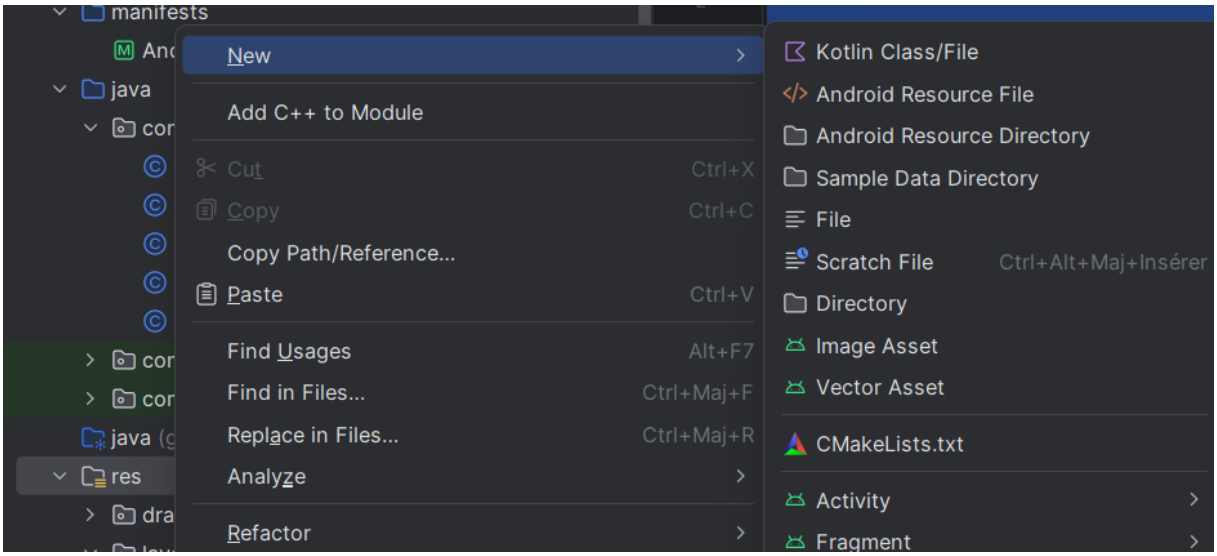
Compte rendu Sae3.02

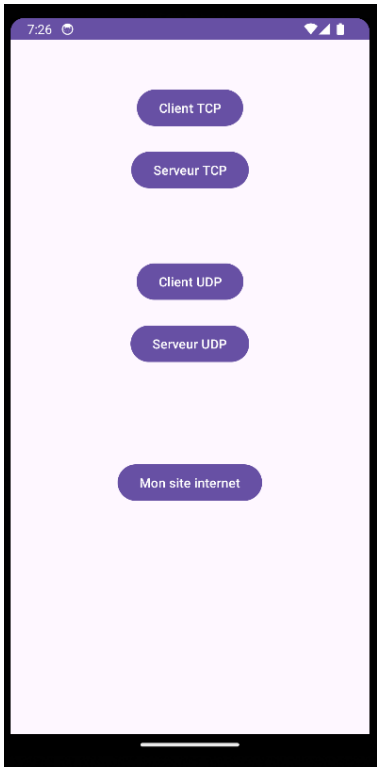
Tout d'abord voici à quoi ressemble mon appli :

Pour modifier le style, je suis allé dans : res => new => Image Asset



Ce qu'il fait que j'ai réussit a changer manuellement son style

Un fois qu'on lance l'application sur mobile, on arrive sur cette page :



Ceci est donc le style de la page principale, définis par le « activity_main.xml »

Dans ce code on peut retrouver :

```
<Button
    android:id="@+id/buttonClient"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Client TCP"
    android:layout_centerHorizontal="true"
    android:layout_marginTop="50dp"
    tools:ignore="HardcodedText" />
```

Le bouton « client TCP »

Avec des définitions de taille, de la position, du type et leur ID qui va permette d'utiliser les boutons.

```
<Button
    android:id="@+id/buttonServer"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_below="@id/buttonClient"
    android:layout_centerHorizontal="true"
    android:layout_marginTop="20dp"
    android:text="Serveur TCP"
    tools:ignore="HardcodedText" />
```

Le bouton qui désigne le serveur TCP

```
<Button
    android:id="@+id/buttonClientUDP"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_below="@id/buttonServer"
    android:layout_centerHorizontal="true"
    android:layout_marginTop="74dp"
    android:text="Client UDP"
    tools:ignore="HardcodedText" />
```

Le bouton qui désigne le client UDP

```
<Button
    android:id="@+id/buttonServerUDP"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_below="@id/buttonClientUDP"
    android:layout_centerHorizontal="true"
    android:layout_marginTop="20dp"
    android:text="Serveur UDP"
    tools:ignore="HardcodedText" />
```

Le bouton qui désigne le serveur UDP

```
<Button
    android:id="@+id/buttonOpenURL"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_alignParentBottom="true"
    android:layout_centerHorizontal="true"
    android:layout_marginBottom="251dp"
    android:text="Mon site internet" />
```

Le bouton qui désigne le site internet

Le .xml permet de faire le style et d'organiser la page en la liant a un .java comme vu ici :

```
<RelativeLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">
```

Ensuite ce qui permet de se déplacer de page en page en cliquant sur les boutons c'est le .java, soit le « MainAcitivity.java » pour ce xml.

Alors d'abord pour simplifier l'explication qu'est ce qu'il y a dans un .java :

Tout d'abord le package :

```
package com.example.sae302;
```

cela nous permet de savoir a quelle projet est ce code java.

