Chat Application

DIALLO Abdoul Aziz

06/10/2023

Chat Application

1. Introduction

Cette suite du projet a été axée sur l'intégration de fonctionnalités majeures, comprenant :

1. Protocole en mode binaire : l'application de messagerie a été dotée d'un protocole de communication utilisant un format binaire, favorisant des échanges de données plus fiables, moins volumineuses et plus efficaces.
2. Transfert de fichier : Une fonctionnalité de transfert de fichier a été implémentée, permettant à deux utilisateurs (server et client) de s'échanger des fichiers (binaires) de différentes tailles et types.
3. Une gestion évoluée des utilisateurs est mise en place, où un serveur reste en attente de l'arrivée des N utilisateurs pour participer à la discussion. Cette gestion garantit une prise en charge efficace lorsqu'un client se déconnecte de la conversation, tout en incluant d'autres fonctionnalités visant à améliorer l'expérience globale des utilisateurs.

2. Machine à états finale.

3. Protocole en mode binaire

Dans ce protocole binaire, j'ai définis 2 type de commande: la commande QUIT et la commande HELO et chacune d'elle est codé sur 1 octet.

Ainsi, pour s'identifier au serveur, si BIN est défini, je déclare un buffer de 1 octet de type `uint8_t` et je le remplis avec la commande correspondante. Et j'utilise un pointeur générique pour passer ce buffer à la fonction `sendto()`, j'ai fait cela afin de pouvoir factoriser le code c'est-à-dire éviter d'écrire `sendto()` deux fois.

```
#ifdef BIN
            uint8_t binBuff[1] = {HELO};
            const void *buff = binBuff;
            size = 1; // 1 octet.
```

```
#else
            const char *messageBuff = "/HELO";
            const void *buff = messageBuff;
            size = 5; // 5 caractères
#endif
```

Vous remarquerez également plus tard que j'ai utilisé la même technique tout juste avant que le serveur reçoit le message d'identification du client. Je prépare un buffer et j'utilise un pointeur générique à ce buffer et selon que le protocole soit binaire ou non, je le remplis en utilisant qu'une seule fois la primitive `recvfrom()` ceci afin d'éviter aussi d'écrire deux fois la primitive `recvfrom()`.

Enfin, lorsque la salle de discussion est créée, il faut et il suffit qu'une instance entre au clavier /QUIT qui a une taille de 5 octets et après j'utilise seulement la première case du tableau pour y mettre la commande QUIT et je rajoute le caractère de fin de chaîne `'\0'` afin que la fonction `strlen()` retourne 1 qui est la nouvelle taille du buffer et je l'envoie ensuite au serveur.

4. Protocole de transfert de fichier fiable en utilisant UDP.

En effet, deux commandes sont indispensables pour le bon fonctionnement de cette fonctionnalité, à savoir :

1. La commande `/GET [file name]`

   Ainsi, pour que le transfert de fichier puisse fonctionner correctement il faudrait qu'un client demande via `sendto()` au serveur le fichier qu'il souhaiterait obtenir en entrant la commande suivante `/GET [file name]`. Et par la suite, je vérifie que le fichier demandé existe bien dans le serveur, si oui alors j'envoie au client le nombre de frame qu'il doit recevoir et j'attends un ack de sa part. Après avoir envoyé le nombre de frames au client et attendu un accusé de réception (ack), le serveur commence à envoyer les frames du fichier demandé séquentiellement.

   - Le serveur lit le fichier par segments (frames) et les envoie au client.
   - Après l'envoi de chaque frame, il attend un ack du client pour confirmer la bonne réception de ce segment spécifique.
   - Si le serveur ne reçoit pas l'ack pour un segment particulier, il le renvoie jusqu'à ce qu'il reçoive l'ack correspondant ou jusqu'à un certain nombre maximal de tentatives (dans notre cas, on a 200 tentatives mentionnées).
   - En cas de réception d'un ack manquant ou incorrect, le serveur garde une trace du nombre de tentatives de renvoi (dropFrame).
   - Si le nombre maximal de tentatives est atteint pour un segment donné sans recevoir l'ack attendu, le serveur peut marquer un drapeau de dépassement de temps (tOutFlag) et arrêter le transfert.

