Rapport IPC - Partie SMTP

Etudiants :

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

Code du Projet : lien GitHub

Table des matières

I - Introduction

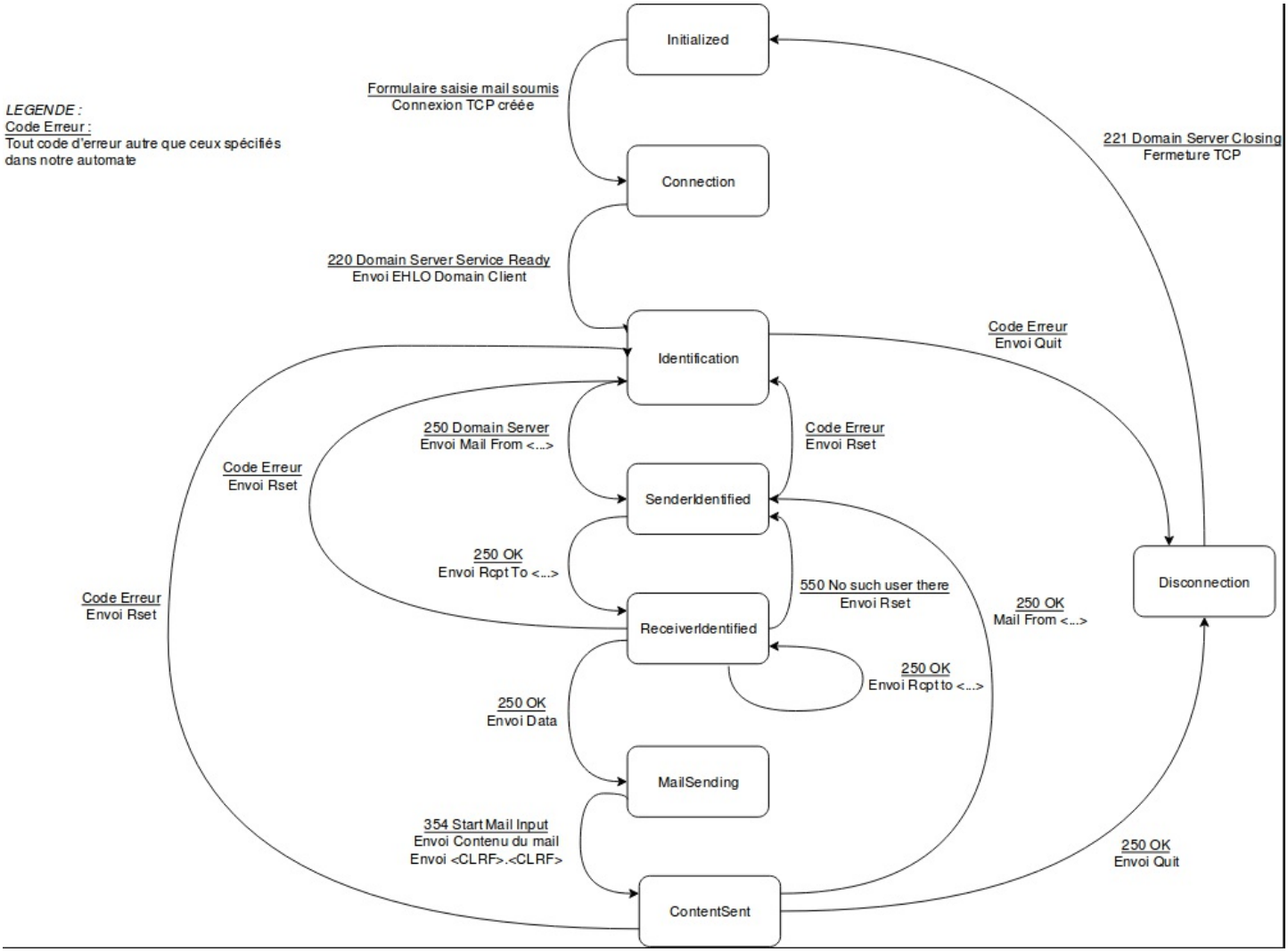
Suite du TP de développement d’un couple client/serveur mail. Cette étape fut la troisième de cette série de TP. Durant la première étape, nous avons dû mettre en place le protocole POP3 pour que le client relève ses mails sur le serveur. La seconde consistait à adapter le code pour utiliser le protocole POP3S afin de sécuriser la connexion du client. Le but de cette dernière étape était de permettre au client d’envoyer des messages à d’autres utilisateurs en permettant au client comme au serveur d’utiliser le protocole SMTP.

Lien vers la norme RFC utilisée.

II - Client

1 - Automate & Table de transitions

Nous n’avons pas commencé le développement du client immédiatement, nous avons dû commencer par faire un automate représentant le fonctionnement attendu pour ce protocole.