Ensuite on retrouve les imports pour définir ce que le code va avoir besoin pour fonctionner

```
import android.annotation.SuppressLint;
import android.content.Intent;
import android.net.Uri;
import android.os.Bundle;
import android.view.View;
import android.widget.Button;
import androidx.appcompat.app.AppCompatActivity;
```

et puis la classe principale

```
public class MainActivity extends AppCompatActivity {
```

Ceci est la chose commune a tous les fichiers java. Ensuite on le code selon nos besoin.

Donc pour rappelle dans la page principale on veut se déplacer vers les différents points de vue.

Alors on va utiliser ce code :

```
@SuppressLint({"MissingInflatedId", "LocalSuppress"}) Button buttonClient =
findViewById(R.id.buttonClient);
buttonClient.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View view) {
        Intent intent = new Intent(MainActivity.this, ClientActivity.class);
        startActivity(intent);
    }
```

Si on le décompose on peut donc expliquer que :

```
@SuppressLint({"MissingInflatedId", "LocalSuppress"}) Button buttonClient =
findViewById(R.id.buttonClient);
```

L'annotation `@SuppressLint` est utilisée pour supprimer les avertissements spécifiques au code, comme les avertissements liés à l'ID manquant ou les suppressions locales. Ce code vise le bouton nommé "buttonClient" en le créant également dans le code.

```
buttonClient.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View view) {
        Intent intent = new Intent(MainActivity.this, ClientActivity.class);
        startActivity(intent);
    }
```

Ce code permet que lorsque le bouton est cliqué, une nouvelle intention (Intent) est créée pour démarrer une nouvelle activité appelée ClientActivity, soit il vise le fichier java ClientActivity. Cette intention est initiée à partir de l'activité actuelle (MainActivity) en utilisant `MainActivity.this`, puis la méthode `startActivity()` est appelée pour démarrer l'activité ClientActivity.

En résumé, ce code relie un bouton à une action qui lance une nouvelle activité lorsque le bouton est cliqué.

Dans mon fichier MainActivity.java, on va alors pouvoir retrouver tous les codes pour les boutons :

Bouton Server TCP :

```
@SuppressLint({"MissingInflatedId", "LocalSuppress"}) Button buttonServer =
findViewById(R.id.buttonServer);
// Ajouter un écouteur de clic au bouton Server
buttonServer.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View view) {
        // Lorsque le bouton Server est cliqué, ouvrir l'activité Server
        Intent intent = new Intent(MainActivity.this, ServerActivity.class);
        startActivity(intent);
    }
```

Bouton Client UDP :

```
@SuppressLint({"MissingInflatedId", "LocalSuppress"}) Button buttonClientUDP =
findViewById(R.id.buttonClientUDP);
/ Ajouter un écouteur de clic au bouton ClientUDP
buttonClientUDP.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View view) {
        // Lorsque le bouton ClientUDP est cliqué, ouvrir l'activité ClientUDP
        Intent intent = new Intent(MainActivity.this, ClientUDPActivity.class);
        startActivity(intent);
    }
```

Bouton Serveur UDP :

```
// Récupérer le bouton Server UDP
@SuppressLint({"MissingInflatedId", "LocalSuppress"}) Button buttonServerUDP =
findViewById(R.id.buttonServerUDP);
// Ajouter un écouteur de clic au bouton ServerUDP
buttonServerUDP.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View view) {
        // Lorsque le bouton ServerUDP est cliqué, ouvrir l'activité ServerUDP
        Intent intent = new Intent(MainActivity.this, ServerUDPActivity.class);
        startActivity(intent);
    }
```

Bouton Internet :

```
@SuppressLint({"MissingInflatedId", "LocalSuppress"}) Button buttonOpenURL =
findViewById(R.id.buttonOpenURL);
buttonOpenURL.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View view) {
        // Lorsque le bouton OpenURL est cliqué, ouvrir l'url
        String url = "http://bastenier-edouard.fr";
        Intent intent = new Intent(Intent.ACTION_VIEW);
        intent.setData(Uri.parse(url));
        startActivity(intent);
    }
```

Ici c'est presque la même chose que pour les autres sauf que lorsque l'on clique au lieu de redirigé vers un fichier java, on le redirige vers un url.

Une fois que l'on toute la page principale est prête, ils faut ensuite crée les différentes pages, une fois les fichier java crée on va tout d'abord les rajouter au « AndroidManifest.xml » ceci permet de déclarer les informations essentielles a une application au système android.

Dedans on va pouvoir définir les pages :

```
<activity
    android:name=".ClientActivity"
    android:exported="true"
    android:label="Client"
    tools:ignore="Instantiatable" />
```

Ce code déclare un fichier java (= une activité) nommée "ClientActivity" dans le fichier AndroidManifest.xml, spécifiant son nom de classe, sa capacité à être exportée (accessible à d'autres applications), son nom affiché et la suppression d'un avertissement lié à son instanciabilité.

