POLYTECH LYON

RAPPORT DE TP POP3

IPC

BENALI Myriam, BESSON Cécile, DELPLANQUE Thibaut, NAAJI Dorian
POLYTECH LYON
4A INFO GROUPE 2

POLYTECH LYON

TABLE DES MATIERES

1.INTRODUCTION

Ce TP a consisté en la réalisation d'un client/serveur POP3 « Post Office Protocol3 » en Java, conformément à la norme RFC 1939.

Ainsi, deux programmes ont été réalisés :

- Le logiciel « Serveur », qui permet de gérer la connexion et le stockage de mails de différents utilisateurs.
- Le logiciel « Client », qui, en se connectant au serveur, permet à chaque utilisateur de récupérer le contenu de sa boîte mail.

Ces programmes communiquent grâce au protocole TCP/IP. Nous avons donc utilisé les sockets INET proposés par Java.

Le serveur est un logiciel qui va écouter l'ensemble des requêtes des clients et les traiter. Il fonctionne en mode concurrent, c'est-à-dire qu'il crée un processus de traitement pour chaque client souhaitant se connecter.

Le client, quant à lui, est une application graphique JavaFX qui permet à l'utilisateur de communiquer avec le serveur. Cette application permet de consulter une messagerie électronique.

2.SERVEUR

2.1. Graphe de l'automate

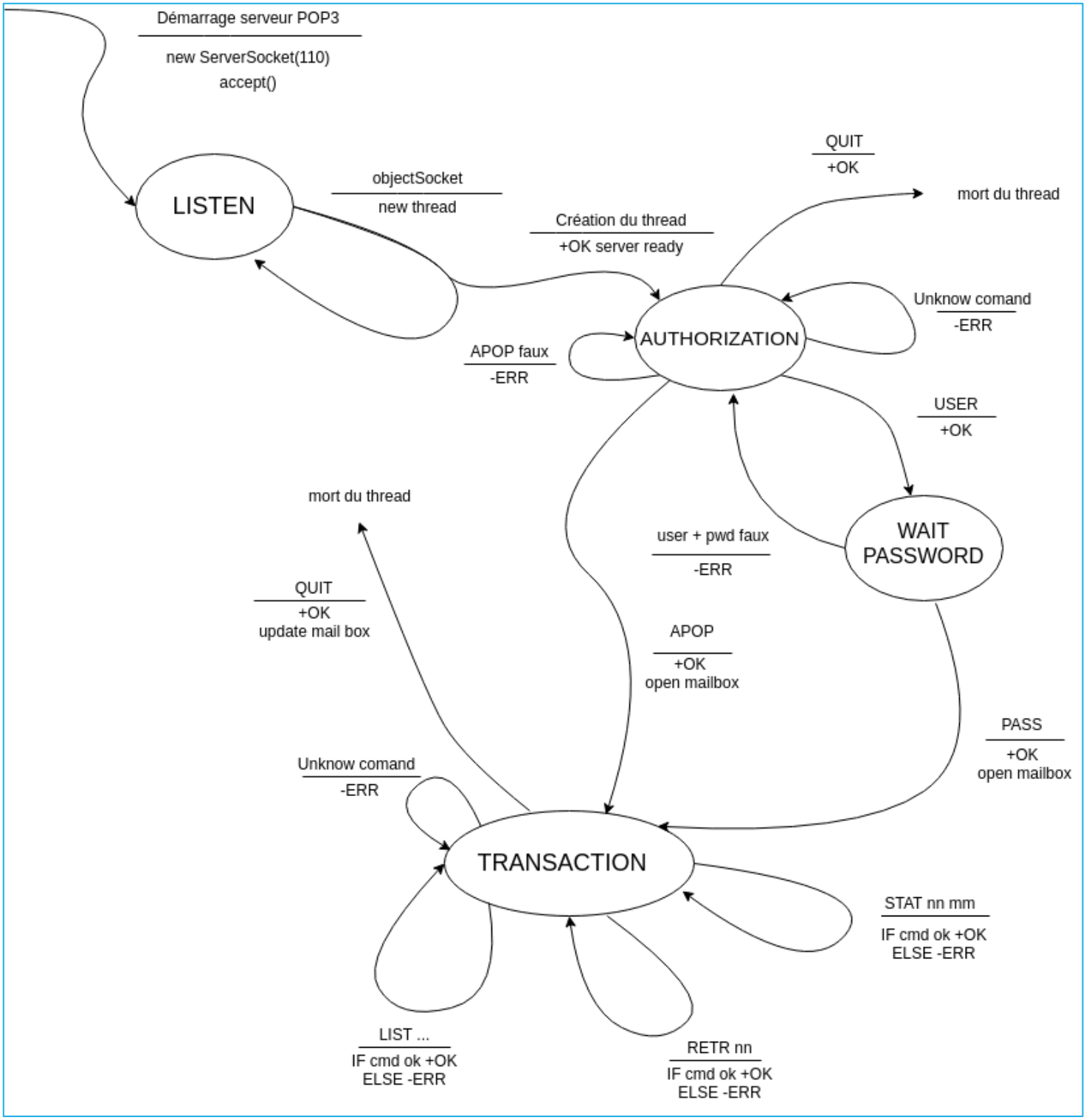


Figure 1 : Graphe de l'automate serveur, représentant les différents états et transitions du logiciel

