IUT de Vélizy-Villacoublay
Département Réseaux et Télécommunications

COMPTE RENDU

SAE DEV

Module RT3SA02

Élaboré par :

Mathias FERNANDES

Mohamed KHAJNANE

Djibril NAMOUNE

2022-2023

Sommaire

1 - Objectif

L’objectif de cette SAE est de réaliser une application de gestion des absences d’élèves basée sur une architecture client-serveur en utilisant les sockets. Et de proposer notre propre protocole pour l'échange d'informations.

Le serveur est écrit en c, et il y a 3 clients : un en c, un en Java et un dernier en Android avec une interface graphique.

Le projet s’est déroulé de la manière suivante :

Dans un premier temps, nous avons codé le serveur en langage C, le code est expliqué plus tard dans ce rapport. Puis, nous avons codé un client en C et en JAVA, qui ont pour but de pouvoir accéder au client et lui faire une requête afin de récupérer la liste des élèves d’une classe (compris dans des fichiers CSV que le serveur est capable de récupérer). Ensuite, le client doit être en mesure de renvoyer au serveur la liste des élèves qui sont présents et absents.

2 - Serveur

Le serveur est un serveur de sockets multi-clients en C. Il utilise des fonctions de socket fournies par les bibliothèques standard de C pour créer un serveur qui peut gérer plusieurs connexions client simultanées.

Qu'est-ce qu'un socket ?

Les sockets sont des flux de données, permettant à des machines locales ou distantes de communiquer entre elles via des protocoles (TCP, UDP). Nous devons créer notre propre protocole lors de cette SAE.

On crée la fonction “doprocessing” afin de gérer les connexions clients lorsqu'elles sont établies. On commence par la déclaration des variables telles que un buffer de taille 256, qui va contenir ce que le client a écrit, une variable ligne de taille 256 également et qui va stocker les lignes lues d'un fichier. On crée aussi 2 pointeurs : un pour le fichier demandé par le client, et l'autre pour le nouveau fichier qui va inscrire la présence des élèves.

Lire depuis un descripteur de fichiers avec read( )

La primitive **read ( )** tente de lire des octets d'un fichier jusqu'à la taille demandée.

```
ssize_t read(int fd, void *buf, size_t count);
```

Ici, nous l'avons utilisée pour lire les données envoyées par le client via le socket. Nous avons une demande de lecture de 256 caractères (la taille du buffer) dans le socket. Les caractères lus sont rendus disponibles dans le buffer.

Ouvrir des fichiers avec fopen( )

La primitive **fopen ( )** permet d'ouvrir un flux de caractère basé sur fichier. On peut choisir entre différents modes d'ouverture du fichier (lecture, écriture ...).

```
FILE *fopen(const char *pathname, const char *mode);
```

Nous avons eu besoin d'appeler la primitive fopen pour 2 raisons. Dans un premier temps, en lecture seule, pour lire le fichier indiqué dans le buffer (le fichier souhaité par le client). Si le fichier demandé n'existe pas, le serveur indique “fichier non trouvé”. Dans un second temps, cette fois en écriture où l'on va créer un fichier texte, nommé "présence" et qui va contenir la liste des élèves et le fait qu'ils soient présents ou absents.

Lire le contenu d'une chaîne de caractère avec fgets( )

La primitive **fgets ( )** lit les caractères d'un fichier jusqu'à la rencontre d'un retour chariot (signe de fin de ligne) ou d'un EOF (end of file = fin de fichier), ou bien jusqu'à ce qu'il ne reste plus qu'un seul caractère libre dans le tableau.

```
char *fgets(char *s, int size, FILE *stream);
```

Nous avons eu besoin d'appeler la fonction fgets afin de pouvoir lire ligne par ligne le fichier demandé par le client grâce à une boucle *while*. Tant que l'on peut lire une ligne du fichier, on l'envoie au client et ensuite on affiche la ligne dans le serveur pour montrer qu'elle a bien été envoyée.