Grâce à cet automate, nous avons réalisé la table de transition du fonctionnement de l'application cliente :

	formulaire mail soumis	220 domain Serveur Service Ready	250 domain Serveur	250 OK	550 No such user here	354 start mail input	221 Service Closing	Code erreur
Initialized	création connexion TCP	Ignore	Ignore	Ignore	Ignore	Ignore	Ignore	Ignore
Connection	Ignore	Envoi EHLO Domain Client	Ignore	Ignore	Ignore	Ignore	Ignore	Ignore
Identification	Ignore	Ignore	Envoi MAIL FROM <>	Ignore	Ignore	Ignore	Ignore	Envoi QUIT
SenderIdentified	Ignore	Ignore	Ignore	Envoi RCPT TO <>	Ignore	Ignore	Ignore	Envoi RSET
ReceiverIdentified	Ignore	Ignore	Ignore	Envoi DATA ou Envoi RCPT TO <>	Envoi RSET	Ignore	Ignore	Envoi RSET
MailSending	Ignore	Ignore	Ignore	Ignore	Ignore	Envoi contenu du mail + "<CRLF> . <CRLF>"	Ignore	Ignore
ContentSent	Ignore	Ignore	Ignore	Envoi QUIT	Ignore	Ignore	Ignore	Envoi RSET
Disconnection	Ignore	Ignore	Ignore	Ignore	Ignore	Ignore	Fermeture TCP	Ignore

2 - Backend

A - SMTP Basique

Dans un premier temps, nous avions un unique domaine à gérer

`email.com`. Ainsi l'implémentation du protocole était relativement simple, il suffisait nous de connaitre l'adresse du serveur, or celle-ci était déjà renseignée pour le fonctionnement des protocoles POP3 et POP3S. Nous n'avions donc pas besoin d'autre nouvelle information que le port de connexion de SMTP sur le serveur.

Afin de permettre à l'utilisateur d'envoyer son message à plusieurs utilisateurs en même temps, nous lui donnons la possibilité de mettre plusieurs destainataires, séparés par des points-virgules. Deux fonctions de la classe `String` de java ont rendu cette fonctionnalité facile à implémenter :

- `String::split(";")` nous permet de découper une chaine de caractères à chaque occurence du caractère `";"`.
- `String::trim()` permet quant à elle de supprimer les espaces en début et fin de chaine de caractères, utile pour avoir une adresse correcte pour le destinnataire, peu importe que l'utilisateur ait décidé de séparer les différentes adresses avec `";"`, `"; "` ou `" ; "`.

B - Fonctionnement avec plusieurs noms de domaines

Nous avons ensuite dû faire fonctionner le client pour qu'il puisse gérer plusieurs noms de domaine (`email.com` et `email.fr`), correspondant à deux serveurs différents. Afin de savoir sur quelle adresse IP et sur quel port envoyer le message selon l'adresse du destinataire, nous avons créé une classe `DNS` permettant de simuler le fonctionnement d'un serveur DNS classique : récupérer l'adresse IP d'un serveur en fonction de son

nom de domaine.

Notre classe `DNS` se compose d'une liste de `ServerIntels`, une classe contenant toutes les informations utiles à propos d'un serveur. Cette liste est lue dans le fichier de configuration `config/DNS.csv`, elle doit être mise à jour après le lancement des serveurs. Les différentes fonctions *public* de cette classe permettent de récupérer les informations qui nous intéressent sur le serveur voulu grâce à son nom de domaine.

```
public class DNS {
    private static List<ServerIntels> servers = getConfig();

    private static List<ServerIntels> getConfig()
    { /* Read "config/DNS.csv" file */ }

    public static String getAddress(String domain)
    throws DNSException
    { /* Code */ }

    public static int getPOP3(String domain)
    throws DNSException
    { /* Code */ }

    public static int getPOP3S(String domain)
    throws DNSException { /* Code */ }

    public static int getSMTP(String domain)
    throws DNSException
```

```
    { /* Code */ }

    public static int getServersNumber()
    { /* Code */ }

    public static List<String> getDomains()
    { /* Code */ }

}
```

Ces informations doivent parfois être mises à jour, dans ce cas il suffit d'aller éditer le fichier de configuration. Celui-ci est au format csv, il est donc simple à modifier.

Son contenu par défaut est

```
Domain name,IP address,POP3,POP3S,SMTP
email.com,127.0.0.1,1210,1211,1212
email.fr,127.0.0.1,1213,1214,1215
```

mais pour rajouter un serveur avec un autre nom de domaine (comme `test.de`) qui tournerait sur une autre machine (d'adresse IP 192.168.43.19 par exemple), il suffit d'ajouter la ligne suivante (en ajustant les ports)

```
test.de,192.168.43.19,1210,1211,1212
```

C - Optimisation

Avec notre premier développement, nous avions créé un unique objet SMTP, qui, pour chaque destinataire, ouvrait une connexion TCP pour envoyer le message et si le destinataire suivant nécessitait la connexion à un autre serveur, il fermait celle précedemment ouverte avant d'en ouvrir une autre sur le nouveau serveur. Étant donné que ceci n'était pas du tout optimisé, nous avons choisi de créer un objet SMTP pour chaque serveur renseigné dans la classe `DNS`. Tous ces objets `SMTP` sont manipulés grâce à la classe `SMTPDispatcher` et stockés dans une `HashMap<String, SMTP>` utilisant le nom de domaine comme clé. Ces objets ne gardent pas les connexions TCP toujours ouvertes étant donné que ceci serait inutile : à l'appel de la fonction `SMTP::SendMail(String targets, MailConvertor mailConvertor)`, la connexion est ouverte, le message envoyé, et la connexion est fermée. De cette façon la connexion ne reste pas toujours ouverte mais le fait d'avoir plusieurs instances nous permet de ne pas avoir à refaire toute la configuration de la connexion à chaque ouverture.