2.2. Table de transition

	LISTEN	AUTHORIZATION	WAIT PASSWORD	TRANSACTION
Processus principal du serveur				

Démarrage du serveur POP3	new ServerSocket(110);
	accept();
	Reste dans **LISTEN**
		impossible	**impossible**	**impossible**

Réception d'un Objet Socket	création d'un nouveau thread pour la communication avec le client Reste dans **LISTEN**	**impossible**	**impossible**	**impossible**
Thread de communication avec un client				
APOP valide	**impossible**	envoie "+OK" passe dans **TRANSACTION**	envoie "-ERR" passe dans **AUTHORIZATION**	envoie "-ERR vous êtes déjà connecté" reste dans **TRANSACTION**
APOP invalide	**impossible**	envoie "-ERR échec de l'authentification" reste dans **AUTHORIZATION**	envoie "-ERR" passe dans **AUTHORIZATION**	envoie "-ERR vous êtes déjà connecté" reste dans **TRANSACTION**
USER	**impossible**	envoie "+OK" passe dans **WAIT PASSWORD**	envoie "-ERR" passe dans **AUTHORIZATION**	envoie "-ERR vous êtes déjà connecté" reste dans **TRANSACTION**
PASS	**impossible**	envoie "-ERR vous devez d'abord envoyer la commande USER" reste dans **AUTHORIZATION**	si USER + PASS valide envoie "+OK" passe dans **TRANSACTION** sinon envoie "-ERR" passe dans **AUTHORIZATION**	envoie "-ERR vous êtes déjà connecté" reste dans **TRANSACTION**
STAT	**impossible**	envoie "-ERR vous devez vous authentifier" reste dans **AUTHORIZATION**	envoie "-ERR vous devez vous authentifier" passe dans **AUTHORIZATION**	envoie "+OK <nb mails> <poids des mails en octets>" reste dans **TRANSACTION**
RETR <nn>	**impossible**	envoie "-ERR vous devez vous authentifier" reste dans **AUTHORIZATION**	envoie "-ERR vous devez vous authentifier" passe dans **AUTHORIZATION**	si RETR valide envoie "+OK ..." sinon envoie "-ERR" reste dans **TRANSACTION**
LIST <nn>	**impossible**	envoie "-ERR vous devez vous authentifier" reste dans **AUTHORIZATION**	envoie "-ERR vous devez vous authentifier" passe dans **AUTHORIZATION**	si LIST valide envoie "+OK ..." sinon envoie "-ERR" reste dans **TRANSACTION**
Commande inconnue	**impossible**	envoie "-ERR commande inconnue" reste dans **AUTHORIZATION**"	envoie "-ERR commande inconnue" passe dans **AUTHORIZATION**	envoie "-ERR commande inconnue" reste dans **TRANSACTION**
QUIT	**Impossible**	envoie "+OK" fermeture de la connexion le thread meurt **FIN**	envoie "+OK" fermeture de la connexion le thread meurt **FIN**	Mise à jour de la boîte mail (suppression des mails marqués) envoie +OK fermeture de la connexion le thread meurt **FIN**

Figure 2 : Table de transition du logiciel serveur, permettant de visualiser l'ensemble des cas possibles