Et on va pouvoir refaire la même chose pour tous :

Serveur TCP :

```
<activity
    android:name=".ServerActivity"
    android:exported="true"
    android:label="Server"
    tools:ignore="Instantiatable" />
```

Client UDP :

```
<activity
    android:name=".ClientUDPActivity"
    android:exported="true"
    android:label="Client"
    tools:ignore="Instantiatable" />
```

Serveur UDP :

```
<activity
    android:name=".ServerUDPActivity"
    android:exported="true"
    android:label="Server"
    tools:ignore="Instantiatable" />
```

On va d'ailleurs retrouver la page par default lors de la création :

Le nom du package :

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    package="com.example.sae302">
```

Le nom du projet :

```
android:label="@string/app_name"
```

qui reprend donc le nom de mon projet soit « Sae3.02 »

```
android:label="Sae3.02"
```

Et la page principale :

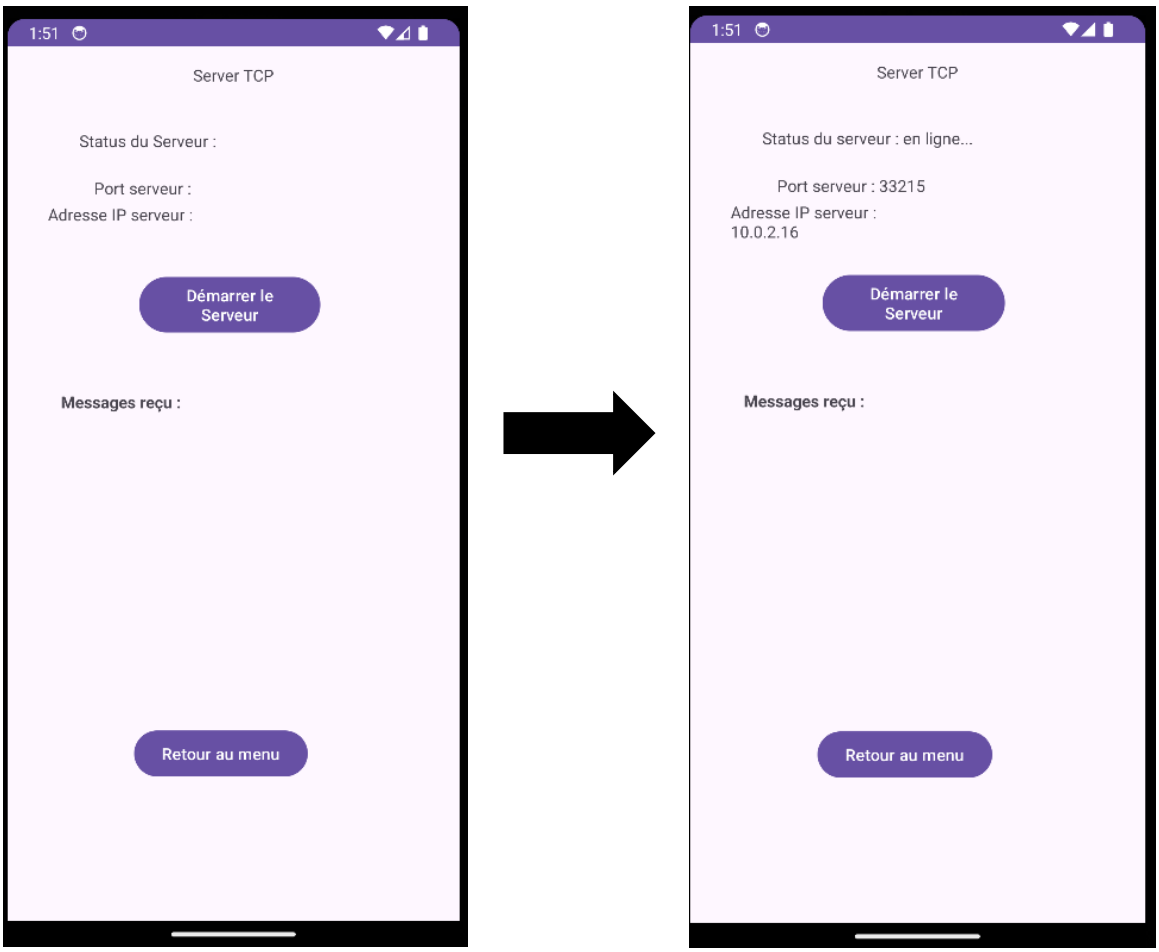
```
<activity
    android:name=".MainActivity"
    android:exported="true">
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
</activity>
```

On peut voir que par rapport au autre elle a « intent-filter » en plus, ce qu'il y a dedans permet de la définir de page de base lorsque l'on lance l'applis, c'est-à-dire que quand on lance l'applis, l'applis nous dirige en premier sur cette page.

Une fois qu'on a déclarer les pages sur le AndroidManifest, et que la page principales est prête, alors on va pouvoir passer au Serveur TCP et a son client.

Pour rappeler, le protocole TCP (Transmission Control Protocol) est un protocole de communication qui assure une transmission fiable et ordonnée des données sur les réseaux informatiques. Il divise les données en segments, gère le contrôle de flux, garantit la livraison des données dans l'ordre correct et fournit des mécanismes de détection et de récupération d'erreurs. TCP est largement utilisé pour les applications nécessitant une fiabilité élevée, telles que le transfert de fichiers, la navigation web et la messagerie électronique. Il permet que lorsque le client se connecte au serveur il reste connecté au serveur et envoie les messages par suite.

Voilà a quoi ressemble la page du serveur TCP :



Passons donc a l'explication du code du serveur : (« ServerActivity.main »)

1. Définition de la class principale

```
package com.example.sae302;

import android.os.Bundle;
import android.view.View;
import android.widget.Button;
import android.widget.TextView;
import androidx.appcompat.app.AppCompatActivity;
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.net.ServerSocket;
import java.net.Socket;
import java.net.InetAddress;

public class ServerActivity extends AppCompatActivity {
```

définition de la class ServerActivity qui correspond a definir le nom du code lui-même.

2. Déclarations des éléments de l'interface utilisateur et des variables de contrôle :

