Océane Jean-François - Chloé Bardot - S2C

Développement ClientPOP3

24/03/2024

Sommaire

1. Introduction

Le projet vise à développer un client POP3 en langage C# avec une interface graphique. Le squelette de départ du projet contient le fonctionnement de base pour gérer la connexion au socket POP3, il se divise en trois classes : Preferences : une classe utilitaire contenu les identifiants de connexion, Communication : une classe qui implémente la logique qui permet de communiqué avec le socket et pour finir la classe ClientPOP3 qui contient l'interface graphique de l'application.

2. Analyse fine du protocole POP3

- Quelles sont les commandes pour lesquelles la réponse du serveur ne contient qu'une seule ligne +OK ?

  Les commandes pour lesquelles la réponse du serveur ne contient qu'une seule ligne +OK sont USER, PASS, STAT, DELE n, LAST, RSET, NOOP et QUIT.

- Quelles sont les commandes pour lesquelles la réponse du serveur contient plusieurs lignes, une ligne +OK suivie de plusieurs autres ?

  Les commandes pour lesquelles la réponse du serveur contient plusieurs lignes, avec une ligne +OK suivie de plusieurs autres, sont LIST, RETR et TOP n.

3. Programmation en C# d'un client POP3

3.1. Partie 1 : Découverte – Conseils essentiels – Codage d'une 1ère fonctionnalité (LIST)

Pour rendre fonctionnel le bouton de la fonctionnalité "List", nous avons complété la boucle de traitement des lignes de réponse du serveur fournie. Plus précisément, nous avons effectué une transformation de chaque ligne en tableau de chaînes grâce à la fonction Split(), permettant ainsi de récupérer séparément le numéro du mail et sa taille. Cela nous a permis d'afficher ces informations de manière plus lisible pour l'utilisateur grâce à la méthode WriteAffichage. Pour passer à la ligne suivante, nous faisons un nouvel appel à la fonction LireLigne().

3.2. Partie 2 : Fonctionnalités obligatoires

3.2.1. Afficher le n_ème message Afficher uniquement la date, le nom de l'expéditeur et le sujet du n-ème message

Pour pouvoir afficher le message n-ème, nous avons ajouté un contrôle "NumericUpDown" ainsi qu'un bouton sur l'interface graphique. Dans un premier temps, nous avons procédé à l'affichage du message dans son intégralité, tel qu'il est reçu du serveur, sans prendre en compte les aspects liés au protocole POP3. Pour ce faire, nous avons créé une méthode permettant d'exécuter la commande "RETR" sur le serveur (Retr1()), prenant en paramètre le numéro du message que l'on souhaite afficher. Cette méthode est appelée lorsqu'on clique sur le bouton "Afficher message" (dans la méthode AfficherMessage_Click). Nous récupérons ce numéro depuis le contrôle NumericUpDown en créant une propriété appelée "NumeroMail" qui retourne la valeur de ce NumericUpDown. En ce qui concerne le contenu de la méthode

Retr1, tout comme pour les méthodes List et Stat, nous commençons par envoyer la commande "RETR" suivie du numéro du message choisi au serveur afin de récupérer le message correspondant. Nous vérifions ensuite que le serveur a bien reçu la commande en vérifiant s'il renvoie bien "+OK". Si tel est le cas, nous affichons toutes les lignes du message à l'aide d'une boucle "tant que" jusqu'à ce que nous rencontrions une ligne où le seul caractère est un point, ce qui signifie que nous avons atteint la fin du message. Nous ajoutons également une condition pour vérifier que la ligne n'est pas vide afin d'éviter de bloquer le programme si le message comporte des sauts de ligne. Enfin, nous affichons chaque ligne grâce à la méthode WriteAffichage de la classe ClientPOP3.

Cependant, cette méthode ne nous permet pas de supprimer les points associés au protocole POP3 dans la fenêtre d'affichage pour l'utilisateur. Pour réaliser cela, il a été nécessaire de repérer les lignes commençant par un point et de retirer ce point superflu à l'aide de la méthode Remove().

