

Rapport IPC

Etudiants :

- JACOUD Bastien
- REMOND Victor
- TAGUEJOU Christian
- TARDY Martial

Projet GIT

- Lien [dépôt GitHub](#)
- Téléchargement [version stable POP3](#)

I - Introduction

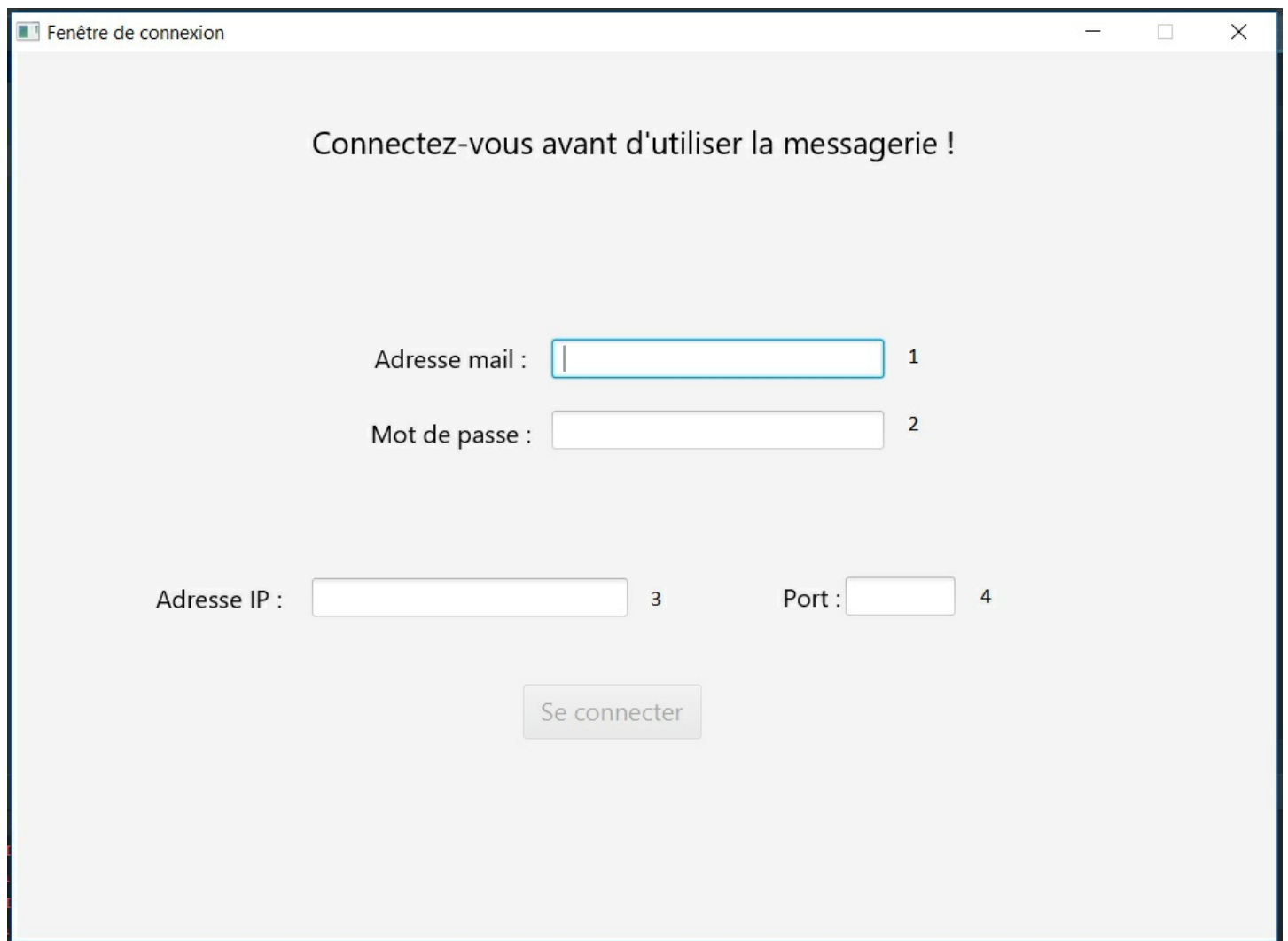
L'objectif de ce TP était dans un premier temps de réaliser un client et un serveur POP3 respectant la norme RFC 1939 trouvable à l'adresse <https://www.ietf.org/rfc/rfc1939.txt>.

