

# RAPPORT DE PROJET

## APPLICATION CHAT

### SERVER C ET CLIENT JAVA

Réalisé par :

EIHOR Ousama

KARAM Hicham

KHADROUF Omar

Master Spécialisé Qualité Logiciel  
Année Universitaire 2011-2012

## Table des matières

<b>Introduction.....</b>	<b>3</b>
<b>Cahier de charge.....</b>	<b>4</b>
Besoins.....	4
Description du serveur.....	5
Description du client.....	5
<b>Serveur en C.....</b>	<b>6</b>
Fonctions.....	6
Description détaillé .....	8
<b>Client en JAVA.....</b>	<b>10</b>
Classes.....	11
Description détaillé .....	12
<b>Conclusion.....</b>	<b>15</b>

# Introduction

Les protocoles de communication comme TCP/IP permettent la communication point à point entre deux applications s'exécutent éventuellement sur deux machines différentes.

Le détail du transfert effectif des données entre deux applications est spécifié par

Le protocole de communication de la couche transport, mais le moment et la façon dont les applications interagissent entre elles sont laissés à la charge du programmeur.

L'architecture client/serveur est devenue la méthode incontournable pour la communication point à point au niveau applicatif, quand le protocole utilisé est de la famille TCP/IP.

Ce modèle est motivé par le fait que TCP/IP ne fournit aucun mécanisme permettant l'exécution automatique d'un programme à l'arrivée d'un message si bien que dans une communication point à point, l'une des applications doit attendre l'initiative de la communication de la part de l'autre application.

❖ Les applications peuvent être classées en deux catégories :

**Les clients** : applications qui prennent l'initiative du lancement de la communication, c'est à dire demande l'ouverture de connexion, l'envoi d'une requête, l'attente de la réponse à la requête, reprise de l'exécution du programme.

**Les serveurs** : applications qui attendent la communication, c'est à dire l'attente d'une demande d'ouverture de connexion, la réception d'une requête et l'envoi d'une réponse

❖ Applications orientées connexion ou sans connexion :

Une application est dite **orientée connexion** si le protocole sous-jacent utilisé est en mode connecté (TCP/IP par exemple).

Une application est dite **orientée sans connexion** si le protocole sous-jacent utilisé est en mode non connecté (UDP/IP par exemple).

L'avantage de l'utilisation d'un protocole comme TCP/IP est la fiabilité : la couche transport effectue elle même le "checksum", la réémission de morceaux de messages perdus, l'élimination des morceaux dupliqués, l'adaptation du débit.

Un protocole comme UDP/IP n'effectue pas ces vérifications qui doivent alors être faites par le protocole de communication de niveau applicatif. Pour cette raison, la programmation de clients ou serveurs en mode non connecté est plus complexe qu'en mode connecté. Cependant, en réseau lo-

cal où le transport est fiable, il peut être avantageux d'utiliser UDP car ce mode de communication demande moins d'opérations que TCP.

# Cahier de charge

## ▣▣ Besoins :

Il s'agit de coder le programme serveur en C et le programme client en JAVA. De cette façon, on peut exécuter le serveur sur une machine Linux pour laquelle on aura compilé et on peut distribuer le programme client en format .class de java à plusieurs personnes qui pourront l'exécuter sur leur plate-forme favorite (si elle dispose d'une machine virtuelle java).

Afin de permettre la communication entre les programmes, vous utilisez un format pour les messages qui est le suivant :

Un octet permettant d'identifier le type de message (chat, lister les clients, quitter, rejoindre).

Un octet permettant de contrôler la réussite de la connexion (si le premier octet est « rejoindre »).

Le contenu applicable à chaque type de message (chat : message, rejoindre : pseudonyme, lister : lister des clients).

## ▣▣ Description du serveur:

Lors du lancement du serveur, celui-ci se met à l'écoute du port indiqué, pour chaque message reçu, il fait le traitement adéquat selon le cas.

Pour stocker la liste des clients, le serveur construit une liste chaînée avec les pseudonymes et les adresse correspondantes. De cette façon lors de la réception d'un message on peut identifier le client qui l'a envoyé.

- Pour les « join », le serveur vérifie que le pseudonyme donné n'est pas encore utilisé, puis il l'ajoute à la liste.
- Les messages « chat », s'ils proviennent d'un client qui a été enregistré, sont envoyés sur toutes les machines du réseau de diffusion. Le texte « From <

pseudonyme > : » est ajouté au début du message pour faciliter l'identification de l'expéditeur aux autres clients.