3.2.2. Afficher uniquement la date, le nom de l'expéditeur et le sujet du n-ème message

Pour afficher uniquement la date, le nom de l'expéditeur et le sujet du n-ème message, nous avons créé une nouvelle méthode similaire à la précédente, nommée "retr2". Dans cette méthode, enlever les points éventuels est inutile car nous n'affichons pas le message en entier. Dans le but de n'afficher que certaines informations, comme la date par exemple, nous avons dû transformer chaque ligne en tableau de chaînes afin de pouvoir récupérer la ligne qui nous intéresse (dans notre exemple, la ligne commençant par "Date"). Ensuite, après avoir récupéré cette ligne, nous avons concaténé le reste du tableau de chaînes à l'aide d'une méthode appelée "Concatener()" afin d'éviter la répétition de code. Nous avons récupéré toutes les lignes qui nous intéressent grâce à une structure de contrôle "switch case". Après avoir parcouru toutes les lignes, nous affichons celles qui nous intéressent du côté du client grâce à la méthode "WriteAffichage()".

3.2.3. Afficher la date, le nom de l'expéditeur et le sujet de tous les messages.

Dans le but d'afficher la date, le nom de l'expéditeur et le sujet de tous les messages, nous avons appelé la méthode "Retr2" précédente pour tous les messages. Pour pouvoir itérer sur tous ces messages, il a été nécessaire de récupérer le nombre de messages. Pour cela, nous avons envoyé la commande "STAT" au serveur et avons transformé la réponse en tableau de chaînes afin de récupérer uniquement le nombre de messages, nécessaire pour notre boucle.

3.2.4. Améliorer l'affichage du n-ème message.

Afin de simplifier l'affichage du message et de structurer les informations de l'en-tête, nous avons créé la méthode "retr3", qui fonctionne de manière similaire à "retr2". Cependant, pour répondre au besoin de clarifier l'en-tête du mail, nous avons décidé de simplifier les informations telles que la date, de supprimer le numéro du message et d'ajouter le contenu du mail. Tout d'abord, pour la date, nous récupérons la date de la même manière que dans la méthode "retr2". Cependant, nous utilisons une méthode fournie par C# pour transformer la date en un format plus court et plus lisible. Pour cela, nous utilisons la classe "DateTime" qui contient la méthode "Parse()", permettant d'appliquer des transformations sur une date au

format de chaîne de caractères et de créer une instance de la classe "DateTime" à partir de la chaîne de caractères, ce qui nous permet d'afficher la date dans le format souhaité, tel que 'JJ/MM/AAAA'. Dans le but de récupérer uniquement le contenu d'un message, nous avons créé la méthode "retrMessage", qui récupère le contenu d'un mail en fonction du numéro passé en paramètre. Cette méthode est utile dans l'affichage du contenu prévu par la méthode "retr3", mais elle peut également être utile dans d'autres méthodes. Il est donc judicieux de la séparer de la méthode "retr3".

La méthode "retrMessage", comme mentionné précédemment, permet de fournir le contenu du mail en fonction du numéro passé en paramètre. Son fonctionnement est légèrement plus complexe que les autres méthodes car elle doit prendre en compte le type de contenu renvoyé par le serveur, que ce soit du "text/plain" (texte simple) ou du "text/html" (HTML), ainsi que du "multipart" qui regroupe plusieurs formats de données, par exemple "text/plain" suivi de "text/html". Pour retourner le contenu multiple, il est nécessaire de récupérer le séparateur (boundary) fourni dans l'en-tête. Nous le récupérons en utilisant son indice (ligne.IndexOf("boundary=")), puis nous ne récupérons que la partie utilisable de la chaîne de caractères et retirons les caractères superflus à l'aide des méthodes "substring" et "trim". Une fois le séparateur obtenu, il suffit de commencer la concaténation du message dès que l'on rencontre le séparateur. Dans le cas où le contenu est simple, la méthode commence la concaténation du contenu à partir de la détection de "Content-Transfer-Encoding:" et "X-AV-Checked" afin de ne pas les inclure dans le contenu. Une fois le contenu récupéré, la méthode le retourne.

