UNIVERSITÀ DI PISA

ChatterBox

Progetto per il modulo di Laboratorio SOL a.a. 2017/18

Jacopo Massa
Matricola 543870
Laboratorio di Sistemi Operativi - Corso A
Dipartimento di Informatica, Università di Pisa

email: jacopomassa97@gmail.com

Indice

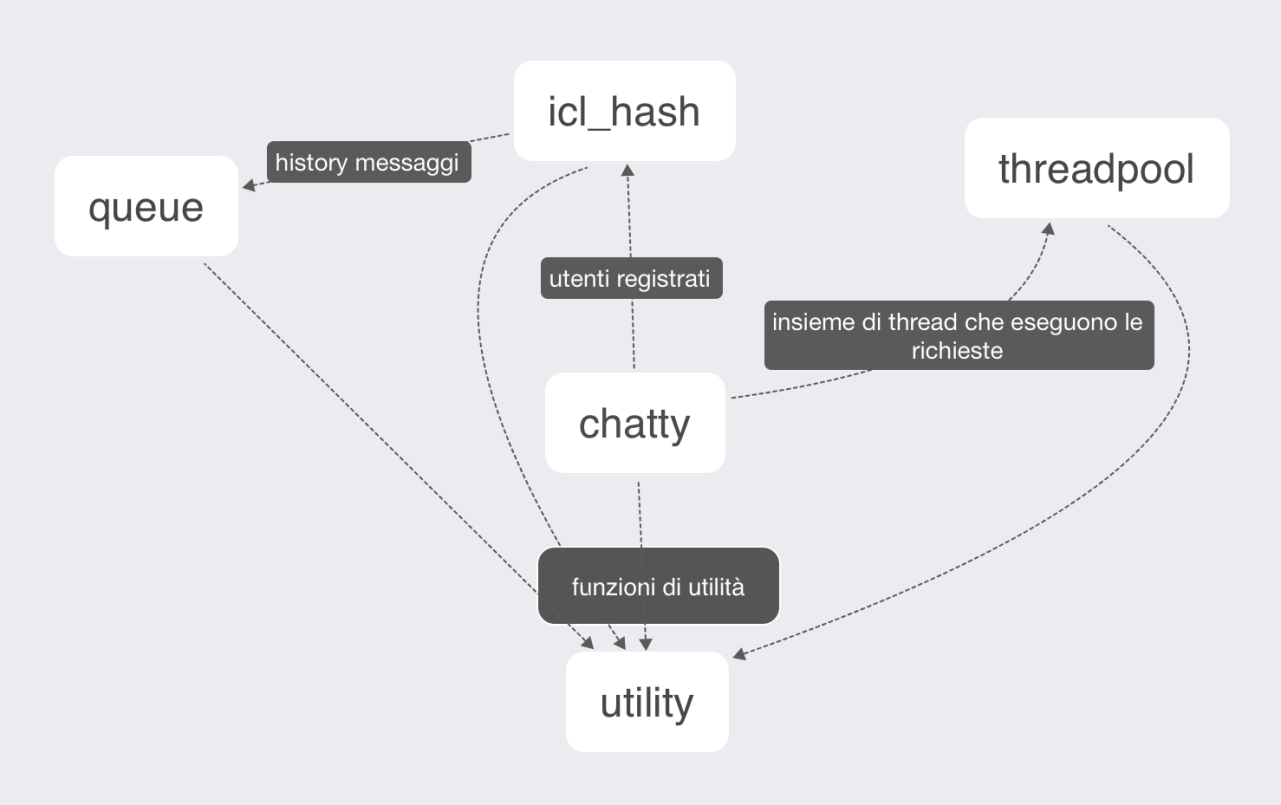
Strutturazione del codice

Non è stato fatto uso di ulteriori librerie, eccetto quella già “impostata” dai docenti tramite Makefile, a cui sono stati aggiunti gli altri file creati in fase di implementazione.
La suddivisione in più moduli è stata fatta in base alle diverse strutture dati o funzioni implementate. Mantenendo fede al criterio seguito durante il corso, ogni modulo prevede:

- un header con i prototipi delle funzioni (eccetto i metodi `static`) e, dove presenti, le dichiarazioni di strutture (e la loro documentazione).
- un file con l’implementazione delle funzioni contenute nel rispettivo header.

