RELAZIONE PROGETTO

SISTEMI OPERATIVI E LABORATORIO :

CHATTY

AUTORE: STEFANO RUSSO

1 SCELTE PROGETTUALI

- Strutture dati
- Gestione della memoria

2 STRUTTURAZIONE DEL CODICE

- Divisione in file

3 PROTOCOLLI DI COMUNICAZIONE

- Terminazione
- Gestione segnali

4 NOTE

1 SCELTE PROGETTUALI

1.1 STRUTTURE DATI

Le strutture dati utilizzate sono:

- **Hash table** : Per la memorizzazione degli utenti e dei messaggi ad essi destinati. Vengono utilizzati due hash table, che presentano comportamenti diversi tra loro. Per quanto riguarda l'hash table degli utenti registrati , viene memorizzato il nickname dell'utente, a cui viene associato il file descriptor attraverso il quale è in comunicazione con il server. L' hash table dei registrati è realizzata con il metodo delle liste di trabocco, ognuna delle quali è sorvegliata da una mutex, che permette un accesso sicuro, bloccando però l'accesso contemporaneo ad elementi diversi ma con un valore hash uguale. Il calcolo del valore hash avviene sommando il valore ASCII dei singoli caratteri della stringa data, e calcolando il resto della divisione del numero ottenuto per il numero di celle massime che l'hash table supporta. L'hash table dei registrati usa anche una variabile che tiene conto del loro numero , sorvegliato da una mutex a parte (**htr_lock**).

  La seconda hash table invece memorizza la history dei messaggi che ogni utente riceve. In questa struttura vengono memorizzati il nome del mittente, del destinatario ed il messaggio ad esso destinato. La sua funzione hash lavora in modo differente, per evitare che due utenti diversi vengano mappati nella stessa cella. L'accesso all'intera hash table viene sorvegliato da un'unica mutex (**htm_lock**) tranne che per la cancellazione dei messaggi di un determinato utente, dove viene utilizzata la mutex relativa all'indice in cui l'utente è stato mappato.

  Il numero massimo di celle che le hash table possono avere, è dato dalla variabile globale **MAX_CELLE** che ha un valore di 5021.

- **Connessi**: Una struttura composta da due array, una contenente i nickname di tutti gli utenti in quel momento connessi, l'altra contenente i file descriptor

relativi agli utenti. L'accesso alla struttura è controllato da una singola mutex (**conn_lock**).

- **MSGS**: Un array globale di tipo **message_t** (definito nel file **message.h**), che memorizza temporaneamente i messaggi, in attesa di essere utilizzati da uno dei thread del threadpool. La struttura fornita di base è stata modificata per tenere il conto del numero di messaggi inseriti nella struttura stessa, e nel caso in cui si arrivi alla fine dell'array, se il numero di messaggi inseriti non è uguale al numero massimo di messaggi, viene riportato a 0 l'indice che eseguirà un ulteriore scansione dell'array, controllando il valore di op dell'header del messaggio e comparandolo con il valore che indica la possibilità di poter riutilizzare quel blocco, tale è indicato su **ops.h** (33).

1.2 GESTIONE DELLA MEMORIA

Le strutture dati utilizzate vengono inizializzate dal **main** in fase di avvio, le stesse saranno poi deallocate alla terminazione in seguito alla ricezione di un apposito segnale, dal **main** stesso . Per gestire la liberazione di buffer inizializzati e utilizzati dalle varie funzioni, vengono salvati i relativi puntatori in un array (**backup**) dichiarato globalmente nel file **connections.h** e sorvegliato da una mutex, sovrascrivendovi iniziando dall'indice 0 qualora si dovesse riempire l'intero rarray, gli elementi dell'array rimasti, vengono liberati alla terminazione insieme al resto delle strutture dati.