Écrire des octets de valeur 0 dans un bloc d'octets avec bzero ( )

La primitive **bzero ( )** met à 0 les n premiers blocs d'octets du bloc pointé par s.

```
void bzero(void *s, size_t n);
```

Il est important de vider le buffer à chaque fois car il faut laisser l'allocation de mémoire vide pour la prochaine ligne.

Fermer un descripteur de fichiers avec close ( )

La primitive **close ( )** ferme le descripteur *fd*, de manière à ce qu'il ne référence plus aucun fichier, et ne puisse être réutilisé.

```
int close(int fd);
```

Nous avons utilisé la fonction **close ( )** pour fermer les sockets.

Programme 1 Notre serveur en C

```c
/* multipleServerSocket.c */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

void doprocessing (int sock)
{
    //Déclaration des variables
    char buffer[256];
    char ligne[256];
    FILE* fichier;
    FILE* presence;

    bzero(buffer,256); //Compléter le buffer avec des 0
    read(sock,buffer,256);
    buffer[strcspn(buffer, "\n")]='\0'; //Remplace les sauts de lignes par le caractère nul dans le buffer
    fichier = fopen(buffer,"r"); //Ouvrir le fichier en lecture seule
    printf("Fichier demandé par le client: %s\n",buffer);

    //Si le fichier entré n'existe pas
    if (fichier == NULL){
        printf("Fichier non trouvé\n");
    }

   //Création d'un nouveau fichier dans lequel on va écrire qui est absent ou présent
    presence = fopen("presence.txt", "w");

    //Tant qu'on parvient à lire une ligne dans le fichier.
    while (fgets(ligne, 256, fichier)){
        bzero(buffer,256);
        write(sock,ligne,strlen(ligne)); //envoi de la ligne au client
        printf("%s",ligne);
        read(sock,buffer,256); //lit la réponse

        //Si le client a noté l'élève absent
        if (strchr(buffer, 'n')){
                strcat(ligne, "absent\n"); //concaténation
                fputs(ligne, presence); //On écrit la chaîne dans le fichier
        }
        //Si le client a noté l'élève présent
        else if (strchr(buffer, 'o')){
                strcat(ligne, "présent\n");
                fputs(ligne, presence);
        }
    }
    //Fermeture du socket et des fichiers
    fclose(fichier);
    fclose(presence);
    close(sock);
}
```

```c
int main( int argc, char *argv[] )
{
    int sockfd, newsockfd, portno;
    unsigned int clilen;

    struct sockaddr_in serv_addr, cli_addr;
    pid_t pid;

    /* Premier appel de la fonction socket() */
    sockfd = socket(PF_INET, SOCK_STREAM, 0);

    /* Initialisation socket structure */
    bzero((char *) &serv_addr, sizeof(serv_addr));
    portno = 5001;

    serv_addr.sin_family = AF_INET;
    serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
    serv_addr.sin_port = htons(portno);

    /* Now bind the host address using bind() call.*/
    bind(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr));

    /* Now start listening for the clients, here
     * process will go in sleep mode and will wait
     * for the incoming connection */
    listen(sockfd,5);
    clilen = sizeof(cli_addr);



  while (1)
  {
      newsockfd = accept(sockfd, (struct sockaddr *) &cli_addr,
                         &clilen);
      /* Create child process */
      pid = fork();

      if (pid == 0)
      {
          /* This is the client process */
          close(sockfd);
          doprocessing(newsockfd);
          exit(0);
      }

      else
      {
          close(newsockfd);
      }
  } /* fin de boucle */
}
```

3 - Client

3.1 - Client en C

Le client se connecte au serveur pour demander un fichier (ici les listes d'élèves) afin qu'il puisse noter leur présence ou leur absence. Il utilise les sockets pour communiquer avec le serveur.

Débuter une connexion sur un socket avec connect ( )

La primitive **connect ( )** connecte le socket référencé par le descripteur de fichier *sockfd* à l'adresse indiquée par *serv_addr*.

```
int connect(int sockfd, const struct sockaddr *addr,
            socklen_t addrlen);
```

Une fois la connexion effectuée, on demande au client le fichier qu'il souhaite avoir afin de faire l'appel. Puis avec la primitive **fgets ( )** vue précédemment, on récupère ce qu'a entré l'utilisateur.