I moduli principali (aggiunti a quelli forniti dai docenti) corrispondono alle strutture dati implementate; in aggiunta a questi, si trovano:

- modulo **utility**, contenente varie funzioni e macro utilizzate all’interno dell’intero progetto;
- modulo **chatty**, contenente le funzioni eseguite dai thread del server (listener, gestore di segnali, e threads del pool) per soddisfare le richieste dei client.



Schema riassuntivo dei moduli implementati (sono omessi quelli forniti dai docenti).

Strutture dati

Oltre a quelle fornite dai docenti, è stata necessaria la progettazione e lo sviluppo di altre strutture dati, di seguito elencate:

- **QUEUE**, una coda circolare implementata tramite una lista, usata per rappresentare la *history* dei messaggi di ogni utente. Una volta raggiunta la dimensione massima (prevista dalla configurazione del server) l'inserimento di nuovi elementi ricomincia dalla testa della coda, ed eventuali vecchi elementi vengono sovrascritti. Questa struttura non prevede l'accesso in mutua esclusione.

- **CONFIG**, struttura contenente i parametri di configurazione del server, inizializzata tramite la lettura del file specificato all'avvio del server.

- **THREADPOOL**, insieme di thread che eseguono una serie di *task* contenuti all'interno di una coda condivisa (non implementata con **QUEUE**) di lunghezza fissa. L'accesso alla coda da parte dei thread è in mutua esclusione.

- **ICL_HASH**, una hash table con un numero di partizioni deciso al momento della creazione, utilizzata per salvare le informazioni e lo stato degli utenti registrati. Le chiamate ai metodi di accesso alla hash table non sono in mutua esclusione.
  Questi sono i valori dei parametri della hash table, scelti per il progetto:
  - **n° di partizioni = n° di thread contenuti nel pool**, in modo tale che (al caso ottimo) ogni thread può lavorare su un utente di una partizione in maniera concorrente agli altri thread. Un numero minore di partizioni avrebbe portato poca efficienza nell'uso del multithreading; d'altro canto un numero maggiore avrebbe aumentato i tempi di accesso alla hash table.
  - **grandezza della hash table = 1024 buckets.** Tale scelta è atta a garantire in qualsiasi situazione (pochi/molti utenti) un buon fattore di carico.

L'implementazione di ogni struttura dati prevede, oltre alle classiche funzionalità per la loro gestione (inserimento, modifica, lettura…) anche dei metodi che permettono di creare, ma soprattutto **distruggere** le strutture dati stesse.

Gestione della memoria

Le strutture dati non sono le uniche ad occupare memoria che dovrà essere liberata. Altre allocazioni di variabili nel corso dell'uso di ChatterBox, vengono sempre seguite da una *free()*, ovvero un'azione di deallocazione di memoria. Inoltre, all'inizio dell'esecuzione del server vengono registrate due funzioni di pulizia della memoria (vedi il file *chatty.c*) che verrano chiamate nel caso di terminazione del server (che sia essa causata da un errore o meno):

- `unlinksocket()`, elimina eventuali precedenti istanze del pocket di comunicazione utilizzato dal server per parlare con i client
- `cleanup()`, libera tutta la memoria allocata fino al punto in cui la funzione viene chiamata

Così facendo ci sia assicura di rendere nuovamente disponibile la memoria utilizzata dal server dopo il suo utilizzo.

Accesso ai files

L'accesso ad ogni file con cui ChatterBox ha a che fare - a partire da quello per la configurazione del server, fino a quelli mandati / richiesti dagli utenti della chat - avviene usando le funzioni standard di libreria bufferizzate *fread()* e *fwrite()*, scelte non tanto per la compatibilità tra sistemi operativi, quanto per la velocità ed efficienza con cui vengono eseguite rispetto alle system call *read* e *write*.

