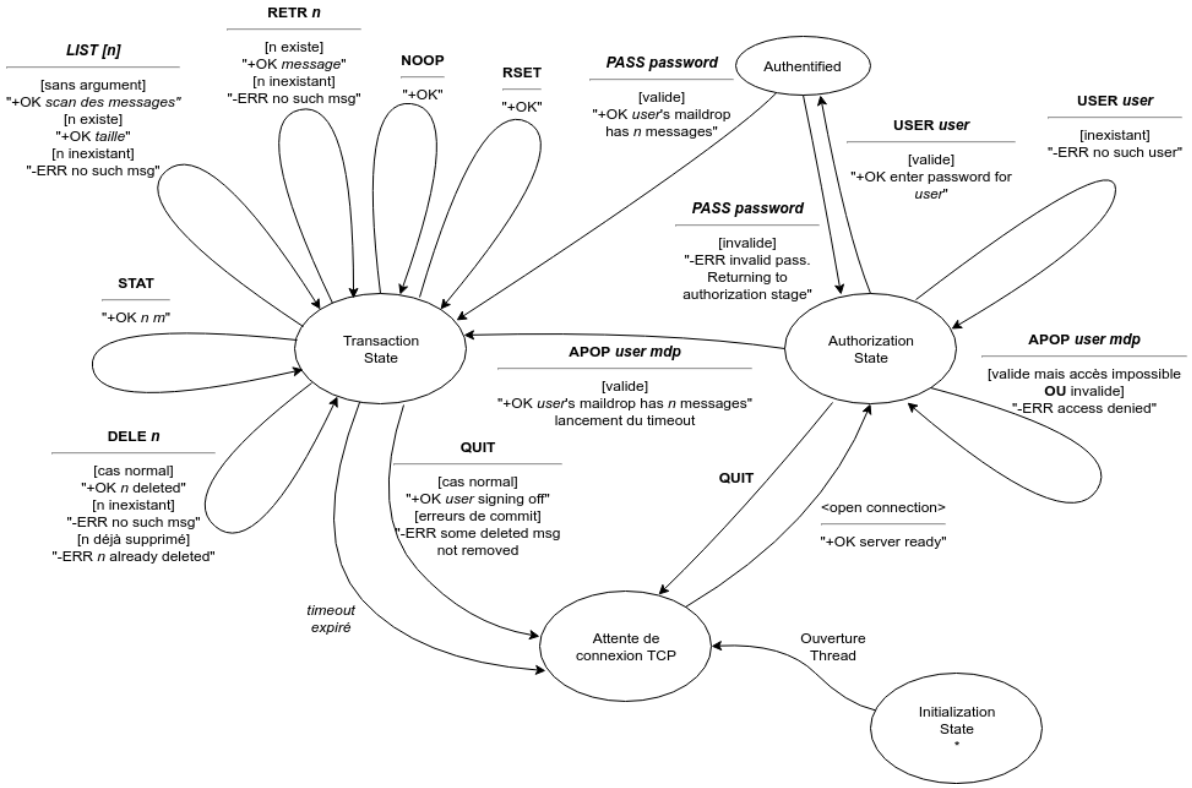
Dossier de programmation - POP3

SUBLET Gary p1506450, VILLERMET Quentin p1507338

Contents

1 Graphe des états (serveur)



2 Table de transitions

Etat	Initialisation	WAITING	AUTHORIZATION	TRANSACTION	UPDATE
Lance le serveur	Ouvre un SocketServer, **WAITING**	"-ERR command unavailable"			
Ouvre communication		"+OK server ready <cypher>", **AUTHORIZATION**			
APOP <user> <pass>		"-ERR command unavailable"	réussi : "+OK maildrop has n messages", **TRANSACTION** sinon : "-ERR permission denied"	"-ERR command unavailable"	"-ERR command unavailable"
STAT		"-ERR command unavailable"	"-ERR command unavailable"	"+OK n m"	"-ERR command unavailable"
RETR n		"-ERR command unavailable"	"-ERR command unavailable"	message n existe : "+OK m octets", "<message>" sinon : "-ERR no such message"	"-ERR command unavailable"
LIST		"-ERR command unavailable"	"-ERR command unavailable"	"+OK n messages, (m octets)", "n m", …	"-ERR command unavailable"
LIST n		"-ERR command unavailable"	"-ERR command unavailable"	message n existe : "+OK n m" sinon : "-ERR no such message, mailbox has n messages"	"-ERR command unavailable"
NOOP		"-ERR command unavailable"	"-ERR command unavailable"	"+OK"	"-ERR command unavailable"
RSET		"-ERR command unavailable"	"-ERR command unavailable"	"+OK mailbox has n messages (m octets)"	"-ERR command unavailable"
DELE n		"-ERR command unavailable"	"-ERR command unavailable"	réussi : "+OK message n deleted" message n déjà supprime : "-ERR message n already deleted" aucun message n : "-ERR no such message"	"-ERR command unavailable"
QUIT		"-ERR command unavailable"	"+OK <name> POP3 server signing off", **WAITING**	**UPDATE**	"-ERR command unavailable"
		"-ERR command unavailable"	"-ERR command unavailable"	"-ERR command unavailable"	aucun message marqués à supprimer : "+OK <name> POP3 server signing off (mailbox empty)", **WAITING** sinon : "-ERR some deleted messages not removed"
Timeout expired				close socket, **WAITING**	

