Relazione sul codice del progetto ChatterBox

Alessandro Meschi 525658

7 maggio 2019

Introduzione

In questo documento vengono descritte le scelte progettuali effettuate nella stesura del codice del progetto **ChatterBox** relativo al corso di Laboratorio di sistemi operativi A.A. 2017/2018.
In particolare vengono elencate le scelte riguardo:

- Le strutture dati utilizzate con le relative dimensioni
- La suddivisione in files e la modularizzazione del codice
- Alcune estensioni facoltative aggiunte

Per generare la documentazione dettagliata di tutto il codice in formato `html` è necessario l'utilizzo del comando `doxygen`. Quindi è necessario impostare la current directory alla cartella dove si è estratto il tarball del progetto, dopodiché è necessario eseguire questo comando:

```
doxygen DoxyFile
```

Alla fine dell'esecuzione verrà creata la cartella `Documentazione/html/`. Questa relazione è ugualmente reperibile in `Documentazione/html/index.html` all'interno della pagina iniziale della documentazione. Nella versione `html` ogni riferimento a strutture dati, funzioni, macro ecc... è opportunamente linkato alle relative pagine di documentazione.

Strutture dati utilizzate

In questo progetto le principali strutture dati sono:

- `incomeList`
  Rappresenta la linea di comunicazione fra il main thread che percepisce richieste sui file descriptors e i worker threads i quali estraggono il valore del file descriptor dalla coda e cominciano il servizio della richiesta. E' implementata come una lista concatenata concorrente di elementi `income`.

- `user`
  Struttura che rappresenta un singolo utente all'interno del sistema. Si tratta di una semplice struttura che raccoglie le informazioni dell'utente fra cui:

  - `delivery`
    Rappresenta il vettore di connessioni al singolo utente (`delivery.h`). La lunghezza di tale vettore e quindi il numero massimo di connessioni simultanee per utente è regolata dalla macro `MAX_USER_SIMULTANEUS_CONNECTIONS`.
  - `membership`
    Rappresenta il vettore di appartenenza del singolo utente (`membership.h`). La lunghezza del vettore e quindi il numero massimo di gruppi a cui un utente può appartenere simultaneamente è regolata dalla macro `MAX_USER_SIMULTANEUS_GROUPS`.
  - `history`
    Rappresenta il vettore degli ultimi messaggi ricevuti dall'utente (`history`). La lunghezza del vettore e quindi il numero massimo di messaggi contenibili nella history è determinata dal valore di impostazione `MaxHistMsgs`.

- `usersTable`
  Rappresenta la tabella dove si tiene traccia di tutti gli utenti registrati, memorizzandoli assieme alle loro informazioni e strutture relative. E' implementata come una hash table concorrente con code di trabocco, più nello specifico consiste di un vettore di `usersList` (le quali rappresentano le code di trabocco); La concorrenza invece è gestita tramite un vettore di locks ognuna delle quali è relativa ad una certa porzione di tabella. Il numero di righe della tabella è regolato dalla macro `USERS_TABLE_SIZE` e la dimensione dei blocchi di mutua esclusione è regolata da `USERS_MUTEX_BLOCK_SIZE`.

- `logsTable`
  Rappresenta la tabella dove si tiene traccia di tutte le istanze di logIn al sistema da parte di utenti registrati. E' implementata come una tabella hash concorrente con hashing diretto, infatti l'accesso alla tabella viene effettuato attraverso il valore del file descriptor del socket sul quale avviene la comunicazione con il clicent. La concorrenza è gestita tramite una lock per ogni slot della tabella (per le scritture e le letture dal socket) e una lock aggiuntiva per rendere le operazioni di login e logout mutuamente esclusive. Il numero di slots della tabella è uguale a: il numero massimo di connessioni contemporanee al server (`MaxConnections`) + il numero di threads nel pool (`ThreadsInPool`) + 1. Questa quantità è dovuta all'assegnazione dei valori dei file de-

scriptors effettuata dal sistema. Per ulteriori chiarimenti vedere la documentazione di `logIn.h`.

