Millot Elisa

SAE3.02 : Développer des Applications communicantes

1. Introduction

Présentation du projet :

L'objectif de ce projet est de développer une application client-serveur capable de communiquer à travers les protocoles TCP et UDP. L'application est conçue pour gérer des connexions multiples, authentifier les utilisateurs et permettre l'envoi et la réception de messages structurés. Ce projet a été développé pour fournir une solution. Les technologies utilisées incluent Java pour la logique de l'application et l'Android SDK pour l'interface utilisateur.

2. Fonctionnalités principales

2.1 Protocole TCP/UDP

L'application supporte à la fois les protocoles TCP (fiable et orienté connexion) et UDP (rapide mais non garanti). L'utilisateur peut choisir le protocole via un RadioGroup dans l'interface.

2.2 Envoi et réception de messages

Les utilisateurs peuvent échanger des messages texte ou des structures de données complexes via des flux sérialisés (ObjectInputStream et ObjectOutputStream).

2.3 Multithreading

Le serveur utilise un ExecutorService pour gérer les connexions multiples simultanément.

2.4 Serveur multi-services

Le serveur interprète les messages envoyés par les clients pour fournir des services adaptés.

2.5 Authentification et gestion des droits d'accès

Un système d'authentification vérifie les identifiants des utilisateurs et leur attribue des droits d'accès (administrateur ou client).

2.6 Interface utilisateur intuitive

L'application inclut des éléments interactifs comme des boutons pour démarrer le serveur, se connecter en tant que client, envoyer des messages, et se déconnecter.

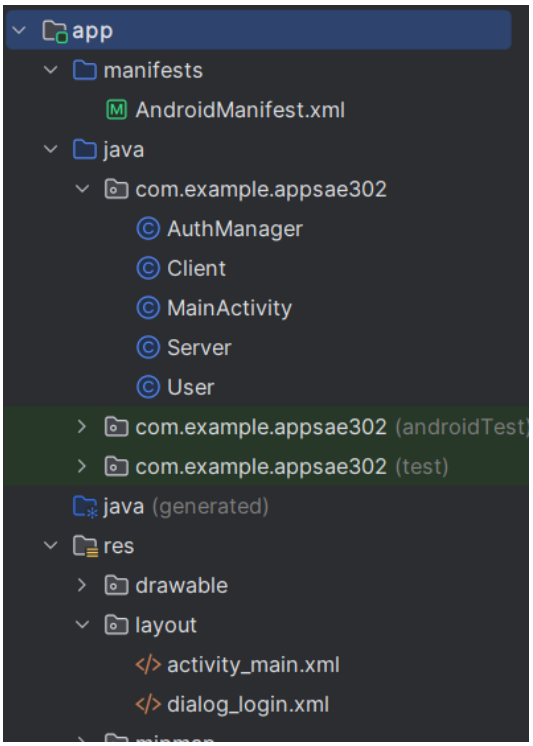
- **Initiative personnelle** :

- Un tableau de bord dynamique affiche des informations comme le nombre de clients connectés, l'adresse IP du serveur et les messages échangés.
- Un switch pour basculer entre le mode clair et le mode sombre.