Pour mener à bien ce projet, nous avons commencé par réfléchir aux schémas des automates découlant de cette norme, avant de nous lancer dans le développement des deux parties du projet en java.

Une fois notre couple Client - Serveur fonctionnel avec le protocole POP3, nous l'avons modifié afin qu'il utilise le protocole POP3S, en remplaçant les commandes *USER* et *PASS* par une unique commande *APOP* et l'utilisation d'un timbre à date.

II - Notice d'utilisation

Afin de pouvoir utiliser la messagerie correctement, l'utilisateur doit tout d'abord passer par une phase d'authentification. Lors de cette authentification, il doit notamment spécifier l'adresse IP et le port du serveur, mais aussi son adresse mail et son mot de passe.



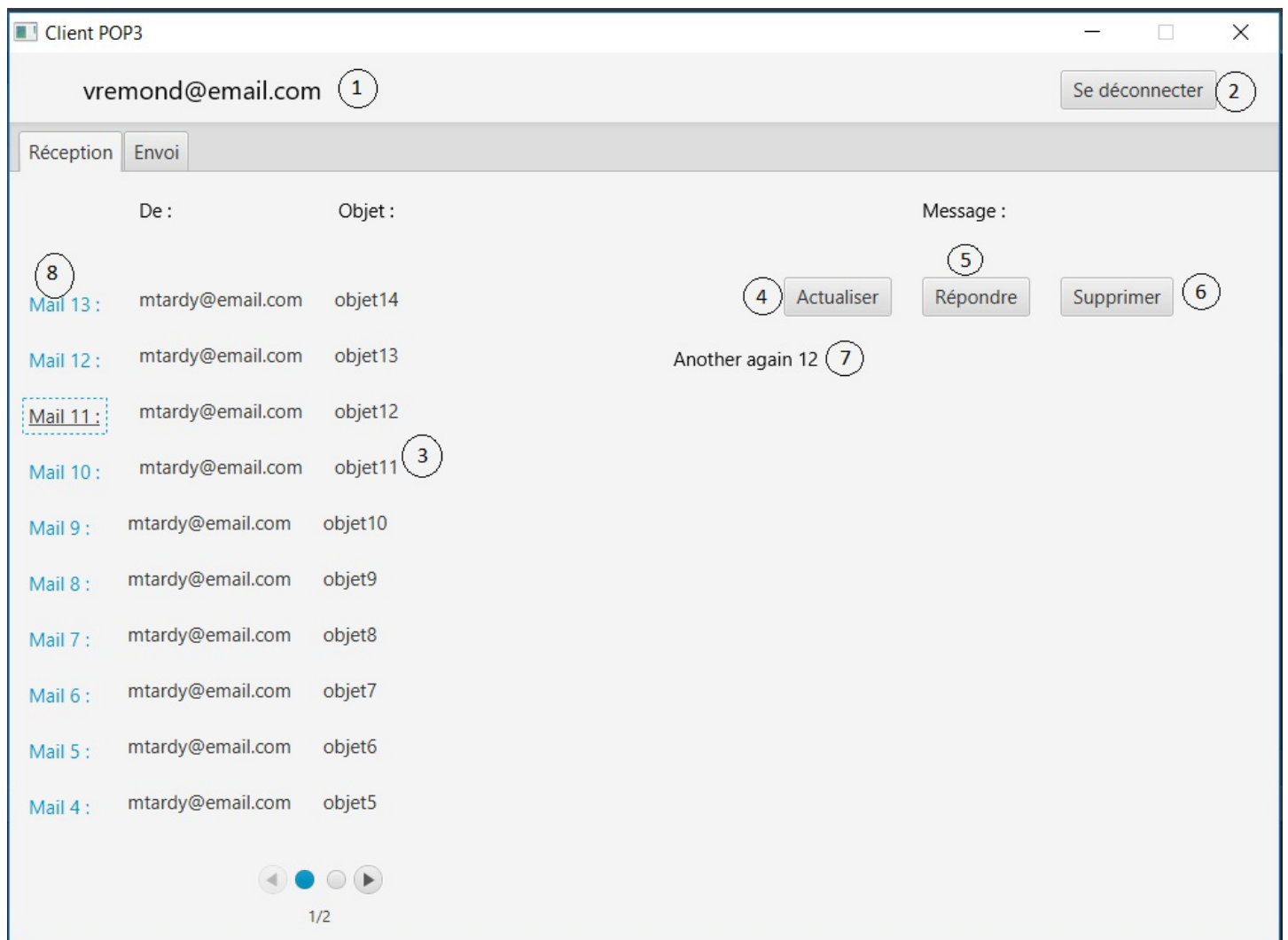
The screenshot shows a window titled "Fenêtre de connexion" with a light gray background. At the top, it says "Connectez-vous avant d'utiliser la messagerie !". Below this, there are four input fields with labels and numbers: "Adresse mail :" followed by a text box labeled "1", "Mot de passe :" followed by a text box labeled "2", "Adresse IP :" followed by a text box labeled "3", and "Port :" followed by a text box labeled "4". At the bottom center, there is a button labeled "Se connecter".

