RELAZIONE PROGETTO CHATTY LABORATORIO SISTEMI OPERATIVI

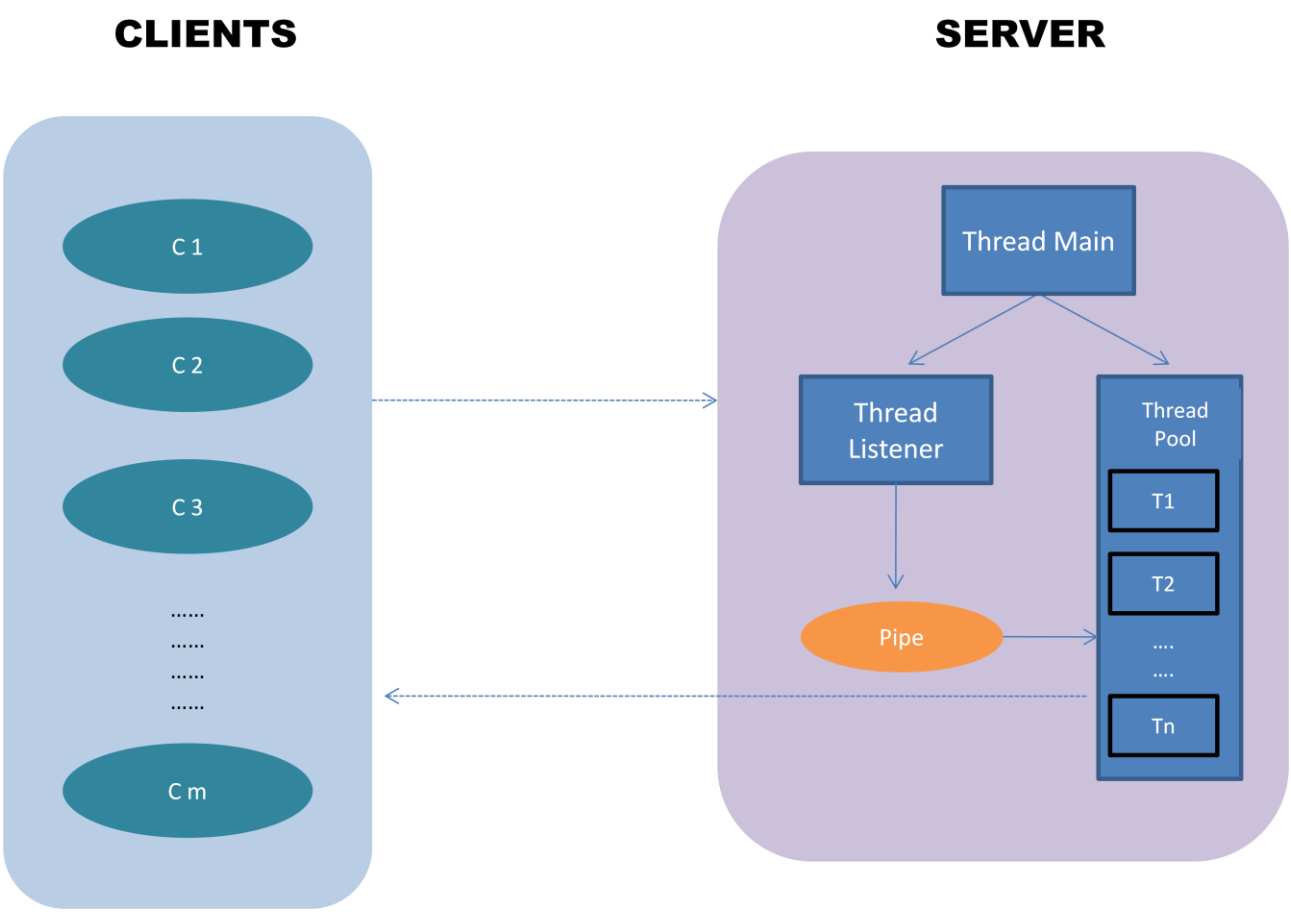
Andrea Pelosi

Sommario

1. ARCHITETTURA DEL SISTEMA

L'applicazione è costruita con architettura Client-Server.
Come richiesto dalle specifiche del progetto, il server in questione non è sequenziale ma è multithreaded, al fine di gestire parallelamente più richieste garantendo performance elevate e la capacità di gestire molti clients contemporaneamente.

Di seguito è disponibile uno schema architetturale esemplificativo dell'applicazione.
Nel paragrafo successivo viene fornita una panoramica delle funzionalità dei threads che emergono da tale schema e dei meccanismi di comunicazione tra i threads stessi e i clients.



2. FUNZIONALITA' DEI THREADS

Il processo server oltre ad avere un thread Main, possiede un thread Listener e un threadPool.

- **Thread Main**: installa il gestore dei segnali, effettua il parsing del file di configurazione, crea e associa al path designato il socket per la comunicazione coi clients, inizializza le strutture dati necessarie, avvia thread Listener e threadPool e prima del termine dell'esecuzione del server si occupa di liberare la memoria dinamica allocata.

