Corso Sistemi Operativi e Laboratorio 2016/2017

Relazione Finale

Luca Corbucci
(Matricola 516450 – Corso A)

Indice

Tutto il progetto è stato sviluppato e testato su Elementary OS 0.4.1.

1. Organizzazione del codice

Chatty.c : File principale del progetto, contiene il Main e avvia i thread

Coda.c / Coda.h : Contengono la specifica e l'implementazione delle funzioni che permettono di creare e di gestire una coda che utilizziamo, nel nostro caso, per inserire i FD dei nuovi client che si connettono al server.

Codamessaggi.c / Codamessaggi.h : Contengono la specifica e l'implementazione delle funzioni che permettono di creare e di gestire una coda che utilizziamo, nel nostro caso, per ricordare i messaggi che l'utente ha ricevuto nel momento in cui non si trovava online.

Connections.c / Connections.h : Contengono la specifica e l'implementazione delle funzioni che permettono la comunicazione tra il server e i vari client.

Icl_hash.c / Icl_hash.h : Contengono la specifica e l'implementazione di una tabella hash. Questi due file sono stati forniti nell'ultima esercitazione (http://didawiki.cli.di.unipi.it/doku.php/informatica/sol/laboratorio17/esercitazionib/esercitazione12) e sono stati poi modificati per aggiungere delle operazioni utili al progetto.

Makefile : Contiene tutte le regole che permettono la compilazione del progetto

MyPthread.c / MyPthread.h : Contengono le classiche funzioni per la gestione dei thread a cui sono stati aggiunti i controlli sui valori restituiti in modo da non doverli controllare ogni volta separatamente.

Parsing.c / Parsing.h : Contengono la specifica e l’implementazione della funzione che viene utilizzata per interpretare il file di configurazione del server

Thread_pool.c / Thread_pool.h : Contengono la specifica e l’implementazione delle funzioni che vengono utilizzate per creare e gestire il pool di thread

Utils.c / Utils.h : Contengono la specifica e l’implementazione di alcune funzioni utili che vengono utilizzate nel progetto

2. Gestione della comunicazione tra client e server

Per iniziare la comunicazione con il server, il client utilizza la funzione openConnection che è stata implementata in connections.c.
Una volta stabilita la connessione il client può iniziare a inviare dei messaggi al server per svolgere svariate operazioni.
Per ogni messaggio inviato dal client, il server manderà una risposta in modo da notificare al client l’esecuzione o il fallimento dell’operazione richiesta.

Quando il client o il server chiudono la connessione, la comunicazione viene interrotta.

Per la comunicazione vengono utilizzate le strutture dati definite in message.h, quindi message_hdr_t, message_data_hdr_t e message_data_t.
Queste strutture dati al loro interno permettono di inserire il nome del client che ha inviato il messaggio e se necessario il nome del client con cui dovrà avvenire la comunicazione.

Per ricevere o inviare messaggi vengono utilizzate le funzioni implementate nel file connections.c.
Se ad esempio il client volesse inviare un messaggio ad un altro client verrà utilizzata la funzione sendRequest che tramite 5 write trasmetterà il messaggio al server che poi lo inoltrerà al destinatario.
Considerato che non sempre all'interno del messaggio sarà presente un testo o un file da inviare come buffer, è stato anche inserito un controllo per evitare di effettuare l'ultima write.
Per la lettura di un messaggio di questo genere verrà usata ReadMsg che tramite 5 read andrà ad estrarre il contenuto del messaggio ricevuto.

3. Strutture dati utilizzate

utenti_registrati : Tabella hash dove vengono memorizzati gli utenti che si sono registrati.
Ogni utente registrato viene salvato in una icl_entry_t, all'interno troviamo:

- Il **nome** dell'utente
- **Il file descriptor dell'utente** che viene assegnato nel momento in cui si registra
- La **codamessaggi** ovvero l'insieme dei messaggi che l'utente ha ricevuto nel momento in cui non era online. Questa coda ha una dimensione massima che viene decisa all'interno del file di configurazione.
  Quando l'utente richiede di visionare i precedenti messaggi, questi vengono scaricati e tolti dalla coda. Se il server termina prima che la

coda sia stata svuotata allora viene avviata una procedura che va a svuotarla per liberare la memoria occupata.

- **L’ID** è l’identificativo che associamo al client e che utilizziamo quando è necessaria la mutua esclusione sulle scritture che fa il server verso un client.

Statistiche : Struttura dati in cui vengono salvate tutte le statistiche relative al server in esecuzione.

Configurazione : Struttura dati in cui vengono salvati tutti i dettagli relativi alla configurazione del server.

Coda_new : La coda in cui inserisco i file descriptor delle nuove connessioni in entrata che vengono accettate dal server. I vari thread poi estraggono da questa coda il file descriptor in modo da effettuare tutte le operazioni richieste.