Sur la capture d'écran ci-dessus, nous observons bien que l'utilisateur doit mentionner l'adresse IP de la machine serveur(3) et le port sur lequel le programme est exécuté(4).

Il doit également inscrire son adresse mail(1) ainsi que son mot de passe(2). Lors du renseignement du mot de passe, ce dernier n'apparaît

pas en clair sur la fenêtre d’affichage, visible par l’utilisateur.

Une fois la phase d’identification et d’authentification achevée, une nouvelle fenêtre s’affiche alors à l’écran. Il s’agit de la messagerie du client, dont voici une capture d’écran :



L’adresse mail de l’utilisateur actuellement connecté(1), ainsi qu’un bouton lui permettant de se déconnecter(2) apparaissent directement sur la fenêtre du client. Il est important de noter qu’il existe deux moyens pour l’utilisateur de se déconnecter : il peut cliquer sur le bouton déconnexion ou alors directement fermer la fenêtre via un clic sur la croix rouge. Dans chacun de ces cas, le client envoie la commande “QUIT” au serveur, qui va

se charger de la suppression des messages marqués.

Une liste, avec pagination automatique, de tous les mails réceptionnés(3) est disponible sur la partie gauche de la fenêtre. Chaque numéro de mail(8) correspond à un lien et permet d'afficher directement via un clic sur ce dernier son contenu dans la partie droite de la fenetre(7).

Le bouton Actualiser(4) permet de réafficher tous les messages réceptionnés, y compris les messages venant tout juste d'être reçus.

Le bouton Répondre(5) nous rediriger automatiquement vers l'onglet Envoi de mail et pré-rempli tous les champs nécessaires à la réponse.

Le bouton Supprimer(6) nous permet de supprimer un mail. Ce dernier apparaît alors en rouge tant que l'utilisateur reste connecté et sera supprimé à la déconnexion du client.

Il est important de noter que la partie envoi de message n'est pas présentée dans ce compte rendu car son implémentation fait l'objet du prochain tp.

II - La partie Client

Le client POP3 avait pour but de permettre à un utilisateur de se connecter au serveur POP3 (en renseignant l'adresse de ce dernier et le port sur lequel il voulait se connecter), de relever ses mails et de pouvoir les afficher.