Ensuite, à l'aide d'une boucle *while*, on indique que tant que l'on peut lire une ligne du fichier, on l'affiche.

Formater et afficher des données avec printf ( )

La primitive **printf ( )** produit une sortie en accord avec le *format* décrit plus bas. La fonction **printf( )** écrit sa sortie sur *stdout*, le flux de sortie standard.

```
printf FORMAT [PARAMÈTRE]...
printf OPTION
```

Ici, nous l'avons utilisé dans la boucle *while* afin de demander au client si son élève est présent ou non. Dans le cas où il est présent, on va ajouter "présent" à la suite du nom de l'élève et s'il est absent, on ajoute "absent".

Écrire dans un descripteur de fichier avec write( )

La primitive **write( )** écrit jusqu'au nombre d'octets souhaité dans le fichier associé au descripteur *fd* depuis le tampon pointé par *buf*. Elle renvoie le nombre d'octets écrits (0 signifiant aucune écriture), ou -1 s'il échoue, auquel cas *errno* contient le code d'erreur.

```
#include <unistd.h>

ssize_t write(int fd, const void *buf, size_t count);
```

Nous avons utilisé cette fonction pour écrire la réponse de l'utilisateur sur le descripteur de fichiers et donc pour pouvoir l'envoyer au serveur.

Programme 2 Notre client en c

```c
/*simpleClientSocket.c*/
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<unistd.h>
#include<sys/types.h>
#include<sys/socket.h>
#include<netinet/in.h>
#include<netdb.h>

int main(int argc,char*argv[])
{
    //Déclaration des variables
    int sockfd,portno;
    char buffer[256];
    char ligne[256];
    char reponse[256];
    struct sockaddr_in serv_addr;
    struct hostent *server;

    if(argc < 3){
       fprintf(stderr,"usage %s hostname port\n", argv[0]);
       exit(0);
    }

    portno = atoi(argv[2]);
```

```c
    /* Create a socket point */
    sockfd = socket(PF_INET,SOCK_STREAM,0);

    server = gethostbyname(argv[1]);

    serv_addr.sin_family = AF_INET;
    bcopy((char *)server->h_addr, //identique à memcpy
                        (char*)&serv_addr.sin_addr.s_addr,server->h_length);

    serv_addr.sin_port = htons(portno);

    //Connection au serveur
    connect(sockfd,(struct sockaddr *)&serv_addr,sizeof(serv_addr));

    /*Demander au client d'entrer le fichier souhaité.
    Ce message sera lu et traité par le serveur*/
    printf("Entrez le nom de fichier du groupe dont vous souhaitez faire l'appel:
");
    bzero(buffer,256);
    fgets(buffer,255,stdin);
    write(sockfd,buffer,strlen(buffer));
```

```
    //Tant qu'on peut lire les lignes, lire la réponse du serveur
    while(read(sockfd,ligne,256)){

       printf("%s",ligne);
       bzero(ligne,256);
       printf("Cet élève est-il présent (o ou n)? \n");//Vérifie la présence de
l'élève
       fgets(reponse,255,stdin); //stocke la réponse dans serveur
       write(sockfd, reponse, strlen(reponse)); //envoi au serveur
       bzero(reponse,256); //On complète le buffer réponse avec des 0

       }

    return 0;
}
```

3.2 - Client en JAVA

De même manière que le client en c, nous avons créé un client en Java afin qu'il demande le fichier de liste d'élèves voulu.

Dans la méthode main, On crée une instance de la classe Client en passant l'adresse IP du serveur en argument. On utilise ensuite la classe Socket pour se connecter au serveur en utilisant l'adresse IP et le port passé en paramètre. On crée également des objets **DataOutputStream** et **DataInputStream** pour envoyer et recevoir les données via les sockets.

On demande ensuite à l'utilisateur d'entrer le fichier souhaité, à l'aide d'un Scanner, afin d'envoyer la requête au serveur. Le code convertit ce message en octets et l'envoie via le **DataOutputStream**. Il envoie également "ok" pour un acquittement.