Les réponses envoyées par le client sont de la forme `COMMANDE <argument optionnel>`. Les réponses sont quant à elles de la forme `"<réussite de l'opération> <information>"`. Les changements d'états sont illustrés par le nouvel état écrit en **gras**.

3 Notice explicative

L'implémentation du serveur/client POP3 a été réalisée à l'aide de la JDK 10 et 11. Il est donc préférable d'avoir une version ≥ 10 pour être sûr de pouvoir utiliser correctement toutes les fonctionnalités du programme. L'ensemble des fichiers exécutables (`.jar`) se trouve dans le dossier `bin/` du projet. Ceux-ci sont directement archivés avec leurs dépendances nécessaires.

3.1 Serveur

Pour lancer le serveur il suffit de se mettre dans le dossier `bin/` du projet et de lancer :

```
java -jar pop3.jar <portnumber>
```

Où `<portnumber>` correspond au port sur lequel on désire faire tourner le serveur. Dans le cas du POP3, ce port est le numéro `110`. Cependant pour des raisons de sécurité, les systèmes de type UNIX demandent des droits supplémentaires pour faire tourner des applications sur les ports en dessous de `1024`.

Pour palier à cela, il faut lancer le programme en lui donnant des accès supplémentaires :

```
sudo java -jar pop3.jar 110
```

Et ainsi le serveur se lancera sur le port `110`.

L'ensemble des éléments envoyés et reçus par le serveur seront écris dans la console comme suit :

```
[  1] : --> APOP dev 78773fa7ce5a595e97a4465d0ce737ab
[  1] : <-- +OK maildrop has 1 message (369 octets)
```

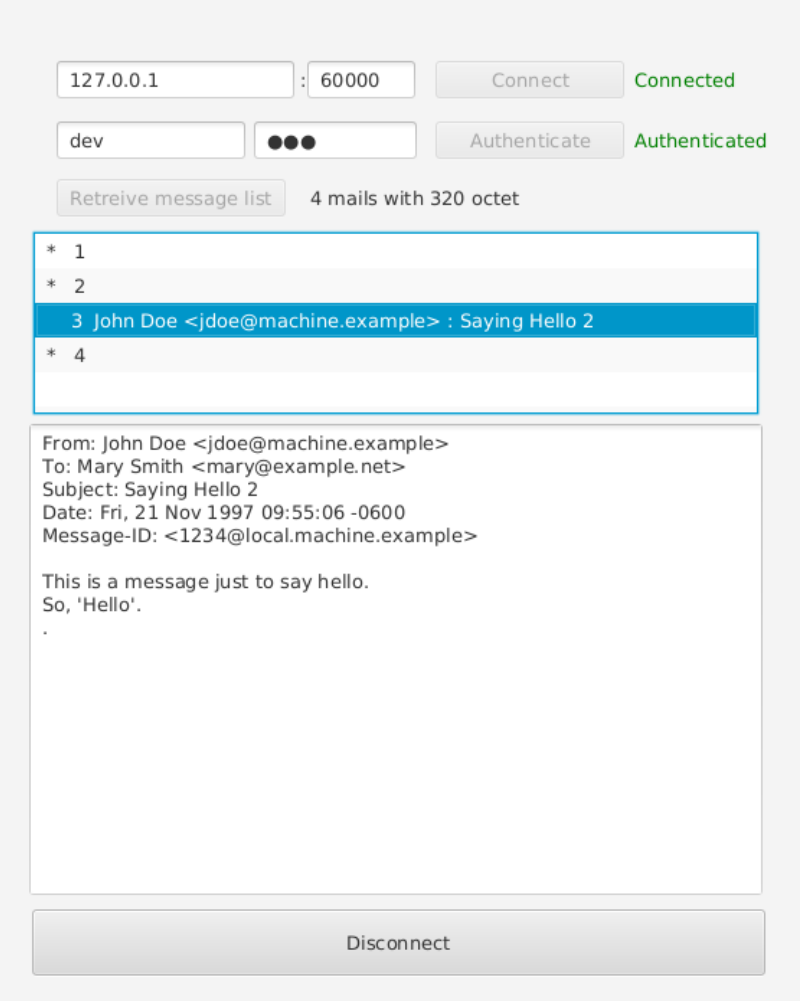
Ici, le `[ 1]` correspond au n-ème client qui se connecte. La flèche allant vers la droite indique une commande a été reçue, et celle qui va vers la gauche indique un message envoyé.