L'objet `MailConvertor` passé en paramètre à la fonction d'envoi de mail ci-dessus est un objet permettant, en lui donnant l'expéditeur d'un mail, son objet et son corps, de générer les lignes à envoyer avec les headers une seule fois. Nous avions ainsi la possibilité de changer le(s) destinataire(s) du mail et de récupérer à nouveau toutes les lignes à envoyer au serveur sans pour autant regénérer la totalité des headers du mail.

Une seconde amélioration a été de regrouper les destinataires par nom de domaine, afin d'ouvrir une seule fois la connexion TCP vers un serveur,

envoyer le mail pour tous les utilisateurs sur ce domaine et fermer la connexion.

3 - Frontend

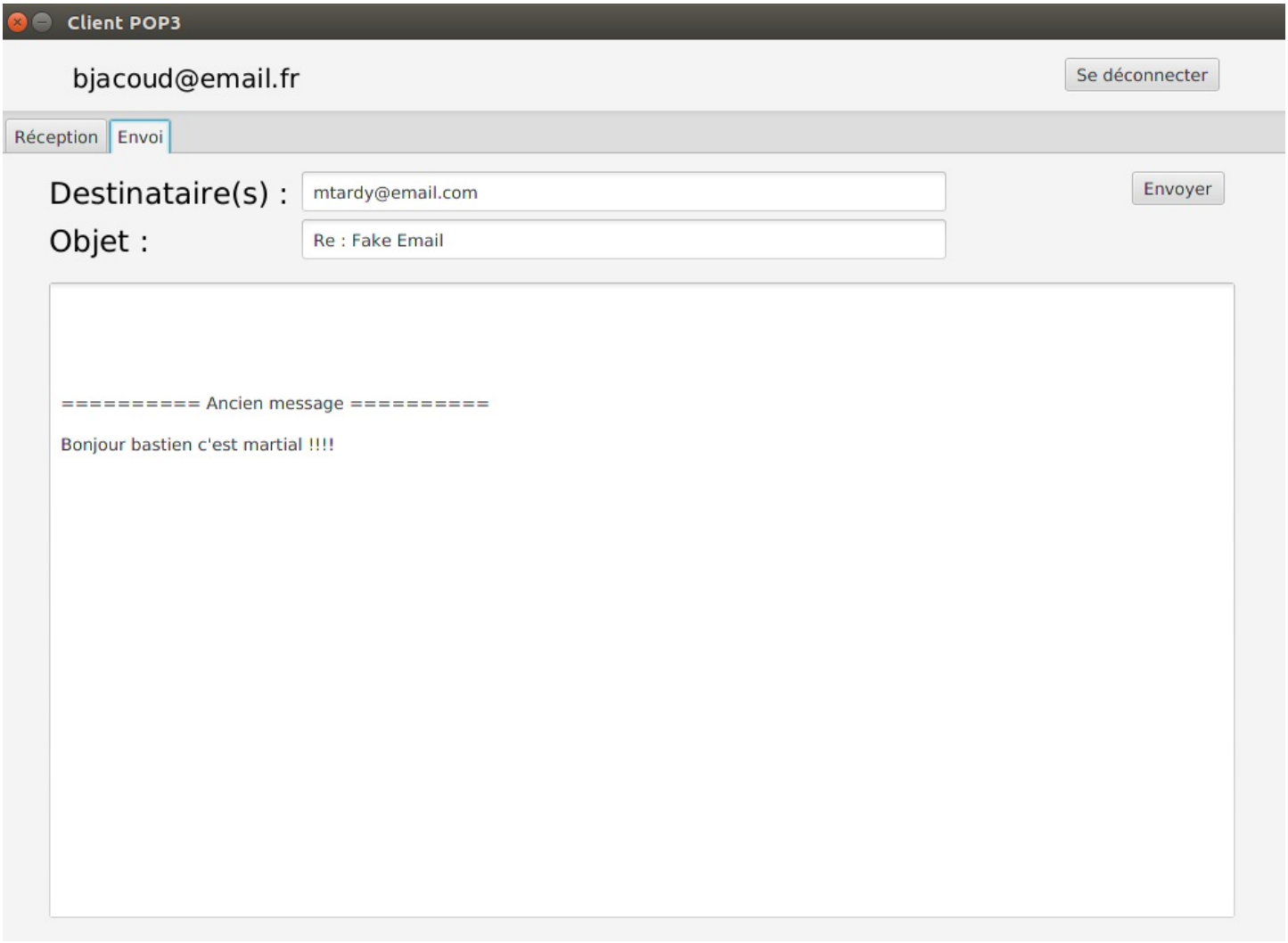
A - Présentation interface graphique

L’interface graphique utilisée dans le cadre du protocole STMP est toujours la même que celle utilisée pour le protocole POP3S. Il est tout de même important de spécifier que le code de cette dernière n’est plus le même que pour le protocole POP3S, car notre interface a été optimisée entre temps.