On rentre ensuite dans une boucle infinie dans laquelle on attend des données du serveur via le **DataInputStream** et affiche ces données à l'écran.
Ensuite nous avons ajouté une condition où l'on vérifie que la taille de la variable n, qui est le nombre d'octets de la ligne lue, n'est pas inférieure à 1. Si elle est inférieure à 1, cela signifie que la ligne est vide donc que le fichier est fini, donc on arrête la lecture.

Enfin, On ferme tous les sockets et les flux de données utilisés avec **close ( )**.

En Java on va utiliser un **Try Catch** pour capturer les exceptions qui peuvent être levées lors de l'exécution du code dans le bloc "try". Les exceptions sont des erreurs qui peuvent survenir à différents moments lors de l'exécution du programme.

Programme 3 Notre client en Java

```java
import java.net.*;
import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.IOException;
import java.net.Socket;
import java.util.Arrays;
import java.util.Scanner;

class Client
{
    private Socket sockfd;
    private DataOutputStream enSortie;
    private DataInputStream enEntree;

    public Client(String hote)
    {
        //Déclaration des variables
        int portno = 5001;
        byte[] buffer;
        String a = "ok";
        byte[] ok;

        try {
            sockfd = new Socket(hote, portno);
            enSortie = new DataOutputStream(sockfd.getOutputStream()); /*données a destination du correspondant*/
            enEntree = new DataInputStream(sockfd.getInputStream()); /*données venant du correspondant*/
            Scanner clavier = new Scanner(System.in);
            System.out.printf("Please enter the message:");
            String str = clavier.nextLine(); //On récupère ce que le client a entré
            buffer = str.getBytes();
            enSortie.write(buffer,0,buffer.length); //On écrit dans le buffer
            ok = a.getBytes();
            while(true){ //Boucle infinie dans lequel on va afficher le fichier

                buffer = new byte[256];
                int n = enEntree.read(buffer,0,buffer.length);
                if (n<1){
                    break;
                }
                System.out.println("n="+ n); //taille de la ligne
                String message = new String(buffer,0,n);
                System.out.println(message);
                enSortie.write(ok,0,ok.length); //acquittement
        }
            //Fermeture des sockets
             enSortie.close();
             enEntree.close();
             sockfd.close();
        }
            catch (IOException e) {
            e.printStackTrace();
        }
    }
    public static void main(String[] args) {
        new Client(args[0]);
    }
}
```

3.3 - Client sur Android

Après avoir créé et configuré le serveur (en C), le premier client (en C), et le deuxième client (en JAVA), on désire maintenant créer un client sous Android. Pour cela, le logiciel que l'on utilisera se nomme Android Studio.
Tout d'abord, il faut créer une nouvelle application sur le logiciel. Puis, la première chose à faire est d'autoriser l'application à utiliser Internet. Pour ce faire, il faut rentrer les lignes suivantes dans le fichier ***AndroidManifest.xml*** de l'application :

