Relazione Progetto SOL

Michele Morisco 505252 Corso A

Nel progetto sono presenti, oltre i files forniti dal kit, alcuni files inediti creati per la realizzazione del server, di cui:

- Il file **configSetup.h** che contiene un'importante dichiarazione della struttura del file di configurazione. In questo modo, da semplificarne la gestione dei dati da cui attingere durante lo sviluppo del server.
- Il **threadPoolHandler.h** rappresenta la gestione dei thread con la relativa definizione di un pool di thread utilizzati, come da specifica, nella realizzazione del server.
- Il file **info_user.h** contiene, invece, le definizioni di strutture importanti per la realizzazione del server: come la hash table degli utenti e dei gruppi.
- Il file **fileManager.h** ha, al suo interno, una raccolta di funzioni per la gestione dei file che utilizzerà il server.
- Infine, il file **hashManager.h** contiene importanti operazioni per la gestione delle hash table.

Entrando nel dettaglio dei files sopracitati; in **info_user.h** si è definito le varie strutture dati. Per questo tipo di problema si è scelto di utilizzare un hash table per gli utenti e per i gruppi, rispettivamente *HT_USER* e *HT_GROUP*. Si è utilizzato questo tipo di struttura data la grande mole di dati che potrebbe usare, soprattutto nelle operazioni di ricerca/inserimento/cancellazione.
Ogni item di *HT_USER* contiene la *key*, rappresentata dall'username dell'utente, e *USER_INFO*, che consiste in un contenitore di informazioni importanti per l'utente quali: il file descriptor associato quando si connette, il numero di messaggi della sua history ed ecc...
Quest'ultimo ha un campo di tipo *MSG_INFO*, che rappresenta un array circolare dei messaggi salvati nella history e, al suo interno, contiene informazioni quali flag per determinare se il messaggio è stato letto e se è un messaggio testuale o un file.
Invece, in *HT_GROUP* è presente il campo *GROUP_INFO* il quale contiene informazioni riguardante il gruppo. Al suo interno si è definito il campo *host* che rappresenta l'utente che ha creato il gruppo. In questo modo, è più semplice, gestire operazioni che solo il creatore del gruppo può eseguire. Inoltre, al suo interno contiene una hash table, *users*, per la gestione degli utenti iscritti al gruppo.

In **threadPoolHandler.h**, si è definito una generica struttura di un pool avente come campi;

- **poolCreator(void *arg)** :- che crea il pool di thread. Viene impostato lo stato detach sui thread che dovranno gestire il funzionamento del server.
- **ThreadF(void *arg)** :- si occuperà di eseguire la funzione passata per parametro.
- **poolDestroy(void *arg)** :- classe che distrugge il pool di thread. Dato che non è possibile fare assunzioni su una eventuale terminazione della funzione che il thread avrebbe eseguito, si è effettuato la fermata dei thread attraverso la cancellazione, anziché con il join. Grazie all'inizializzazioni e alle cleanup, che si occupano del detach della memoria occupata dal thread alla sua cancellazione si eviterà memory leak. In conclusione, per garantire il detach completo dei thread, il programma si sospende un secondo prima della sua terminazione.
- **clean_upPool(void *arg)** :- La cleanup viene eseguita alla chiusura del thread.

Nel file **fileManager.h**, ci sono solamente due funzioni:

- **removeDirectory(const char *path)** :- essa ha lo scopo di eliminare ricorsivamente una directory con tutti i suoi file e/o directory al suo interno. Si è scelto di creare questo tipo funzione per pulire le directory dove si immagazzinano i files degli utenti per evitare che da un test all'altro restituisca un esito sfasato.

Per eliminare la directory, bisogna cancellare prima tutti i file al suo interno. Per far ciò, si deve esplorare tutto l'albero della directory da eliminare, enumerando i figli della directory. Quando si troverà un figlio che è una directory si chiama la funzione ricorsivamente, altrimenti si elimina il file al suo interno. Per enumerare le cartelle si utilizzano le funzioni *opendir*, *readdir* e *closedir*. Nella funzione si è scelto di utilizzare *stat()* e *S_ISDIR(stat.st_mode)* per controllare che il path sia una directory poiché il membro *d_type* nella *struct dirent* non è supportato da molti sistemi.