De plus, on utilise cette fois ci, non pas le premier onglet, listant tous les mails reçus par l’utilisateur, mais le second onglet qui nous permettra d’envoyer un mails à un ou plusieurs destinataires.

Vous trouverez ci-dessous un aperçu de notre fenêtre d’envoi de mails :

Afin d’améliorer considérablement l’expérience de l’utilisateur, le bouton “Répondre”, présenté lors de la partie POP3S redirige automatiquement l’émetteur sur l’onglet d’envoi de mails, et ce en remplissant automatiquement les champs nécessaires :



Ainsi, notre utilisateur n’a plus qu’a écrire sa réponse à l’endroit adéquat pour que son destinataire puisse savoir de quel mail est issue la réponse.

De plus, si le champ “Objet” n’est pas rempli par l’émetteur lors de l’écriture du mail, une confirmation lui sera demandée afin d’envoyer un mail ne comportant pas de mention “Objet”.

B - Gestion de plusieurs adresses mail valides ou non

Bien évidemment, notre service de messagerie étant capable d’envoyer des mails à plusieurs destinataires, appartenant éventuellement à différents

serveurs,
nous avons du traiter l'existence ou non de chacune des adresses mails des destinataires.

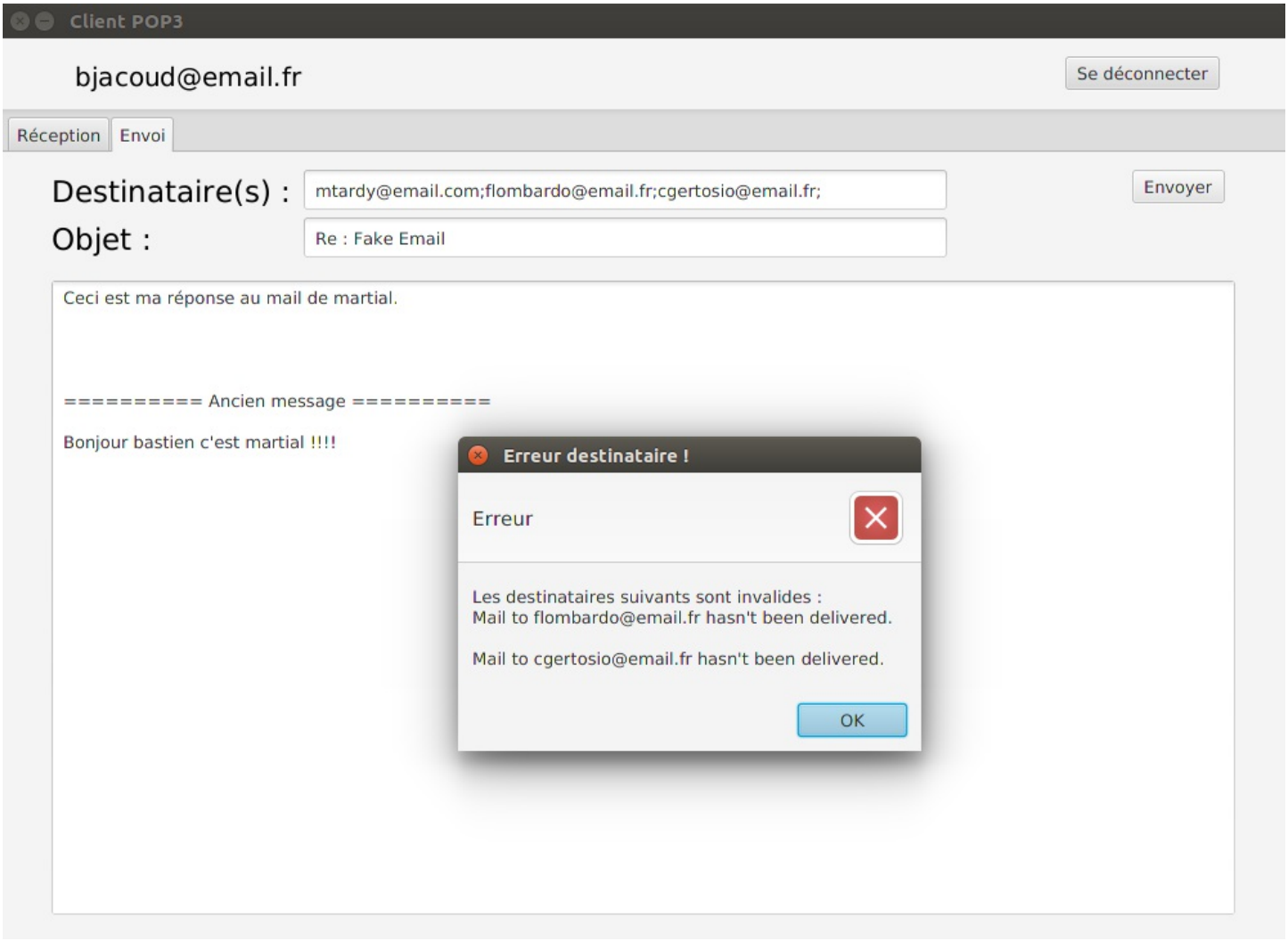
Dans un premier temps, nous réalisons, à l'aide d'une expression régulière, un test nous permettant de savoir si la syntaxe de chacune des adresses mails insérées est correcte ou non.

```
private static final String _MAIL = "(?:[a-z0-9!#$%&'*+/=?^_`
{|}~-]+(?:\\.[a-z0-9!#$%&'*+/=?^_`{|}~-]+)*|\"(?:[\\x01-\\x08
\\x0b\\x0c\\x0e-\\x1f\\x21\\x23-\\x5b\\x5d-\\x7f]|\\\\[\\x01-
\\x09\\x0b\\x0c\\x0e-\\x7f])*\")@(?:(?:[a-z0-9](?:[a-z0-9-]*[
a-z0-9])?\\.)+[a-z0-9](?:[a-z0-9-]*[a-z0-9])?|\\[(?:(?:25[0-5
]|2[0-4][0-9]|[01]?[0-9][0-9]?)\\.){3}(?:25[0-5]|2[0-4][0-9]|
[01]?[0-9][0-9]?|[a-z0-9-]*[a-z0-9]:(?:[\\x01-\\x08\\x0b\\x0c
\\x0e-\\x1f\\x21-\\x5a\\x53-\\x7f]|\\\\[\\x01-\\x09\\x0b\\x0c
\\x0e-\\x7f])+)\\])";

public static boolean CheckMails(String mails){
    String tabmails[] = mails.split(";");
    for(String mail : tabmails){
        if(!CheckMail(mail.trim())){
            return false;
        }
    }
    return true;
}
```