Mutua Esclusione e Concorrenza

Per implementare la mutua esclusione (non fornita da tutte le strutture dati), si fa uso di variabili mutex ogniqualvolta si accede (in lettura o scrittura) ad una struttura condivisa. Nello specifico sono presenti:

- **una mutex per ogni partizione** della hash table;
- **32 mutex per il canale di comunicazione** tra i client e il ThreadPool (il socket). 32 mutex sono state reputate sufficienti per avere una certa velocità di risposta, nonostante la concorrenza tra thread;
- **una variabile di condizione nel ThreadPool**, che serve a segnalare ai thread la presenza di un nuovo task da eseguire, o a far terminare tutto i ThreadPool.

Architettura del server

Alla base di ChatterBox c'è il **multithreading**, quindi si è deciso di assegnare ogni aspetto cruciale della gestione del server, ad un thread o componente diverso:

- il **socket** è il mezzo tramite il quale gli utenti comunicano tra loro attraverso il server.
- il **ListenerThread** è quello che gestisce la connessione da parte degli utenti.
- il **SignalManagerThread**, come suggerisce il nome, è quello che si occupa della gestione dei segnali (vedi sez. Gestione dei segnali);
- il **ThreadPool** è l'insieme di thread che eseguono le richieste ricevute dai client.
- una **pipe bidirezionale** funge da canale tra il ThreadPool e il ListenerThread;

Protocollo di comunicazione

1. Il ListenerThread cattura una richiesta da parte di un client e:
    - se è una nuova richiesta di connessione
        - assegna a quel client un descrittore;
        - aggiunge il client tra quelli da "ascoltare";
    - se invece, il client voleva notificare al server una scrittura sul socket
        - legge la richiesta del client e la aggiunge alla coda condivisa del ThreadPool;
        - elimina il descrittore del client da quelli da "ascoltare".

2. Segnalata la presenza di un *task*, uno dei thread del pool lo prende in carico, eseguendolo e rispondendo al client tramite il socket.
   Il protocollo di comunicazione implementato prevede l'invio di un header contenente l'esito della richiesta (fallimento / successo).
   In caso di successo, se sono previsti dati, questi vengono mandati con una seconda scrittura sul socket.

3. Indipendentemente dall'esito della richiesta, il thread che l'ha terminata segnala il fatto al ListenerThread, inserendo nella pipe il descrittore associato al client che aveva fatto la richiesta.

4. Il ListenerThread cattura una lettura sulla pipe, e comincia a riascoltare il client con il descrittore letto.

5. La comunicazione con un client viene interrotta quando esso non scrive più nulla sul socket (system call read torna 0), oppure quando il socket viene chiuso dal client (errore catturato sempre con la system call read).

6. Il server termina la sua esecuzione alla ricezione di uno dei segnali SIGTERM, SIGINT, SIGQUIT (vedi sez. Gestione dei segnali e Protocollo di Terminazione), oppure in caso di errore.

Gestione dei segnali

Il server può ricevere un ristretto gruppo di segnali, che sono mascherati a tutti i thread in esecuzione, in modo tale che la loro gestione sia assegnata esclusivamente al SignalManagerThread.
Proprio da questo thread parte la terminazione del server richiesta dall'utente, che viene descritta nella sezione successiva.