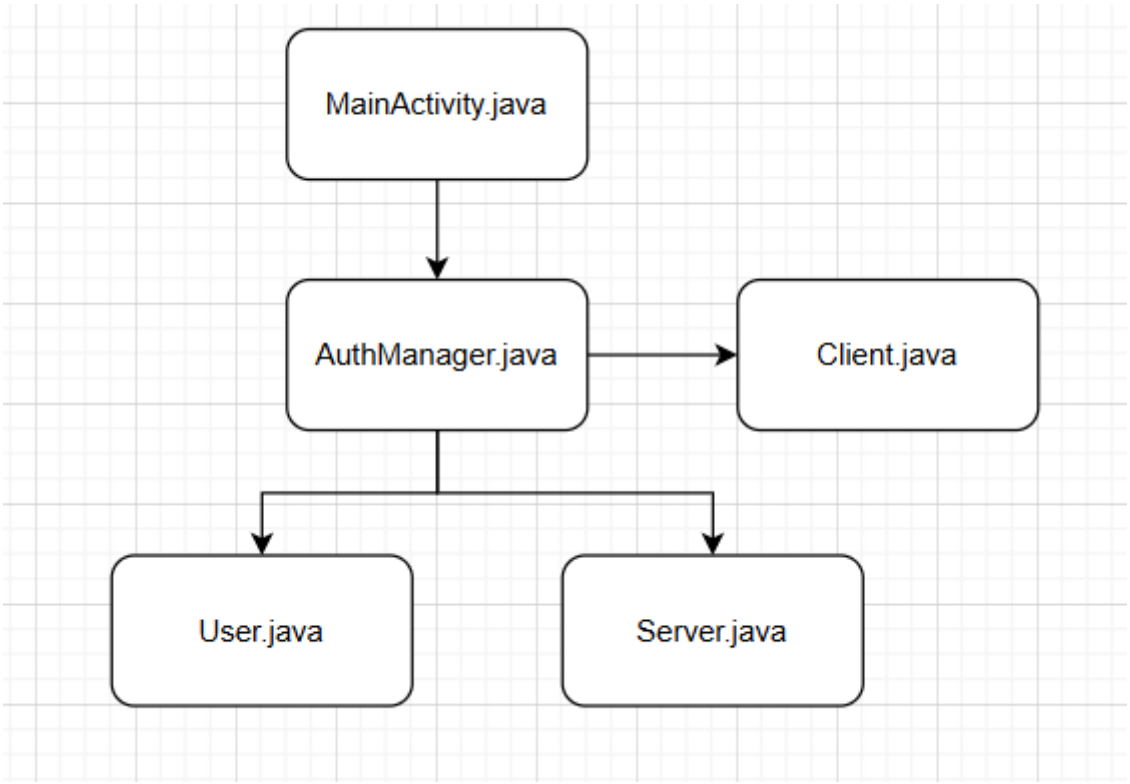
3. Architecture de l'application

3.1 Structure des dossiers et fichiers :

- *MainActivity.java* : Gère l'interface utilisateur et les événements principaux
- *Client.java* : Gère les connexions et les communications du côté client.
- *Server.java* : Gère les connexions et les communications du côté serveur.
- *AuthManager.java* : Gère l'authentification et les droits d'accès
- *User.java* : Représente un utilisateur avec son nom, son niveau d'accès et son mot de passe.



3.2 Diagramme de l'architecture :



4. Détails d'implémentation

MainActivity.java

Responsabilités :

- Afficher l'interface utilisateur.
- Gérer les événements d'authentification, de connexion et d'envoi de messages.
- Fournir un tableau de bord dynamique avec des informations sur l'activité du serveur.

Méthodes principales :

showLoginDialog(boolean isAdmin): Affiche une boîte de dialogue pour l'authentification.

```java
private void showLoginDialog(boolean isAdmin) {
    LayoutInflater inflater = LayoutInflater.from( context: this);
    View loginView = inflater.inflate(R.layout.dialog_login,  root: null);
    AlertDialog.Builder builder = new AlertDialog.Builder( context: this);
    builder.setView(loginView);

    final EditText inputUsername = loginView.findViewById(R.id.input_username);
    final EditText inputPassword = loginView.findViewById(R.id.input_password);

    builder.setTitle(isAdmin ? "Connexion Administrateur" : "Connexion Client")
            .setPositiveButton( text: "Se Connecter", (dialog, which) -> {
                if (authManager == null) {
                    textStatus.setText("Erreur interne : AuthManager n'a pas été initialisé.");
                    return;
                }
                String username = inputUsername.getText().toString();
                String password = inputPassword.getText().toString();
                currentUser = authManager.authenticate(username, password);
                if (currentUser == null) {
                    textStatus.setText("Accès refusé : Utilisateur ou mot de passe incorrect.");
                } else if (isAdmin && !authManager.hasAccess(currentUser,  requiredAccessLevel: "admin")) {
                    textStatus.setText("Accès refusé : Droits administrateur requis.");
                } else if (!isAdmin && authManager.hasAccess(currentUser,  requiredAccessLevel: "admin")) {
                    textStatus.setText("Accès refusé : Droits client requis.");
                } else {
                    textStatus.setText("Connexion réussie.");
                    if (isAdmin) {
                        startServer();
                    } else {
                        showClientInput();
                    }
                }
            })
            .setNegativeButton( text: "Annuler", (dialog, which) -> dialog.cancel())
            .show();
}
```