Deux possibilités s'offrent alors à nous : soit les adresses mails des destinataires possèdent une syntaxe correcte, auquel cas nous essayons d'envoyer le mail à chaque destinataire. Soit une ou plusieurs adresses mails possèdent une syntaxe incorrecte, auquel cas nous affichons un message d'erreur à l'utilisateur.

Si la syntaxe de chaque adresse écrite par l'utilisateur est correcte, nous essayons d'envoyer le mail à chaque utilisateur ainsi mentionné. Si un utilisateur est mentionné mais n'existe pas au niveau de notre serveur, un message est envoyé à l'utilisateur. Ce message indique les adresses mails de tous les utilisateurs inexistants :

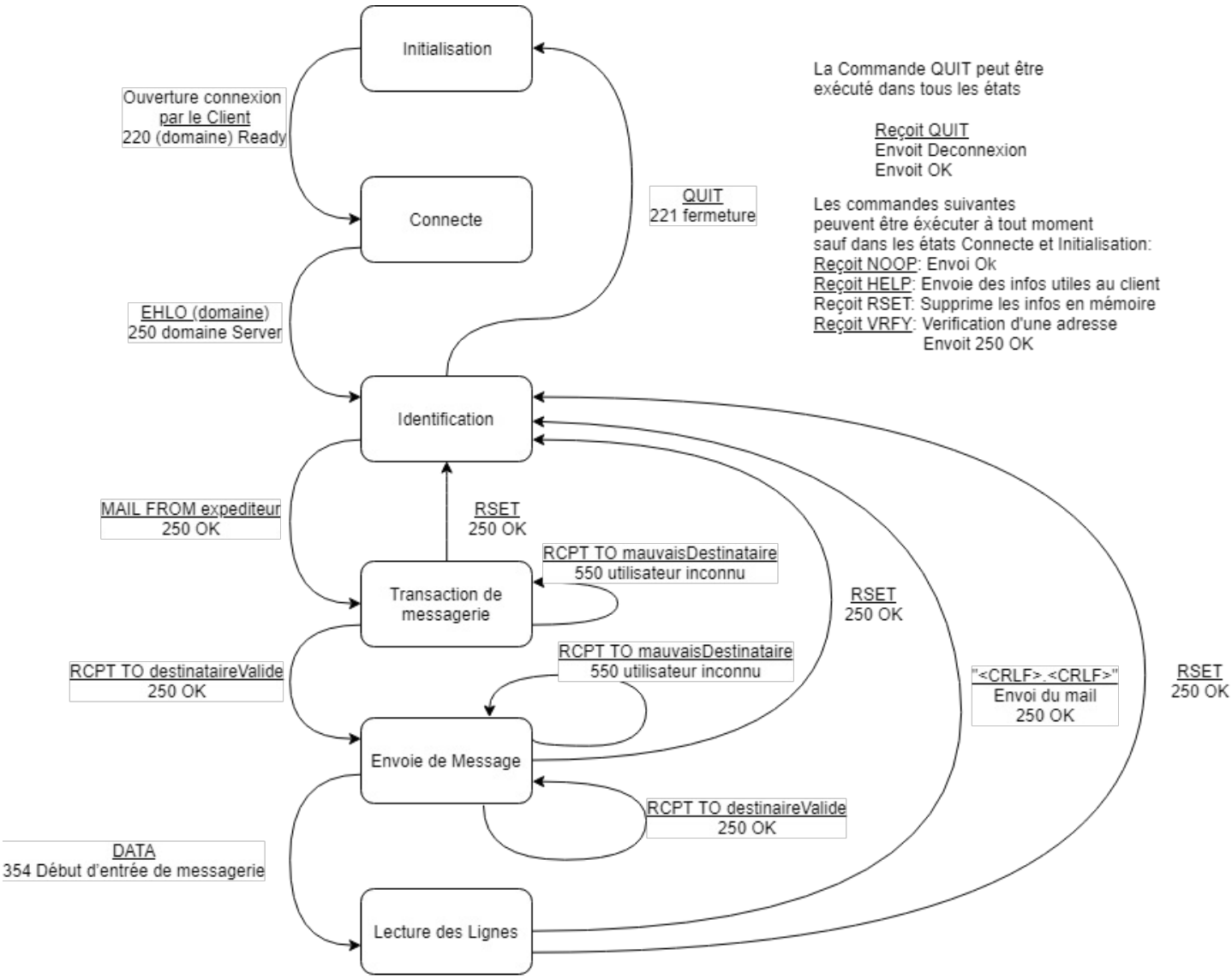


L'émetteur est alors informé de chaque destinataire n'ayant pas pu recevoir son mail.

III - Serveur

1 - Automate & Table de transitions

Avant de commencer l'implémentation du Serveur, nous avons réalisé l'automate de celui-ci. Nous nous sommes basés sur la documentation du protocole SMTP.



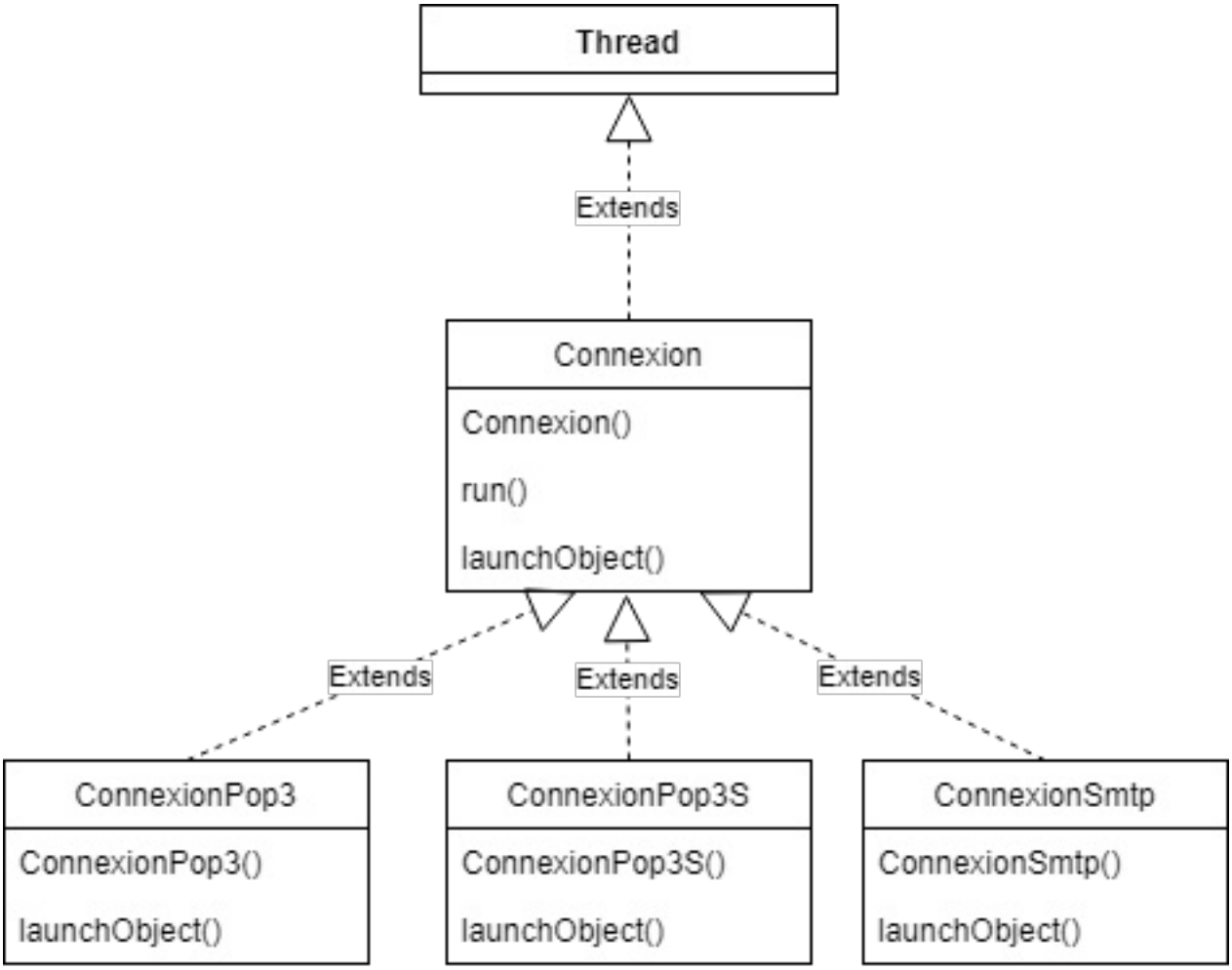
A partir de cet automate, nous avons pu réaliser sa table de transition:

	EHLO	MAIL FROM	RCPT TO	DATA	QUIT	RSET	CLRF
Connecte	identification 250 OK	IGNORE	IGNORE	IGNORE	IGNORE	IGNORE	IGNORE
Identification	IGNORE	ajoute expéditeur à un nouveau email 250 OK	IGNORE	IGNORE	Retour initialisation 221 fermeture	IGNORE	IGNORE
Transaction de messagerie	IGNORE	IGNORE	Verification de l'existence du destinataire + ajoute destinataire si correct 250 OK 550 utilisateur inconnu	IGNORE	Retour initialisation 221 fermeture	Retour etat Identification 250 OK	IGNORE
Envoi de message	IGNORE	IGNORE	Verification de l'existence du destinataire + ajoute destinataire si correct 250 OK 550 utilisateur inconnu	débute de lire les lignes 354 Début d'entrée de messagerie	Retour initialisation 221 fermeture	Retour etat Identification 250 OK	IGNORE
Lecture des lignes	IGNORE	IGNORE	IGNORE	IGNORE	Retour initialisation 221 fermeture	Retour etat Identification 250 OK	Envoi du mail 250 OK

2 - Backend

A - Fonctionnement avec plusieurs serveurs

Les serveurs SMTP, POP3 et POP3S doivent fonctionner ensemble, le but étant que l’utilisateur pourra choisir lors de sa connexion, s’il veut se connecter au POP3 ou au SMTP. Pour éviter toutes incompatibilités lors du l’exécution du main, on a utilisé la structure ci-dessus:



La méthode `run()` de la classe Connexion contient une boucle infini pour que le serveur puisse accepter toutes les connexions tant que celles-ci se font sur le bon port. Ainsi, lorsqu'on exécute le main, les 3 serveurs se mettent en écoutent sur les différents ports qu'on leur a assigné. Le serveur POP3 écoute sur le port 1210, le POP3S sur le 1211 et le SMTP sur le port 1212.

```
ConnexionPOP3 connexionPOP3 =
    new ConnexionPOP3(domain);
connexionPOP3.start();
ConnexionPOP3S connexionPOP3S =
    new ConnexionPOP3S(domain);
connexionPOP3S.start();
```

```
ConnexionSMTP connexionSMTP =
    new ConnexionSMTP(domain);
connexionSMTP.start();
```

Lorsqu'un client va se connecter à l'un des serveurs, la méthode launchObjet() va s'exécuter pour créer un objet correspondant au serveur choisit. Par exemple, le serveur SMTP exécutera cette méthode:

```
protected void launchObject(Socket socket)
throws IOException {
    ObjetSmtpConnecte objetConnecte =
        new ObjetSmtpConnecte(socket, domain);
    objetConnecte.start();
}
```

Les identifiants des utilisateurs pour chaque domaine sont enregistrés dans des fichiers différents et chaque utilisateur possède un fichier dans lequel sera écrit les messages reçus.

B - Implémentation du serveur SMTP

Pour le développement SMTP, nous nous sommes servi de ce que nous avions fait pour POP3, donc la structure du projet est similaire. Le serveur étant concurrent, lorsqu'un client se connectera sur le port 1212, un thread sera créé dans la classe ObjetSmtpConnecte pour lui permettre de communiquer avec le serveur. Elle est aussi chargé de faire respecter l'automate du serveur. Les méthodes `send()` et `receive()` qui

permettent la communication entre le serveur et le client sont implémentées dans la classe Tcp(). Cette classe sera instanciée dans le constructeur de ObjetSmtpConnecte.

Pour simplifier l'organisation du code, les messages d'erreurs et de confirmation que peuvent envoyer le serveur ont été stockés dans la classe ReponseServeur.

```
public class ReponseServeur {
    public final static String SMTP_SERVER_READY =
        "220 Simple Mail Transfer Service Ready";
    public final static String SMTP_500_UNKNOWN_COMMAND =
        "500 Erreur de syntaxe, commande non reconnue";
    public final static String SMTP_221_CLOSING =
        "221 fermeture";
    public final static String SMTP_250_OK =
        "250 OK";
    public final static String SMTP_550_UNKNOWN_USER =
        "250 utilisateur inconnu";
    public final static String SMTP_354_START_READING =
        "354 debut de lecture";
    /* Etat */
    public final static String SERVER_READY =
        "Initialisation";
    public final static String SERVER_CONNEXION =
        "Connexion";
    public final static String SERVER_IDENTIFICATION =
        "Identification";
```

```
    public final static String SERVER_TRANSACTION =
        "Transaction de Messagerie";
    public final static String SERVER_ENVOIE =
        "Envoie de Message";
    public final static String SERVER_LECTURE =
        "Lecture des Lignes";

    public final static String SMTP_CRLF = "\\r\\n";
}
```