Il n'était pas nécessaire de permettre à celui-ci d'envoyer des nouveaux messages ou de lui laisser la possibilité de supprimer ses messages.

1 - Algorithmme

Ci-dessous le code Java basique d'un main de client POP3

```
public static void main(String[] args) {  
    //Création d'un objet client POP3  
    POP3 client = new POP3();  
    boolean connected = false;  
    while(connected == false) {  
        //Connexion du client au serveur, via son adresse IP  
        et le numéro de port souhaité  
        client.joinServer("192.168.43.18", 1210);  
        //Authentification de l'utilisateur  
        connected = client.authenticate("vremond@email.com",  
"Else");  
    }  
    //Passage à la boucle principale du client POP3  
    client.transactions();  
    //Fermeture du client  
    client.close();  
    //Fin de l'execution  
}
```

Ainsi que le code Java basique de la gestion des transactions par le client

```
class POP3 {  
    public POP3() {
```

```

        /* Initialisation du client
        */
    }

    protected String getFromUser() {
        /* Récupère la commande à executer via l'interface u
utilisateur
        * Varie selon l'interface homme-machine utilisée.
        */
        return command;
    }

    /* Autres fonctions */

    public transactions() {
        String command;
        boolean quit = false;
        while(quit == false) {
            command = this.getFromUser();
            //Récupération de chaque mot de la commande indiv
iduellement
            String[] splitCommand = command.split(" ");
            //Analyse du premier mot, forcé en minuscule pour
ignorer la casse
            switch(splitCommand[0]) {
                case "quit":
                    this.quit();
                    quit = true;

```

```
        break;

    case "stat":
        this.stat();
        break;

    case "list":
        this.list();
        break;

    case "dele":
        // On n'exécute la commande que si un arg
        ument a été passé en paramètre en entrée.

        // La validité de cet argument sera quant
        à elle vérifiée dans la fonction.

        if(splitCommand.length > 1) {
            this.dele(splitCommand[1]);
        }

        break;

    case "retr":
        // On n'exécute la commande que si un arg
        ument a été passé en paramètre en entrée.

        // La validité de cet argument sera quant
        à elle vérifiée dans la fonction.

        if(splitCommand.length > 1) {
            this.retr(splitCommand[1]);
        }

        break;

    case "noop":
        this.noop();

        break;
```

```
        case "rset" :  
            this.rset();  
            break;  
        default:  
            break;  
    }  
}  
}
```

On notera que dans la fonction ci-dessus, les codes d'actualisation des informations visibles par l'utilisateur ne sont pas détaillés étant données qu'ils sont totalement différents selon le style d'affichage utilisé (invité de commandes ou interface graphique). De plus ils n'apporteraient pas forcément d'informations utiles pour comprendre le fonctionnement du protocole.

2 - Développement

Dans la phase de développement, nous avons fait le choix de créer une classe gérant le protocole TCP afin d'effectuer la connexion avec le serveur. Cette classe **TCP** est elle même utilisée par notre classe **POP3**, ainsi lorsque la fonction de connexion de POP3 est appelée, la classe fait elle même appel à TCP.

De même, pour l'envoi de commande, le client POP3 utilise la fonction *send* définie dans la classe **TCP** et la fonction *receive* pour récupérer les

réponses envoyées par le serveur.

D'autres fonctions définies dans la classe `POP3` permettent d'envoyer une commande au serveur et d'attendre la réponse, voire même de tester si celle-ci est positive avant de la retourner.

Pour effectuer divers vérifications, nous avons créé un package `utilities` contenant une classe définissant diverses méthodes statiques permettant de faire des tests via des expressions régulières, telles que vérifier qu'une réponse du serveur commence bien par *+OK* ou encore découper la chaîne de caractères qui représente le mail afin de ne récupérer que les informations qui nous intéressent (*expéditeur, objet, message, etc...*).