startServer(): Démarre le serveur avec gestion multithreading.

```java
private void startServer() {
    if (!isPortAvailable(serverPort)) {
        runOnUiThread(() -> textStatus.setText("Port déjà utilisé. Choisissez un autre port."));
        return;
    }

    new Thread(() -> {
        server = new Server(serverPort);
        server.setCallback(new Server.ServerCallback() {
            7 usages
            @Override
            public void onStatusUpdate(String status) {
                runOnUiThread(() -> {
                    textStatus.append("\n" + status);
                    scrollViewStatus.fullScroll(View.FOCUS_DOWN);
                });
            }

            2 usages
            @Override
            public void onMessageReceived(String message) {
                messages.add(message);
                runOnUiThread(() -> {
                    textStatus.append("\nMessage reçu : " + message);
                    scrollViewStatus.fullScroll(View.FOCUS_DOWN);
                });
            }
```

```
            });
            server.start();
            isServerRunning = true;
            handler.post(dashboardUpdater);
            runOnUiThread(() -> {
                textDashboard.setVisibility(View.VISIBLE);
                textStatus.setVisibility(View.VISIBLE);
                textStatus.setText("Serveur démarré.");
                showServerIpAddress(serverPort);
            });
        }).start();
    }
```

sendMessage(): Envoie un message au serveur ou à d'autres clients.

```
    private void sendMessage() {
        String ip = editIp.getText().toString();
        String portStr = editPort.getText().toString();
        String message = editMessage.getText().toString();

        if (ip.isEmpty() || portStr.isEmpty() || message.isEmpty()) {
            runOnUiThread(() -> textStatus.setText("Veuillez remplir tous les champs."));
            return;
        }

        int port;
        try {
            port = Integer.parseInt(portStr);
        } catch (NumberFormatException e) {
            runOnUiThread(() -> textStatus.setText("Port invalide."));
            return;
        }

        boolean isTcp = radioGroupProtocol.getCheckedRadioButtonId() == R.id.radio_tcp;

        new Thread(() -> {
            client = new Client(ip, port, isTcp);
            client.setUser(currentUser);
            String connectionResult = client.connect();
            runOnUiThread(() -> textStatus.setText(connectionResult));
```

```
            if (connectionResult.equals("Connexion réussie.")) {
                String protocol = client.isTcp() ? "TCP" : "UDP";
                String fullMessage = "[Utilisateur : " + currentUser.getUsername() + "] [Protocole : " + protocol + "] " + message;
                client.sendMessage(fullMessage);
                runOnUiThread(() -> {
                    textStatus.append("\nMessage envoyé : " + fullMessage);
                    scrollViewStatus.fullScroll(View.FOCUS_DOWN);
                });
                isConnected = true;
            }
        }).start();
    }
```

disconnectClient(): Déconnecte le client du serveur proprement.

```
323         private void disconnectClient() {
324             if (client != null) {
325                 new Thread(() -> {
326                     client.sendMessage("Déconnexion de l'utilisateur.");
327                     client.disconnect();
328                     runOnUiThread(() -> {
329                         textStatus.append("\nDéconnecté du serveur.");
330                         scrollViewStatus.fullScroll(View.FOCUS_DOWN);
331                     });
332                     isConnected = false;
333                 }).start();
334             }
335         }
```

Client.java

***Responsabilités** :*

- Établir une connexion au serveur en TCP ou UDP.
- Envoyer des messages au serveur.
- Se déconnecter proprement.

