MailServer - Protokoll

1. Kurzbeschreibung

Die Kommunikation zwischen Mailserver und Mailclient findet über Sockets statt. Dabei wird jeweils von Server und Client abwechselnd gesendet und empfangen, um eine synchronisierte Kommunikation zu gewährleisten.

Der Server ist zudem Multiuser-Fähig. Es können sich also auch mehrere Clients auf den Server verbinden und gleichzeitig Operationen durchführen können.

2. Verfügbare Operationen

Der Austausch von Informationen basiert auf mehreren verfügbaren Operationen. Diese Operationen beinhalten folgende Befehle:

- **Login:** führt eine Benutzerauthentifizierung mit Username und Passwort gegen den LDAP Server der FH Technikum Wien durch und speichert bei erfolgreichem Login den Benutzernamen für weitere Operationen
- **Send:** sendet eine Nachricht mit Empfänger, Betreff, Inhalt und optional einer Datei als Mail-Anhang an den Server, welcher diese in einem personenbezogenen Ordner im Mailspool-Verzeichnis abspeichert
- **List:** fordert vom Server eine Liste an Nachrichten (nur Betreff-Zeile) für den eigenen Benutzer an und zeigt diese beim Client an
- **Read:** fordert eine bestimmte Nachricht mit Sender, Empfänger, Betreff und Inhalt vom Server an, zeigt diese beim Client an und lädt falls vorhanden und vom Client erwünscht den Datei-Anhang dieser Nachricht vom Server herunter um sie beim Client lokal abzuspeichern
- **Del:** löscht eine bestimmte Nachricht aus dem Mailspool-Verzeichnis beim Server
- **Quit:** beendet die Kommunikation zwischen Client und Server, schließt den Socket und beendet das Client-Programm

Um eine dieser Operationen (mit Ausnahme von Login) ausführen zu können muss zuerst die Operation „Login“ erfolgreich durchgeführt werden.

3. Ablauf einer Operation

Der grundlegende Ablauf einer dieser Operationen ist wie folgt:

1. Der Client gibt einen der obigen Befehle in der Konsole ein
2. Der Befehl wird an den Server gesendet
3. Der Server empfängt diesen Befehl und sucht die dazugehörige Operation
4. Der Server fordert alle benötigten zusätzlichen Informationen für den jeweiligen Befehl vom Client an
    a. Der Server sendet pro benötigter Information eine Aufforderung an den Client, diese einzugeben
    b. Der Client gibt die geforderte Information ein und sendet diese zurück an den Server
    c. Der Server empfängt die Information vom Client, überprüft sie auf Korrektheit und speichert sie lokal zur späteren Verwendung ab
5. Sobald alle benötigten Informationen für den jeweiligen Befehl erhalten, überprüft und abgespeichert wurden, führt der Server die jeweilige Operation durch
6. Der Server sendet eine Status-Nachricht an den Client, um diesen darüber zu informieren, ob die Operation erfolgreich war oder fehlgeschlagen ist
7. Der Client empfängt die Status-Nachricht vom Server und zeigt diese an

4. Blacklist bzw. blockierte IP-Adressen

Am Server existiert eine Blacklist, die IP-Adressen beinhaltet, die sich vorübergehend nicht auf den Server verbinden dürfen. Sobald ein Client drei Mal hintereinander die „Login“ Operation durchführt und dabei nicht erfolgreich ist, wird aus Sicherheitsgründen die IP-Adresse dieses Clients für eine gewisse Zeit gesperrt. In diesem Fall wird die entsprechende IP-Adresse, sowie die Zeit, ab der die Sperre wieder aufgehoben wird, in einer unordered_map in der Server-Klasse abgespeichert. Außerdem werden diese Informationen in eine Datei „blacklist.txt“ geschrieben, die sich am Server im Mailspool-Verzeichnis befindet. In dieser Datei wird pro gesperrter IP-Adresse eine Zeile in folgendem Format geschrieben: [IP_ADDRESS] [TIMEVALUE]

Um sicherzugehen, dass aktuell gesperrte Clients den Server nicht nutzen können, wird direkt nach dem Verbinden eines neuen Clients überprüft, ob sich dieser in der Map der gesperrten Clients befindet und ob in diesem Fall die Zeit, zu der die Sperre wieder deaktiviert wird, noch nicht erreicht wurde. Sollte dies der Fall sein, wird die Verbindung mit dem Client geschlossen. Sollte sich der Client in der Map befinden, die Zeit allerdings schon abgelaufen sein, wird der Client aus der Map entfernt und die Verbindung bleibt aufrecht, sodass der Client nun wieder Operationen am Server ausführen kann.