   En plus, lors de la réception de tous les acks attendus pour chaque frame (segments de fichier), le serveur désactive l'option de timeout pour la

communication avec ce client spécifique.

Finalement, le serveur informe le client de la bonne réception de tous les segments ou, dans le cas d'un dépassement de temps ou d'un échec de transfert pour une raison quelconque, annonce que le fichier n'a pas été envoyé. Et il faudra également noter que c'est le client qui envoi la commande /GET sinon ça ne marchera pas.

Par ailleurs, voici les bouts de code du dépot GITHUB que j'ai modifié et adapté à mon programme :

- Pour la commande `/GET [file name]` :

  Coté server, nous avons :

```
        /*-----------------"get case"--------------------------*/

  printf("Server: Get called with file name --> %s\n", ...);

  if (access(flname_recv, F_OK) == 0)
  { // Check if file exist

      int total_frame = 0, resend_frame = 0, drop_frame = 0,
            t_out_flag = 0;
      long int i = 0;

      stat(flname_recv, &st);
      f_size = st.st_size; // Size of the file

      t_out.tv_sec = 2;
      t_out.tv_usec = 0;

      ....
      ....
      ....

      fclose(fptr);
      // Disable the timeout option
      t_out.tv_sec = 0;
      t_out.tv_usec = 0;
      setsockopt(sfd, SOL_SOCKET, SO_RCVTIMEO, (char *)&t_out,
                  sizeof(struct timeval));
  }
  else
  {
      printf("Invalid Filename\n");
  }
```

  Et coté client, nous avons :

```
// The client.
  long int total_frame = 0;
  long int bytes_rec = 0, i = 0;

   // Enable the timeout option if client does not respond
  t_out.tv_sec = 2;
  setsockopt(cfd, SOL_SOCKET, SO_RCVTIMEO, (char *)&t_out,
           sizeof(struct timeval));

   // Get the total number of frame to recieve
  recvfrom(cfd, &(total_frame), sizeof(total_frame), 0,
           (struct sockaddr *)&from_addr, (socklen_t *)&length);

  // Disable the timeout option
  t_out.tv_sec = 0;
  setsockopt(cfd, SOL_SOCKET, SO_RCVTIMEO, (char *)&t_out,
              sizeof(struct timeval));

   ....
   ....
   ....

  else
  {
      printf("File is empty\n");
  }
```

2. La commande `/PUT [File name]` :

   Cette commande permet à un client d'envoyer un fichier au server, on commence par construire le chemin à partir du nom de fichier obtenu précédemment, ensuite on vérifie si le fichier existe dans le dossier `./clientFiles` si non on affiche un message indiquant qu'il existe pas sinon on procède au traitement du fichier pour l'envoi qui se resume à récupérer la taille du fichier pour déterminer le nombre de segments (frames) à envoyer via `sendto` et on attends un accusé de reception de ce nombre de frame de la part du server pour confirmation de la bonne reception. Ensuite, on commence à envoyer les "frames" au serveur. Après chaque envoi, on attend un ACK correspondant du serveur. Si le client ne reçoit pas l'ACK correspondant, il réessaie un nombre limité de fois et le serveur de son coté reçoit les frames du client et envoie des ACKs correspondant pour chaque frame reçue. Il vérifie également les doublons et les pertes potentielles de "frames"; et cela se fait dans cette boucle :

```
    /*Recieve all the frames and send the acknowledgement
    sequentially*/
 for (i = 1; i <= totalFrame; i++)
```

```
{
    memset(&sframe, 0, sizeof(sframe));

    // Recieve the sframe
    CHECK(recvfrom(sockfd, &(sframe), sizeof(sframe), 0,
                   (struct sockaddr *)&clientStorage,
                   (socklen_t *)&clientLen));
    // Send the ack
    CHECK(sendto(sockfd, &(sframe.ID), sizeof(sframe.ID),
                 0, (struct sockaddr *)&clientStorage,
                 clientLen));

    /*Drop the repeated sframe*/
    if ((sframe.ID < i) || (sframe.ID > i))
    {
        i--;
    }
    else
    {
        /*Write the recieved data to the file*/
        fwrite(sframe.data, 1, sframe.length, sfptr);
        printf("sframe.ID ----> %ld    sframe.length ---->"
               "%ld\n",
               sframe.ID, sframe.length);

        bytes_rec += sframe.length;
    }

    if (i == totalFrame)
        printf("File recieved\n");
}
```