```
private Button buttonStartServer;
private TextView textViewIPAddress;
private TextView textViewPort;
private Button buttonBackToMain;
private TextView textViewServerStatus;
// Déclaration de la TextView pour afficher les messages reçus
private TextView textViewReceivedMessages;

private boolean serverRunning = false;
private ServerSocket serverSocket;
private boolean firstConnection = true;
```

Ces lignes déclarent les éléments de l'interface utilisateur (boutons et textViews) ainsi que quelques variables de contrôle pour gérer l'état du serveur.

3. Initialisation des éléments de l'interface utilisateur :

```
buttonStartServer = findViewById(R.id.buttonStartServer);
textViewIPAddress = findViewById(R.id.textViewIPAddress);
textViewPort = findViewById(R.id.textViewPort);
buttonBackToMain = findViewById(R.id.buttonBackToMain);
textViewServerStatus = findViewById(R.id.textViewServerStatus);
// Initialisation de la TextView pour afficher les messages reçus
textViewReceivedMessages = findViewById(R.id.textViewReceivedMessages);
```

Ces lignes initialisent les éléments de l'interface utilisateur en récupérant les références des vues depuis les fichiers de présentation XML.

4. Configuration des écouteurs de clic pour les boutons :

```
buttonStartServer.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View view) {
        if (!serverRunning) {
            startServer();
        }
    }
});
```

```
buttonBackToMain.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View view) {
        finish();
    }
});
```

Ces lignes définissent les écouteurs de clic pour les boutons "Start Server" et "Back To Main". Lorsque le bouton "Start Server" est cliqué, la méthode `startServer()` est appelée pour démarrer le serveur. Lorsque le bouton "Back To Main" est cliqué, l'activité est terminée.

5. Méthode “startServer()” :

```
private void startServer() {
    new Thread(new Runnable() {
        @Override
        public void run() {
            try {
                serverSocket = new ServerSocket(0);
                serverRunning = true;

                runOnUiThread(new Runnable() {
                    @Override
                    public void run() {
                        textViewIPAddress.setText("Adresse IP serveur : " +
getLocalIpAddress());
                        textViewPort.setText("Port serveur : " +
serverSocket.getLocalPort());
                        textViewServerStatus.setText("Status du serveur : en
ligne...");
                    }
                });
```

Cette méthode crée un nouveau thread pour gérer les connexions entrantes sur un socket. Elle crée un `ServerSocket` pour écouter les connexions entrantes sur un port déterminé. Ensuite, elle met à jour l'interface utilisateur avec l'adresse IP et le port du serveur, ainsi que son statut en ligne.

```
while (serverRunning) {
                    Socket clientSocket = serverSocket.accept();
                    // Crée un nouveau thread a chaque client
                    Thread clientThread = new Thread(new
ClientHandler(clientSocket));
                    clientThread.start();

                    // Afficher une notification lorsque le client se connecte
                    final String clientAddress =
clientSocket.getInetAddress().getHostAddress();
                    runOnUiThread(new Runnable() {
                        @Override
                        public void run() {
                            textViewReceivedMessages.append("Client connecté: " +
clientAddress + "\n");
                        }
                    });
                }


            } catch (IOException e) {
                e.printStackTrace();
            } finally {
                closeServerSocket();
            }
```

```
        }
    }).start();
}
```

Dans cette partie, le serveur entre dans une boucle qui attend les connexions entrantes des clients. Lorsqu'une connexion est acceptée, un nouveau thread `ClientHandler` est créé pour gérer cette connexion client. De plus, une notification est affichée dans l'interface utilisateur indiquant que le client est connecté. Cette boucle s'exécute tant que le serveur est en cours d'exécution.

6. Méthode “closeServerSocket()” :

```
private void closeServerSocket() {
    try {
        if (serverSocket != null && !serverSocket.isClosed()) {
            serverSocket.close();
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

Cette méthode est utilisée pour fermer le `ServerSocket` lorsque le serveur est arrêté ou qu'une exception se produit.

7. Méthode “getLocalIpAddress()” :

```
private String getLocalIpAddress() {
    try {
        InetAddress inetAddress = getWifiInetAddress();
        if (inetAddress != null) {
            return inetAddress.getHostAddress();
        }
    } catch (Exception e) {
        e.printStackTrace();
    }
    return null;
}
```

Cette méthode est utilisée pour obtenir l'adresse IP locale du périphérique.

8. “Méthode “getWifiInetAddress()” :

```
private InetAddress getWifiInetAddress() {
    android.net.wifi.WifiManager wifiManager = (android.net.wifi.WifiManager)
getApplicationContext().getSystemService(WIFI_SERVICE);
    android.net.DhcpInfo dhcpInfo = wifiManager.getDhcpInfo();
    if (dhcpInfo != null) {
        int ip = dhcpInfo.ipAddress;
        try {
            return InetAddress.getByAddress(new byte[]{(byte) (ip & 0xff), (byte)
(ip >> 8 & 0xff), (byte) (ip >> 16 & 0xff), (byte) (ip >> 24 & 0xff)});
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
    return null;
}
```

Cette méthode est utilisée pour obtenir l'adresse IP locale du périphérique en utilisant les informations DHCP.

