MSG:

un semplice server di scambio messaggi di testo

Progetto del modulo di laboratorio dei corsi di SO A/B 2009/10

Alessandro Lenzi
438142 (corso A)

Scelte di implementazione

In generale, il progetto è stato sviluppato nell'ottica di tenere il più possibile separati semantica e sintassi, in modo che quest'ultima potesse cambiare con maggiore libertà. Spesso questo ha portato a scelte di implementazione più complesse; un esempio di queste “problematiche” può essere il Writer Buffer, che poteva essere implementato molto più semplicemente con un array di stringhe già formattate per l'inserimento nel file di log, mentre l'utilizzo di una struttura apposita che tenga ben separati i diversi dati permette – in teoria – di elaborare ulteriormente le informazioni (es: a fini statistici) senza dover disassemblare i dati già formattati.

La parte sull'accesso in mutua esclusione alle socket dei client è stata aggiunta alla libreria comsock.h.

Il buffer per la scrittura dei messaggi ha una sua libreria apposita, mentre il resto delle funzioni che si utilizzano nel server e nel client, o che comunque non trovavano una collocazione logica all'interno di comsock o di msgbuffer, fanno parte di msglib.h.

Writer Buffer e libreria msgbuffer.h

La struttura dati che è stata usata per la comunicazione tra i thread worker e il thread writer(che si occupa di scrivere i messaggi sul file di log) è un buffer circolare. Di default, la costante che ne definisce la dimensione è pari a 64, che sembra un numero sufficientemente grande per un numero ragionevole di client.

Oltre all'array di message_t_expande, di dimensione (ovviamente) variabile in base alla richiesta del programmatore, la struttura contiene una variabile di mutua esclusione e due condition, una su cui si sospenderanno i lettori e una per gli scrittori.

Ogni elemento di questo buffer è un message_t_expanded, che è una espansione della struttura dati per i messaggi su cui si basa la libreria comsock.h: ogni message_t_expanded ha, difatti, anche un campo per indicare il nome del mittente e del destinatario.

Oltre al buffer circolare, ogni struttura message_buffer contiene anche una mutex e due variabili di condizione, una stante a indicare “buffer pieno” e l'altra “buffer vuoto”.

Lettura e scrittura avvengono in mutua esclusione attraverso le funzioni fornite nella libreria, ovvero write_Buffer e read_Buffer.

Sono inoltre disponibili funzioni per la conversione da message_t e per la liberazione efficiente della memoria. Inoltre, all'atto della scrittura sul buffer, l'allocazione dello spazio per il buffer e per i nomi utenti memorizzati nella “message_t” avvengono dinamicamente, tenendo in considerazione lo spazio precedentemente occupato per gestire al meglio la memoria ed evitare continue allocazioni e liberazioni.

Si noti che la struttura dati non prevede, durante la sua esistenza, modifiche strutturali. Non si è ritenuto quindi necessario gestire questo caso.

Socket Lock ed espansioni alla libreria comsock.h

Nell'invio dei messaggi da un client a un altro, si presenta un problema di mutua esclusione: difatti, non essendo la write implementata su comsock atomica, bensì composta di tre diverse chiamate, è necessario assicurarsi che ogni “worker” (il thread che gestisce ogni client) abbia accesso esclusivo sulla socket del destinatario fino al termine delle scritture.

Il problema poteva risolversi in vari modi, dall'utilizzo di un semaforo per ogni socket a un lock complessivo su una struttura dati.

La scelta che è stata fatta in questa implementazione è stata utilizzare per la gestione dell'accesso una lista, che in ogni elemento contiene il file descriptor della socket e un indicazione di occupato

(1) o di libero(0). La lista è racchiusa in una struttura dati socket_lock, che contiene una variabile di mutua esclusione e due di condizione, oltre a delle indicazioni sulla sua dimensione, a un intero che indica il numero di socket in uso e un altro che permette di bloccare la struttura qualora dovesse essere modificata nella sua interezza.

Il meccanismo di accesso a una socket si basa su due fasi: la prima, di prenotazione, incrementa la variabile waitingUse, che rappresenta il numero di utenti che sono in attesa di fare una scrittura sulla socket. Quando si ottiene l'accesso effettivo alla socket, l'intero che rappresenta se la socket è occupata viene posto a uno, e il numero waitingUse viene decrementato. Da questo momento in poi l'accesso alla socket è unico per il thread che ha l'ha richiesto.

Le operazioni di inserimento o cancellazione, richiedono difatti che nessun thread stia utilizzando la struttura dati, che durante queste modifiche potrebbe essere inconsistente. Questo tipo di accesso consta perciò di due fasi: la prima, in cui si verifica che non ci siano prenotazioni accettate sulla lock o in attesa di esserlo, e una seconda, che verifica che non si stiano facendo modifiche strutturali sulla socket lock. Una volta posta la variabile di modifica (edit) a uno, non si permette ad altri thread di richiedere operazioni di scrittura o di accesso. Nel momento in cui tutte le operazioni terminano, l'operazione può essere svolta e successivamente il ciclo di funzionamento normale della socket_lock prosegue normalmente.