- **readConfigFile(char *path)** :- questa funzione permette di leggere il file di configurazione ed estrapolarne i dati. Per ottenere le informazioni al suo interno si è deciso, inizialmente, di leggere ogni riga del testo ignorando le righe con cui iniziano con '#' o che siano vuote e prendendo solamente in considerazione quelle senza. Dopo aver controllato che la riga può contenere un dato importante, viene utilizzato *strtok_r()* per ottenere prima il tipo di dato, dopodiché suddividere la riga rimanente in token. Quando si è ottenuto l'informazione viene confrontata in un ifthenelse dove ogni valore viene salvato nella struttura *ConfigureSetup* definita in **configSetup.h**.

Infine, nell'ultimo file citato, ovvero **hashManager.h** sono contenute le seguenti funzioni:

- **new_ht_user()** / **new_ht_group()** :- crea un hash table per gli utenti/gruppi con una dimensione fissata pari a 50 tramite la funzione **ht_user_sized(const int size)**/**ht_group_sized(const int size)**.
- **insertUser(ht_user *ht, const char *key, USER_INFO **user)** :- inserisce un nuovo utente all'interno del hash table. Prima dell'operazione di inserimento, controlla il numero di utenti inseriti in modo tale da raddoppiare la dimensione della struttura dati. Si è scelto questo approccio in modo da evitare che man mano che si aggiungano utenti si creino troppe collisioni, e di conseguenza, aumenti il tempo per aggiungere un elemento nella hash table. Per ridimensionare la struttura con la nuova dimensione si utilizza la funzione **ht_user_resize(ht_user *ht, const int size)**. Tale funzione crea una nuova hash table il quale viene popolata con i dati appartenenti alla vecchia hash table. Per l'inserimento del nuovo utente, si prepara prima l'item tramite la funzione **new_item_user(const char *key, USER_INFO *user)**, essa permette di creare un nuovo elemento utente di tipo *item_user* per poi inserirlo all'interno della struttura dati. Dopodiché, si cerca l'indice per l'inserimento tramite la funzione **get_hash(const char *s, const int num_buckets, const int attempt)**; **f_hash(const char *s, const int a, const int n)** prende una stringa e restituisce un numero tra 0 e n, ovvero la *size* della hash table. Purtroppo, utilizzando questa tecnica il tasso di collisioni è alto; per ovviare questo problema si è scelto di adottare il metodo double hashing ad indirizzo aperto. Di fatto, in *get_hash* si utilizza due volte la funzione *f_hash* per calcolare l'indice di un item che dovrebbe essere salvato dopo *i* collisioni.
- **insertGroup(ht_group *ht, const char *key, GROUP_INFO **group)** :- inserisce un nuovo gruppo all'interno della hash table,
- **searchUser(ht_user *ht, const char *key)**/**searchGroup(ht_group *ht, const char *key)** :- ricerca l'utente/gruppo all'interno della hash table.
- **searchUserByFd(int fd, ht_user *ht)** :- ricerca un utente tramite il file descriptor associato.
- **removeUser(ht_user *htUser, ht_group *htGroup, const char *key)** :- elimina un utente salvato nella hash table. Prima della cancellazione viene controllato il numero degli utenti inseriti in modo tale da dimezzare la struttura dati nel caso fossero pochi. Quando si elimina un utente, prima di tutto, si cancellano tutti i messaggi pendenti inviati e che non sono ancora stati letti tramite la funzione **deleteMsgs(ht_user *ht, const char *key)**. Per cancellare i messaggi, si scorre tutta la hash table e si scorre tutta la history di ogni utente per verificare se si trovano messaggi ancora non letti inviati dall'utente da eliminare. Inoltre, tale utente viene eliminato dalle liste degli utenti appartenenti ai gruppi tramite la funzione **deleteUserGroup(ht_group *ht, const char *key)**. Quando l'utente viene eliminato, si assegna nell'indice dell'utente cancellato un place-holder *DELETED_ITEM_USER*, che rappresenterà un item vuoto.
- **removeGroup(ht_group *ht, const char *key)** :- elimina un gruppo salvato nella hash table.
- **userConnection(ht_user *ht, const char *key, int fd)** :- tale funzione permette di aggiornare il file descriptor di un utente.
- **userMsg(USER_INFO *user, MSG_INFO newMsg)** :- essa permette di aggiungere un messaggio della history di un utente. Per questo tipo di problema si è scelto di implementare la history come una array circolare, l'indice *user->currentIndex* indica l'ultima posizione in cui è stato inserito il