- Lors de la réception d'un message de « list » de la part d'un client, le serveur lui répond avec un autre message du même type contenant une copie en format texte de sa liste de clients.
- Pour les messages de type « leave », après avoir vérifiés que le client était bien enregistré, le serveur procède à la suppression du client de la liste et lui envoie un message de confirmation.

Si le serveur est interrompu, pour éviter la présence de client orphelin, il envoie un message du type « leave » à tous les clients enregistrés juste avant de finir son exécution.

## ▣▣ Description du client:

Le client est composé de plusieurs classes : socket\_client, InterfaceGraphique, IGListener et le Timer.

- Socket\_client est la classe principale qui contient la méthode main. Elle est chargée de démarrer le client, créer la connexion vers le serveur puis de recevoir tous les messages provenant du serveur. Lors de l'envoi de la requête de connexion au serveur, elle démarre un objet de la classe Timer juste avant de se mettre à l'écoute de la possible réponse. De cette façon, si le serveur n'est pas présent, le client ne reste pas bloqué indéfiniment, parce que le Timer force la sortie du programme après 5 secondes. Une fois connecté, socket\_client reçoit les messages du serveur et fait le traitement adéquat pour chacun d'entre eux. Il démarre aussi l'interface graphique et lui associe un IGListener.
- InterfaceGraphique sert à saisir les actions et messages de l'utilisateur et à afficher les conversations.
- IGListener est une classe qui permet de gérer les événements produits par les composants de l'interface graphique. C'est cette classe qui se charge d'envoyer les messages au serveur. Pour cela elle possède une référence au socket d'émission.
- Timer est une classe très simple qui est lancée dans un thread. Si elle n'est pas arrêtée au bout de 5 secondes, elle fait arrêter le programme pour éviter une situation de blocage quand le serveur ne répond pas ou n'est pas présent.

# Serveur en C

## Fonctions :

```
typedef struct _s_client  
  
int create_server(int port)  
  
int server_accept(int main_sock,int timeout)  
  
int send_msg(int sock,char *msg)  
  
int send_all(char *msg, int not_to)  
  
void client_quit(s_client *me, char *msg)  
  
void *interact(void *param)  
  
int main(int argc, char **argv)
```

## Description détaillé :

**La fonction create\_server sert à accepter une connexion avec ou sans timeout**

```
int create_server(int port)  
{  
    int sock,optval = 1;  
    struct sockaddr_in sockname;
```

```

if((sock = socket(PF_INET,SOCK_STREAM,0))<0)
{
    printf("Erreur d'ouverture de la socket");
    exit(-1);
}
setsockopt(sock,SOL_SOCKET,SO_REUSEADDR,&optval,sizeof(int));
memset((char *) &sockname,0,sizeof(struct sockaddr_in));
sockname.sin_family = AF_INET;
sockname.sin_port = htons(port);
sockname.sin_addr.s_addr = htonl(INADDR_ANY);
if(bind(sock,(struct sockaddr *) &sockname, sizeof(struct sockaddr_in)) < 0)
{
    printf("Erreur de bind!");
    exit(-1);
}
if(listen(sock,LS_CLIENT_NB) < 0){
    printf("listen error!");
    exit(-1);}
return sock;
}
    
```

**La fonction send\_msg envoie une chaîne de caractère à un client.**

```

int send_msg(int sock,char *msg)
{
    return send(sock,msg,strlen(msg),0);
}
    
```

**La fonction send\_all : envoie un message à tout le monde sauf au socket not\_to**

```

int send_all(char *msg, int not_to)
{
    int i;

    pthread_mutex_lock(&mutex);    // debut de la section critique
    for(i=0;i<first_free;i++)
    {
        if(clients[i]->sock != not_to)
            send_msg(clients[i]->sock,msg);
    }
}
    
```

```
pthread_mutex_unlock(&mutex); // fin de la section critique

return 0;

}
```