***Méthodes principales** :*

connect(): Démarre une connexion au serveur.

```
35      public String connect() {
36          try {
37              if (isTcp) {
38                  tcpSocket = new Socket(serverIp, serverPort);
39                  out = new ObjectOutputStream(tcpSocket.getOutputStream());
40                  in = new ObjectInputStream(tcpSocket.getInputStream());
41                  System.out.println("Connexion TCP réussie au serveur " + serverIp + " sur le port " + serverPort);
42              } else {
43                  udpSocket = new DatagramSocket();
44                  System.out.println("Connexion UDP prête pour le serveur " + serverIp + " sur le port " + serverPort);
45              }
46              return "Connexion réussie.";
47          } catch (IOException e) {
48              e.printStackTrace();
49              return "Erreur de connexion : " + e.getMessage();
50          }
51      }
```

sendMessage(Object message): Envoie un message au serveur.

```java
public void sendMessage(Object message) {
    try {
        if (isTcp) {
            if (tcpSocket != null && !tcpSocket.isClosed() && out != null) {
                out.writeObject(message);
                out.flush();
                System.out.println("Message envoyé : " + message);
            } else {
                System.err.println("Connexion TCP non établie ou fermée.");
            }
        } else {
            if (udpSocket != null && !udpSocket.isClosed()) {
                byte[] buffer = message.toString().getBytes();
                DatagramPacket packet = new DatagramPacket(buffer, buffer.length, InetAddress.getByName(serverIp), serverPort);
                udpSocket.send(packet);
                System.out.println("Message envoyé : " + message);
            } else {
                System.err.println("Connexion UDP non établie ou fermée.");
            }
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

disconnect(): Ferme la connexion.

```java
public void disconnect() {
    try {
        if (isTcp) {
            if (tcpSocket != null && !tcpSocket.isClosed()) {
                sendMessage("Déconnexion de l'utilisateur.");
                tcpSocket.close();
            }
            System.out.println("Déconnecté du serveur TCP.");
        } else {
            if (udpSocket != null && !udpSocket.isClosed()) {
                udpSocket.close();
            }
            System.out.println("Déconnecté du serveur UDP.");
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
}
}
```

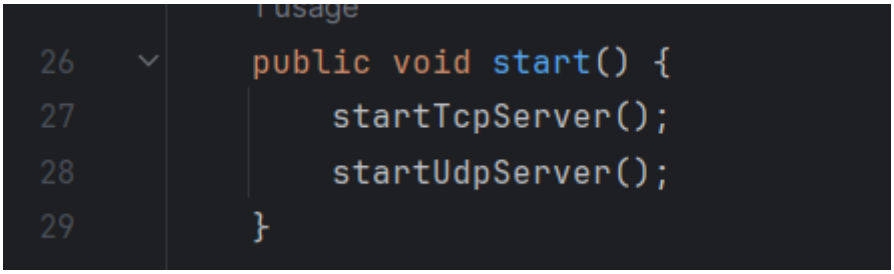
Server.java

Responsabilités *:*

- Gérer les connexions clients en multithreading.
- Fournir des services adaptés aux messages reçus.
- Compter les clients uniques et le total des messages échangés.

Méthodes principales :

start(): Lance les serveurs TCP et UDP en parallèle.



```
1 usage
public void start() {
    startTcpServer();
    startUdpServer();
}
```

```
private void startTcpServer() {
    new Thread(() -> {
        try (ServerSocket serverSocket = new ServerSocket(port)) {
            if (callback != null) callback.onStatusUpdate("Serveur TCP en attente de connexions...");
            while (!Thread.currentThread().isInterrupted()) {
                Socket clientSocket = serverSocket.accept();
                String clientIp = clientSocket.getInetAddress().getHostAddress();
                uniqueClients.add(clientIp);
                connectedClients.incrementAndGet();
                threadPool.execute(new ClientHandler(clientSocket, callback, connectedClients, totalMessages));
            }
        } catch (IOException e) {
            if (callback != null) callback.onStatusUpdate("Erreur : " + e.getMessage());
        }
    }).start();
}
```

```
private void startUdpServer() {
    new Thread(() -> {
        try (DatagramSocket serverSocket = new DatagramSocket(port)) {
            if (callback != null) callback.onStatusUpdate("Serveur UDP en attente de connexions...");
            byte[] buffer = new byte[1024];
            while (!Thread.currentThread().isInterrupted()) {
                DatagramPacket packet = new DatagramPacket(buffer, buffer.length);
                serverSocket.receive(packet);
                String clientIp = packet.getAddress().getHostAddress();
                uniqueClients.add(clientIp);
                connectedClients.incrementAndGet();
                threadPool.execute(new UdpClientHandler(packet, serverSocket, callback, connectedClients, totalMessages));
            }
        } catch (IOException e) {
            if (callback != null) callback.onStatusUpdate("Erreur : " + e.getMessage());
        }
    }).start();
}
```

getUniqueClients(), getTotalMessages(): Fournissent des statistiques sur l'activité du serveur.