3. CLIENT

3.1. Graphe de l'automate

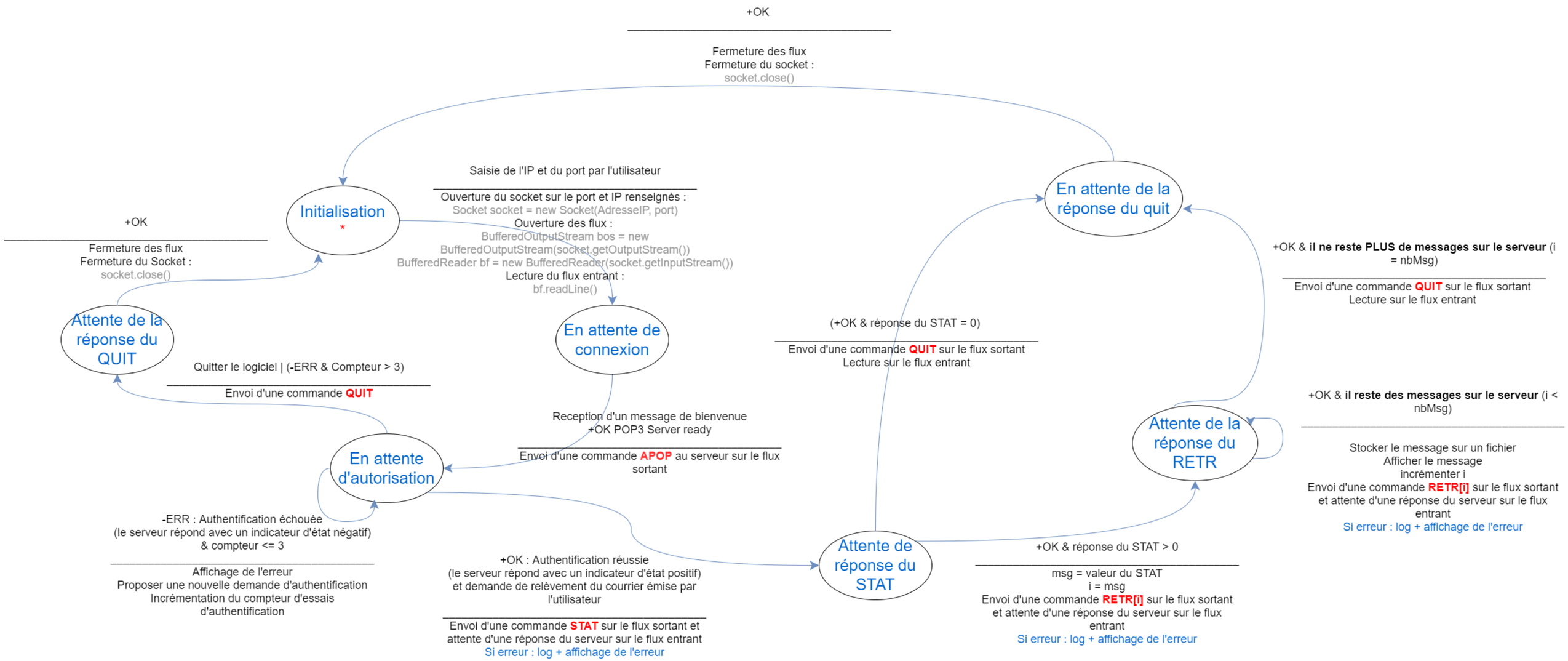


Figure 3 : Graphe de l'automate client, représentant les différents états et transitions du logiciel

3.2. Guide utilisateur

Lors du lancement de l'application, une fenêtre s'affiche proposant à l'utilisateur de saisir le port et l'IP.

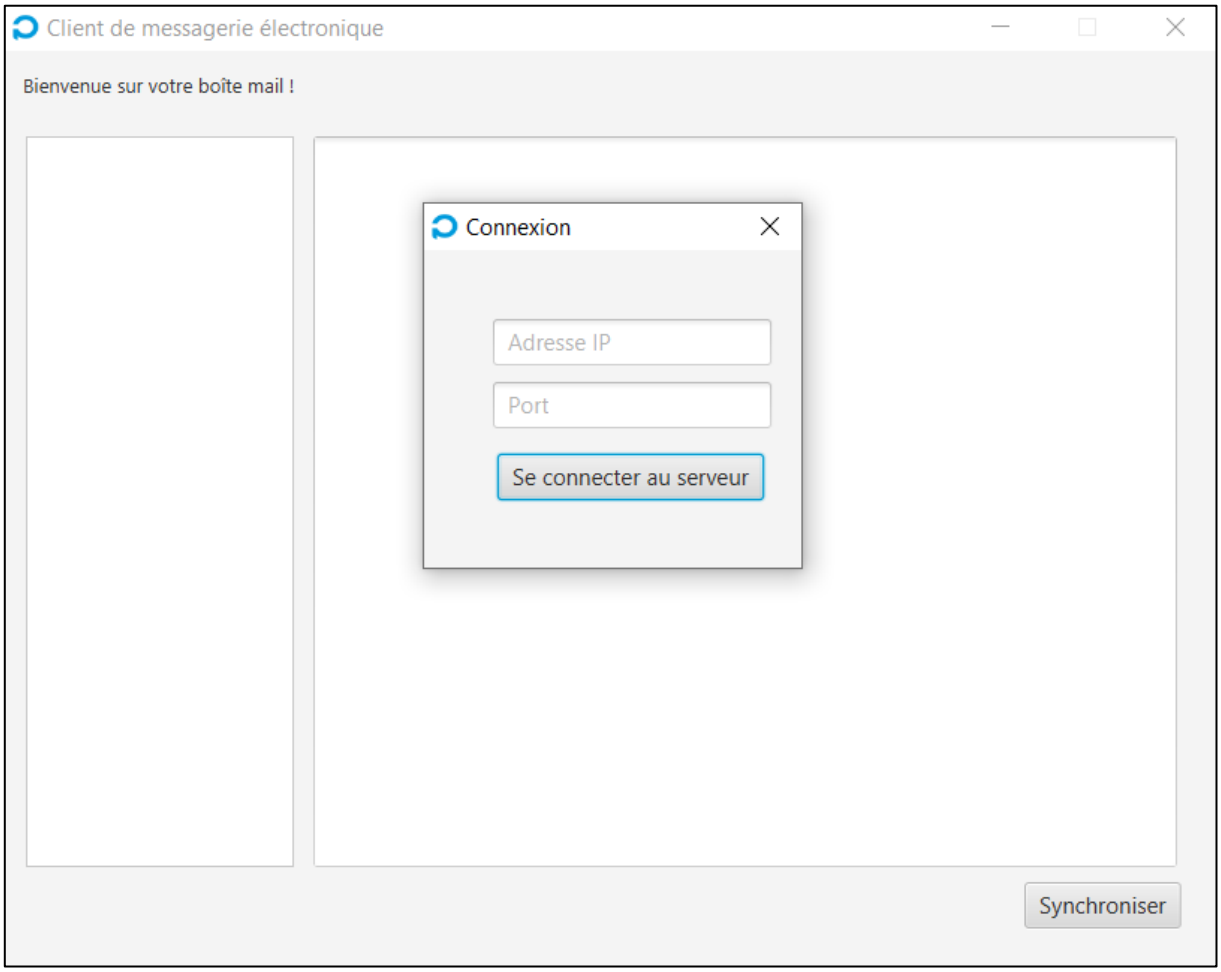


Figure 4 : Lancement de l'application client

Si les champs sont incorrects, un pop-up avec le message approprié s'ouvre, en fonction de ce qui a été mal renseigné par l'utilisateur :