4. Gestione dei thread e della mutua esclusione

4.1 Thread

All’interno del server vengono utilizzati i seguenti thread:

- **Thread main:** si occupa di accettare le connessioni dei vari client.
  Per accettare le connessioni viene utilizzata la select.
  All’arrivo di un nuovo client controlliamo per prima cosa se abbiamo già superato il massimo numero di client che il server può gestire.
  Se superiamo il limite allora inviamo al client un messaggio di fallimento OP_FAIL, altrimenti inseriamo nella coda condivisa con i thread worker del pool il nuovo File Descriptor.

- **Pool di thread:** formato da un numero di thread definito nel file di configurazione. Ognuno dei thread si occupa di estrarre dalla coda condivisa il file descriptor di una connessione con il client e poi di svolgere la prima operazione richiesta dal client, finita questa rimette il client nella maschera della select.
- Thread per la gestione dei segnali

4.2 Mutua Esclusione

- **Mtx_registrati** : Si tratta di un array di mutex con cui gestiamo l'accesso e la modifica degli elementi presenti all'interno della tabella hash degli utenti registrati.
  Si è scelto di utilizzare un array di mutex e non una singola mutex per tutta la struttura per far in modo che contemporaneamente i thread worker possano accedere ad aree diverse della tabella hash senza doversi necessariamente fermare garantendo in questo modo un parallelismo dei lavori effettuati.
- **Mtx_write** : Si tratta di un array di mutex (il numero degli elementi è minore dell'Mtx_registrati) con cui gestiamo l'invio dei messaggi dal server ai client registrati.
  È stato necessario inserire queste mutex perché altrimenti capitava che il server inviasse nello stesso momento due messaggi allo stesso client causando la perdita di uno dei due.
  Per la mutua esclusione sulle scritture ho utilizzando un identificativo per ogni utente registrato in modo da eseguire la lock su:

  mtx_write[id_client%numero_di_mutex_write]

- **mtxMain**: Mutex utilizzata per garantire la mutua esclusione nel momento in cui devo andare ad inserire e a leggere un file descriptor contenuto nella coda condivisa tra il main e i thread worker.
- **MtxStats**: Mutex utilizzata per garantire la mutua esclusione nel momento in cui devo andare a modificare o a leggere la struttura dati delle statistiche.

5. Gestione dei segnali

Per gestire i segnali viene utilizzato un thread che rimane sempre attivo eseguendo la funzione sigwait.
Ho scelto di utilizzare un thread perché in questo modo si possono utilizzare anche funzioni non signal safe e perché è possibile accedere con più facilità alle variabili globali.

Quando avvio il server maschero SIGPIPE per evitare di rimanere bloccato nel caso in cui dovesse essere effettuata una scrittura su un socket chiuso.
Poi nel thread vengono gestiti:

- **SIGUSR1**: Il segnale che permette la creazione e il riempimento di un file di “log” in cui salvare i dati di utilizzo del server
- **SIGTERM**, **SIGQUIT** e **SIGINT**: i tre segnali che vanno a terminare il server.
  Nel momento in cui arriva un segnale di questo genere viene settata a 1 la variabile avvio che controlla il while presente nel main e quindi non sono più accettate nuove connessioni, tutti i thread worker vengono risvegliati e anche loro terminano.
  Prima di terminare il server viene liberata tutta la memoria occupata..

6. Modifiche apportate al file icl_hash.c / icl_hash.h

Il file icl_hash per la gestione delle tabelle hash è stato modificato per introdurre alcune funzioni utili:

- void * icl_hash_updatefd(icl_hash_t * ht,int fd, void * key);
- int icl_hash_update_id(icl_hash_t *ht, void * data, int id);
- int icl_add_message(icl_hash_t *ht, void * key, message_t msg);
- int isOnline(icl_hash_t *ht, void * key);
- int goOnline(icl_hash_t *ht, void * key, int fd);
- int goOffline(icl_hash_t *ht, void * key);
- int isRegistrato(icl_hash_t *ht, void * key);
- int icl_get_id(icl_hash_t *ht, char * user);

Il funzionamento delle procedure è spiegato all’interno dei due file icl_hash.

7. Possibile implementazione dei gruppi

Avendo consegnato il progetto per l’appello di Luglio non ho implementato la gestione dei gruppi, ho comunque pensato al modo in cui realizzarlo e avevo anche iniziato a scrivere le operazioni di creazione ed eliminazione.

Per gestire tutti i gruppi che gli utenti possono creare avevo ipotizzato di produrre una lista di strutture dati in cui ogni elemento identifica il singolo gruppo.

La struttura del singolo elemento della lista di gruppi avrebbe avuto i seguenti campi:

- Nome del creatore del gruppo
- Nome del gruppo
- Lista di utenti del gruppo, in questo caso avevo inizialmente pensato di settare un limite massimo di utenti che possono essere inseriti in un gruppo poi invece ho scelto di usare una lista in modo da non avere questo limite.