### La fonction client\_quit : fait la gestion de fin de connexion d'un client

```
/* gestion de fin de connection d'un client */

void client_quit(s_client *me, char *msg)
{
    /*dans cette Fonction on a gerer les 2 cas du sortie du client,,
    *s'il quitte le programme son Laisser de message,
    *(si le client ne veut pas Envoyer un Message un Message avant le <<Leave>>
    ou s'il est interrompu)
    *ou bien s'il Vent Envoyer un Dernier Message en Quittant l'application
    *
    */

    int i,j;

    char buf[8192+1];

    //On Concatene le Pseudo et le Message du Client qui quitte l'application dans
    la variable Buf

    if(msg)snprintf(buf,8192,"%s nous quitte...(%)s\r\n",me->pseudo,msg);

    else    snprintf(buf,8192,"%s nous quitte...\r\n",me->pseudo);

    buf[8192] = '\0';

    send_all(buf,me->sock);//l'Envoie du Message à tous les Client de notre Liste

    pthread_mutex_lock(&mutex); // debut de la section critique
```



// recherche de l'index de notre Client dans le tableau pour reorganiser le Tableau  
Après la suppression

```
for(i=0;(clients[i]->sock != me->sock);i++);
```

```
close(me->sock); //la Fermeture du Socket
```

//Liberer la Memoire des 2 pointeur

```
free(me->pseudo);
```

```
free(me);
```

for(j=i+1;j<first\_free;j++) // on reorganise le tableau en decalant les clients  
situes apres celui qui a Quitter

```
{
```

```
    clients[j-1] = clients[j];
```

```
}
```

```
nb_clients--;
```

```
first_free--;
```

```
pthread_mutex_unlock(&mutex); // fin de la section critique
```

```
printf("Un client a Quitter ... %d clients\n",nb_clients);
```

```
}
```

//l'Envoie du Message a tous les client sauf "Me",celui qui l'a Envoyer !

```
}
```

### **La fonction interact : interaction avec le client (thread)**

/\* interaction avec le client (thread) \*/

```
void *interaction(void *param)
```

```
{
```

```
    int sck = *((int *) param);
```

```
char msg[4096+1];

char msg_to_send[8192+1];

s_client *me = NULL;

char *buf = NULL;

int len;

int i;

//Allocation du Memoire pour 1 nouveau Client

me = (s_client *) malloc(sizeof(s_client));

//Le Cas D'erreur on Affiche le Msg d'erreur et en Ferme la Socket et le Thread
et le Nbr du client diminue de 1

if(!me)

{

    printf("\nErreur d'allocation memoire!\n");

    close(sck);

    nb_clients--;

    pthread_exit(NULL);

}

//Si nn on Initialise Les Champs de la Cstructure Client !!

bzero(me,sizeof(s_client));

//REception du 1er Message du Client,,le Message contient le Login du Client

len = recv(sck,msg,4096,0);
```

//le Cas d'erreur ..

```
if(len <= 0)
```

```
{
```

```
    printf("\nErreur\n");
```

```
    close(sck);
```

```
    free(me);
```

```
    me = NULL;
```

```
    nb_clients--;
```

```
    pthread_exit(NULL);
```

```
}
```

```
msg[255] = '\0';    // on limite le pseudo a 255 caracteres
```

```
for(i=0;(msg[i]!='\0') && (msg[i]!='r') && (msg[i]!='n') && (msg[i]!='t');i++);
```

```
msg[i] = '\0';    // on isole le pseudo
```

// debut de la section critique pour l'ajout d'un Seul client a la fois

```
pthread_mutex_lock(&mutex);
```

```
for(i=0;i<first_free;i++)
```

//il faut comparer si le Pseudo Existe déjà Utiliser par un autre Client

```
    if(!strcmp(msg,clients[i]->pseudo))//si le Cas
```

```
{
```

//On Envoie un Message qui "Pseudo deja utilise! !..." + fermeture de la connection

```
    send_msg(sck, "\r\nPseudo deja utilise! Deconnection...");

    close(sck);

    free(me);

    nb_clients--;

    pthread_mutex_unlock(&mutex);    // fin de la section critique

    pthread_exit(NULL);

}

}

pthread_mutex_unlock(&mutex);    // fin de la section critique

//S'il n y a pas d'erreur on Affecte les Valeur du Client en Cours au Client "Me"

me->id = pthread_self();

me->sock = sck;

me->pseudo = strdup(msg);

pthread_mutex_lock(&mutex);    // debut de la section critique

//on ajoute "Me" a la 1ere case Vide de Notre Tableau de Clients

clients[first_free] = me;
first_free++;

pthread_mutex_unlock(&mutex);    // fin de la section critique
```

```
send_all(msg_to_send,INVALID_SOCKET);
```

//Debut de la Conversation du Client Avec Notre Serveur en Lui Envoyant le Message Help qui Contient les commandes Possible

```
send_msg(me->sock,HELP_MSG);
```