9. Déclaration de la classe "ClientHandler" :

```
private class ClientHandler implements Runnable {
    private Socket clientSocket;

    public ClientHandler(Socket clientSocket) {
        this.clientSocket = clientSocket;
    }
```

Dans cette partie, la classe `ClientHandler` est déclarée comme une classe interne de la classe principale `ServerActivity` et implémente l'interface `Runnable`. Le constructeur prend un objet `Socket` représentant la connexion du client et l'assigne à la variable `clientSocket`.

10. Méthode "run()" :

```
@Override
public void run() {
    BufferedReader inputReader = null;
    try {
        inputReader = new BufferedReader(new
InputStreamReader(clientSocket.getInputStream()));
        final String clientAddress =
clientSocket.getInetAddress().getHostAddress();
```

Dans la méthode `run()`, un objet `BufferedReader` est créé pour lire les données entrantes à partir du flux d'entrée associé au socket du client. L'adresse IP du client est également obtenue à partir du socket et stockée dans `clientAddress`.

```
while (serverRunning) {
    final String receivedMessage = inputReader.readLine();
    if (receivedMessage != null) {
```

La boucle `while` s'exécute tant que le serveur est en cours d'exécution (`serverRunning` est vrai). À chaque itération, la méthode lit une ligne du flux d'entrée, représentant un message du client.

```
if (receivedMessage.trim().startsWith("[") ||
receivedMessage.trim().startsWith("{")) {
            continue;
        } else if (receivedMessage.trim().startsWith("\"")) {
            final String modifiedMessage = receivedMessage.replaceAll("\"", "");
            runOnUiThread(new Runnable() {
                @Override
                public void run() {
                    textViewReceivedMessages.append(clientAddress + " : " +
modifiedMessage + "\n");
                }
            });
        } else {
            final String modifiedMessage = receivedMessage.replaceAll("[{};,]",
"");
            runOnUiThread(new Runnable() {
                @Override
                public void run() {
                    textViewReceivedMessages.append(clientAddress + " : " +
modifiedMessage + "\n");
                }
            });
        }
    }
}
```

Cette partie gère le traitement du message reçu :

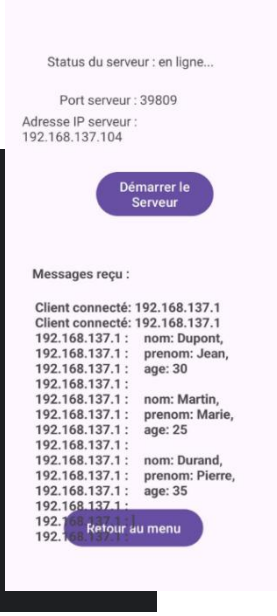
- Si le message commence par `[` ou `{`, il est dans un format spécial et est ignoré.

- Si le message commence par `"`, il est dans un format spécifique et les caractères `"` sont supprimés.

- Sinon, les caractères `{`, `}`, `;` et `,` sont supprimés.

- Ensuite, le message modifié est ajouté à `textViewReceivedMessages` pour affichage dans l'interface utilisateur. La mise à jour de l'interface utilisateur est effectuée sur le thread principal avec `runOnUiThread()`.

Ce qui lui permet de traiter les messages en json :




```
} catch (IOException e) {
            e.printStackTrace();
        } finally {
            try {
                if (inputReader != null) {
                    inputReader.close();
                }
                if (clientSocket != null && !clientSocket.isClosed()) {
                    clientSocket.close();
                }
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }

}
```

Dans la partie `catch`, toute exception liée à la lecture du flux d'entrée est imprimée. Enfin, dans la clause `finally`, les ressources sont nettoyées :

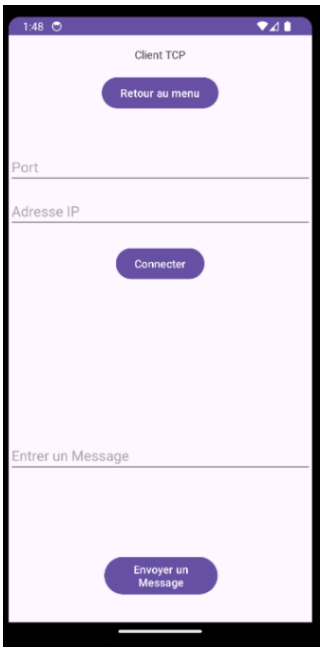
- Le `BufferedReader` est fermé.

- Le socket client est fermé s'il n'est pas déjà fermé.

En résumer, ce code fonctionne comme un serveur. Lorsqu'il démarre, il attend les connexions entrantes des clients. Chaque client est géré par un thread distinct (`ClientHandler`). Les messages envoyés par les clients sont reçus et affichés dans l'interface utilisateur. Une fois terminée, l'application ferme proprement les connexions.

Passons ensuite au client TCP,

voilà à quoi ressemble la page client TCP :



Et maintenant expliquons le code :

1. Définition de la class principale

```
package com.example.sae302;

import android.os.Bundle;
import android.view.View;
import android.widget.Button;
import android.widget.EditText;
import android.widget.TextView;
import androidx.appcompat.app.AppCompatActivity;
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStream;
import java.net.Socket;

public class ClientActivity extends AppCompatActivity {
```

2. Déclarations des éléments de l'interface utilisateur et des variables du client :

```
private EditText editTextPort;
private EditText editTextIPAddress;
private Button buttonConnect;
private EditText editTextMessage;
private Button buttonSendMessage;
private TextView textViewStatus;

private Socket clientSocket;
private BufferedReader inputReader;
private OutputStream outputStream;
```

Ces lignes déclarent les éléments de l'interface utilisateur (zones de texte, boutons et texteView) ainsi que les variables nécessaires pour gérer la connexion et les échanges avec le serveur.