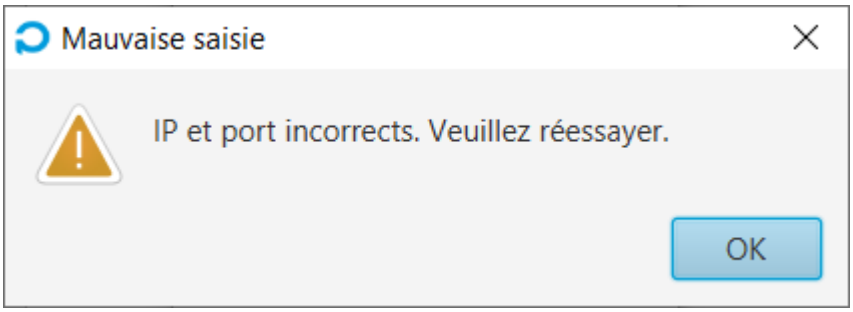


Figure 5 : Message d'alerte IP et port incorrects

Une fois que l'adresse IP et le port ont été correctement renseignés, le serveur va envoyer un message de bienvenue au client : « +OK POP3 Server ready ». S'ouvre alors une fenêtre d'authentification, qui va permettre d'envoyer une commande **APOP** au serveur sur le flux sortant.

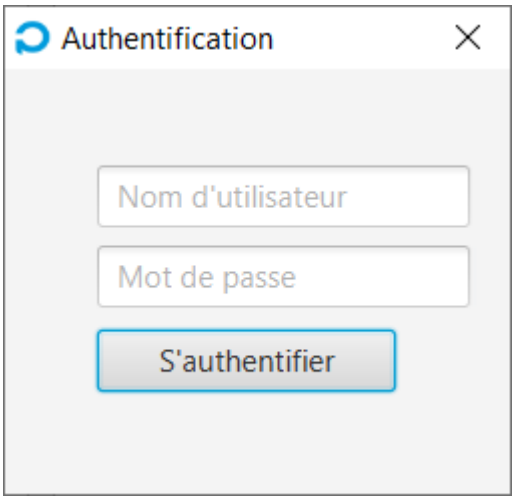


Figure 6 : Fenêtre d'authentification

Une fois la commande APOP émise, le serveur la traite et y répond. Si sa réponse est négative, « -ERR », on en informe l'utilisateur

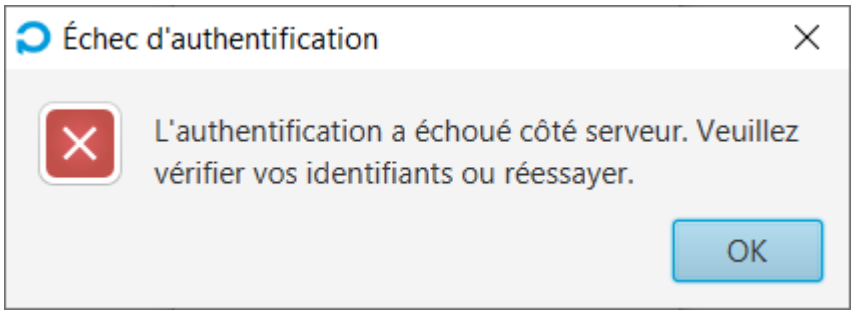


Figure 7 : Message d'échec d'authentification

Si la demande d'authentification a échoué trois fois, une fenêtre en avertit l'utilisateur, puis une commande QUIT est émise et le logiciel se ferme.

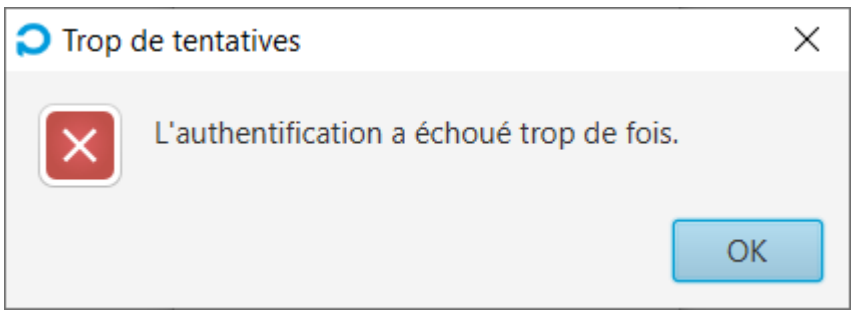


Figure 8 : Message d'erreur en cas de nombreux échecs d'authentification

En revanche, si la demande d'authentification a réussi, le logiciel client reçoit un « +OK » de la part du serveur. Le client envoie alors une commande **STAT** sur le flux sortant qui nous donne des informations sur les messages contenus sur le serveur (le numéro du message et sa taille).

Le serveur répond +OK avec un nombre de messages supérieur à 0 sur le serveur alors le client va envoyer une commande **RETR**[numéro du message] tant qu'il reste des messages sur le serveur.