- `groupsTable`
  Rappresenta la tabella dove si tiene traccia di tutti i gruppi registrati nel sistema dagli utenti. Nella struttura ogni gruppo viene accompagnato dalla lista dei suoi membri e altre informazioni di base. L'implementazione della tabella dei gruppi rispecchia completamente la struttura della tabella degli utenti. Il numero di righe della tabella è regolato dalla macro `GROUPS_TABLE_SIZE` e la dimensione dei blocchi di mutua esclusione è regolata da `GROUPS_MUTEX_BLOCK_SIZE`.

- `fdSet`
  Rappresenta la struttura che raccoglie l'insieme dei file descriptors da monitorare e altre informazioni per permettere la corretta gestione delle strutture utili alla system call `select()`.

Suddivisione del codice

Il codice prodotto è stato suddiviso in modo tale da essere modulare. Nello specifico il codice delle funzoni relative alle strutture dati è stato suddiviso su diversi files: uno per struttura dati principale. Inoltre la maggior parte delle definizioni di macro sono raccolte in `config.h`, le funzioni di utilità sono invece raccolte in `utils.h`, le funzioni che estraggono le impostazioni di sistema e ne stampano i valori sono raccolte in `settings.h`, le funzioni che implementano il protocollo di comunicazione parte client sono raccolte in `connections.h` mentre quelle parte server sono raccolte in `communication.h` insieme ad altre funzioni che svolgono operazioni comunicando con i clients, infine la funzione di routine dei threads worker è raccolta nel file `listener.h`. Durante l'esecuzione del Makefile viene generata una libreria statica `libchatty.a` che contiene il codice compilato di tutte le funzioni menzionate sopra.

Flusso di esecuzione

Il programma server inizia la sua esecuzione dalla funzione `main` definita all'interno del file `chatty.c`, durante la sua esecuzione attiva un pool di tread che eseguono tutti la solita funzione di routine (listener). Analizziamo il loro flusso di esecuzione separatamente.

Thread Main

Inizialimente si effettua il parsing dei parametri passati da riga di comando, controllando che non ci siano opzioni incomplete o sconosciute, dopodiché si procede con il parsing del file di configurazione e quindi con l'impostazione dei valori di setting del server (`settings.h`). A questo punto vengono inizializzate in quest'ordine le principali strutture dati del programma:

- `incomeList`
- `usersTable`
- `logsTable`
- `groupsTable`

Vengono rimossi eventuali files residui da vecchie esecuzioni del programma e subito creati il nuovo file delle statistiche e la directory per il salvataggio dei files scambiati. Viene impostata la gestione dei segnali e vengono inizializzati i parametri dei threads. A questo punto vengono lanciati i vari threads worker. Viene creato il socket per la connessione di client dopodiché inizia il **ciclo di ascolto del server**. In questo ciclo il server controlla inizialmente i flag di terminazione e di ricezione del segnale SIGUSR1:

- Se il flag di terminazione è `true` allora **si esce dal ciclo**, inserisce il messaggio di terminazione nella `incomeList`, attende la terminazione dei threads e chiama una per una le funzioni di pulzia della memoria per ogni struttura dati precedentemente inizializzata.
  Qui il programma termina.
- Se il flag di ricezione del segnale `SIGUSR1` è `true` allora vengono appesi i valori delle statistiche attuali al file delle statistiche.
  Il ciclo ricomincia da capo.

Si crea una copia del set di files descriptor di sistema (`masterSet`) che viene passata alla system call `select()` al termine della quale vengono esaminati i files descriptor residenti nell'insieme per vedere se c'è una nuova connessione pendente o se è possibile effettuare una lettura da una connessione già stabilita:

- Se viene rilevata un anuova connessione pendente questa viene accettata, l'accettazione restituisce il valore di un file descriptor che viene inserito nell'insieme dei files descriptor da monitorare (`masterSet`).