3. Initialisation des éléments de l'interface utilisateur et ajout des écouteurs de clic :

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_client);

    editTextPort = findViewById(R.id.editTextPort);
    editTextIPAddress = findViewById(R.id.editTextIPAddress);
    buttonConnect = findViewById(R.id.buttonConnect);
    editTextMessage = findViewById(R.id.editTextMessage);
    buttonSendMessage = findViewById(R.id.buttonSendMessage);
    textViewStatus = findViewById(R.id.textViewStatus);

    buttonConnect.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View view) {
            connectToServer();
        }
    });

    buttonSendMessage.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View view) {
            sendMessage();
        }
    });
```

```
// Ajouter le bouton pour revenir à la page principale
    Button buttonBackToMain = findViewById(R.id.buttonBackToMain);
    buttonBackToMain.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View view) {
            finish(); // Ferme l'activité actuelle pour revenir à la principale
        }
    });
}
```

Ces lignes initialisent les éléments de l'interface utilisateur en récupérant les références des vues depuis les fichiers de présentation XML. De plus, elles ajoutent des écouteurs de clic aux boutons "Connect" et "Send Message".

4. Méthode "connectToServer()" :

```
private void connectToServer() {
    new Thread(new Runnable() {
        @Override
        public void run() {
            boolean connectionEstablished = false;

            try {
                int port = Integer.parseInt(editTextPort.getText().toString());
                String ipAddress = editTextIPAddress.getText().toString();

                clientSocket = new Socket(ipAddress, port);
                inputReader = new BufferedReader(new
InputStreamReader(clientSocket.getInputStream()));
                outputStream = clientSocket.getOutputStream();

                runOnUiThread(new Runnable() {
                    @Override
                    public void run() {
                        textViewStatus.setText("Connecter au server");
                    }
                });

                connectionEstablished = true;

                while (connectionEstablished) {
                    try {
                        final String receivedMessage = inputReader.readLine();
                        if (receivedMessage != null) {
                            runOnUiThread(new Runnable() {
                                @Override
                                public void run() {
                                    textViewStatus.setText("Message reçu : " +
receivedMessage);
                                }
                            });
                        }
                    } catch (IOException e) {
                        e.printStackTrace();
                        // Gérer la déconnexion du client ici
                        runOnUiThread(new Runnable() {
                            @Override
                            public void run() {
                                textViewStatus.setText("Connexion perdu");
                            }
                        });
                        connectionEstablished = false; // Terminer la boucle en cas
d'erreur
                    }
                }
```

```
            } catch (final IOException e) {
                e.printStackTrace();
                runOnUiThread(new Runnable() {
                    @Override
                    public void run() {
                        textViewStatus.setText("Erreur de connexion: " +
e.getMessage());
                    }
                });
            }
        }
    }).start();
}
```

Cette méthode est appelée lorsque le bouton "Connect" est cliqué. Elle crée un nouveau thread pour gérer la connexion au serveur. Elle tente de se connecter au serveur en utilisant l'adresse IP et le port fournis. Si la connexion est établie, elle affiche un message de réussite et commence à écouter les messages du serveur en continu.

5. Méthode "sendMessage()" :

```
private void sendMessage() {
        new Thread(new Runnable() {
            @Override
            public void run() {
                try {
                    String message = editTextMessage.getText().toString();
                    outputStream.write((message + "\n").getBytes());
                    outputStream.flush();

                    runOnUiThread(new Runnable() {
                        @Override
                        public void run() {
                            textViewStatus.setText("Message envoyé");
                        }
                    });

                } catch (IOException e) {
                    e.printStackTrace();
                    runOnUiThread(new Runnable() {
                        @Override
                        public void run() {
                            textViewStatus.setText("Erreur envoie du message");
                        }
                    });
                }
            }
        }).start();
    }
}
```

Cette méthode est appelée lorsque le bouton "Send Message" est cliqué. Elle crée un nouveau thread pour envoyer le message saisi au serveur. Elle écrit le message dans le flux de sortie du socket et affiche un message de réussite ou d'erreur.

En résumé, ce code représente une application Android côté client qui peut se connecter à un serveur distant, envoyer des messages.

Vous avez une vidéo si vous le désirez (« Vidéo TCP ») et également la video pour le json

Une fois le client/serveur TCP sont fait avec le multithreading ainsi que sa manière de traiter les json pour garder que les données dans le serveur, On peut alors passer au client/serveur UDP :

Pour rappeler, le protocole UDP (User Datagram Protocol) est un protocole de communication léger et non fiable utilisé pour l'envoi de données sur un réseau IP. Contrairement au TCP, UDP ne garantit pas la livraison des données ni l'ordre dans lequel elles sont reçues. Il est souvent utilisé pour les applications où une perte de données occasionnelle est acceptable, telles que la diffusion en continu, les jeux en ligne et la voix sur IP. UDP est rapide et efficace, mais ne fournit pas de mécanismes de contrôle de flux, de retransmission ou de gestion de la congestion. De plus le protocole se connecte dans le serveur seulement lorsque qu'il a un message a transmettre et se déconnecte directement après. Ce qui signifie qu'il n'y pas besoin de faire un multithreading.

Voyons donc le code du serveur : (« ServerUDPActivity.java »)

1. Définition de la class principale et du package