4. Extensions

Dans le but d'améliorer l'application, nous allons implémenter plusieurs extensions. Tout d'abord, nous ajouterons une bannière de connexion permettant de saisir les informations de connexion et de choisir le serveur auquel on souhaite se connecter. Ensuite, nous mettrons en place un système de déconnexion/reconnexion en créant un bouton dédié. Nous clarifierons également l'affichage des messages en simplifiant leur présentation. De plus, nous permettrons l'affichage du contenu d'un message lorsque l'utilisateur clique sur ce dernier. Enfin, nous créerons un bouton permettant la suppression des messages dans la listBox afin de nettoyer l'interface et d'éviter de devoir faire défiler sur une longue durée pour trouver l'information recherchée.

4.1. Ajouter une bannière de connexion

Pour ajouter la bannière de connexion, nous avons créé une fenêtre de dialogue qui s'ouvre lorsque l'application s'initialise. Ainsi, le code permettant de lancer et de récupérer l'exécution du dialogue se trouve dans la classe "Communication", dans la méthode "Initialize". Afin de prévenir d'éventuelles saisies de données invalides, nous avons implémenté un bloc "try and catch" qui, lors de la demande de connexion via la méthode "connect()" du socket, permet, en cas d'exception, d'afficher une MessageBox donnant des informations sur l'erreur survenue.

La classe "Bannière", qui gère l'interface de la fenêtre de dialogue de connexion, est composée de trois propriétés retournant le contenu des trois textBox de la fenêtre de dialogue qui permettent de gérer un champ de connexion au socket. Nous y retrouvons l'adresse du serveur, nommée "textMachine", l'identifiant de l'utilisateur, nommé "textIdentifiant", et le mot de passe de l'utilisateur, nommé "textMdp ". La méthode "textOnChange" permet de prendre en compte les

champs de saisie de la fenêtre de dialogue et de vérifier s'ils ne sont pas vides. Si l'un ou plusieurs d'entre eux sont vides, alors nous désactivons le bouton permettant de valider l'envoi des données.

4.2. Ajouter la déconnexion / reconnexion :

Nous allons maintenant mettre en place le bouton de déconnexion/reconnexion. Tout d'abord, nous ajoutons un bouton à l'interface du client, puis nous créons la méthode qui réagira aux clics sur ce bouton, nommée "buttonDeco_Click". Cette méthode utilise la méthode "Quit()" de la classe "Communication" et rappelle également la méthode "initialize", également présente dans la classe "Communication". Une fois cela fait, l'utilisateur a la possibilité de se déconnecter puis de se reconnecter via la bannière de connexion qui se réouvre suite à l'appel de la méthode "initialize". Cependant, un problème subsiste : lorsque l'on ferme la bannière en cliquant sur la croix, cela fait planter l'application. Cela provient du fait que même si on ferme la bannière, la méthode "initialize" continue à être exécutée provoquant ainsi une erreur dû à l'utilisation de méthodes liés au socket. Pour remédier à cela, nous allons créer une propriété dans le client, nommée "Connecter", qui nous permettra de savoir si la connexion au socket est active ou non. Nous modifions légèrement la méthode "initialize" pour qu'elle retourne un booléen qui sera récupéré pour connaître l'état de la connexion au socket. Une fois cela fait, nous conditionnons l'utilisation des méthodes liées au socket grâce à la propriété "Connecter", permettant ainsi d'éviter les erreurs. Nous avons également veillé à désactiver les boutons du client, sauf celui de reconnexion et celui pour quitter l'application, lorsque la personne se déconnecte mais ne se reconnecte pas. De plus, nous avons fait varier le texte du bouton en fonction de l'état de connexion du client. L'activation et la désactivation des boutons du client ont été centralisées dans la méthode "GestionBoutons", prenant un booléen en paramètre et le transmettant aux différents boutons via la méthode "Enable" de la classe "Button".