- Se viene rilevata una possibile scrittura da una connessione già stabilita viene rimosso il file descriptor corrispondente dall'insieme di file descriptors da monitorare (`masterSet`) e viene inserito nella lista di comunicazione con i threads (`incomeList`).

Il ciclo ricomincia.

Thread listener

La funzione `listener` inizia eseguento subito un ciclo potenzialmente infinito nel quale si porcede estraendo un file descriptor da servire dalla `incomeList` di sistema (`service`) attraverso la chiamata alla funzione `popIncome()`. Effettuata l'estazione si controlla se il valore restituito:

- Se è il valore di terminazione (`INCOME_LIST_TERM`) il ciclo viene interrotto.
  Qui la funzione `listener()` termina.

- Se è il valore di un semplice file descriptor allora la funzione comincia ad interagire con il client connesso al server attraverso file descriptor estratto.

Inizialmente si legge il messaggio di richiesta del client attraverso la funzione `sReadMsg()`:

- Se viene restituito un valore minore di zero allora si procede con l'operazione di logout e poi disconnessione qualora il client da cui si leggeva fosse stato loggato, altrimenti direttamente con la disconnessione qualora il client da cui si leggeva o non era registrato o non aveva eseguito il login. In entrambi questi casi qui il cliclo ricomincia.

- Se viene restituito un valore maggiore di 0 (sperabilmente la dimensione del messaggio) viene analizzato per casi il tipo di messaggio in base al campo op di `message_hdr_t`.

A questo punto si analizza il messaggio letto per casi:

- `REGISTER_OP`
  Consiste nella richiesta di registrazione, viene chiamata la funzione `registerUser()`. Se la funzione fallisce per qualche motivo (vedere la documentazione di `registerUser`) si invia un messaggio di operazione fallita al client, altrimenti
  si passa al caso successivo.

- `CONNECT_OP`
  Consiste nella richiesta di logIn, viene chiamata la funzione `logInUser()`. Se la funzione fallisce per qualche motivo (vedere la documentazione di `logInUser`) si invia un messaggio di operazione fallita al client, altrimenti
  si passa al caso successivo.

- `USRLIST_OP`
  Consiste nella richiesta della lista degli utenti loggati al sistema, viene chiamata la funzione `getLoggedUsersList()` per generare la lista dopodiché viene inviata al client con un messaggio di operazione avvenuta con successo (`OP_OK`).
  Qui l'analisi dei casi si interrompe.

- `POSTTXTALL_OP` e `POSTTXT_OP`
  Consiste nella richiesta di invio di un messaggio di tipo testo da un utente ad un altro utente oppure ad un gruppo di utenti oppure a tutti gli utenti registrati. Anche se le tre operazioni sono chiaramente distinte, queste rientrano nel solito caso all'interno di questa analisi poiché la discriminazione del destinatario/i verrà fatta all'interno della funzione di invio del messaggio. Per prima cosa viene verificato se il mittente è effettivamente registrato: In caso negativo viene inviato un messaggio di fallimento dell'operazione e
  qui l'analisi dei casi si interrompe.
  Altrimenti viene invocata la funzione `ship`:

  - Se la funzione fallisce per qualche motivo (vedere la documentazione di `ship`) allora viene inviato un opportuno messaggio di errore al mittente (`OP_NICK_UNKNOWN` o `OP_MSG_TOOLONG` o `OP_FAIL` a seconda dei casi) e
    qui l'analisi dei casi si interrompe.
  - Se l'invio è avvenuto con successo allora viene inviato al mittente un messaggio di operazione avvenuta con successo (`OP_OK`) e
    qui l'analisi dei casi si interrompe.

- `GETPREVMSGS_OP`
  Consiste nella richiesta dello storico dei messaggi da parte di un utente, viene chiamata la funzione `sendUserHistory()` che invia uno per uno tutti i messaggi presenti nello storico dell'utente che ne ha fatto richiesta. Se la funzione fallisce per qualche motivo (vedere la documentazione di `sendUserHistory()`) allora viene inviato un messaggio

di operazione fallita. In ogni caso
qui l'analisi dei casi si interrompe.