```
package com.example.sae302;

import android.os.Bundle;
import android.view.View;
import android.widget.Button;
import android.widget.TextView;
import androidx.appcompat.app.AppCompatActivity;
import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.Inet4Address;
import java.net.InetAddress;
import java.net.NetworkInterface;
import java.net.SocketException;
import java.util.Enumeration;

public class ServerUDPActivity extends AppCompatActivity {
```

Ce code représente une application Android qui agit en tant que serveur UDP, capable de recevoir des datagrammes UDP (User Datagram Protocol). Voici une explication détaillée :

2. Déclarations des éléments de l'interface utilisateur et des variables du serveur :

```
private TextView textViewServerStatus;
private TextView textViewUDPPort;
private TextView textViewUDPIP;
private TextView textViewReceivedMessagesUDP;

private DatagramSocket serverSocket;
```

Ces lignes déclarent les éléments de l'interface utilisateur (textViews) ainsi que la variable `serverSocket` nécessaire pour gérer la connexion du serveur.

3. Initialisation des éléments de l'interface utilisateur et ajout des écouteurs de clic :

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_server_udp);
```

```
    textViewServerStatus = findViewById(R.id.textViewServerStatus);
    textViewUDPPort = findViewById(R.id.textViewUDPPort);
    textViewUDPIP = findViewById(R.id.textViewUDPIP);
    textViewReceivedMessagesUDP = findViewById(R.id.textViewReceivedMessagesUDP);

    Button buttonStartUDPServer = findViewById(R.id.buttonStartUDPServer);
    buttonStartUDPServer.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View view) {
            startUDPServer();
        }
    });

    Button buttonBackToMainUDPServer =
findViewById(R.id.buttonBackToMainUDPServer);
    buttonBackToMainUDPServer.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View view) {
            finish();
        }
    });
}
```

Ces lignes initialisent les éléments de l'interface utilisateur en récupérant les références des vues depuis les fichiers de présentation XML. De plus, elles ajoutent des écouteurs de clic aux boutons pour démarrer le serveur UDP et revenir à l'écran principal.

4. Méthode `startUDPServer()` :

```
private void startUDPServer() {
    new Thread(new Runnable() {
        @Override
        public void run() {
            try {
                serverSocket = new DatagramSocket(0); // Utilise un port aléatoire
                byte[] receiveData = new byte[1024];

                runOnUiThread(new Runnable() {
                    @Override
                    public void run() {
                        textViewServerStatus.setText("Status du serveur : en
ligne...");
                        textViewUDPPort.setText("Port serveur : " +
serverSocket.getLocalPort());
                        textViewUDPIP.setText("Adresse IP serveur: " +
getLocalIpAddress());
                    }
                });
                while (true) {
                    DatagramPacket receivePacket = new DatagramPacket(receiveData,
receiveData.length);
                    serverSocket.receive(receivePacket);
                    final String message = new String(receivePacket.getData(), 0,
receivePacket.getLength());
                    final String clientAddress =
receivePacket.getAddress().getHostAddress();

                    runOnUiThread(new Runnable() {
                        @Override
                        public void run() {
                            if (firstConnection) {
                                textViewReceivedMessagesUDP.append("Client connecté
: " + clientAddress + "\n");
                                firstConnection = false;
                            } else {
                                textViewReceivedMessagesUDP.append(clientAddress +
```

```
" : " + message + "\n");
                                }
                            }
                        });
                }
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    }).start();
}
```

Cette méthode est appelée lorsque le bouton "Start UDPServer" est cliqué. Elle crée un nouveau thread pour gérer la réception des datagrammes UDP. Elle initialise un socket UDP sur un port aléatoire, récupère les données reçues, puis les affiche dans l'interface utilisateur.

5. Méthode `getLocalIpAddress()` :

```
private String getLocalIpAddress() {
    try {
        for (Enumeration<NetworkInterface> en =
NetworkInterface.getNetworkInterfaces(); en.hasMoreElements(); ) {
            NetworkInterface intf = en.nextElement();
            for (Enumeration<InetAddress> enumIpAddr = intf.getInetAddresses();
enumIpAddr.hasMoreElements(); ) {
                InetAddress inetAddress = enumIpAddr.nextElement();
                if (!inetAddress.isLoopbackAddress() && inetAddress instanceof
Inet4Address) {
                    return inetAddress.getHostAddress();
                }
            }
        }
    } catch (SocketException e) {
        e.printStackTrace();
    }
    return null;
}
```

Cette méthode permet de récupérer l'adresse IP locale du serveur en parcourant les interfaces réseau disponibles et en sélectionnant l'adresse IPv4 non bouclable.

6. Méthode `onDestroy()` :

```
@Override
protected void onDestroy() {
    super.onDestroy();
    if (serverSocket != null && !serverSocket.isClosed()) {
        serverSocket.close();
    }
}
```

Cette méthode est appelée lorsque l'activité est détruite. Elle assure que le socket du serveur est correctement fermé pour libérer les ressources réseau.

En résumé, ce code représente une application Android qui agit en tant que serveur UDP, capable de recevoir des datagrammes UDP et d'afficher les messages reçus dans l'interface utilisateur de manière asynchrone.