En outre, une fois que le client envoie tous les frames sont envoyés avec succès et que les ACKs correspondants sont reçus, il termine le transfert de fichier. Quant au serveur, après avoir reçu tous les “frames” attendu, il assemble les données et écrit le fichier.

En effet, les bouts de code qui m’ont effectivement permis d’implementer cette commande sont les suivantes :

Du coté du client, j’en ai pris :

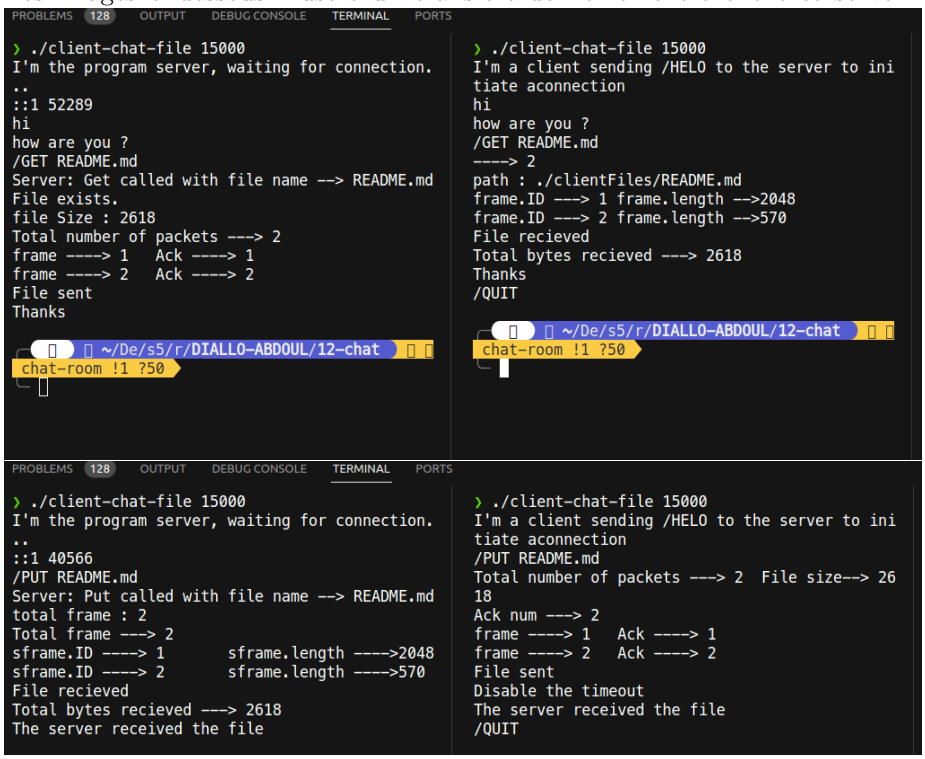
```
/*-----------------"put case"---------------------------------*/

 else if ((strcmp(cmd, "put") == 0) && (flname[0] != '\0')){

 if (access(flname, F_OK) == 0)
        { // Check if file exist
```

```
            int total_frame = 0, resend_frame = 0,
               drop_frame = 0, t_out_flag = 0;
            long int i = 0;

            stat(flname, &st);
            f_size = st.st_size; // Size of the file

            ....
            ....
            ....

            fclose(fptr);

            printf("Disable the timeout\n");
            t_out.tv_sec = 0;
            setsockopt(cfd, SOL_SOCKET, SO_RCVTIMEO,
                       (char *)&t_out, sizeof(struct timeval));
        }
}
```