- `UNREGISTER_OP`
  Consiste nella richiesta di deregistrazione da parte di un utente, viene chiamata la funzione `unregisterUser()`. Se la funzione fallisce allora viene inviato un messaggio di fallimento dell'operzaione al client che l'ha richiesta, in caso di successo invece viene inviato un messaggio di operazione avvenuta con successo. In ogni caso
  qui l'analisi dei casi si interrompe.

- `POSTFILE_OP`
  Consiste nella richiesta di invio di un file da un utente ad un altro utente oppure ad un gruppo di utenti. Per prima cosa viene verificato se il mittente è effettivamente registrato, in caso negativo viene inviato un messaggio di fallimento dell'operazione e
  qui l'analisi dei casi si interrompe.
  Altrimenti Viene estratto il nome del file dal path (`getNameFromPath`) presente nel messaggio di richiesta, dopodiché viene composto il path della copia del file che dovrà risiedere nella cartella dei file scambiati (`DirName`). A questo punto avviene il trasferimento del file che viene letto dal server tramite la funzione `sReadFile()`. Se avviene un errore durante la lettura viene inviato un opportuno messaggio di errore al client che ha inoltrato la richiesta e
  qui l'analisi dei casi si interrompe.
  Se invece la lettura avviene correttamente, si verifica subito se il file è già presente nella cartella `DirName`:

  – Se il file è già presente allora quello appena letto viene scartato.
  – Se invece non è già presente viene creato il file e scritto con i dati appena letti.

  A questo punto viene prearato un messaggio di notifica per il destinatario/i e inviato tramite la funzione `ship()`. Viene verificato il successo della funzione come nel caso di `POSTTXT_OP` e `POSTTXTALL_OP`. Viene liberata la memoria occupata dai vari buffers e
  qui l'analisi dei casi si interrompe.

- `GETFILE_OP`
  Consiste nella richiesta di download di un file da parte di un utente. Per prima cosa viene verificato se il mittente è effettivamente registrato, in caso negativo viene inviato un messaggio di fallimento dell'operazione

e
qui l'analisi dei casi si interrompe.
Altrimenti Viene composto il path del file che, se esiste, risiede in `DirName`.

  - Se il file non esiste viene inviato un messaggio di errore al mittente (`OP_NO_SUCH_FILE`) e
    qui l'analisi dei casi si interrompe.
  - Se invece il file esiste questo viene aperto, mappato in memoria e inviato al client tramite un messaggio di operazione avvenuta con successo (`OP_OK`).
    Qui l'analisi dei casi si interrompe.

- `CREATEGROUP_OP`
  Consiste nella richiesta di creazione di un nuovo gruppo. Inizialmente viene invocata la funzione `createGroup()` per l'aggiunta del gruppo alla tabella dei gruppi di sistema (`grTable`). Se la funzione fallisce viene inviato un opportuno messaggio di errore (`OP_NICK_ALREADY` o `OP_FAIL`) e
  qui l'analisi dei casi si interrompe. Altrimenti l'utente che ha inoltrato la richiesta viene aggiunto al gruppo tramite la funzione `joinUser()`.
  Ugualmente qui se la funzione fallisce viene inviato un opportuno messaggio di errore (`OP_FAIL`) e
  qui l'analisi dei casi si interrompe.

- `ADDGROUP_OP`
  Consiste nella richiesta di adesione ad un gruppo da parte di un utente. Viene chiamata la funzione `joinUser()` se la funzione fallisce viene inviato un opportuno messaggio di errore (`OP_NICK_UNKNOWN` o `OP_FAIL`) e
  qui l'analisi dei casi si interrompe.
  Se la funzione ha avuto successo viene inviato un messaggio di operazione avvenuta (`OP_OK`) e
  qui l'analisi dei casi si interrompe.