En ce qui concerne la transition du client de POP3 vers POP3S, elle a été extrêmement facile à opérer. En effet nous avons juste eu à créer une classe `POP3S` héritant de la classe `POP3` et redéfinissant la fonction d'authentification. Ainsi nous avons eu un client POP3S fonctionnel en récupérant la quasi totalité du code déjà mis en place précédemment, les seules modifications à apporter étant la récupération d'un timbre à date (extrait de la réponse du serveur grâce à une des fonctions du package `utilities` évoqué ci-dessus) et l'envoi d'une commande *APOP*, encryptée en MD5 grâce au timbre à date récupéré précédemment, à la place des commandes *USER* et *PASS* utilisées par le protocole POP3 classique.

De plus les classes TCP, POP3 et POP3S ont été codées de sorte à émettre des exceptions avec un message d'explication en cas d'erreur et à propager ces dernières jusqu'à ce qu'elles puissent être affichées. A chaque propagation, un message supplémentaire est ajouté à l'exception si

cela est jugé nécessaire.

3 - Partie Graphique

L'interface graphique du client à été réalisé grâce à la bibliothèque graphique JavaFX. Les différents composants de chacune des fenêtres ont été créés grâce à SceneBuilder et sont stockées dans les fichiers FXML correspondants.

Les vérifications sont effectuées à la fois côté client et côté serveur, dans la mesure du possible. Par exemple, l'utilisateur ne peut accéder au bouton de connexion que si les différents champs ont été correctement remplis. Pour cela, nous utilisons différentes Regex pour vérifier si une adresse mail ou une adresse IP est correcte. Il s'agit des vérifications effectuées côté client. De l'autre côté nous traitons les réponses renvoyés par le serveur pour faire remonter les erreurs sous forme d'exceptions. Lorsqu'une exception est levée, une alertbox est affichée à l'écran de l'utilisateur pour lui indiquer l'erreur qui a eu lieu.

D'autres alertbox peuvent également suivenir pour demander confirmation à l'utilisateur, par exemple lors de la suppression d'un message.

Tout ceci va donc rendre plus agréable l'utilisation du logiciel par l'utilisateur.

III - La partie Serveur

La partie Serveur a été faite en deux étapes. Dans un premier temps, nous

avons géré la connexion de l'utilisateur par le protocole TCP, puis la communication entre le client et le serveur grâce aux commandes POP3.

1 - Connexion TCP

Avant qu'un utilisateur n'essaye de se connecter, on lance le serveur sur un port choisi. Pour cela, on instancie la classe `SocketServer`. Cet objet prendra en charge la transmission des données.

Nous avons défini une boucle sans fin pour que le serveur puisse accepter toutes les connexions tant que celles-ci se font sur le bon port.

```
try{
    ServerSocket serverSocket = new ServerSocket(port
);
    System.out.println("Lancement du serveur sur le p
ort " + port);
    while (true){
        Socket socketClient = serverSocket.accept();
        Tcp t = new Tcp(socketClient);
        t.start();
    }
}catch(Exception e){
    e.printStackTrace();
}
```

Lorsqu'un client se connecte sur le bon port, la méthode `accept()` de la classe `SocketServer` va retourner un objet `Socket` représentant la

connexion du client. Pour pouvoir traiter plusieurs connexion à la fois, un thread va être créé à partir de cet objet. Lorsque le thread est lancé, la méthode `run()` de la classe `Tcp` s'exécute. Dans cette méthode on instancie la classe `ObjetConnecte` qui va par la suite communiquer avec l'utilisateur en utilisant les méthodes `receive()` et `send()`.

2 - POP3

1 - Développement

Pour la création du serveur POP3 nous avons suivi l'automate créé lors de la première séance, c'est à dire la gestion des états et des commandes en fonction de ces états.

Pour chaque état certaines commandes sont utilisables. On retrouve dans l'état autorisation la commande `USER` et `QUIT`, dans l'état authentification la commande `PASS` et `QUIT`, pour finir dans l'état transaction les commandes `QUIT`, `RSET`, `DELE`, `STAT`, `LIST`, `NOOP`, `UIDL`. Chaque commande correspond à une action sur le serveur.

