CHATTERBOX

di Fabio Catinella, Matricola: 517665, Corso: B

INTRODUZIONE

Chatterbox è una classica chat tra più utenti. Un utente invia un messaggio e il server lo inoltra al diretto interessato (possono essere anche più d'uno). Ometterò in questa relazione i comandi che chatty soddisfa (POSTTXT_OP etc…) in quanto forniti dai docenti; mi concentrerò invece sulle scelte progettuali fatte per realizzarlo.

Funzionamento generale

Il server appena viene avviato, cerca il file.conf da cui estrarre i parametri di configurazione. Una volta configurato si mette ad ascoltare sul socket definito nel file di configurazione; quando una richiesta arriva da parte di un client la fa servire ad uno dei suoi thread worker che esaudiranno la richiesta e poi si metteranno in attesa di altri comandi. Se il ricevente di un messaggio è offline, quest'ultimo viene scritto nella sua history dei messaggi ricevuti, fino ad un massimo stabilito dal file di configurazione. Alla ricezione di SIGINT il server termina, assicurandosi di aver terminato anche i thread worker e di aver pulito la memoria. Se invece arriva un SIGUSR1 il thread il server scrive le statistiche delle operazioni fatte fino a quel punto in un file.

Strutture dati usate (strutture.h)

HashElem: la struttura alla base del progetto è una tabella Hash che contiene gli utenti registrati, ha una dimensione di 1024 e gestisce le collisioni usando le liste di trabocco. Ogni utente è rappresentato da una struct contenente molti parametri, in particolare contiene (oltre ad altri):

- isON che indica se un utente è online in quel momento
- channel che indica su quale file descriptor è connesso
- isGroup se l'elemento in tabella hash ha questo valore a 1, allora è un gruppo
- ricevuti lista contenente i messaggi ricevuti mentre era offline

FileDescriptorQueue: è la coda dei file descriptor, è realizzata come array circolare ed è sincronizzata mediante due variabili di condizione, una per l'operazione PUSH ed una per l'operazione PULL. Il loro meccanismo è spiegato più avanti nella sezione *Concorrenza*.

LinkEl: è la struct che ho usato per crearmi la lista degli utenti Online e per i Gruppi. L'elemento principale di questa struct è il puntatore *memberLink* che punta direttamente ad

un elemento della tabella hash. In questo modo si può accedere agli utenti in questa lista passando direttamente da qui senza fare inutili ricerche e controlli per ogni utente della tabella hash.

Le altre strutture sono fondamentalmente liste concatenate.

Funzioni degne di nota

readAll / writeAll: visto la non atomicità della read / write, è stato necessario definire queste due funzioni ausiliari che mantengono traccia di quanti byte letti/scritti in modo da non avere mai letture/scritture incomplete a causa di una deschedulazione al momento sbagliato. Entrambe restituiscono il numero di byte letti/scritti.

crea_lista_utenti: la necessità di una stringa su cui sono scritti tutti gli utenti online mi ha portato alla realizzazione di questa funzione. La modalità di lettura di questa stringa da parte del client mi ha guidato nell'implementazione di *crea_lista_utenti*. Infatti mentre il client legge stringhe come blocchi di MAX_NAME_LENGHT+1, la mia funzione scrive a blocchi anch'essi di MAX_NAME_LENGHT+1; per farlo rialloco molte volte la stringa allungandola del necessario e scrivendo il nuovo nome a partire dalla posizione giusta (multiplo di MAX_NAME_LENGHT+1).

threadOp: funzione che viene eseguita dai worker, ne parlerò dopo con un paragrafo dedicato.

ThreadOp (funzione dei worker)

Questa funzione cicla finché il *clean_bit* è diverso da 1, estrae dalla coda dei file descriptor pronti e legge un messaggio chiamando poi un'altra funzione che esaudirà la richiesta contenuta nel messaggio. Dopo che la prima richiesta è stata soddisfatta threadOp continua a leggere e se i byte che ha letto sono >0 allora soddisfa la richiesta letta, se invece legge 0 byte, e quindi EOF, chiude la connessione e disconnette il mittente dei messaggi precedenti. Quando legge -1 infine, chiude il fd corrispondente in modo che la *accept* possa riassegnarlo ad un altro utente che ne fa richiesta. Appena il *clean_bit* è settato ad 1 esce da ogni ciclo e si chiude. Il modo con cui estrae dalla coda dei file descriptor è sincronizzato tramite variabili di condizione e lock.