- `DELGROUP_OP`
  Consiste nella richiesta di dissociazione da un gruppo da parte di un utente. Viene chiamata la funzione `disjoinUser()` se la funzione fallisce viene inviato un opportuno messaggio di errore (`OP_NICK_UNKNOWN`) e
  qui l'analisi dei casi si interrompe.

Se la funzione ha avuto successo viene inviato un messaggio di operazione avvenuta (`OP_OK`) e
qui l'analisi dei casi si interrompe.

- `REMOVEGROUP_OP`
  Consiste nella richiesta di cancellazione di un gruppo. Viene chiamata la funzione `deleteGroup()`. Se la funzione fallisce viene inviato un messaggio di errore al client che ha inoltrato la richiesta (`OP_NICK_UNKNOWN`) e
  qui l'analisi dei casi si interrompe.
  Altrimenti viene inviato un messaggio di oprazione avvenuta con successo (`OP_OK`) e
  qui l'analisi dei casi si interrompe.

A questo punto viene liberata la memoria occupata dalle varie strutture allocate durante l'esecuzione delle operazioni dopodiché il file descriptor viene reinserito nel set (`masterSet`).
Qui il ciclo ricomincia.

Estensioni

Gestione errori

Relativamente alla gestione degli errori, all'interno del programma è stato assunto un modello di restiruzione di valori da parte delle funzioni per indurre coerenza in tutte le sue singole parti. I valori di ritorno delle funzioni, che sono sempre interi, vanno interpretati come segue:

- ≥ 0 Se l'esecuzione della funzione è avvenuta con successo
- -1 Se si è verificato un errore fatale
- <-1 Se si è verificato un errore non fatale

All'interno del file `errors.h` sono elencati tutti i codici di errore interni e relativi ad ogni possibile causa di inadempienza delle funzioni. Relativamente ai codici di errore sono anche elencati, uno per uno, tutti i possibili messaggi di errore. Tutto questo per avere una coerenza di valutazione dell'errore al ritorno delle funzioni, inoltre in questo modo si facilita la gestione dell'errore e la stampa dei messaggi di errore.

LogIn simultanei

All'interno del progetto si è gestito nel modo ritenuto più corretto lo scenario di più client loggati al servizio con lo stesso username, questo giustifica la presenza della struttura `delivery` all'interno della struttura `user`. Innanzitutto si è dato un limite al numero di client loggati al solito utente, definito dalla macro `MAX_USER_SIMULTANEUS_CONNECTIONS`. In generale la gestione di questo fatto funziona come ci si aspetta: se un messaggio viene inviato ad un utente con più clients loggati, allora la notifica di messaggio arriverà a tutti i clients, se un client richiede lo storico dei messaggi questo verrà inviato solo a lui e non a tutti gli altri client loggati con il solito utente, ecc... La parte più delicata risiede nella desregistrazione dell'utente che ha più client loggati, per cui si è deciso di disconnettere (brutalmente) tutti i clients tranne quello che ha fatto richiesta di deregistrazione, il quale deve ricevere il messaggio di operazione avvenuta, fatto ciò si procede come una normale disconnessione del client rimasto. Per maggiori dettagli leggere la documentazione di `delivery.h`.

Stampa di messaggi colorati su terminale

Per ottenere una lettura veloce ed efficace dei messaggi di output su terminale da parte del server, è stato implementato all'interno del progetto un sistema di stampa formattata che si basa sul concatenamento di alcune stringhe all'inizio e alla fine del messaggio che si intende stapare. I codici di formattazione e le descrizione delle funzioni di stampa sono reperibili nella documentazione del file `utils.h`. Questa funzionalità non è completamente portabile e il suo comportamento può variare fra i diversi emulatori di terminale, è stata quindi resa una funzione impostabile tramite ozione (`-c`):

```
./chatty -f ./DATA/chatty.conf1 -c
```