Et finalement, du coté du serveur j'en ai pris :

```
/*--------------------------"put case" -------------------------*/

    else if ((strcmp(cmd_recv, "put") == 0) &&
               (flname_recv[0] != '\0'))
    {

        printf("Server: Put called with file name --> %s\n", f);

        long int total_frame = 0, bytes_rec = 0, i = 0;

        // Enable the timeout option if client does not respond
        t_out.tv_sec = 2;
        setsockopt(sfd, SOL_SOCKET, SO_RCVTIMEO, (
                   char *)&t_out, sizeof(struct timeval));

        ....
        ....
        ....

        else
        {
            printf("File is empty\n");
        }
    }
```

Les images ci-dessous illustre un transfert de fichier entre client et server :



5. Gestion avancées des utilisateurs.

Pour cette fonctionnalité, j'avais besoin d'une variable initialisée à 0 lorsque le serveur est lancé qui sert de compteur pour chaque nouveau utilisateur rejoignant le chat mais il s'est avéré qu'à chaque nouvelle instance d'un processus client la variable qui faisait le compte est remis à nouveau 0. Pour résoudre ce problème, j'ai envisagé d'utiliser un segment de mémoire partager appeler `/table`. Ce segment contient une structure spécifique qui comprend les informations nécessaires suivantes:

```
struct ClientInfo
{
    struct sockaddr_storage address; // We store the client address.
    char username[MAX_USERNAME_LEN]; // we store the client username.
};

struct Table
{
    int maxClients;   // MAX user that can join the chat room.
    int countClients; //  current user in the chat room
    sem_t semCountClient; // a semaphore to handle the client counting.
```

```
    struct ClientInfo clientsLst[];  // A list of all client joining the
                                     // chat.
} Table;
```

Avec cette approche, une instance du programme qui tourne en mode server et toutes les autres instances auront access à ces précieuses données. Ainsi, lorsque le serveur est lancé, je crée le segment et j'initialise les champs comme suit:

```
    // We init here the client counts otherwise each time a client join
    table->countClients = -1; // we start counting from 0 (1 client)
    table->maxClients = MAX_CLIENTS;
    CHECK(sem_init(&table->semCountClient, 1, 1));
```

La décision d'initialiser le champ `countClients` à `-1` découle de deux raisons principales. Tout d'abord, ultérieurement dans le code, lorsque l'arrivée d'un client est enregistrée, cette variable est incrémentée, passant ainsi à une valeur de `0`. Deuxièmement, cette variable joue également le rôle d'indice dans le tableau des clients. En commençant par `-1`, elle est prête à être utilisée comme index une fois incrémentée, permettant ainsi une gestion fluide des données relatives aux clients dans le programme.

Vous remarquerez aussi que j'ai déclaré deux variables de type `struct Table` qui sont `table` et `tableClient`, la raison de ces deux declarations réside dans le fait qu'en admettant qu'on a uniquement la variable `table` initialisée par le serveur, le programme en mode client utilisera cette adresse ne pointant pas au bon droit dans le segment et donc j'obtenais des valeurs aléatoires dans les champs de la structure.

L'une des nouveautés principales dans cette fonctionnalité est que pour qu'un nouveau client puisse s'annoncer, ce n'est pas plus commande `/HELO` qui est utilisée, il s'agit maintenant de `/HELO FROM [client username]`. Et ensuite le serveur se permet de bien sauvegarder l'adresse du nouveau client et son nom en vérifiant bien sur que ce nom n'est pas vide et qu'il y aussi de la place dans le salon de discussion. C'est pourquoi j'ai décidé d'ajouter la structure suivante :

```
struct FullMessage
{
    char welcomeMessage[MAX_WELCOME_MSG];
    char username[MAX_USERNAME_LEN];   // username .
    char message[MAX_MSG_LEN];    // message entered by the user.
};
```