Passons donc ensuite au Client UDP :

1. Définition de la class principale et du package

```
package com.example.sae302;

import android.os.Bundle;
import android.view.View;
import android.widget.Button;
import android.widget.EditText;
import android.widget.Toast;
import androidx.appcompat.app.AppCompatActivity;
import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.InetAddress;

public class ClientUDPActivity extends AppCompatActivity {
```

Ce code représente une application Android qui agit en tant que client UDP, capable de se connecter à un serveur UDP distant et d'envoyer des datagrammes UDP. Voici une explication détaillée :

2. Déclarations des éléments de l'interface utilisateur :

```
private EditText editTextUDPPort;
private EditText editTextUDPIP;
private EditText editTextUDPMessage;
private Button buttonConnectUDP;
private Button buttonSendUDPMessage;
```

Ces lignes déclarent les éléments de l'interface utilisateur nécessaires pour saisir l'adresse IP et le port du serveur, ainsi que le message à envoyer, ainsi que les boutons pour établir la connexion et envoyer le message.

3. Initialisation des éléments de l'interface utilisateur et ajout des écouteurs de clic :

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_client_udp);

    editTextUDPPort = findViewById(R.id.editTextUDPPort);
    editTextUDPIP = findViewById(R.id.editTextUDPIP);
    editTextUDPMessage = findViewById(R.id.editTextUDPMessage);
    buttonConnectUDP = findViewById(R.id.buttonConnectUDP);
    buttonSendUDPMessage = findViewById(R.id.buttonSendUDPMessage);

    buttonConnectUDP.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View view) {
            connectToUDPServer();
        }
    });

    buttonSendUDPMessage.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View view) {
            sendMessageToUDPServer();
        }
    });
```

```
    Button buttonBackToMainUDP = findViewById(R.id.buttonBackToMainUDP);
    buttonBackToMainUDP.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View view) {
            finish();
        }
    });
}
```

Ces lignes initialisent les éléments de l'interface utilisateur en récupérant les références des vues depuis les fichiers de présentation XML. De plus, elles ajoutent des écouteurs de clic aux boutons pour établir la connexion avec le serveur et envoyer le message ainsi que retourner à la page principale.

4. Méthode `connectToUDPServer()` :

```
private void connectToUDPServer() {
    final String serverIP = editTextUDPIP.getText().toString();
    final int serverPort = Integer.parseInt(editTextUDPPort.getText().toString());

    new Thread(new Runnable() {
        @Override
        public void run() {
            try {
                DatagramSocket socket = new DatagramSocket();
                byte[] sendData = new byte[1024];

                InetAddress serverAddress = InetAddress.getByName(serverIP);
                DatagramPacket sendPacket = new DatagramPacket(sendData,
sendData.length, serverAddress, serverPort);

                socket.send(sendPacket);
                socket.close();
                runOnUiThread(new Runnable() {
                    @Override
                    public void run() {
                        Toast.makeText(ClientUDPActivity.this, "Connecter au Server
UDP", Toast.LENGTH_SHORT).show();
                    }
                });
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    }).start();
}
```

Cette méthode est appelée lorsque le bouton "Connect UDP" est cliqué. Elle crée un nouveau thread pour établir la connexion avec le serveur UDP en envoyant un datagramme vide.

5. Méthode `sendMessageToUDPServer()` :

```
private void sendMessageToUDPServer() {
    final String serverIP = editTextUDPIP.getText().toString();
    final int serverPort = Integer.parseInt(editTextUDPPort.getText().toString());
    final String message = editTextUDPMessage.getText().toString();

    new Thread(new Runnable() {
        @Override
        public void run() {
            try {
                DatagramSocket socket = new DatagramSocket();
```

```
                byte[] sendData = message.getBytes();

                InetAddress serverAddress = InetAddress.getByName(serverIP);
                DatagramPacket sendPacket = new DatagramPacket(sendData,
sendData.length, serverAddress, serverPort);

                socket.send(sendPacket);
                socket.close();
                runOnUiThread(new Runnable() {
                    @Override
                    public void run() {
                        Toast.makeText(ClientUDPActivity.this, "Message envoyer au
Server UDP", Toast.LENGTH_SHORT).show();
                    }
                });
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    }).start();
}
```

Cette méthode est appelée lorsque le bouton "Send UDP Message" est cliqué. Elle crée un nouveau thread pour envoyer le message saisi au serveur UDP en utilisant un datagramme UDP.

En résumé, ce code représente une application Android qui agit en tant que client UDP, capable de se connecter à un serveur UDP distant et d'envoyer des datagrammes UDP de manière asynchrone.

Pour résumer :

La page principale :

- permet a travers les boutons de se déplacer à travers les différentes pages

Le Serveur TCP :

- Le serveur TCP attend les connexions entrantes des clients et gère chaque client dans un thread distinct (`ClientHandler`). (multithreading)

- Les messages envoyés par les clients sont reçus et affichés dans l'interface utilisateur.

- L'application ferme proprement les connexions une fois terminée.

- traite les messages en JSON

Le Client TCP :

- Le client TCP est une application Android qui peut se connecter à un serveur distant et envoyer des messages.

- L'utilisateur saisit l'adresse IP et le port du serveur, puis peut envoyer des messages qui seront transmis au serveur.

Serveur UDP :

- Le serveur UDP est une application Android qui agit en tant que serveur UDP, capable de recevoir des datagrammes UDP.

- Il affiche les messages reçus dans l'interface utilisateur de manière asynchrone.

Client UDP :

- Le client UDP est une application Android qui agit en tant que client UDP, capable de se connecter à un serveur UDP distant et d'envoyer des datagrammes UDP.

- L'utilisateur saisit l'adresse IP et le port du serveur, puis peut envoyer des messages qui seront transmis au serveur.

PS :

Pour information vous pouvez retrouver tout mon projet de la sae3.02 sur Github sous le nom de projet :

https://github.com/Imferno3969/Application-android