4.3. Améliorer globalement l'affichage des messages

Afin de rendre l'affichage des messages plus clair, nous avons créé une autre méthode dans la classe "Communication" nommée "retrSimple". Cette méthode permet d'afficher sur une seule ligne les informations essentielles liées à un message. Son fonctionnement est similaire aux autres méthodes "retr", la différence réside dans la forme de l'affichage. Pour accéder au contenu du message, nous avons implémenté l'événement "listBoxAffichage_SelectedIndexChanged", qui permet d'obtenir un retour d'information lors du changement de sélection sur une listBox. Ce changement nous permet de récupérer la listBox en castant l'objet sender en listBox (ListBox listBox = (ListBox)sender ;). Ensuite, nous pouvons récupérer la propriété "SelectedItem" contenue dans la classe "listBox". Dans le cas où "SelectedItem" contient le numéro d'un message (SelectedItem.Contains("Num:")), cela signifie que "SelectedItem" est un message simple. Nous pouvons donc, grâce à la fonction "Split(' ')", récupérer le numéro du message, puis extraire le contenu du message via la méthode "retrMessage" précédemment codée. Une fois le contenu récupéré, nous le tronquons à 300 caractères pour éviter qu'il ne soit trop long dans l'affichage de la MessageBox.

4.4. Nettoyer l'affichage

Comme dernière extension, nous avons ajouté la possibilité de nettoyer l'affichage de la listBox, car lorsque de nombreux messages s'accumulent, il devient difficile d'utiliser l'application. Pour ce faire, nous avons ajouté un bouton dans la classe Client, puis nous avons ajouté la méthode liée à

l'événement de clic sur ce bouton. Nous utilisons ensuite la méthode "Clear()", accessible depuis la liste des éléments de la listBox (listBoxAffichage.Items.Clear()).

5. Difficulté rencontrée lors de la réalisation du projet

Nous allons maintenant aborder les difficultés que nous avons rencontrées lors de ce projet. Tout d'abord, il y a eu l'apprentissage de l'utilisation du protocole POP3, suivi de la récupération des informations sous forme de texte brut, puis la gestion de l'état de connexion lié au socket, et enfin la conversion de la date du format String au format DateTime.

Pour prendre en main l'utilisation du protocole POP3, nous avons dû tester les différentes méthodes disponibles afin de comprendre leur fonctionnement.

Lors du développement des divers besoins de l'application, nous avons été confrontés au problème de récupération des données dans le texte brut renvoyé par le serveur. Pour résoudre ce problème, nous avons utilisé les méthodes proposées par C# pour extraire du texte à partir d'une chaîne de caractères, ce qui nous a permis de trier les informations souhaitées.

En mettant en place la déconnexion/reconnexion de l'application, nous avons rencontré des problèmes d'utilisation de certaines méthodes faisant appel au socket, notamment les méthodes "LireLigne" et "EcrireLigne" qui interagissent avec le serveur. Cependant, si le serveur n'est pas accessible en raison d'une rupture de connexion, nous avons dû réfléchir à un système permettant de préserver l'application contre d'éventuelles coupures de connexion, comme une déconnexion. Pour cela, nous avons mis en place une propriété qui renvoie un booléen reflétant l'état de la connexion. Ainsi, nous avons pu conditionner l'utilisation des méthodes posant problème et limiter leur utilisation uniquement lorsque l'état de connexion est actif (Connecter = true).

Enfin, lors de l'affichage des données, nous avons voulu simplifier l'affichage de la date en utilisant la classe DateTime native à C# pour convertir la date donnée dans le texte brut en une date plus concise. Cependant, pour certains formats de date rencontrés, nous avons rencontré une erreur de conversion qui nous a contraints à conditionner l'utilisation de la conversion de la date.

6. Conclusion

En conclusion, ce projet nous a permis de mettre en œuvre la création d'un client de messagerie POP3 via une interface graphique, de comprendre le fonctionnement du protocole POP3 par la pratique, et de découvrir le fonctionnement des sockets en C#.