- **Thread Listener**: è sempre in ascolto tramite una select di nuovi clients che vogliono connettersi e di clients già connessi in precedenza, per i quali si attende che il File Descriptor a loro associato sia pronto per operazioni di lettura. Se l'operazione di selezione ha successo e ci sono FD pronti in lettura, scrive tali FD in una pipe.

- **ThreadPool**: ogni thread del ThreadPool attende di leggere sulla pipe un FD. Una volta ottenuto tale FD, si occupa di leggere la prima richiesta del client relativo a questo FD e di elaborarla; se tale operazione ha successo il FD viene rimesso nel set di ascolto del thread Listener e verrà selezionato nuovamente, scritto nella pipe ed estratto da un altro thread del Pool per la lettura della seconda richiesta (analogamente avviene per tutte le richieste di un client). Ogni thread si occupa dunque di una sola richiesta alla volta. Sarà direttamente il thread del Pool a comunicare al client l'esito della richiesta dopo averla elaborata.

3. COMUNICAZIONE INTER PROCESSO E FRA THREADS

La comunicazione tra il server e i clients avviene mediante una socket con dominio AF_UNIX, ovvero tra processi su una stessa macchina. Client e server si scambiano messaggi definiti dalla struttura *Message*, per maggiori informazioni su tale struttura consultare la sezione relativa alle strutture dati.

I vari threads all'interno del server utilizzano meccanismi di comunicazone, in particolare thread Listener e thread del ThreadPool comunicano tramite una pipe senza nome.

4. GESTIONE TERMINAZIONE E SEGNALI

All'avvio del server viene installato un signal handler per tutto il processo. In particolare il segnale SIGPIPE viene ignorato, i segnali SIGINT, SIGTERM, SIGQUIT e SIGUSR1 vengono gestiti come richiesto dalla specifica del progetto.

La terminazione del server avviene solamente a seguito di ricezione di uno dei segnali tra SIGINT, SIGTERM e SIGQUIT. Quando uno di questi segnali viene ricevuto, viene eseguito un *Graceful ShutDown*: il thread Listener esce dal ciclo while e prima di terminare chiude i FD che potrebbero essere rimasti aperti, i threads del ThreadPool escono dal ciclo while e il thread Main, dopo aver effettuato le join dei threads creati, libera la memoria dinamica allocata e l'applicazione termina.
Si noti che alla variabile che ogni volta è controllata nella guardia dei cicli è stato aggiunto il qualificatore *volatile*, in modo che eventuali ottimizzazioni del codice ad opera del compilatore non rendano l'applicazione inaffidabile sostituendo a tale variabile una guardia sempre vera.

All'arrivo di SIGUSR1, Chatty stampa le statistiche del server come richiesto.

5. GESTIONE CONCORRENZA

Essendo il server multithreaded, si è resa necessaria una gestione della concorrenza per evitare stallo oppure situazioni inconsistenti sui dati che il server gestisce.

La comunicazione tra thread Listener e threadPool è un'istanza del ben noto problema Produttore-Consumatore. In questo caso abbiamo un produttore (il Listener) e più consumatori (threads del Pool). Per la gestione di tale problema ho utilizzato una lock e una variabile di condizione: il produttore scrive un FD nella pipe e con una signal risveglia uno dei consumatori in wait sulla variabile di condizione *cond_pipe.* Il consumatore legge dalla pipe un FD e lo usa per le sue operazioni; se non c'è nulla da leggere, si mette in attesa su *cond_pipe*.

In ogni struttura dati (quando è risorsa condivisa) è stata presa in considerazione la gestione della concorrenza; maggiori dettagli sono forniti nella sezione delle strutture dati utilizzate. In linea generale ho utilizzato solamente lock e variabili di condizione necessarie per evitare inconsistenze nei dati cercando però di tenere l'overhead dovuto alla gestione della concorrenza il più basso possibile.

6. STRUTTURE DATI UTILIZZATE

Di seguito le principali strutture dati usate.

- **HashTable (utenti registrati):** E' usata per la gestione degli utenti registrati. Alla chiave logica Utente è associato il valore Userdata relativo a quell'utente. Per l'implementazione mi sono servito della tabella hash *icl_hash* suggerita a lezione, scegliendo come dimensione della tabella l'intero 1024, sempre suggerito a lezione. Per rendere possibile un accesso concorrente alla HashTable, ho allocato un array di locks di dimensione (dim. Tabella)/(num. Max di connessioni consentite) in modo che un thread ottenga accesso in mutua esclusione solo alla sezione che gli

interessa. Un thread locka una sezione e non un singolo bucket per evitare eccessivo overhead ad ogni operazione sulla tabella.

- **Lista (utenti connessi):** E' usata per tenere traccia degli utenti connessi in un dato momento a Chatty. Si ricorda che un utente è considerato connesso solo fino a quando una delle sue richieste è in fase di elaborazione. La disconnessione da parte del client (e la seguente rimozione dalla lista) non avviene esplicitamente con una richiesta ma implicitamente quando la read sul FD relativo a quel client non ha successo. Quando è richiesta un'operazione sulla lista, essa viene lockata e unlockata al termine dell'operazione.