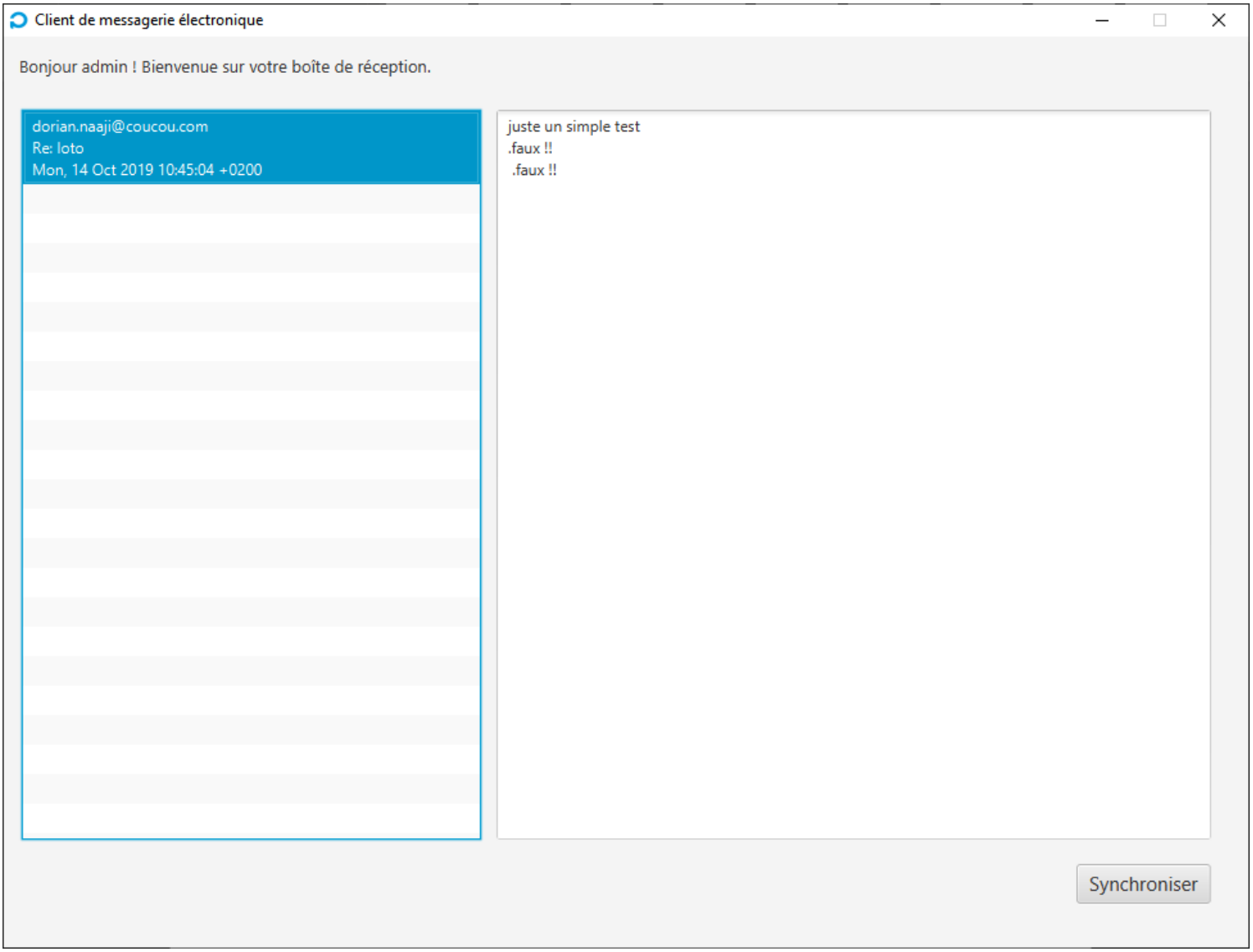


Figure 9 : Consultation de la messagerie électronique

4. SÉCURISATION

Notre application client / serveur POP3 communique grâce au protocole TCP, et, jusqu'à maintenant, aucune sécurisation n'avait été mise en place lors des échanges entre le client et le serveur. Nous faisions une vérification de l'identité du client, en lui demandant un **LOGIN** et un **PASSWORD** (plus précisément son **HASH**) à l'aide de la commande **APOP** (ou du couple **USER** et **PASS**), pour lui permettre d'accéder seulement à ses mails. Seulement, deux problèmes s'opposent à nous.

Premièrement, lors de la connexion du client, le **LOGIN** et **PASSWORD** circule en clair sur le réseau (ou le hash du **PASSWORD** lorsqu'on utilise **APOP**). Ce qui permet à un pirate informatique se trouvant sur le même réseau, de capter le **LOGIN** + **PASSWORD** et de les réutiliser plus tard (ou pour **APOP**, de récupérer **LOGIN** + **HASH** et de les réutiliser, car oui, notre serveur, pour la commande **APOP** accepte juste le **LOGIN** et le **HASH**, qu'importe l'origine de celui-ci).

Secondement, outre ce problème d'authentification, notre canal de communication n'est pas sécurisé, tous ce qui circule dessus y est écrit en clair. Sans même que notre pirate informatique vole et se connecte avec notre identité, il lui suffit juste d'attendre qu'un client se connecte et requête ses mails au serveur pour voir tout le contenu circuler sur le réseau.