Comme expliqué précédemment la classe `ObjetConnecte` contient une instance de la classe `Tcp` qui permet d'échanger les messages avec le client.

Lorsque la connexion `tcp` est effectuée la fonction `Launch` de `ObjetConnecte` est appelée. Elle prévient le client que la connexion est bien effective puis attendra les commandes demandées par le client. Lors de la réception d'une commande le Serveur rentre dans la fonction correspondante à l'état actuel du serveur stocké dans la variable `m_etat`.

Dans chaque fonction d'état nous appelons la fonction correspondante à la commande si l'état le permet sinon nous répondons un message indiquant que la commande n'est pas valide.

2 - Algorithmme

On retrouve tout d'abord la fonction principale de `ObjetConnecte` ainsi que son constructeur:

```
public ObjetConnecte(Tcp tcp) {  
    //Initialisation du tcp et des autres variables  
}  
  
public void Launch() {  
    //Initialisation du serveur à l'état AUTHORISATION  
    m_etat = etatAutorisation;  
  
    //reponse apres connexion tcp  
    m_tcp.Send("+OK" + " POP3 server ready");  
  
    //boucle principale  
    while (m_continuer) {  
  
        //Attente de la reception d'un message venant d'un client  
  
        String input = m_tcp.Receive();
```

```

        //Récupération de la commande, vérifications et traitement
        // séparation de la commande et des parametres dans deux tableaux

        //En fonction de l'etat actuel du serveur
        switch (m_etat) {
            case etatAutorisation:
                response = this.AuthorisationState(command, parameters);
                break;
            case etatAuthentification:
                response = this.AuthenticationState(command, parameters);
                break;
            case etatTransaction:
                response = this.TransactionState(command, parameters);
                break;
            default: //etat non reconnu
                response = "-ERR";
                break;
        }

        //reponse au Client
        m_tcp.Send(response);
    }

    // on est sorti de la boucle m_continuer on ferme le serveur

```

```
    affiche("Fin de POP3");  
}
```

Puis les fonctions de chaque état:

```
//Etat autorisation  
protected String AuthorisationState(String command, String[]  
parameters) {  
    //reception de la commande USER  
    if (command.equals("USER")) {  
  
        if(parameters.length < 1) {  
            //si il manque des parametres on retourne un mess  
age d'erreur  
            return "-ERR" + " manque parametre.";  
        }  
  
        //Verification que le nom d'utilisateur passé en para  
mètre est bien dans nos données  
        if(this.checkUser(username)) {  
            //on passe dans l'état authentication  
            m_etat = etatAuthentification;  
            return "+OK";  
        } else {  
            //si l'utilisateur n'est pas correct on retourne  
un message d'erreur  
            return "-ERR" + " username is not valid";  
        }  
    }  
}
```

```
}  
  
//Sinon retourne erreur de commande  
return "-ERR commande non valide";  
}  
  
//Etat authentication  
protected String AuthenticationState(String command, String[]  
parameters) {  
    //reception de la commande PASS  
    if(command.equals("PASS")) {  
  
        if(parameters.length < 1) {  
            // si il manque des paramètres on retourne un mes  
sage d'erreur  
            return "-ERR manque parametre.";  
        }  
  
        //verification que le mot de passe correspond au mot  
de passe de l'utilisateur qui vient d'etre rentré  
        if(this.checkPass(password)) {  
            //on recupere les emails de l'utilisateur on  
passe dans l'etat transaction  
            m_etat = etatTransaction;  
            //retour message positif  
            return "+OK";  
        }  
    } else if(command.equals("QUIT")) {  
        return this.quit();  
    }  
}
```



```
//sinon commande non valide
return "-ERR commande non valide";
}

//etat transaction
protected String TransactionState(String command, String[] parameters) {
    //reception des differentes commandes POP3 et appel des fonctions correspondantes
    if(command.equals("QUIT")) {
        return this.quit();
    } else if(command.equals("RETR")) {
        if(parameters.length < 1) {
            return "-ERR manque parametre.";
        }
        return retr(parameters[0]);
    } else if(command.equals("NOOP")) {
        return noop();
    } else if(command.equals("RSET")) {
        return rset();
    } else if(command.equals("DELE")) {
        if(parameters.length < 1) {
            return "-ERR manque parametre.";
        }
        int indice = parameters[0];
        return dele(indice);
    } else if(command.equals("LIST")) {
        return list();
    }
}
```

```
    } else if(command.equals("UIDL")) {  
        return uidl();  
    } else if(command.equals("STAT")) {  
        return stat();  
    } else {  
        //sinon retourne commande non valide  
    }  
}
```