- **Set(FD attivi):** Insieme di File Descriptor attivi. Essendo una risorsa condivisa, un thread lo può modificare se necessario previa acquisizione di una lock dedicata.

- **Conf_values:** Struttura che contiene i valori necessari per la configurazione del server, viene aggiornata in fase di avvio dopo aver parsato il file di configurazione.

- **Message:** Struttura per la codifica di un messaggio per la comunicazione tra server e clients. E' costituita da Header (contenente mittente del messaggio e tipo di operazione) e Data (contenente destinatario, lunghezza buffer dati e buffer dati).

- **Statistics:** Struttura che memorizza le statistiche del server. E' una risorsa condivisa, un thread può aggiornare una o più statistiche dopo aver ottenuto la lock.

- **Userdata:** Struttura che è associata ad ogni utente registrato a Chatty. Principalmente vi son memorizzati username dell'utente, FD relativo all'ultima comunicazione col client dell'utente, e history dei messaggi ricevuti. Più threads potrebbero accedere concorrentemente alla struttura dati, pertanto anche in questo caso l'accesso avviene solo dopo aver acquisito la lock relativa a Userdata. Per ulteriori dettagli sul funzionamento dello storico dei messaggi di un utente, consultare la sezione 10.

7. DIVISIONE IN FILE

Poiché l'applicazione consta di numerose funzionalità, la scrittura del codice è stata suddivisa in diversi file in modo da semplificarne la leggibilità e gestione. In particolare vi sono file scritti per:

- effettuare parsing del file di configurazione
- implementare protocollo di comunicazione tra clients e server
- implementare la tabella hash degli utenti registrati
- implementare la lista degli utenti connessi
- controllare sistematicamente la correttezza delle SC (si guardi sezione 10)
- gestire i dati relativi agli utenti
- gestire stampa delle statistiche
- gestire messaggi e tipo delle operazioni dei messaggi

Il file chatty.c raccoglie tutti i moduli e fornisce le funzionalità richieste.

Per analizzare il dettaglio dei file, delle strutture dati con i relativi campi è possibile consultare la documentazione generata con l'applicazione *Doxygen*.
Tale documentazione è presente in formato .html nella cartella documentazione_doxygen.

8. GESTIONE DELLA MEMORIA

Per evitare operazioni su memoria "non pulita" dopo ogni operazione di allocazione di memoria, si effettua una fill dello spazio allocato con il byte 0 tramite la funzione memset.

Tra le varie funzioni implementate vi sono funzioni per ripulire la memoria delle principali strutture dati utilizzate alla chiusura del server.

9. ACCESSO AI FILES

Oltre che in fase di ricezione e invio di file richiesti dai clients, il server accede ai files all'avvio e alla ricezione del segnale SIGUSR1.
In fase di avvio effettua un parsing del file passato come argomento per ottenere informazioni indispensabili alla configurazione del server.
Quando riceve il segnale SIGUSR1 il server stampa su file le statistiche raccolte fino al momento della ricezione del segnale.

10. ULTERIORI SCELTE PROGETTUALI E CHIARIFICAZIONI

Per il controllo sistematico delle System Calls sono state scritte delle macro che si trovano negli header *macrosctest.h* e *macrothread.h* . In caso di fallimento della funzione chiamata si stampa su STDERR l'errore.

Lo storico messaggi relativo ad un utente memorizza al più *MaxHistMsgs* messaggi, ovvero il massimo numero di messaggi che si possono trovare in uno storico (informazione reperita dal file di configurazione).
All'arrivo di un nuovo messaggio da memorizzare, tale messaggio viene aggiunto allo storico e si memorizza anche se esso è stato letto dall'utente oppure no. Quando lo storico di un utente è pieno, se arriva un nuovo messaggio si possono verificare due situazioni:

- L'utente ha letto uno o più messaggi dello storico; in tal caso uno di questi messaggi viene sovrascritto e l'operazione ha successo.

- Tutti i messaggi nello storico sono ancora da leggere; in tal caso il messaggio non viene aggiunto allo storico, il server stampa a video che un inserimento nella history

è fallito. Il FD del mittente del messaggio viene ascoltato nuovamente poiché il fatto che l'operazione non sia andata a buon fine non dipende da un suo comportamento scorretto.

11. TESTING E DEBUGGING

L'applicazione è stata sviluppata e testata su una VM con Linux distribuzione Ubuntu Release 18.10. Si è verificato anche che i test fossero funzionanti sulla VM utilizzata per il corso, distribuzione Xubuntu.
Il debugging è stato per la maggior parte effettuato usando il software GDB e il software Valgrind-Memcheck per l'analisi della memoria dinamica.

12. DIFFICOLTA' INCONTRATE

Le maggiori difficoltà nello sviluppo del progetto sono state rilevate nel superamento del test 4. In particolare non è stato semplice eliminare alcuni leak anche a causa di una scarsa leggibilità degli errori segnalati da Valgrind.