Comme dit plus haut, la classe ObjetSmtpConnecte va permettre la communication entre le client et le serveur. Afin de pouvoir traiter toutes les requêtes de l'utilisateur, l'ensemble des traitements possibles est contenue dans une boucle `while(continuer)` ('continuer' etant un booléèn ayant pour valeur par défaut true) qui ne s'arretera que lorsque l'utilisateur émettra une requête QUIT. Dans un premier temps, on commence par initialiser l'état du serveur comme c'est marqué dans l'automate. Ensuite, on recupère la commande et les paramètres envoyés par le client à travers la méthode `receive()`. Selon l'état dans lequel se trouve le serveur, la méthode correspondante sera appelée et prendra en paramètre les informations envoyées par l'utilisateur.

```
switch (etatServeur) {
    case SERVER_CONNEXION:
        reponseServeur = this.connexion(
                    command,
                    parameters);
```

```
            break;
        case SERVER_IDENTIFICATION:
            reponseServeur = this.identification(
                        command,
                        parameters);
            break;
        case SERVER_TRANSACTION:
            reponseServeur = this.transaction(
                        command,
                        parameters);
            break;
        case SERVER_ENVOIE:
            reponseServeur = this.envoie(
                        command,
                        parameters);
            break;
        case SERVER_LECTURE:
            reponseServeur = this.lecture(
                        command,
                        parameters);
            break;
        default:
            reponseServeur = SMTP_500_UNKNOWN_COMMAND;
    }
```

Chaque état a un ensemble de commande qui est possible d'exécuter. Par exemple, dans l'état Identification, il sera possible pour l'utilisateur

d’utiliser les commandes MAIL, RSET et QUIT.

```
private String identification(String command,
                                    String[] parameters) {
    switch (command){
        case "MAIL":
            return commandeMailFrom(parameters);
        case "QUIT":
            return commandeQuit();
        case "RSET":
            return commandeRset();
        default :
            return SMTP_500_UNKNOWN_COMMAND;
    }
}
```

Si l’utilisateur envoie la commande QUIT, la méthode `commandeQuit()` va s’exécuter. Le booléen `continuer` devient false ce qui va entrainer l’arret de la boucle `while`. Le serveur va se mettre dans l’état d’initialisation et va retourner à l’utilisateur un message de fermeture.

```
private String commandeQuit() {
    continuer = false;
    etatServeur = SERVER_READY;
    return SMTP_221_CLOSING;
}
```

Lorsque le traitement est fini, on utilise la méthode send() de la connexion tcp du client pour lui renvoyer la réponses:

```
tcp.send(reponseServeur);
```

L'ensemble des commandes fonctionnent de la même manière.

IV - Utilisation

1 - Le Serveur

Pour lancer le serveur, il suffit d'exécuter via une console `Serveur.jar`, situé dans le répertoire `Binaires/Serveur/`. Attention, pour son bon fonctionnement, toujours l'exécuter dans ce repertoire (à côté du répertoire `data/`). Par défaut, le serveur se lance avec le nom de domaine `email.com`. Pour choisir le nom de domaine, il faut le passer en arguement.

Par exemple :

```
java -jar Serveur.java
```

Lance le serveur avec comme nom de domaine `email.com`

```
java -jar Serveur.java "email.fr"
```

Lance le serveur avec comme nom de domaine `email.fr`

Vous devriez ensuite avoir l'affichage suivant, afin de savoir sur quels ports peut se connecter le client :

```
java -jar Serveur.java "email.fr"

Lancement du domaine email.fr
Attente de connexion POP3 au port 1210
Attente de connexion POP3S au port 1211
Attente de connexion SMTP au port 1212
```

`email.fr` et `email.com` sont les deux seuls noms de domaine disponnibles. Si vous mettez un nom de domaine qui n'existe pas, c'est le nom de domaine `email.com` qui sera utilisé.

2 - Le Client

L'exécutable du client (*Client.java*) se trouve lui dans le répertoire `Binaires/Client/`.

Avant de l'exécuter il est conseillé de vérifier que le fichier de configuration `Binaires/Client/config/DNS.csv` est correctement rempli.

Il n'est pas necessaire de le lancer à la console, un double clic suffit à lancer l'interface graphique. Il peut cependant être lancé avec la

commande :

```
java -jar Client.java
```

V - Conclusion

Nous avons finalement mis au point un serveur permettant de gérer une messagerie identifiée par un nom de domaine sur un serveur, ainsi qu'un client permettant d'y accéder à distance afin de relever son courrier et/ou d'envoyer de nouveaux messages.

Ce TP nous aura apporté un bon nombre de connaissances sur le fonctionnement des protocoles de communication et les normes que les régissent. Il nous aura aussi permis de revoir le principe de fonctionnement d'un serveur concurrent.