Gruppi

I gruppi sono gestiti come utenti speciali della tabella hash, infatti questi "utenti" hanno settato il parametro IsGroup. In base a questo parametro la funzione "Worker" fa cose diverse; per esempio se l'operazione è una POSTTXT e il mittente è un gruppo, il server inoltrerà il messaggio a tutti gli utenti appartenenti a quel gruppo.

CONCORRENZA

Non si può parlare di concorrenza senza parlare di lock, per questo inizio proprio da lì.

Lock

Visto la natura del progetto, e comunque della hash table con liste di trabocco, l'idea di usare una lock singola per ogni utente è stata subito abbandonata; in generale i benefici tratti dal fare una cosa del genere non sono sufficienti per giustificarla. Per questo, su consiglio del professore, ho usato un numero fisso di lock, 32 nel mio caso. Ognuna di esse blocca 32 liste di trabocco. Tutte le volte che devo accedere ad un utente devo bloccare, oltre alla sua, 31 liste di trabocco. Può sembrare poco efficiente ma visto che il carico di utenti connessi contemporaneamente non è mai superiore a qualche decina, questo non impatta molto sulle prestazioni. Per fare in modo che i soliti utenti prendessero sempre le solite lock ho semplicemente aggiunto "%32" alla fine della funzione di hashing, ho dovuto anche fare attenzione a non far prendere due volte la stessa lock dal solito utente.

Invece per le altre strutture condivise ho usato una sola lock per ognuna. In particolare:

- Lock_set (set per la Select)
- Lock_listaOn (lista utenti online)
- Lock_threadAttivi (numero di thread attivi)
- Lock_fdqueue (coda File descriptor)
- Lock_stat (statistiche)

Variabili di condizione

Oltre alle lock ho usato anche due variabili di condizione per sincronizzare la coda dei file descriptor. In particolare il thread main e i vari thread worker possono sospendersi sulle rispettive variabili di condizione :

- Il thread main si sospende su *cond_fd_queue_Push* quando la coda è piena
- I thread worker si sospendono su *cond_fd_queue_Pull* quando la coda è vuota

Le signal vengono inviate dualmente: il main sveglia i worker quando inserisce in coda e viceversa.

Select e File descriptor

Ho usato una select e un array circolare per gestire la base della concorrenza. Tramite la select il thread main sceglie i file descriptor e li mette nella coda. Quest'ultimi saranno estratti dal thread Worker e gestiti. La coda dei file descriptor ha dimensione MaxConnections.

GESTIONE SEGNALI E CONCLUSIONI

Gestione segnali

I segnali gestiti sono SIGUSR1, SIGINT,SIGQUIT e SIGTERM.

SIGINT, SIGQUIT e SIGTERM condividono lo stesso handler, che banalmente setta ad 1 il clean_bit, fa solo questo per rispettare la buona regola di avere handler veloci. Una volta settato ad 1 saranno il main e i worker ad agire di conseguenza avviando la procedura di terminazione e di pulizia.

L'handler che gestisce SIGURS1 agisce in modo simile però settando il Stat_bit, in questo caso il thread main stampa le statistiche nel file delle statistiche

Script

Lo strumento che più ho usato nella realizzazione dello script è sicuramente il pattern matching, usandolo insieme a "cat" sono riuscito velocemente ad estrarre DirName dal file di configurazione. Invece mi sono servito di "date" e "stat" per estrapolare l'età dei file in secondi. Dividendo per 60 e facendo una differenza, sono riuscito ad avere sufficienti dati per fare quello che il testo chiedeva; eliminare i file più vecchi di "t".

Listener.c

Ho scritto questo file per superare il test4.

Conclusioni

Sono consapevole del fatto che questa relazione non può rimpiazzare l'intera lettura del codice sorgente del progetto, ma spero di aver dato almeno un'idea di cosa ho fatto e del perché l'ho fatto. Detto questo, come disse Linus Torvalds (e come è ricordato su Didawiki):

"Talk is cheap. Show me the code"

Buona lettura. Fabio Catinella