4.1. Timbre à date

Pour notre premier problème, la mise en place d'un "timbre à date" nous permet d'assurer que celui qui s'authentifie est bien la seule personne à connaître le mot de passe. Le principe du "timbre à date" consiste, pour le serveur, à générer un timbre à date et de le communiquer lors du message de bienvenue du serveur. Ce timbre doit être différent à chaque génération et respecte la forme suivante :

- **pid** : l'identifiant du processus
- **hh** : heure à laquelle le timbre est généré
- **mm** : minutes à laquelle le timbre est généré
- **ss** : secondes à laquelle le timbre est généré
- **hostname** : nom de domaine ou adresse du serveur

Avec ces différentes informations, nous pouvons générer un "timbre à date" différent à chaque fois.

Exemple d'un "timbre à date" généré à 13h34 et 45 secondes en localhost, avec le pid du serveur égal à 16964

```
<16964:13:34:45@localhost>
```

Du coté de notre client, lorsqu'il recevra le message de bienvenue, avec le "timbre à date", il devra renvoyer son **LOGIN** + le **HASH** du timbre à date concaténé avec son mot de passe. Pour que le serveur puisse vérifier si l'authentification est valide, il devra lui aussi, générer le **HASH** du timbre à date concaténé avec le mot de passe du client et enfin comparer les deux hash finaux.

En faisant en sorte que chacune des deux parties (client et serveur) réalise cette opération, générer le **HASH** d'une chaîne (qui est unique pour chaque opération) avec un élément secret, seulement connue des deux parties, nous somme sûr des points suivants :

- Seul le client et le serveur peuvent réaliser cette opération, ils sont les seuls à connaître l'élément secret.
- Même si le pirate récupère le "timbre à date", il ne pourra pas générer le **HASH** avec l'élément secret, puisqu'il ne l'a pas en sa possession.
- Même si le pirate récupère le **HASH** final, il ne pourra pas s'en servir, car le **HASH** dépend d'un timbre à date qui est unique à chaque opération et nécessite de régénérer le **HASH** final.

Dans notre cas, l'élément secret partagé entre le client et le serveur est le **PASSWORD**, mais stocker le mot de passe en clair coté serveur peut poser quelques problèmes de sécurité, nous pourrions alors garder seulement le **HASH** du mot de passe. Ce qui, pour notre méthode de "timbre à date" nécessite que l'élément secret partagé soit le **HASH** du mot de passe et non plus le **PASSWORD**.

4.2. Connexion TLS

Dans le cas de notre second problème, pour que la communication soit sécurisée entre le client et le serveur. Nous avons mis en place une connexion TLS grâce à l'utilisation de **SSLSERVERSOCKET**. Le principe consiste à échanger une clef maître à l'aide d'un chiffrement asymétrique, puis, pour chaque message envoyé entre le client et le serveur, de chiffrer ces messages à l'aide de la clef maître avec un chiffrement symétrique.

Lors de l'échange de la clef maître, un handshake est réalisé, où le client et le serveur vont se mettre d'accord sur une "cipher suite" à utiliser (c'est à dire l'algorithme d'échange de clés et d'authentification, de chiffrement par bloc et de code d'authentification de message). Ensuite, une étape d'authentification par certificat numérique est réalisée pour certifier de l'identité du serveur et ainsi garantir que le client envoie ses informations à la bonne personne.

Nous pouvons différencier deux types de "cipher suite", celle avec "anon" pour "anonyme", qui n'oblige pas d'utiliser un certificat, et les autres. Dans notre cas, nous avons utilisé un "cipher suite" sans "anon", du fait que notre version de java ne permette pas un accès facile aux "cipher suites anon".

Pour ce faire, nous avons préféré passer par les certificats auto-signés à l'aide de la commande KEYTOOL. Cette commande nous a permis de générer deux KEYSTORES, un pour le client et un pour le serveur, permettant de stocker clefs et certificats. Coté serveur, nous avons généré un certificat propre à celui-ci que nous avons ensuite importé dans le KEYSTORE du client.

Enfin, côté java, nous avons précisé les KEYSTORE à utiliser pour le client et le serveur à l'aide de la commande `System.setProperty(...) ;`

Ce qui nous donne pour le serveur :

```
System.setProperty("javax.net.ssl.keyStoreType","jks");
// chemin du keystore serveur
System.setProperty("javax.net.ssl.keyStore","./conf/server.jks");
// mot de passe du keystore
System.setProperty("javax.net.ssl.keyStorePassword","password");
```

Puis pour le client :

```
System.setProperty("javax.net.ssl.keyStoreType","jks");
// chemin du keystore client
System.setProperty("javax.net.ssl.trustStore", "./conf/client.jks");
// mot de passe du keystore
System.setProperty("javax.net.ssl.trustStorePassword", "password");
```

5. DOSSIER DE PROGRAMMATION

5.1. Notice explicative du fonctionnement du programme serveur

Le programme côté serveur est organisé sous la forme de différents packages :

- Le package Application :

Il contient les trois classes principales. La classe **serveur** sert à établir une connexion sécurisée grâce à un SSL serveur socket (cf. paragraphe précédent). Dès qu'une demande de connexion est réceptionnée, une instance de la classe **connexion** est créée et est passée en paramètre d'un nouveau thread. C'est ce mécanisme qui nous permet à plusieurs clients de se connecter en même temps sur notre application.