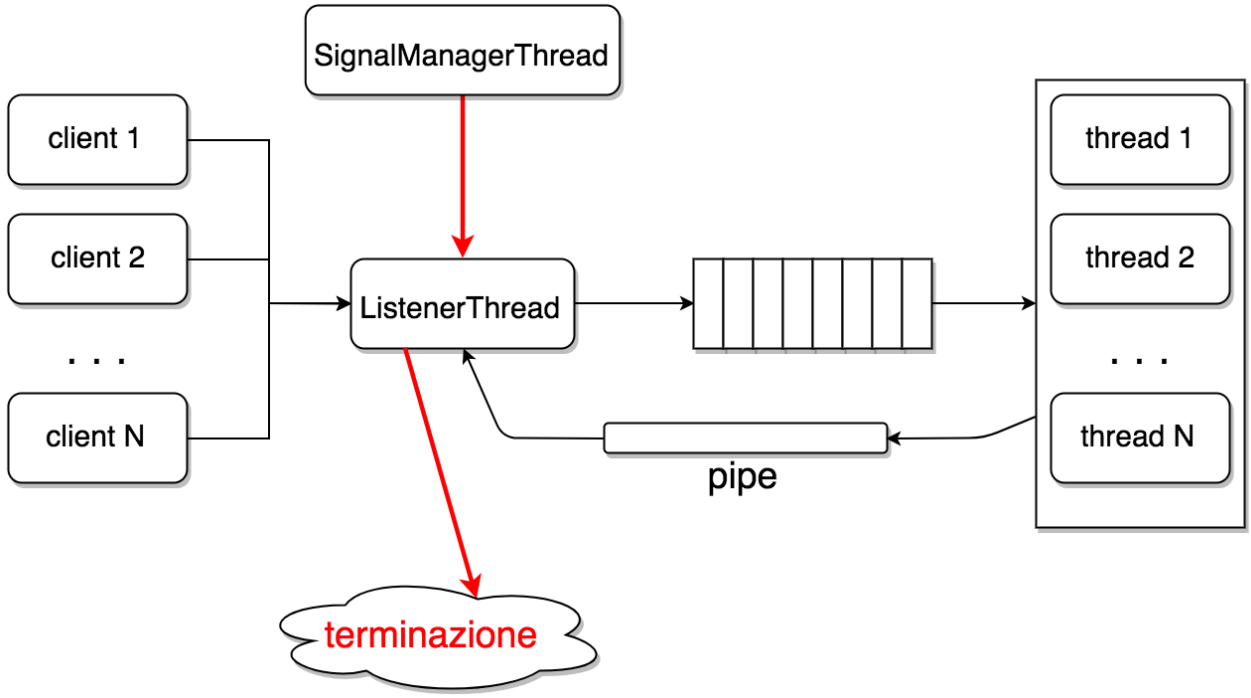
Protocollo di terminazione

Si è scelto di attendere la fine dell'esecuzione del ListenerThread, che viene "ucciso" dal SignalManagerThread, quando questo riceve un segnale atto a terminare il server.
Per rendere ciò possibile, è stato cambiato il tipo di cancellazione del ListenerThread da "deferred" (quella di default per i nuovi thread) ad "asynchrounous". Ciò permette di terminare il thread in qualsiasi momento, senza dover prima arrivare ad un *cancellation point*.
Quando il SignalManagerThread "uccide" il ListenerThread, il controllo ritorna nel main dove (tramite una `join`) si attendeva la fine del ListenerThread. Subito dopo seguono le operazioni di liberazione della memoria, chiamate dopo il `return` del main.

Il server può terminare anche quando avviene un errore.
Le principali cause sono:
- mancanza di memoria (errore della malloc)
- system call (read, write, fread, fwrite).

Alla pagina seguente si trova uno schema illustrativo del funzionamento di ChatterBox.

Schema riassuntivo



Note

- Coerentemente al concetto di “history”, la coda dei messaggi pendenti di ogni utente non viene eliminata fino alla sua deregistrazione, o fino alla terminazione del server.

- Il ThreadPool usato all’interno del progetto è una reimplementazione di un pool di thread generico sviluppato da Mathias Brossard. La versione originale è reperibile qui.

- Le scelte dei valori massimali (vedi file *config.h*) e di size delle strutture sono stati fatti nell’ottica di far funzionare il server con migliaia di utenti e quindi, decine di migliaia di richieste.

- Il server risulta funzionante sia sulla macchina virtuale *Xubuntu* fornita dai docenti, sia su macchina virtuale *Ubuntu 16.04*, con prestazioni pressoché identiche.

- Il progetto è completo di documentazione, generata tramite *Doxygen*, e consultabile aprendo il file `index.html`, nella sottocartella `doc`.
  Inoltre, all’interno di ogni file sono commentate le parti di codice principali o passaggi poco chiari.