionWriter = null;
        }
    }

    public void printOptions() {
        System.out.println("\n");
        System.out.println("Choose an action: ");
        System.out.println("\t" + "1. Set user");
        System.out.println("\t" + "2. Connect to radio");
        System.out.println("\t" + "3. Disconnect from radio");
        System.out.println("\t" + "4. Exit");
    }

    public boolean mainMenu(Scanner in) throws IOException,
            InterruptedException {
        String choiceStr = in.nextLine();
        int choice = Integer.parseInt(choiceStr);
        switch (choice) {
            case 1:
                System.out.print("Please specify a username: ");
                String username = in.nextLine();
                setUsername(username);
                break;
            case 2:
                System.out.println("Please specify a radio: ");
                String radio = in.nextLine();
                setRadio(radio);
                if (¬requestConnectionToRadio()) {
                    break;
                }
                listenToRadio();
                scheduleKeepAlive();
                break;
            case 3:
                stopListeningToRadio();
                stopKeepAlive();
                break;
            case 4:
                System.out.println("Press CTRL+C to exit");
                return true;
            default:
                System.out.println("ERROR: Invalid option");
                break;
        }

        printOptions();
        return false;
    }

    @Override
    protected void close() throws IOException, TimeoutException {
```

```java
        super.close();
        if (transmissionWriter ≠ null) {
            transmissionWriter.close();
        }
    }


    public static void main(String[] argv) throws Exception {

        Scanner in = new Scanner(System.in);

        Client client = new Client(Configuration.RabbitMQHost);
        client.printOptions();

        boolean end = false;
        while (¬end) {
            try {
                end = client.mainMenu(in);
            } catch (NoSuchElementException e) {
                end = true;
            }
        }
    }

}
```

**Table of Contents**