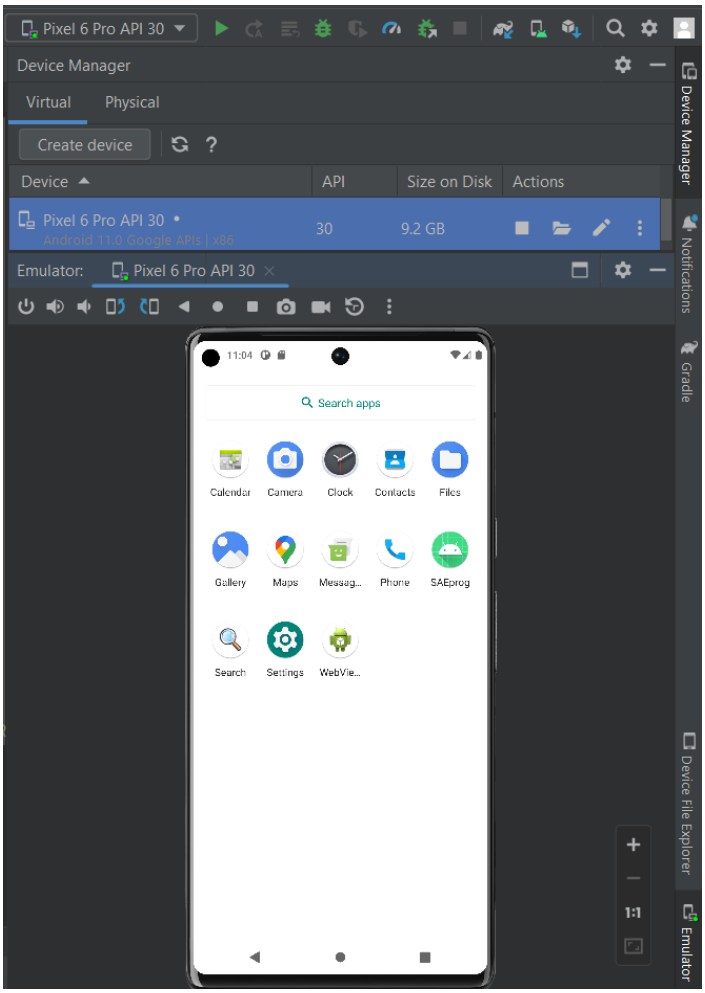
Programme 4 Fichier ***AndroidManifest.xml*** autorisant l'accès à Internet.