```java
    public int getTotalMessages() {
        return totalMessages.get();
    }

    1 usage
    public int getUniqueClients() {
        return uniqueClients.size();
    }
```

AuthManager.java et User.java

- **AuthManager** :

Contient une liste d'utilisateurs préconfigurés.

Valide les identifiants et définit les niveaux d'accès (client ou admin).

```java
public class AuthManager {
    6 usages
    private Map<String, User> users;

    1 usage
    public AuthManager() {
        users = new HashMap<>();
        users.put("admin", new User( username: "admin",  accessLevel: "admin",  password: "adminpass"));
        users.put("client1", new User( username: "client1",  accessLevel: "client",  password: "client1pass"));
        users.put("client2", new User( username: "client2",  accessLevel: "client",  password: "client2pass"));
    }

    1 usage
    public User authenticate(String username, String password) {
        if (users == null) {
            throw new IllegalStateException("La liste des utilisateurs n'est pas initialisée.");
        }
        User user = users.get(username);
        return (user != null && user.getPassword().equals(password)) ? user : null;
    }

    2 usages
    public boolean hasAccess(User user, String requiredAccessLevel) {
        if (user == null) return false;
        return user.getAccessLevel().equals(requiredAccessLevel);
    }
}
```

- **User** :

Classe représentant un utilisateur avec ses attributs (nom, mot de passe, niveau d'accès).

```java
public class User {
    2 usages
    private String username;
    2 usages
    private String accessLevel;
    2 usages
    private String password;

    3 usages
    public User(String username, String accessLevel, String password) {
        this.username = username;
        this.accessLevel = accessLevel;
        this.password = password;
    }