messaggio nell'array. Questo tipo di scelta, ci evita di scorrere ogni volta tutto l'array quando si inserisce un nuovo messaggio e, dunque, il tempo per l'inserimento/cancellazione sarà costante. Quando la history è piena, per semplicità, si è pensato di eliminare il messaggio anche se non è ancora stato letto.

- **addUserToGroup(GROUP_INFO *group, char *key)** :- tale funzione garantisce che un utente venga iscritto ad un gruppo.
- **removeUserToGroup(GROUP_INFO *group, const char *user)** :- permette di disiscrivere un utente da un gruppo.
- **delete_ht_user(ht_user *ht)/delete_ht_group(ht_group *ht)** :- funzione per cancellare l'hash table degli utenti/gruppi.
- **printListUser(ht_user *curr, int *dim)** :- crea la lista degli utenti connessi da inviare al client. Si scorre l'intera hash table e si verifica che ogni utente abbia un file descriptor diverso da -1, dato che rappresenta un utente non connesso. Dopodiché, si concatena l'username alla stringa da restituire come risultato. Per facilitarne la concatenazione si è pensato di inserire spazi nella stringa con l'utilizzo della funzione d'ausilio **setBlank(char *str, int size)**.

Gli altri file presenti nel progetto sono:

- Il file **chatty.c** che contiene il codice del server, dunque tutta la gestione dei segnali, delle lock, delle variabili di condizione e l'inizializzazione dei thread che gestiranno le richieste da parte dei client.
- Il file **connections.c** ha, al suo interno, le implementazioni delle funzioni definite nel file connections.h, come da specifica.

Nel file **connections.c** si è aggiunto una sola funzione in più rispetto alle specifiche, ovvero, **sendHeader(long fd, message_hdr_t *msg)** che invia solamente lo header del messaggio al server. Per la lettura e la scrittura dei byte, viene utilizzato *readn* e *writen*:

- **readn(int fd, char *buf, size_t n)** :- è una classe che attende di ricevere tutti i byte richiesti,
- **writen(int fd, char *buf, size_t n)** :- essa attende che consegni al buffer di sistema tutti i byte richiesti.

Invece all'interno del file **chatty.c** si sono definite alcune funzioni d'ausilio per gestire le richieste tra il client e server.
Inizialmente, quando si attiva il server viene letto il file di configurazione e, con i dati appena acquisiti, si crea la cartella dove risiederanno i file inviati agli utenti. Per questo tipo di problema, si è creata un'altra directory che permette di salvare i messaggi ricevuti, rappresentata da una cartella rinominata con l'username dell'utente. Si è scelto questo approccio poiché salvando i messaggi all'interno di un array circolare creava molti errori durante la richiesta di *GETPREVMSGS_OP*, quali per esempio perdita di messaggi o nessun salvataggio di essi nella history. Invece, usando questo tipo approccio e salvando in *MSG_INFO* il percorso dove il quale risiede tale messaggio garantiva che venisse salvato correttamente senza errori.

Dopodiché, vengono create e inizializzate le due hash table per gli utenti e i gruppi.