2 STRUTTURAZIONE DEL CODICE

2.1 DIVISIONE IN FILE

Il grosso del codice è contenuto nel file **listenerworker.c** , che contiene le funzioni **th_Listener** e **Gestione** che sono le funzioni eseguite rispettivamente dal thread Listener, che viene creato nel main, e resta in ascolto delle richieste da parte dei client e attraverso una select inserisce le richieste, nella struttura threadpool, che le memorizza in una coda, dove saranno soddisfate dai thread del pool, e l'altra eseguita dai thread che soddisfano le richieste nella coda. La coda è definita come campo della struttura **threadpool_t** in **threadpool.h ,** che è a sua volta una struttura contenente un file descriptor , una funzione ed un indice per l'array **MSGS** , che verranno settati dal thread Listener al momento di creazione e aggiunta del task, richiesto da un determinato client, nella coda.

Le varie funzioni di supporto sono divise tra i file : **hash_tables.c , ops.c , connections.c** , **connessi.c** , **threadpool.c** , **parsing_config.c** , **message.h** e **stats.h**

- **hash_tables.c** : Contiene la definizione della struttura hash table e le varie funzioni ad esso relative : creazione, distruzione, calcolo valore hash , inserimento e cancellazione.
- **ops.c** : Contiene le funzioni chiamate per l'esecuzione delle richieste da parte dei client
- **connections.c** : Contiene le funzioni relative alla scrittura e lettura da socket per le comunicazioni tra client e server.
- **connessi.c** : Contiene la definizione della struttura connessi e le funzioni ad esso relative
- **threadpool.c** : Contiene la definizione del pool di thread e le funzioni ad esso relative : creazione , distruzione , inserimento e gestione dei thread.
- **parsing_config.c** : Contiene la funzione di parsing utilizzata per ottenere dal file passato come argomento, la configurazione da utilizzare.
- **message.h** : Contiene la definizione delle strutture che compongono un messaggio , e le funzioni per settarne i vari campi.
- **stats.h** : Contiene le funzioni di stampa e gestione delle statistiche del server.

Il file **chatty.c** contiene l'inizializzazione di variabili globali, e la funzione main che si occupa di inizializzare le strutture dati, spownare il thread Listener e mettersi in attesa di ricevere un segnale che verrà gestito dalla funzione **sig_handler** definita nello stesso file.

Il file **config.h** contiene la maggior parte delle **#define** utilizzate nel codice.

Il file **extern.h** contiene le definizioni delle variabili globali esterne utilizzate da più file .

3 PROTOCOLLI DI COMUNICAZIONE

3.1 TERMINAZIONE

Per la terminazione è stata implementato un thread che viene creato nel momento in cui arriva un segnale di terminazione, eseguendo la funzione **ending** , definita in **chatty.c** , che funge da client e scrive al server un messaggio a cui il thread Listener risponde chiudendo tutte le connessioni aperte e terminandosi. I thread che nello stesso momento si trovano ad eseguire un comando, terminano automaticamente se trovano settata la variabile **segnale_exit**. Viene infine chiamata la funzione di distruzione del threadpool che risveglia tutti i thread in attesa in modo che possano terminare.

3.2 GESTIONE DEI SEGNALI

I segnali sono gestiti tramite una variabile globale inizializzata nel file **chatty.c** (**segnale_exit**) , che viene settato con il valore del segnale ricevuto. E' stato settato un handler apposito per i segnali **SIGINT** , **SIGQUIT** , **SIGTERM** e **SIGUSR1** ,che inserisce il valore del segnale ricevuto nella variabile **segnale_exit**. Verranno stampati i valori delle statistiche del server nel caso in cui **segnale_exit** sia settato con valore **SIGUSR1**, si procederà alla terminazione del programma nel caso di **SIGINT** , **SIGQUIT** e **SIGTERM**. Sono inoltre stati ignorati alcuni segnali quali **SIGPIPE** , **ECONNRESET** , **ECONNREFUSED** ed **EPIPE** , per evitare la chiusura dei socket di comunicazione.

4 NOTE

Il programma è stato testato sulla macchina virtuale fornita durante il corso (xubuntu).

Il Test 5 (**teststress.sh**) nelle righe 59 e 60, i comandi -C chiamati da utente 1 e 2, provocano un errore con il client (opzione -k non specificata), portando cosi' gli utenti che dovrebbero fallire nel mandare il messaggio "ciao" a mandarlo invece con successo.