    1 usage
    public String getUsername() { return username; }

    1 usage
    public String getAccessLevel() { return accessLevel; }

    1 usage
    public String getPassword() { return password; }
}
```

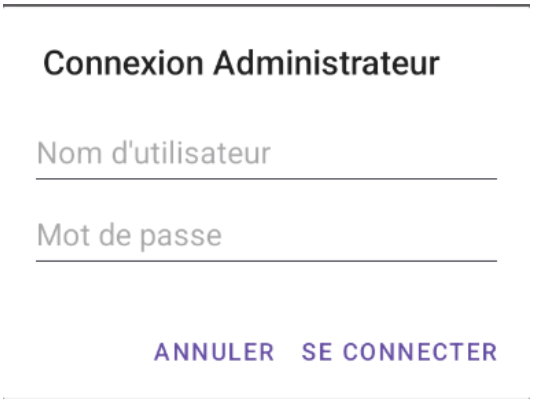
5. Captures d'écran

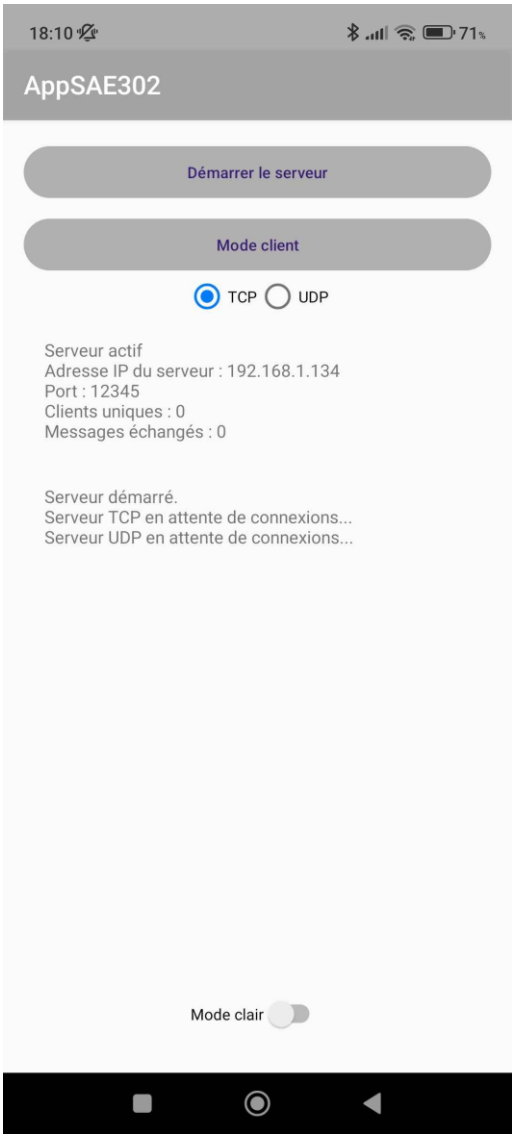
1. Interface utilisateur :

Voici ce que nous avons quand nous démarrons l'application

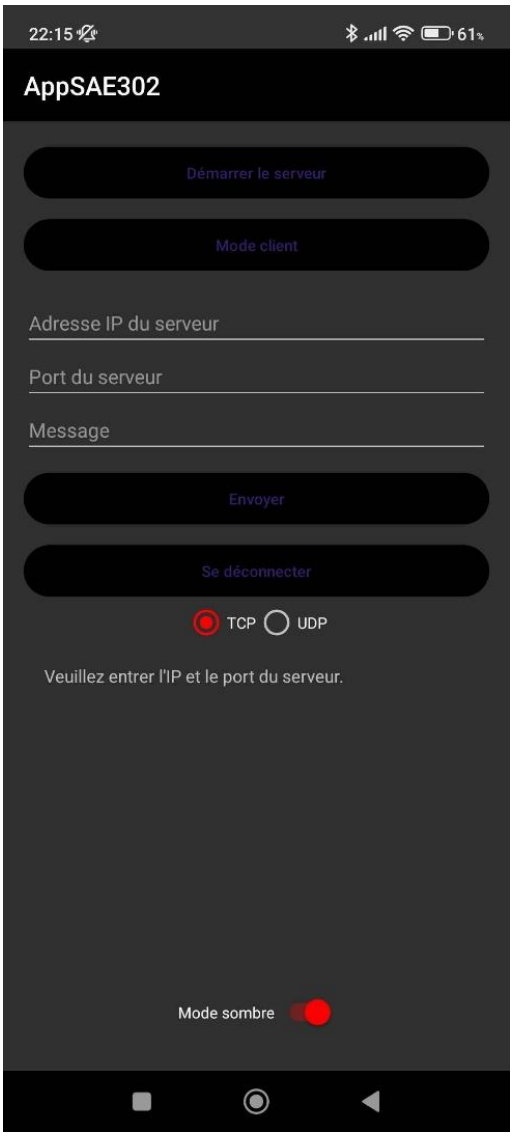
Quand nous démarrons le serveur et quand nous démarrons le client :

Entrez les comptes valides soit pour se connecter au serveur, soit au client.



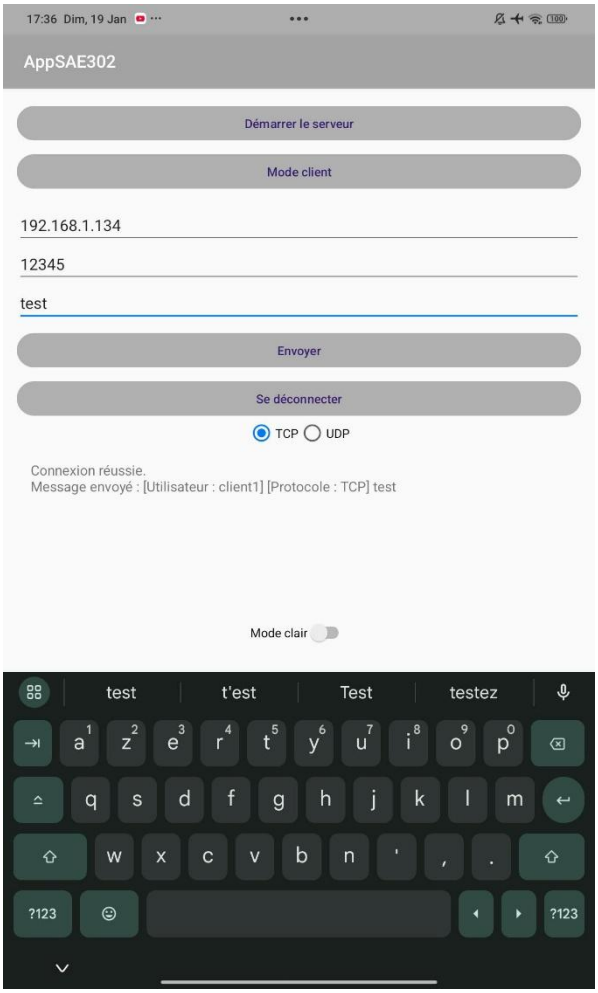


Mode serveur

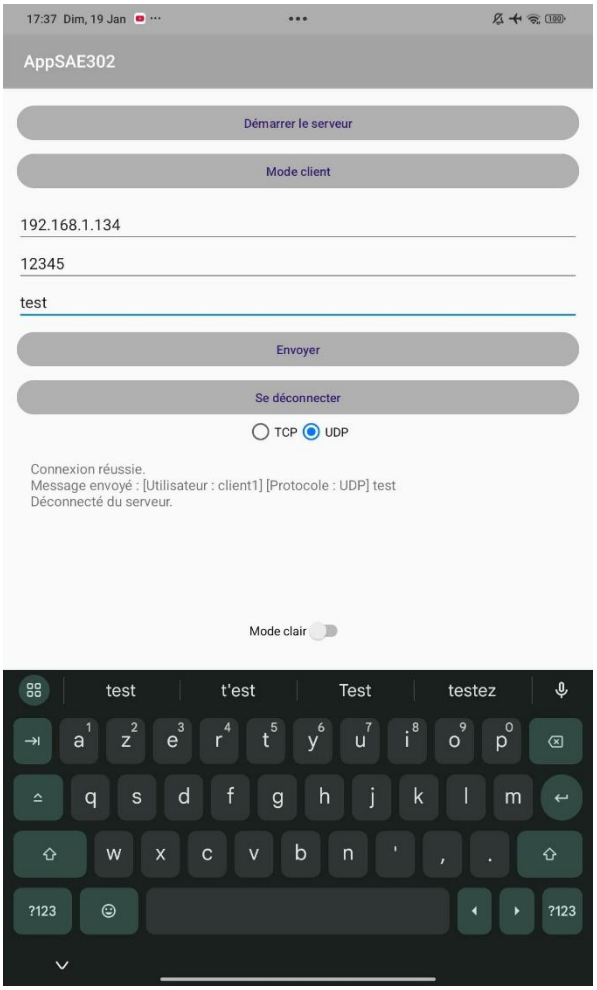


Mode Client

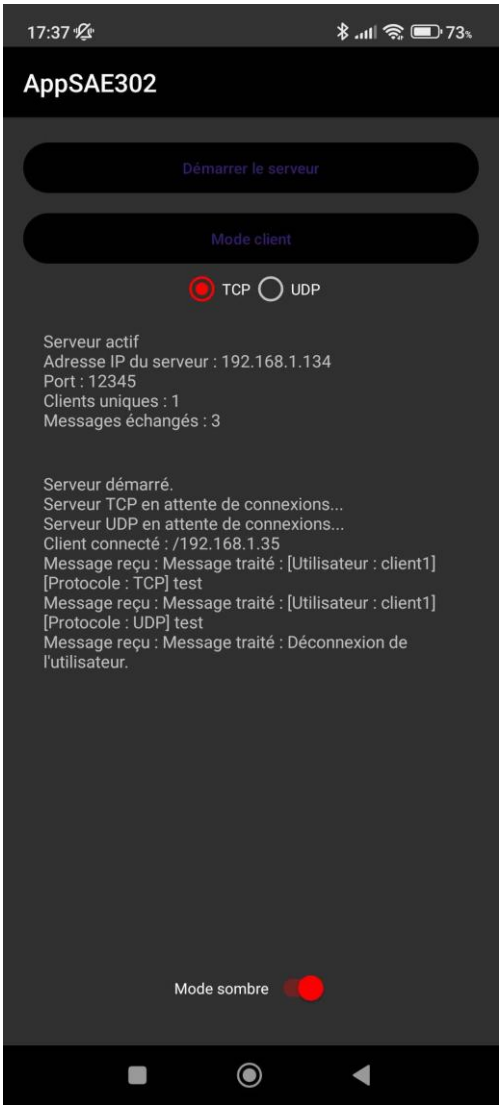
Enfin quand le client envoie un message au serveur :



Mode client TCP



Mode client UDP



Mode serveur

6. Résolutions des erreurs rencontrées

- Protocole TCP/UDP

Problème : Initialement, l'application permettait à l'utilisateur de sélectionner à la fois le protocole TCP et UDP simultanément, ce qui était une source d'ambiguïté dans le traitement des messages.

Solution : Un RadioGroup a été utilisé pour garantir qu'un seul protocole puisse être sélectionné à la fois. Ce composant permet d'assurer une meilleure gestion des choix de l'utilisateur.

- Double clic sur « Démarrer le serveur »