- **connectClient()** :- la classe che contiene la gestione della connessione del server. Appena si entra in questa classe, inizializzo l'attributo dei mutex con il tipo *PTHREAD_MUTEX_ERRORCHECK*, perché durante la chiusura di un thread del pool, quando veniva chiamata la funzione di cleanup, accadeva che si richiamava un unlock di una possibile mutex non lock quindi Valgrind restituiva un errore. Grazie ad *ERRORCHECK*, viene controllato che la mutex sia lock, altrimenti, gestisce l'errore e viene ignorato; in questo modo, Valgrind non restituisce più tale errore. Dopodiché, viene inizializzato l'insieme per la gestione dei segnali che verranno gestiti dal server. Durante l'intera esecuzione delle operazioni dei thread, il thread che viene eseguito dal main, rimane l'unico in ascolto dei segnali attraverso la *sigwait*. Infatti, sarà proprio tale thread, a chiedere la generazione delle statistiche e a chiudere i thread in caso di segnale di terminazione. Tutti gli altri thread hanno una maschera totale dei segnali.

- **spawn_thread(void *arg)** :- è la funzione che viene chiamata dai singoli thread creati dal pool. Per realizzare la gestione delle richieste dei client si è pensato di utilizzare i thread (definiti nel file **threadPoolHandler.h**) che ascoltano uno alla volta il canale di comunicazione, dopodiché dopo aver accettato una richiesta da un client, passeranno l'ascolto a un altro thread. Per implementare ciò, si è scelto di implementare una lock *mutexAccept* e la sua variabile di condizione *conditionAccept* e la variabile *acceptMux*. Grazie al metodo della select ci permette di identificare, ad ogni richiesta arrivata, se essa è una nuova connessione, una disconnessione oppure un messaggio da gestire. Se fosse una nuova connessione, si controlla prima di tutto, se è possibile accettare nuove connessioni, altrimenti, viene restituito un errore al client che ha provato a connettersi. Se l'esito è positivo, incrementa il numero di utenti connessi e inserisce il file descriptor nell'insieme tramite *FD_SET*, dopodiché si gestisce il tipo di richiesta che ha fatto il client. Invece, nel caso si dovesse ricevere una richiesta di messaggio o chiusura del canale di comunicazione, si utilizza la mutua esclusione sul quel file descriptor tramite un array di mutex, denominato *connfdList*, insieme alla lock *mutexUser* e la variabile di condizione *conditionUser*. Questo array è essenziale poiché nella i-esima posizione contiene lo stato del file descriptor *i-esimo*; lo stato può assumere tre valori: 0, quando il fd non è occupato da nessun client; -1, quando è viene abbandonato (temporaneamente) da un thread durante l'invio di un messaggio; maggiore di 0, quando contiene l'ID del thread che lo sta occupando tramite la funzione *pthread_self()*. Dopo aver finito con ciò, al passo successivo ci si occupa se il client ha qualche altra richiesta o vuole deregistrarsi oppure ha chiuso la connessione; grazie, alla lettura del messaggio dal fd, se fallisce vuol dire che il canale è chiuso, altrimenti, gestisco l'operazione letta dal messaggio tramite la funzione **cmd_server(message_t msg, int fd)** che prende come input il messaggio appena letto e il fd del client che lo ha richiesto.
- **cmd_server(message_t msg, int fd)** :- All'interno di questa funzione, si è creato un menù che filtra le operazioni da eseguire dalla richiesta del client; la funzione creerà un messaggio di risposta al client con l'esito dell'operazione ed, eventuali, dati da consegnare. Nella richiesta *REGISTER_OP* e *UNREGISTER_OP* si crea/cancella la directory con l'username dell'utente all'interno della cartella "History" per registrare i messaggi degli utenti. Nel comando *CONNECT_OP*, quando l'utente è già connesso, si è scelto di mandare un errore inedito introdotto nel file **ops.h**, ovvero, *OP_CONNECTED_ALREADY*. Purtroppo, il client non potrà gestire questo nuovo errore dato che la struttura si basa sulle specifiche iniziali del progetto. Per garantire la mutua esclusione durante le operazioni sul hash table si è utilizzato la lock *mutexUser* dato che sono operazioni eseguite dagli utenti, invece, per garantire la mutua esclusione durante l'aggiornamento delle statistiche si è utilizzato la lock *mutexStats* e per garantire la mutua esclusione per le operazioni dei file si è utilizzato la lock *mutexFile*. Nell'operazione *DELGROUP_OP*, si disiscrive un utente da un gruppo e, riguardo la cancellazione di un gruppo avviene soltanto quando l'host vuole cancellarsi dal gruppo (o se tale utente si vuole deregistrare dal server).
  Per la richiesta *GETPREVMSGS_OP*, si devono ottenere tutti i messaggi della history di un utente. Grazie all'array MSG_INFO, dove si è salvato il percorso del file di ogni messaggio, viene copiato il path dove risiede il messaggio per poi accederci. Si controlla se l'accesso al file ha esito positivo, dopodiché, si controlla se il messaggio è stato letto oppure no. Questo permette di aggiornare il file delle statistiche sul numero dei messaggi consegnati. Dopo ogni verifica, si legge il messaggio con la funzione *readMsg* e lo si alloca a *bufferMsg*.
  Riguardo all'invio dei messaggi (sia testuali che files) tra gli utenti, che avviene tramite la richiesta *POSTTXT_OP*, *POSTTXTALL_OP* e *POSTFILE_OP*, si controlla inizialmente se il destinatario si tratta di un gruppo o di un utente. Dopodiché, qualsiasi è il risultato viene inviato il messaggio all'utente tramite la classe **sendMessagge(message_t msg, op_t op)**.