Le champ `welcomeMessage[MAX_WELCOME_MSG]` ne peut contenir que deux messages possibles :

- "[user name] join the server." que le serveur envoie à tous les autres clients sauf le dernier qui vient de joindre le salon. Cela se concrétise grace à cette boucle.

```
    for (int i = table->countClients - 1; i >= 0; i--)
```

```
{
    // We inform all of them except the last one
    // to join.
    CHECK(sendto(sockfd, &(fullMessage), sizeof(fullMessage), 0,
                 (struct sockaddr *)&table->clientsLst[i].address,
                 sizeof(table->clientsLst[i].address)));
    // sleep(1);
}
```

- "/QUIT": Dans le cas où il n'y a plus de place dans le salon le serveur décrémente `countClients` vu que ce client ne va rester et l'envoie ce message et qui est en plus notifié par le message "This group is full, you cannot join it", ensuite, on `munmap()`, on ferme son descripteur et en fin il quitte le salon. Tout cela se passe dans ces deux blocs de code :

  – Travail du serveur :

```
...
else
{
    // We decrement or we will have destination addr error
    // in the main loop, because we increment it above even
    // though there weren't a place for him.
    CHECK(sem_wait(&table->semCountClient));
    table->countClients--;
    CHECK(sem_post(&table->semCountClient));

    printf("Server is full.\n");

    struct FullMessage fullMessage = {0};
    strcpy(fullMessage.welcomeMessage, "/QUIT");

    // We make him to quit the server.
    CHECK(sendto(sockfd, &(fullMessage), sizeof(fullMessage),
                 0, (struct sockaddr *)&clientStorage,
                 sizeof(clientStorage)));
    // we have to tell him to quit.
    fflush(stdout);
}
```

  – Travail du client :

  Je commence par vérifier que ça vient du client.

```
//
 if (strlen(buffFullMessage.welcomeMessage) > 1)
{
    if (strncmp(buffFullMessage.welcomeMessage, "/QUIT", 5) == 0)
    {
```

```
            printf("This group is full, you cannot join it.\n");
            // Unmap the shared memory object
            CHECK(munmap(tableClient, stbufshmClient.st_size));

            // Close the shared memory object
            CHECK(close(shmfdclient));
            running = 0;
        }
        else
        {
            printf("%s", buffFullMessage.welcomeMessage);
        }
    }else
    ...
```

Dès lorsqu'il y a deux utilisateurs dans le salon, ils peuvent s'envoyer mutuellement des messages. Afin de savoir qui envoie le message, j'ai utilisé le mecanisme suivant, comme le nom du client est déjà enregistré dans la variable `clientUsername` à son arrivé je le stocke dans le champ `username` et le message dans le champ `message` de la structure `struct FullMessage` qui est ensuite enovoyé à tout le monde sauf l'envoyeur. J'avais pensé à une première approche qui pouvait nous permettre de s'en passer de l'utilisation de la structure `struct FullMessage` en écrivant directement dans un buffer de type `char` le nom de l'expediteur et le message comme suit `sprintf(messageComplet, "%s: %s", clientUsername, message)` . Mais le problème avec cette proche est qu'au niveau des destinataires, il serait super compliqué de savoir si `messageComplet` contient la commande "/QUIT", d'ou l'idée de separer le `nom` et le `message`. Je pointe du doigt cette partie du code:

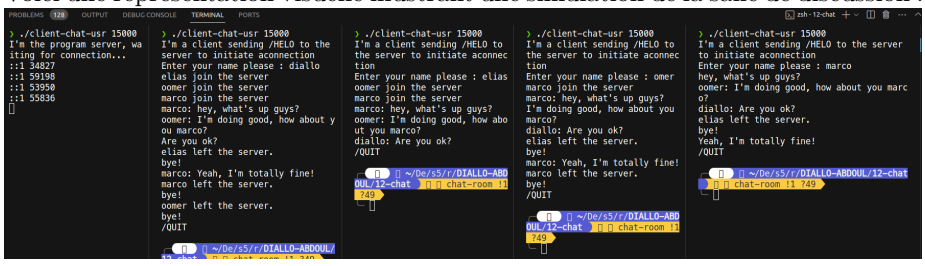
```
    // A message from a user.
    if (strncmp(buffFullMessage.message, "/QUIT", 5) == 0)
    {
        printf("%s left the server.\n", buffFullMessage.username);
        printf("bye!\n");
    }
    else
    {
        printf("%s: %s", buffFullMessage.username,
               buffFullMessage.message);
    }
```