Problème : Lorsqu'un utilisateur cliquait plusieurs fois sur le bouton « Démarrer le serveur », cela provoquait des erreurs, notamment l'utilisation multiple du même port.

Solution : Une variable booléenne isServerRunning a été introduite pour empêcher le redémarrage d'un serveur déjà actif. Si un utilisateur tente de redémarrer le serveur alors qu'il est en cours d'exécution, un message d'erreur est affiché.

- Transition entre mode serveur et client

Problème : Passer du mode serveur au mode client sans réinitialiser les threads actifs causait des conflits ou des comportements inattendus.

Solution : Lors de chaque changement de mode, les threads existants sont correctement arrêtés, et les variables associées sont réinitialisées. Cela garantit une transition fluide et sans erreur entre les deux modes.

- Récupération de l'adresse IP de la machine

Problème : L'identification de l'adresse IP correcte était complexe, notamment sur les appareils Android qui disposent souvent de plusieurs interfaces réseau (WiFi, données mobiles, etc.).

Solution : L'API NetworkInterface a été utilisée pour parcourir toutes les interfaces réseau disponibles et retourner l'adresse IP non locale de l'appareil. Cette approche assure que l'adresse correcte est affichée dans le tableau de bord.

- Validation des champs côté client

Problème : Si un utilisateur tentait d'envoyer un message sans remplir tous les champs requis (IP, port, ou contenu du message), l'application pouvait crasher.