3.2 Client

3.2.1 Interface graphique

L'un des exécutables pour le client dispose d'une interface graphique qui prend en charge l'ensemble des commandes demandées.

Celle-ci a été développée à l'aide de la bibliothèque JavaFX.



Pour la lancer il suffit de lancer cette commande :

```
java -jar JavaFXApp.jar
```

Grâce à cette interface, l'utilisateur peut ainsi sélectionner manuellement toutes les informations demandées pour se connecter et s'authentifier au serveur. Il peut ensuite, en cliquant sur les mails, les télécharger directement et les afficher dans l'encadré du dessous.

Une fois sa session terminée, il peut se déconnecter du serveur grâce au bouton `Disconnect`, et se reconnecter à un serveur (potentiellement différent) à sa guise.

Cette interface prend aussi en charge l'ensemble des erreurs et des exceptions qui sont levées, soit par le serveur, soit par le client lui-même.

3.2.2 Interface minimaliste

Le client possède aussi une interface minimaliste qui s'exécute comme un script. Dans le dossier `bin/`, il suffit de lancer :

```
java -jar client.jar <adresse> <port> <username> <password>
```

En remplaçant bien sûr les différents paramètres :

```
java -jar client.jar 127.0.0.1 110 dev dev
```

Le programme va alors se connecter et s'authentifier au serveur. Il va ensuite télécharger l'ensemble des mails disponibles pour l'utilisateur.

3.2.3 Sauvegarde vers un fichier

Les deux programmes précédents sauvent automatiquement les mails téléchargés dans le fichier `email.json` (qui est automatiquement créé si il n'existe pas) dans le dossier où se trouve l'exécutable.

Les mails enregistrés est de la forme suivante :

```
{
    "header": {
        "From": "John Doe \u003cjdoe@machine.example\u003e",
        "To": "Mary Smith \u003cmary@example.net\u003e",
        "Subject": "Saying Hello 1",
        "Date": "Fri, 21 Nov 1997 09:55:06 -0600",
        "Message-ID": "\u003c1234@local.machine.example\u003e"
    },
    "content": "This is a message just to say hello.\r\nSo, \u0027Hello\u0027.",
    "raw": "From: John Doe \u003cjdoe@machine.example\u003e\r\nTo: Mary Smith \u003cmar
    y@example.net\u003e\r\nSubject: Saying Hello 1\r\nDate: Fri, 21 Nov 1997 09:55:06 -0600
    \r\nMessage-ID: \u003c1234@local.machine.example\u003e\r\n\r\nThis is a message just to
    say hello.\r\nSo, \u0027Hello\u0027.\r\n.\r\n"
},
```

Chacun des mails enregistré dans `email.json` comporte une section `header`, qui comprend l'ensemble des headers du mail, chacun analysés pour faire des recherches plus facilement par la suite. On retrouve aussi le corps du mail dans la section `content`. Enfin dans la section `raw`, on retrouve

l'intégralité du mail tel qu'il a été reçu.

3.2.4 Connexion au travers d'une application tierce

Il est aussi possible de se connecter au serveur par le biais d'une application tierce comme `netcat`, `putty` ou encore `telnet`.

Voici un exemple avec le programme `netcat` :

```
nc localhost 110 -4 -C
```

Avec cette commande, netcat va alors se connecter au port `110` sur le serveur hébergé sur `localhost`.

L'argument `-4` permet de dire à netcat de préférer se connecter en utilisant le protocole `IPv4` plutôt que `IPv6` (par défaut) (même si le serveur prend en charge l' `IPv6`);

L'argument `-C` est ici indispensable car il permet de remplacer les simple sauts à la ligne `\n` (lorsqu'on appuie sur `Entrée`) par des `\r\n` (`CRLF`) comme le stipule la RFC1939.