5. Multiuser-Fähigkeit

Die Multiuser-Fähigkeit wurde mithilfe von Threads implementiert. Dabei wird beim Server jedes Mal, sobald sich ein Client verbindet, ein neuer Thread erstellt, in dem die Kommunikation mit dem jeweiligen Client abgearbeitet wird. Alle angeforderten Operationen des Clients werden in diesem Thread durchgeführt.

Sobald der Client die „Quit“ Operation durchführt, wird die Verbindung auf beiden Seiten geschlossen und der Thread endet. Sollte der Client das Programm schließen, ohne die dafür erstellte „Quit“ Operationen zu verwenden, wird ebenfalls die Verbindung serverseitig geschlossen und der Thread beendet, sobald vom Server bemerkt wird, dass der Socket des Clients nicht mehr verfügbar ist.

Da im Programm mit mehreren Threads gearbeitet wird, wird bei bestimmten Operationen in der Server-Klasse ein Mutex verwendet, um die Ausführung von einzelnen Code-Teilen zu schützen. Diese beinhalten das Schreiben und Lesen in und aus der Map, die die blockierten Benutzer beinhaltet, sowie das Schreiben und Lesen in und aus dem File, in dem die blockierten Benutzer abgespeichert sind. Für Map und File wurde dementsprechend jeweils ein Mutex erstellt, die bei den entsprechenden Operationen gesperrt bzw. anschließend entsperrt werden.

Außerdem musste auf eine eindeutige Benennung der Mail-Dateien am Server geachtet werden, sodass es nicht zu Problemen bei der Erstellung der Dateien kommt, sollten mehrere Clients gleichzeitig eine „Send“ Operation mit demselben Empfänger durchführen. Dies wurde gelöst, indem die Mail-Dateien immer nach folgendem Schema benannt werden:
[TIMECODE_OF_CREATION]_[ID_OF_THREAD]
Da hier sowohl das Erstellungsdatum als auch die ID des Threads, der den Befehl durchführt, verwendet werden, kann es zu keinen Konflikten mit gleichzeitigen Operationsdurchführungen von mehreren Clients kommen.

6. Socket Implementierung

Das Erstellen eines neuen Sockets am **Server** wurde folgendermaßen umgesetzt:

```
int create_socket = socket (AF_INET, SOCK_STREAM, 0);
```

Erstellt einen neuen Socket mit IPv4 Adressen und TCP Protokoll. Dabei wird ein Integer-Wert zurückgegeben, der als Socket-Descriptor dient.

```
memset(&address, 0, sizeof(address));
address.sin_family = AF_INET;
address.sin_addr.s_addr = INADDR_ANY;
address.sin_port = htons (port);
if (bind (create_socket, (struct sockaddr *) &address, sizeof (address)) != 0) {
    perror("bind error");
    return EXIT_FAILURE;
}
```

Hier wird zunächst ein struct mit Informationen befüllt (IPv4 Adressen, alle erlauben und Port für Server-Socket festlegen).
Anschließend wird bind aufgerufen, um den vorher erstellten Socket nun auch an einen Port zu binden, um ihn später verwenden zu können.

```
listen (create_socket, 5);
```

Hier wird nun definiert, dass der Socket maximal 5 gleichzeitige Requests einreihen soll.

```
new_socket = accept (create_socket, (struct sockaddr *) &cliaddress, &addrlen );
```

Anschließend kann über einen accept-call solange gewartet werden, bis ein Client versucht sich zum Server zu verbinden. In diesem Fall wird ein neuer Socket-Descriptor für die weitere Kommunikation mit diesem Client zurückgegeben.

```
close (create_socket);
```

Am Ende des Programms wird mit einem Aufruf von close der Server-Socket geschlossen.

Am **Client** wird ebenfalls mit einem Aufruf zu socket ein neuer Socket erstellt. Anschließend wird allerdings nicht bind aufgerufen, sondern connect (mit Server-IP und Server-Port als Parameter), um sich mit einem Remote-Server zu verbinden.

```
connect (create_socket, (struct sockaddr *) &address, sizeof (address))
```

Sobald connect aufgerufen wurde, kann nun über den Socket mit dem Server kommuniziert werden.