Solution : Des vérifications ont été ajoutées dans la méthode sendMessage() pour garantir que tous les champs soient correctement remplis avant de tenter un envoi. Si un champ est vide, un message d'erreur clair est affiché, et l'application continue de fonctionner normalement.

7. Conclusion

Ce projet a permis de développer une application répondant aux besoins de communication client/serveur tout en explorant des concepts avancés comme le multithreading et l'authentification. Les objectifs principaux ont été atteints :

- Support des protocoles TCP et UDP.
- Transmission de données simples et complexes.
- Gestion efficace de plusieurs connexions.
- Prise d’initiative avec l’implémentation d'un tableau de bord dynamique.
- Ajout d'une personnalisation via le mode clair/sombre.

Le projet a renforcé ma compréhension des communications réseau et de la conception d’applications Android, tout en mettant en pratique des solutions aux problèmes rencontrés.

8. Annexes

Outils et bibliothèques utilisés :

Java : Langage principal pour le développement de l'application.
Android SDK : Framework pour le développement d'applications Android.
Android Studio : Environnement de développement intégré (IDE) utilisé pour écrire, tester et déployer l'application.
ExecutorService : Bibliothèque Java utilisée pour le multithreading côté serveur.

Références :

Cours, TD/TP
Documentation officielle Java / Android
Tutoriels et forums en ligne