Une fois la connexion établie, il suffit d'écrire à la main les commandes à envoyer.

4 Base de données et pérennisation

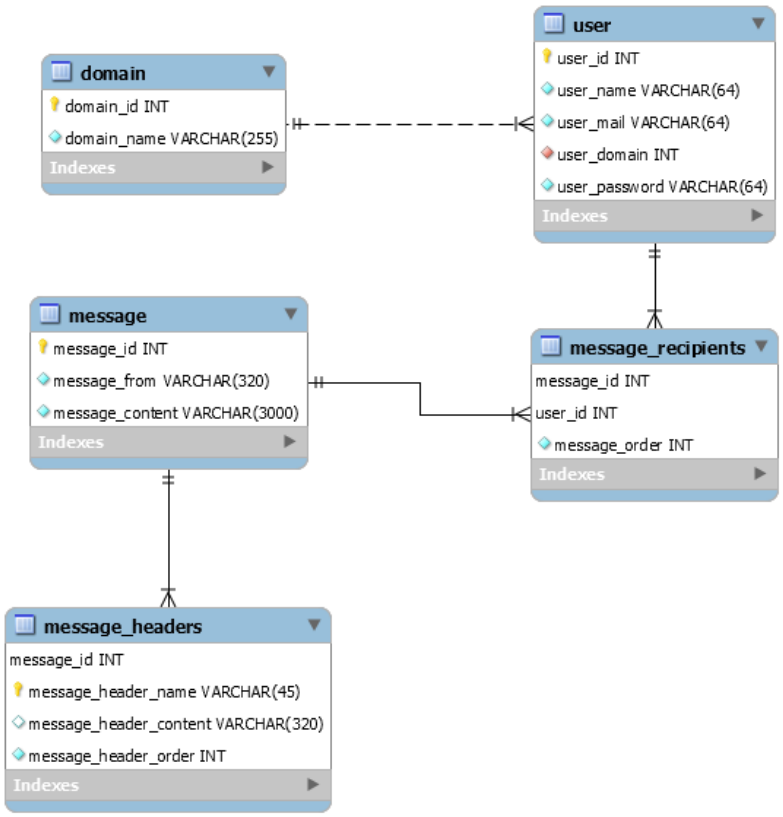


Figure 1: Schéma conceptuel de la BD

En plus de l'enregistrement des données sous format `json`, nous avons fait le choix de pérenniser les données au sein d'une base MySQL. Pour ce faire, une classe côté serveur encapsule les attributs et opérations courantes effectuées sur la BD.

Le schéma conceptuel ci-dessus se suffit à lui-même en terme d'explications, à quelques détails près : par exemple, le choix a été fait de conserver les domaines des e-mails dans une table à part, par optimisation mais aussi pour favoriser l'utilisation de cette même BD pour le serveur SMTP.

On notera aussi que les headers ont eux aussi une table qui leur est réservée, pour éventuellement mener des recherches plus efficaces sur certains critères de mails au sein de la base.

La structure est sinon relativement classique, un message étant lié à un ou plusieurs utilisateurs en réception, chaque utilisateur ayant ses attributs stockés en base. L'ajout de cette BD permet une implémentation plus fidèle à un véritable serveur POP, avec un stockage longue durée de données envisageable.

5 Sécurité

5.1 Authentification

Lors de l'authentifications, le serveur envoie un seed de la forme `<6084.1554706149@localhost>`, auquel le client devrai ajouter son mot de passe et hasher le tout au format `MD5` pour ensuite le renvoyer au serveur.

5.2 SSL

Pour sécuriser l'application, nous avons mis en place le système de chiffrage SSL, sans passer par une authentification par certificat.

Pour y parvenir, nous avons mis en place cette fonction :

```
private String[] getCipherSuite(SSLServerSocket ss) {
    // retourne seulement les cipher qui ne necessitent pas
    // de certificat (ceux qui contiennent ‘anon‘)
    return Arrays
        .stream(ss.getSupportedCipherSuites())
        .filter(c -> c.contains("anon"))
        .toArray(String[]::new);
}
```

qui va filtrer l'ensemble des `cipher suite` disponibles et ainsi n'utiliser que celles qui contiennent un `anon` dedans.

Les sockets utiliseront à présent uniquement le SSL pour communiquer (du côté client et serveur).