3 - Développement et algorithme POP3S

POP3S est défini par la classe `ObjetConnecteSecurise` et est une classe qui va hériter de `ObjetConnecte`.

On retrouve donc les mêmes attributs et quelques changements de fonctions. En effet nous avons enlevé l'état authentication et remplacé les commandes PASS et USER par APOP. Pour protéger l'authentification de l'utilisateur, le serveur va implémenter la commande APOP qui mettra un timbre-à-date dans son message de bienvenue. La syntaxe de ce timbre-à-date est la suivante:

<identificateur-de-processus.heure@nom-de-l'hôte>.

La méthode `generateTimbre()` se charge de générer ce timbre-à-date et l'envoie à l'utilisateur. L'utilisateur va encrypter cette chaîne de caractère en appliquant l'algorithme MD5 sur le timbre-à-date, suivi du "secret partagé". Le secret partagé est une chaîne de caractère connu uniquement par le serveur et le client; dans notre cas, le secret partagé sera le mot de passe du client.

APOP prend en paramètre le nom d'utilisateur ainsi que le mot de passe encrypté. Cette somme de contrôle sera renvoyé au serveur.

Le serveur va de son coté de générer une somme de controle à partir du timbre-à-date précemment envoyé à l'utilisateur et du mot de passe de l'utilisateur stocké dans la base de donnée. Il va ensuite comparer sa somme de contrôle à celle renvoyé par l'utilisateur. Si les deux chaines de caractères sont égales, alors l'utilisateur pourra passer à l'état suivant.

Par exemple on envoie le timbre date lorsque la connexion est effectuée avec le client.

```
try {  
    this.m_tcp.Send("+OK" + " POP3 server ready " + generate  
    Timbre());  
} catch (Exception e) {  
    e.printStackTrace();  
}
```

Pour cela nous avons créé des fonctions pour former ce timbre date:

```
//Etat autorisation  
  
protected String AuthorisationState(String command, String[]  
parameters) {  
    //on recupere la commande APOP  
    if (command.equals("APOP")) {  
        // Si il n'y a pas deux parametres ( nom d'utilisateur
```

r et mot de passe encrypté) on retourne une erreur.

```
    if(parameters.length <= 1) {
        return "-ERR" + " manque parametre.";
    }
    String username = parameters[0];
    String password = parameters[1];
    out.println("username : " + username);

    //on verifie si l'utilisateur existe
    if(this.checkUser(username)) {
        try {
            //on verifie si le mot de passe existe
            if(this.decrypteTimbre(password)) {
                //on recupere les emails de l'utilisateur
                //on passe à l'etat transaction
                m_etat = etatTransaction;
                return "+OK";
            } else {
                return "-ERR" + " password is not valid";
            }
        } catch (NoSuchAlgorithmException e) {
            e.printStackTrace();
        }
    } else {
        return "-ERR" + " username is not valid";
    }
}

//Sinon on retourne que la commande n'est pas valide
```

```
        return "-ERR" + " command \"" + command + "\" doesn't see  
m valid";  
    }
```