- **sendMessagge(message_t msg, op_t op)** :- tale classe permette di salvare ogni messaggio nella history di un utente e di notificarlo, se connesso. Prima di tutto, viene preparato il messaggio da salvare e/o inviare e, dopodiché, si controlla se l'utente sia connesso al server. Per l'invio di un messaggio al fine di evitare problemi di stallo, viene rilasciata momentaneamente la mutua esclusione sul fd corrente. Se il destinatario non è online, il messaggio viene salvato nella cartella il quale verrà poi letto da esso tramite il comando apposito per la history dei messaggi. Difatti, si è scelto di salvare in un file associato all'utente in modo da garantire che i messaggi rimangano salvati anche quando il server viene chiuso. Per poterlo salvare, si prepara il percorso: il nome del file, ovvero di ogni messaggio, sarà rinominato "msg" + l'indice corrente a cui punta il messaggio. E

dopo l'esito positivo del salvataggio viene salvato tale percorso in *MSG_INFO* tramite la funzione **userMsg(USER_INFO *user, MSG_INFO newMsg)**. Ovviamente, ad ogni utente che viene deregistrato, ogni messaggio pendente viene cancellato.

Script Bash

Lo script **scriptArchivio.sh** inizialmente controlla se non ci siano argomenti oppure se l'utente abbia utilizzato come primo argomento "-help",
Per implementare ciò ho utilizzato una condizionale con la guardia: *$# -eq 0* per indicare il numero di argomenti che sia uguale a 0.
Se fosse vero mostro nello standard output come utilizzare lo script. Dopodiché, faccio un po' di controlli, del tipo se l'utente immette più di due argomenti, se il file di configurazione immesso esista e sia regolare oppure che il file non sia vuoto.
Per cui, salvo il numero intero del secondo argomento in una variabile e confronto se il numero sia minore e uguale a -1. Se la condizione è vera avverto l'utente che non ha immesso un numero positivo.
Prima di cercare il percorso dove risiede la directory con i file da inviare agli utenti, salvo in due variabili differenti la stringa "DirName" e "="; dopodiché, utilizzando l'opzione *grep* cerco la riga in cui inizia con la parola "DirName" e la salvo in una variabile.
Grazie ai due prefissi salvati precedentemente li utilizzo per estrapolare dalla variabile *var1*, solamente il percorso dove risiede la directory *DirName*.
Infine, controllo se la variabile *t* sia uguale a 0 in modo che io stampi tutti i file presenti all'interno della directory, altrimenti, creo un archivio vuoto e, poi, utilizzando l'opzione *find* cerco i file e le directory che sono più vecchi di *t* minuti e li archivio aggiungendoli all'archivio vuoto; dopodiché li cancello definitivamente. Nel caso in cui l'archivio è rimasto vuoto allora lo elimino.

Ambiente compilazione ed esecuzione codice

Il codice è stato sviluppato e testato su Ubuntu nella macchina virtuale rilasciata dai professori.