La classe connexion est la classe la plus importante de l'application. En effet, c'est elle qui est responsable de la communication avec le client. Toutes les commandes envoyées par le client passent par sa méthode *automate* qui se charge de réaliser le bon traitement en fonction de la demande de l'utilisateur. La classe contient ainsi toutes les méthodes permettant au client de l'authentifier ainsi que de consulter sa boite mail.

La classe commande, qui définit un objet représentant la commande envoyée par le client. Il est ainsi plus aisé d'accéder au mot clé d'une commande ainsi qu'à ses différents paramètres.

- le package Mailbox :

Il contient la classe Mail qui regroupe toutes les caractéristiques d'un mail. On y trouve aussi la classe MailBox qui permet de charger une boite de réception et de lire les mails qu'elle contient.

- le package Service :

Il contient les classes UserHandler et ConfigHandler qui sont des singletons. Ce sont des classes qui permettent d'avoir accès aux users, aux password ou aux différentes variables de configuration partout dans le code, de façon simple et rapide.

- le package Utils qui contient une unique classe, Popsecurity qui permet de chiffrer, en MD5 notamment.

Nous pouvons également trouver un dossier ressources qui stocke les boites mails des différents utilisateurs et leur contenu, ainsi que la liste des utilisateurs et de leur mot de passe. Enfin, le dossier config qui contient un fichier JSON de configuration (port, hostname, mailbox path, etc.).

5.2. Notice explicative du fonctionnement du programme client

La programmation côté client repose sur les états et événements définis par l'automate en partie 3.1.

Étant donné que nous avons opté pour l'implémentation d'une interface graphique, nous avons souhaité respecter certains principes de programmation.

Le code est donc divisé en deux parties bien distinctes :

- la partie **traitement métier** ou ***back-end***,
- la partie **interface utilisateur** ou ***front-end / graphical user interface***.

Le schéma ci-après (**FIGURE 10**) détaille cela visuellement.

La couche de traitement métier contient l'ensemble des algorithmes liés au traitement de l'automate, afin de respecter le protocole POP3. Elle communique avec les interfaces JavaFX, afin d'afficher les informations à l'utilisateur :

- demandes de connexion / d'authentification,
- affichage des mails,
- affichage des erreurs,
- etc.