//Debut de la Communication l'Envoi et la Reception des Messages et des Commandes

```
while(1)
```

```
{    bzero(msg,sizeof(msg));
```

```
len = recv(sck,msg,4096,0);//Reception d'un Message
```

```
if(len <= 0)//Verifier s'il y a des Erreur
```

```
{
```

```
    client_quit(me,"Erreur reseau");
```

```
    pthread_exit(NULL);
```

```
}
```

```
msg[len] = '\0';
```

// si le message Commance par "/" ca ve dire qu'il s'agit d'une commande

```
if(msg[0] == '/')
```

```
{
```

```
    int valid_command = 0; //Variable pour contoler si une  
commande est Valide ou pas
```

```
if(!strcmp(msg, "/quit", 5))
// sortie "propre" du serveur (avec ou sans message)

{

    int i;

    //le Cas du sortie Avec Message

    if(msg[5]==' ')
    {

        for(i=6;(msg[i]!='\0') && (msg[i]!='r') &&
(msg[i]!='n') && (msg[i]!='t');i++);

        msg[i]='\0';

        //la Fonction Client_quit pour Quitter la Conversation et l'Envoie du Message

        client_quit(me,&msg[6]);
    }

    else client_quit(me,NULL); //si non ,on Quitte sans
Message

    valid_command = 1;

    pthread_exit(NULL);

}

// obtenir la liste des Clients Connecté sur le serveur

if(!strcmp(msg, "/list", 5))
{
```

```

pthread_mutex_lock(&mutex); // debut de la section critique

/*on doit parcourir le Tableau des Clients,
*

*et on Va Envoyer la Liste au Seulement au client "Me"
(Celui qui a Demander la Liste) *

*la Liste Sera sous la Forme " Client <indice du Client> :
<Pseudo> " *

*/

for(i=0;i<first_free;i++)

{   char listesClient[100];

    snprintf(listesClient,100,"Client %d : %s ",
(i+1),clients[i]->pseudo);

    send_msg(me->sock,listesClient);

    send_msg(me->sock,"\r\n");

}

pthread_mutex_unlock(&mutex); // fin de la section critique

valid_command = 1;

}

//pour Obtenir la liste des Commandes Possible

if(!strcmp(msg,"?",2))
{

    send_msg(me->sock,HELP_MSG);

    valid_command = 1;

```

```
    }

    if(!valid_command) // commande invalide

        send_msg(sck,"Commande non valide!\r\n");

    }

    else
        // message teste normal ,envoyer a tout les Client connecté

    {

        //le Message a Envoyer aura la forme "From <Pseudo> :
        Mssage" , la Forme est Citer dans le Cahier des Charges

        snprintf(msg_to_send,sizeof(me->pseudo)+sizeof(msg),"From <
        %s > : %s\n",me->pseudo,msg);

        msg_to_send[sizeof(me->pseudo)+sizeof(msg)] = '\0';

        //l'Envoie du Message a tous les client sauf "Me",celui qui l'a Envoyer !

        send_all(msg_to_send,me->sock);

    }

}

return NULL;

}
```

### La fonction principale main

```
int main(int argc, char **argv)
```



```
{
    int server,sck;
    pthread_t th_id;
    printf(BANNER);
    server = create_server(PORT);
    while(1)
    {
        sck = server_accept(server,0);
        if(sck == INVALID_SOCKET)
        {
            printf("\nErreur de accept()!\n");
            exit(-1);
        }
        if(nb_clients < MAX_CLIENTS)
        {
            pthread_create(&th_id,NULL,interact,(void *)&sck);
            nb_clients++;
            printf("Nouveau client! %d clients\n",nb_clients);
        }
        else close(sck);
    }
    return 0;
}
```

# Client en JAVA

## A. Classe :

### **Connexion :**

Contient la declaration et l'initialisation de la Communication avec le serveur.

### **Reception:**

Contient un Thread qui rest en Ecoute,lors du reception d'un Message du Serveur il l'affiche dans le Texte Area reservé pour le chat.

### **FenChat :**

Contient les Buttons et les champs textes reservé pour la saisit d'un Message et l'Affichage des Messages Reçus des autres clients.

### **FenInfo:**

La 1er Fentre de Chat,pour la saisie du Pseudo du Client et l'adresse du serveur auquel ce dernier veut se connecter.

### **Main:**

la class Principale de notre projet.