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools">

    <uses-permission android:name="android.permission.INTERNET" />
```

La deuxième étape est de générer un appareil Android virtuel qui nous permettra de tester notre application au fur et à mesure de son développement. Rendons nous donc dans la section **Device Manager**. Et il faudra ensuite cliquer sur **Create Device**. On peut désormais lancer notre émulateur android.

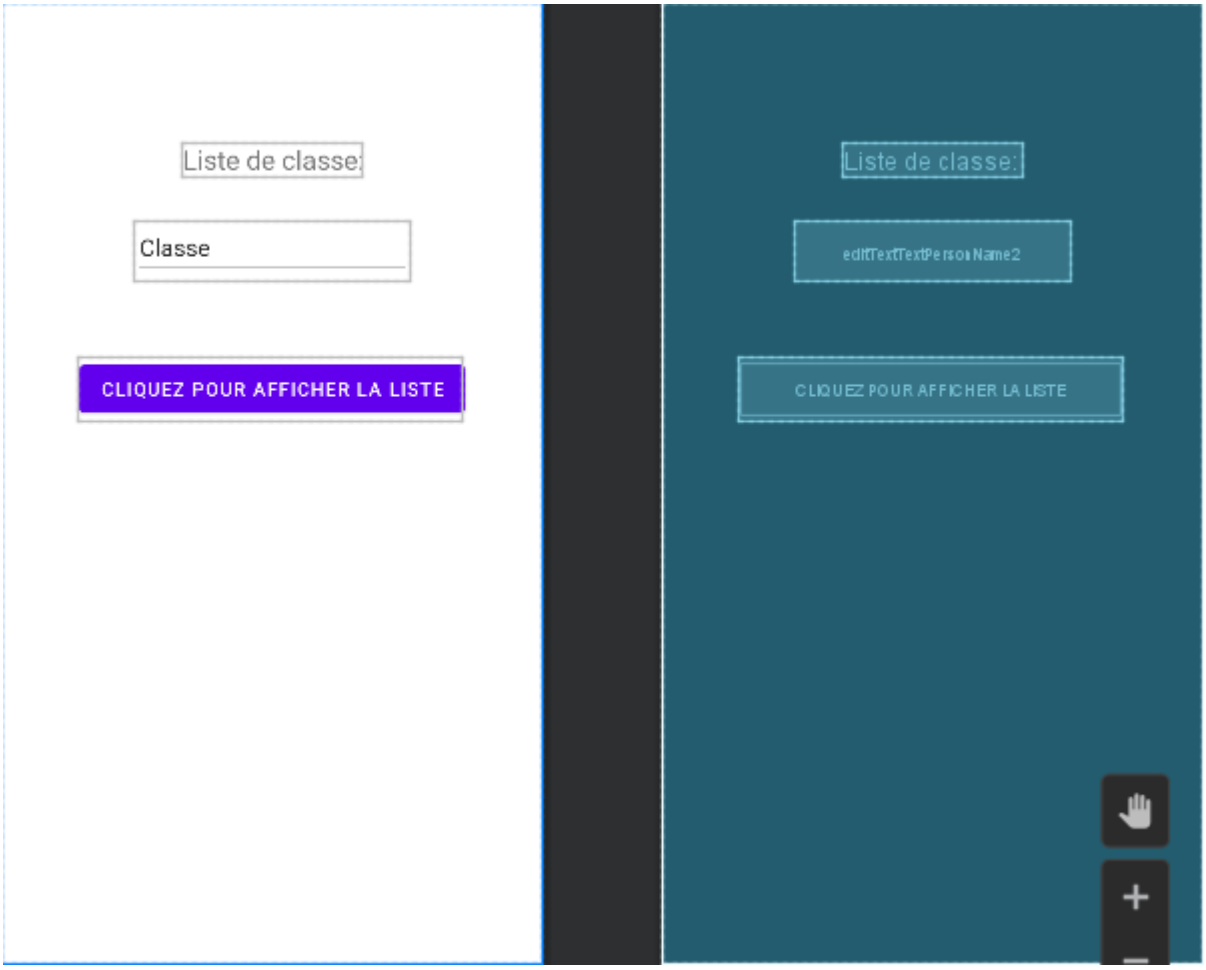
Capture 1 Exemple d'émulation d'un smartphone ***Pixel 6 Pro*** sur Android Studio.



Le développement d'une application android se déroule sur 2 fichiers principalement. Un fichier qui permet d'apporter des modifications graphiques à l'interface de notre application à savoir : le fichier activity_main.xml. Et un fichier qui contient notre code JAVA qui permettra ici de contenir une partie du code client en JAVA décrit précédemment mais adapté à l'interface graphique de notre fichier XML pour que l'application soit plus intuitive pour le client.

Attelons nous tout d'abord à savoir quels types d'outils graphiques (widgets) nous allons utiliser dans notre fichier XML.

Capture 2 Voici ce à quoi devrait ressembler le Design de notre application.



A partir de ce visuel graphique, on obtient un code activity_main.xml. Les éléments graphiques que l'on souhaite utiliser sont les suivants : Un textView qui correspondra à l'affichage du texte **"Liste de classe",** un EditText qui nous permettra de **rentrer un texte dans un champ** (le nom du fichier csv que le client souhaite en l'occurrence). Un Button qui permettra de valider le contenu du texte entré dans le EditText et de l'envoyer au serveur.

Programme 5 Code XML du **textView** d'ID textView

```
<TextView
    android:id="@+id/textView"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Liste de classe:"
    android:textSize="20sp"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent"
    app:layout_constraintVertical_bias="0.151" />
```

Programme 6 Code XML du **Button** d'ID ButtonID1

```
<Button
    android:id="@+id/ButtonID1"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Cliquez pour afficher la liste"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintHorizontal_bias="0.495"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent"
    app:layout_constraintVertical_bias="0.396" />
```

Programme 7 Code XML du **editText** d'ID editTextTextPersonName2

```
<EditText
    android:id="@+id/editTextTextPersonName2"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:ems="10"
    android:inputType="textPersonName"
    android:text="Classe"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintHorizontal_bias="0.497"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent"
    app:layout_constraintVertical_bias="0.24" />
```

Nous allons désormais passer au code JAVA du client Android Studio. La première étape est de configurer notre classe MainActivity qui représente notre classe principale dans laquelle on mettra les méthodes et les variables qui définissent le comportement de l'application lorsqu'elle est lancée. Elle hérite de la sous classe **AppCompatActivity** qui est une sous classe de android.app.Activity.

Programme 8 extrait du code JAVA de la classe **MainActivity**