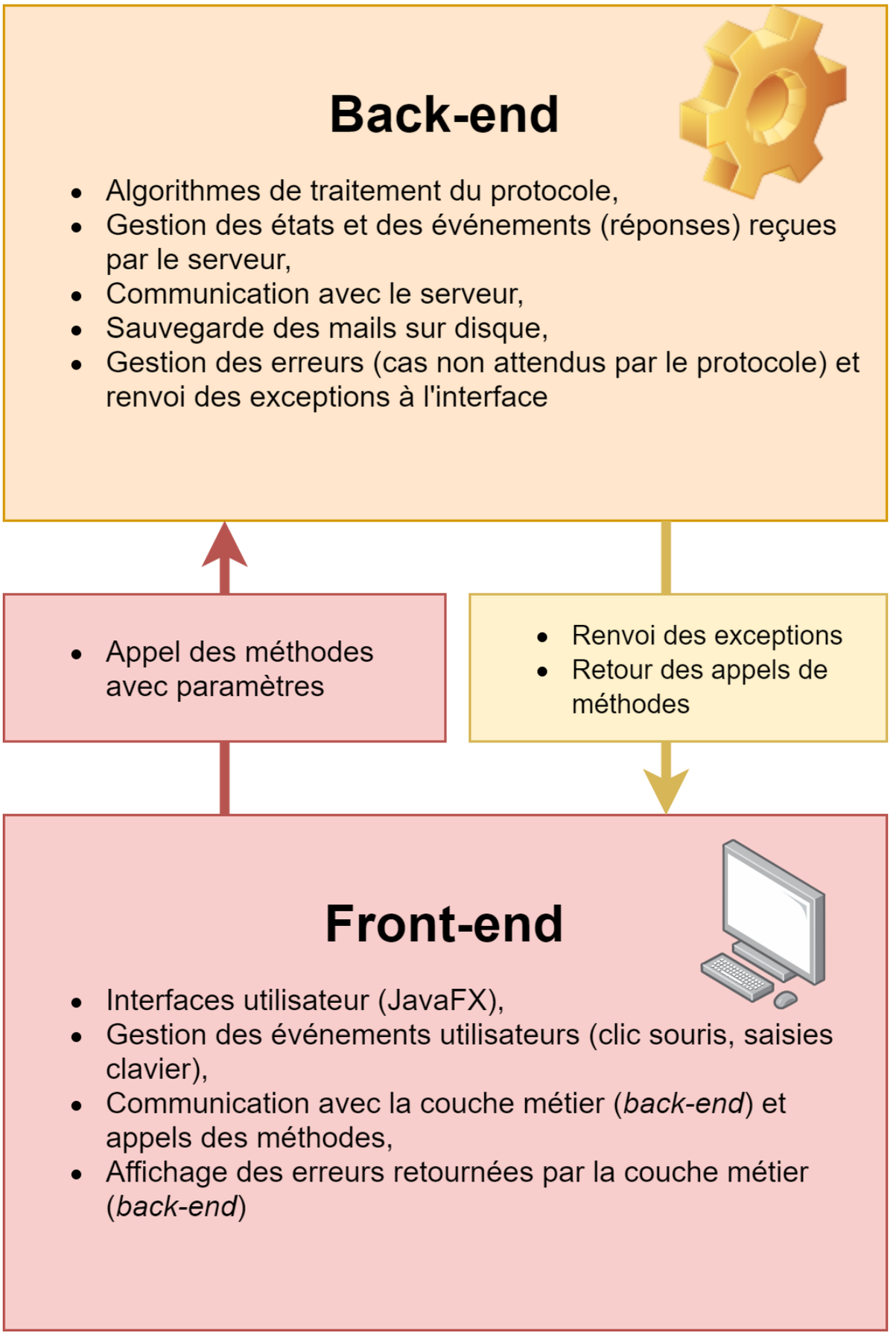


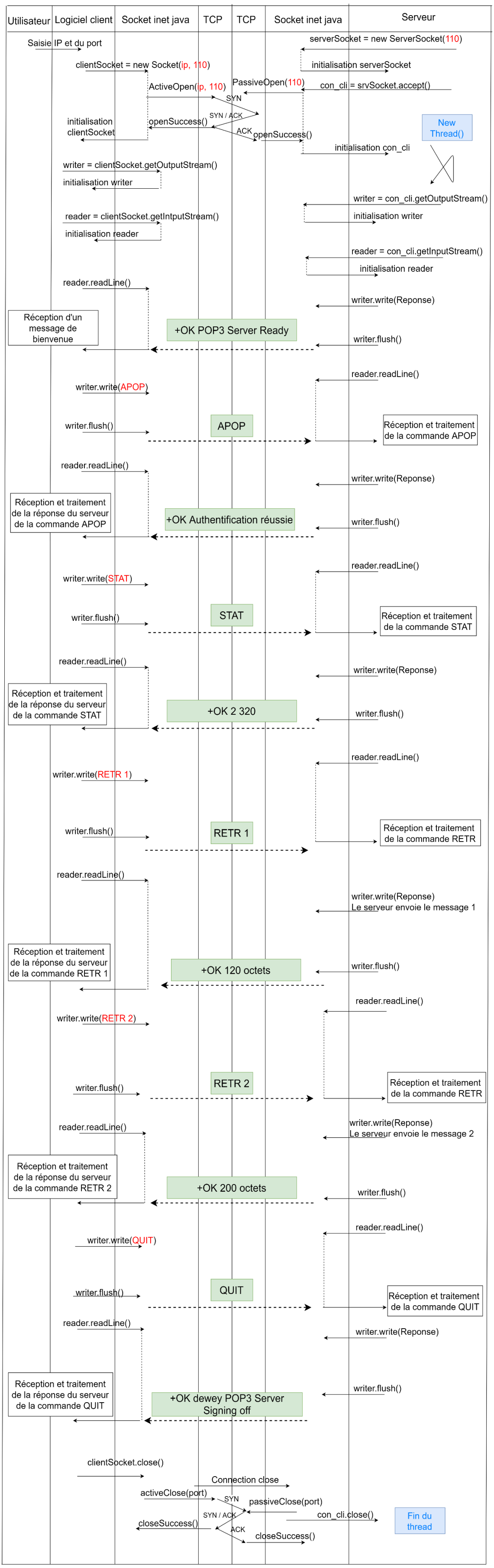
Figure 10 : Schéma représentant la communication entre la couche métier et la couche interface

La couche métier implémente les méthodes suivantes du protocole :

- **APOP**
- **STAT**
- **RETR**
- **QUIT**

Il est tout à fait possible d'ajouter la gestion d'autres méthodes au sein du programme ; nous avons implémenté seulement les méthodes nécessaires au bon fonctionnement du logiciel.

5.3. Scénario



Scénario de la récupération de 2 messages (un de 120 octets et un de 200 octets) par le client chez le serveur sans perte d'informations

6. CONCLUSION

Nous avons donc réalisé une application client/serveur POP3 “Post Office Protocol3” en Java, en respectant la norme RFC 1939.

Dans un premier temps nous avons réalisé par binôme :

- Un logiciel serveur concurrent qui permet de gérer les connexions de plusieurs clients simultanées ainsi que le stockage du contenu de différentes boites de réception.
- Un logiciel client qui s’authentifie au serveur et est capable de récupérer le contenu de sa boite mail.

Dans un second temps, nous avons été amenés à sécuriser notre application :

- Nous avons utilisé les SSL sockets proposés par JAVA afin d’établir une connexion sécurisée entre le serveur et ses différents clients.
- Nous sommes allés jusqu’à mettre en place un certificat autosigné qui nous a permis d’utiliser des ciphersuites plus sécurisées que les ciphersuites anonymes.

Ce projet fut très intéressant à réaliser en raison de la diversité des connaissances qu’il nous a apportées. D’une part, nous avons appris à réaliser un logiciel conforme à une analyse sous la forme d’automate d’états. D’autre part, la réalisation de ce projet nous a permis de nous familiariser avec l’un des deux protocoles de courrier électronique les plus utilisés, le protocole POP3, puis avec sa version sécurisée, POP3S.