### **Timer :**

Clontient un thread qui Calcule le temps d'attente apres l'envoi ou la reception d'un message. le client ne reste pas bloquer indéfiniment au cas d'erreur sur le Serveur, parce que le Timer force la sortie du programme après 5 secondes.

## B. Description détaillé :

La classe Connexion

```
import java.net.* ;
import java.io.* ;
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;
public class connexion {
    private Socket socket ;
    private DataInputStream entree;
    public PrintStream sortie;
    private reception rc;
    //private TextArea chat;
    private String nomClient;

    public connexion(TextArea chat,String serveur,String nom) {
        try {
            socket = new Socket(serveur,3557) ;
            entree=new DataInputStream(socket.getInputStream());
            sortie=new PrintStream(socket.getOutputStream());
            sortie.println(nom);sortie.flush();
            nomClient=entree.readLine();
            chat.setText("Vous êtes connecté avec "+nomClient+"\nDébut de la
discussion...\n");
            rc=new reception(entree,chat,nomClient);
        }
        catch ( IOException e) {System.out.println(e); }
    }
    public PrintStream recupererSortie()  {

        return sortie;
    }
}
```

La classe Timer

```
import java.net.* ;

import java.io.* ;

import javax.swing.*;

public class Timer extends Thread

{
```

```
private int i;

private JFrame F;

public Timer(JFrame E)
{
    F=E;
    i=0;
    start();
}

public void Initialise()
{
    i=0;
}

public void run()
{
    while(true)
    {
        try
        {
            sleep(1000);
        }
        catch(Exception e)
        {
            System.out.println("Erreur Sleep... \n"+e);
        }
    }
}
```

```

    }

    if(i>60)

    {

        F.dispose();

        JOptionPane.showMessageDialog(null,"il Faut se Reconnecter
au serveur","Erreur pseudo",JOptionPane.WARNING_MESSAGE);

        new fenInfo();

        stop();

    }

}

}

}

```

### La classe Reception

```

import java.net.* ;
import java.io.* ;
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;

public class reception extends Thread
{
    private DataInputStream entree;
    private TextArea chat;
    private String recu;
    private String nomClient;

    public reception(DataInputStream DIS,TextArea texte,String nom)
    {
        entree=DIS;
        chat=texte;
        nomClient=nom;
        start();}
}

```

```

public void run(){
    String texte;

    try{
        while (true)
        {
            recu=entree.readLine();
            chat.setText(chat.getText()+nomClient+"
:\n"+recu+"\n");
        }
        catch(IOException e) { System.out.println(e);System.exit(0);} }

```

La classe FenInfo

```

import javax.swing.*.*;
import java.awt.event.*;
import java.awt.*.*;

public class fenInfo extends JFrame implements ActionListener
{
    private JButton Lancer=new JButton("Lancer le Tchat");
    private JLabel labelNom=new JLabel("Identifiant de connexion");
    private JLabel labelServeur=new JLabel("Adresse du serveur");
    private JTextField textNom=new JTextField(15);
    private JTextField textServeur=new JTextField(15);

    public fenInfo()
    {
        super("Informations de connexion");
        setBounds(350,200,250,250);
        setVisible(true);
        setAlwaysOnTop(true);
        setLayout(new FlowLayout ());
        labelNom.setForeground(Color.blue);
        labelServeur.setForeground(Color.blue);
        add(labelNom);add(textNom);
        add(labelServeur);add(textServeur);
        add(Lancer);
        Lancer.addActionListener(this);}
    public void actionPerformed(ActionEvent e)    {
        String label = e.getActionCommand();
        if (label.equals("Lancer le Tchat")){
            dispose();
            new fenChat(textNom.getText(),400,100,350,600,textServeur.getText());} }
}

```

# Conclusion

Après une semestre de formation, la semestre la plus riche pour nous tant sur le plan académique que sur le plan professionnel en terme de connaissances techniques et découverte de nouvelles solutions informatiques, Ce projet représente un espace d'expression des compétences acquises et de concrétisation du premier objectif de la formation qui est le savoir faire.

Certes un premier travail moyennant des outils nouveau dans un temps relativement très limité ne peut être un oeuvre finale, mais il représente une source de motivation pour amplifier au fur et à mesure nos compétence à la lumière des principes de base maîtrisés et méthodes de travail assimilées.

Espérant donc que ce modeste travail soit à la hauteur, et satisfaisant pour toutes les personnes qui ont aidé à son élaboration.