De plus, que se passe-t-il lorsqu'un utilisateur quitte le salon ? Si c'est le dernier à avoir rejoint qui sort du salon, alors on peut simplement réduire la taille du tableau contenant les adresses des clients (`struct ClientInfo clientsLst[]`). Mais si, dans un salon de 5 utilisateurs, le 2ème quitte, dans ce cas précis, il faudra effectuer plus qu'une simple décrémentation. L'algorithme que j'ai

implémenté est le suivant : dans notre exemple, je déplace le dernier utilisateur, qui est le 5ème, à la place du 2ème, puis je réduis la taille du tableau. Vous trouverez ci-dessous l'implémentation de l'algorithme précédemment décrit :

```
    // Same here folks.
    if (strncmp(message, "/QUIT", 5) == 0)
    {
#ifdef USR
        // before quitting we have the decrement the countClients var
        // we do not only decrement we have to move the last client
        // to at the place of the one leaving the room.
        CHECK(sem_wait(&tableClient->semCountClient));
        if (tableClient->countClients >= 1)
        {
            // There is atleast two clients (0 and 1)
            for (int i = 0; i <= tableClient->countClients; i++)
            {
                if (strcmp(tableClient->clientsLst[i].username,
                           fullMessage.username) == 0)
                {
                    // we replace the client whose gone by the last one.
                    tableClient->clientsLst[i] =
                      tableClient->clientsLst[tableClient->countClients];
                    tableClient->countClients--;
                    break;
                }
            }
        }
        else
        {
            tableClient->countClients--;
        }
        CHECK(sem_post(&tableClient->semCountClient));
#endif
        running = 0;
    }
```

Voici une représentation visuelle illustrant une simulation de la salle de discussion :



6. Conclusion

L'implémentation du projet a réussi à atteindre les fonctionnalités prévues, mais elle a rencontré certaines limitations et défis.

Tout d'abord, pour le mode binaire, la factorisation du code afin d'éviter au maximum les instructions redondantes a été le principal défi pour moi concernant cette partie. Comme vous l'avez vu, je n'ai utilisé aucune structure, juste deux constantes `QUIT` et `HELO`, chacune ayant une taille d'un octet.

En ce qui concerne le transfert de fichiers, qui représente les deux tiers du code, je trouve qu'il y a des instructions répétitives dans la gestion des commandes `/GET [nom du fichier]` et `/PUT [nom du fichier]`. Je pourrais donc normalement le factoriser davantage. L'autre problème est que c'est commande doivent être lancé par le client et il faut savoir identifier le terminal du serveur et du client qui est le deuxième programme à être lancer et cela n'est pas du tout une façon souple de le faire et donc c'est pas "USER FRIENDLY".

Finalement, une limitation majeure pour le cas de N utilisateurs est que ce projet ne fonctionnerait que sur un seul ordinateur en raison du segment de mémoire partagée. De plus, il y a quatres subtilités dont je suis vraiment fier. Tout d'abord, l'idée d'utiliser un segment de mémoire partagée, car je pensais que je n'allais pas vraiment trouver de solution. Ensuite, savoir à quelle partie du code il fallait écrire le code de création et d'initialisation du segment, qui devrait se faire uniquement par le serveur. Ensuite, la nécessité d'avoir deux variables `table` et `tableClient` de type `struct Table`. Enfin, l'utilisation de l'algorithme pour gérer le départ d'un utilisateur du salon de discussion.