```
public class MainActivity extends AppCompatActivity {
    2 usages
    private Button monBouton;
    2 usages
    private EditText monText;
    private TextView textView;
```

Ensuite, on configure notre méthode OnCreate() qui a pour but de **mettre en place l'interface graphique, à initialiser les variables.** C'est dans cette méthode que l'on va lier nos variables JAVA aux identifiants des éléments de l'interface graphique XML.

Programme 9 extrait du code JAVA de la méthode **OnCreate**

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    monText = findViewById(R.id.editTextTextPersonName2);
    monBouton = findViewById(R.id.ButtonID1);
```

On va désormais créer une méthode pour mettre en place le système d'interaction avec le serveur lorsque le client cliquera sur le bouton. Cette méthode nous l'avons appelée OnClick. C'est aussi ici que l'on stockera ce que le client aura écrit dans le EditText.

Il faudra également renseigner ici l'adresse IP du serveur auquel on souhaite se connecter. (Ici, on souhaite se connecter à une machine virtuelle qui se situe sur le même PC. Son IP est 192.168.0.162)

On crée un nouveau Thread, qui contiendra une méthode run qui créera un nouvel objet de la classe Client avec les paramètres hostname et valeur (pour envoyer au serveur ce qu'a renseigné le Client dans le champ editText). **Start()** execute le thread.

Programme 10 extrait du code JAVA de la méthode **onClick() et run()**

```java
monBouton.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View view){
        String valeur = monText.getText().toString();
        System.out.println(valeur);

        String hostname = "192.168.0.162";
        new Thread(new Runnable() {
            @Override
            public void run() { new Client(hostname, valeur); }
        }).start();
```

La suite du code est récupérée du code client JAVA. Les explications sont exactement les mêmes par conséquent, il faut se rendre sur la sous partie **3.2 Client en JAVA.**

Voici un aperçu de l'application lorsque l'on exécute le programme.

Nous n'avons pas réussi à afficher sous forme de liste, le contenu de la variable message qui contient elle-même le contenu du fichier CSV. Nous avons rencontré des erreurs que nous n'avons pas réussi à résoudre de type :

Erreur FATAL EXCEPTION

```
E FATAL EXCEPTION: Thread-2
  Process: com.example.saeprog, PID: 20342
  java.lang.RuntimeException: Can't create handler inside thread Thread[Thread-2,5,m
```

4 - Makefile

Qu'est-ce qu'un makefile ?

Les Makefiles sont des fichiers utilisés par la fonction make afin d'automatiser un ensemble d'actions permettant la génération de fichiers, la plupart du temps résultant d'une compilation.

Maintenir des groupes de programmes avec make ( )

La primitive **make ( )** sert à déterminer automatiquement quelles sont les parties d'un gros programme qu'il faut compiler, et d'exécuter les commandes appropriées.

```
make [OPTION]... [TARGET]...
```

Nous avons créé un Makefile afin de compiler les programmes du client en c et du serveur.

Programme 11 Notre Makefile pour le client et le serveur

```
Voici un exemple de makefile qui peut être utilisé pour compiler les fichiers
Client.c et Serveur.c:

CC = gcc
CFLAGS = -Wall -g

all: Client Serveur

Client: Client.o
	$(CC) $(CFLAGS) -o Client Client.o

Client.o: Client.c
	$(CC) $(CFLAGS) -c Client.c

Serveur: Serveur.o
	$(CC) $(CFLAGS) -o Serveur Serveur.o

Serveur.o: Serveur.c
	$(CC) $(CFLAGS) -c Serveur.c

clean:
	rm -f *.o Client Serveur
```

5 - Conclusion

Cette SAE nous a permis de consolider nos bases de programmation en c et en Java, notamment avec la création de la connexion client/serveur à l'aide de sockets. Nous avons pu également avoir une première approche de l'application Android Studio.

Le travail de recherche en autonomie sur Android Studio nous a permis d'en apprendre davantage et d'acquérir de nouvelles compétences qui pourront nous être utiles à l'avenir. De plus, des compétences personnelles comme l'autonomie et le fait de travailler en groupe ont également été sollicitées durant ce projet.