L'unica problematica di questa implementazione si è presentata nella fase iniziale, dove sembrava piuttosto inutile e faticoso scorrere l'intera lista per ogni scrittura sulla socket, specie perché questa ricerca non poteva che seguire a una sulla tabella hash.

Ho perciò modificato il campo payload della tabella hash in modo che fosse un elem_t**, che punta all'elemento della socket_lock corrispondente all'utente il cui username è contenuto in hash_table->key. L'indicazione di disconnesso è ovviamente con payload == NULL. La correlazione tra le due strutture dati è mostrata nell'Illustrazione 1.

La scelta del tipo del payload è dovuta alle necessità di poter gestire le due strutture dati con un sufficiente grado di separazione: se fosse stato solo un elem_t* l'eliminazione della tabella avrebbe comportato l'eliminazione di un elemento nella struttura socket lock, e viceversa l'eliminazione dell'elemento dalla socket_lock avrebbe lasciato inconsistente il puntatore nella tabella.

Il resto delle funzioni di comsock aderiscono alle specifiche del progetto.

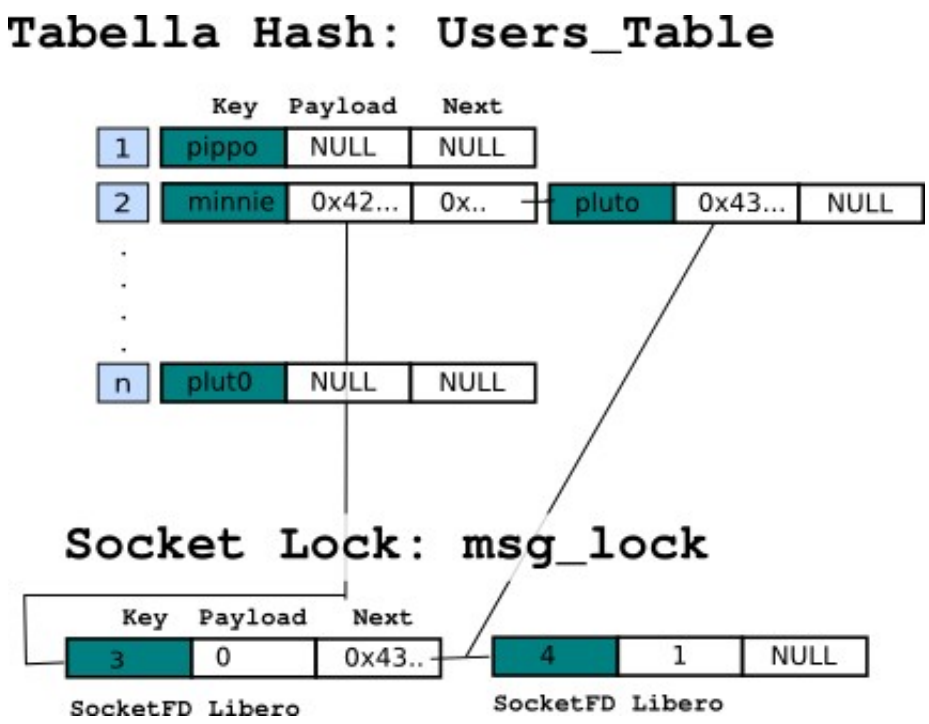

Illustrazione 1: La figura rappresenta una situazione ipotetica in cui nella tabella sono caricati gli username di userslist (la disposizione dell'immagine non è reale) e ci sono due utenti connessi, minnie e pluto. Inoltre, attualmente qualcuno sta effettuando una scrittura sulla socket con file descriptor 4, riferita al client pluto.

Lista degli utenti connessi

Ho previsto una stringa statica, accessibile da tutti i thread, con memorizzata all'interno la stringa contenente gli username degli utenti connessi. Questa implementazione permette di modificare la lista solo quando un utente si connette o si disconnette, invece di dover scorrere l'intera tabella ogni qual volta un utente richiede la lista degli utenti. Sempre per risparmiare dal punto di vista computazionale, ho preferito cancellare o aggiungere l'username alla stringa anziché scorrere l'intera tabella.

Si accede alla lista degli utenti connessi in mutua esclusione attraverso listWait e listSignal, funzioni chiamate automaticamente in refreshUserList, che prende come primo parametro l'username da aggiungere-togliere e come secondo la costante ADD o REMOVE.

Tabella Hash

La tabella hash è implementata con la struttura descritta sopra, in cui accanto a una chiave, che è l'username dell'utente e viene caricato inizialmente all'atto della lettura del file contenente la lista degli utenti autorizzati, c'è un elemento che è un riferimento alla struttura socket lock che gestisce l'accesso in scrittura alla socket.

Per modificare questo elemento (che è l'unica operazione che può avvenire durante l'esecuzione del server) nel server è stata prevista una variabile UT_inUse (tabella utenti in uso) a cui si accede in modo mutuamente esclusivo utilizzando una mutex e una condition, e che blocca l'accesso all'intera tabella. L'accesso alla tabella hash si ottiene con la funzione tableWait e si rilascia con tableSignal.

Strutturazione del Server

Il server, a regime, è composto di un numero variabile di thread. Viene difatti attivato un thread per ogni utente che si connette (il numero massimo sarà quindi sancito dalla lunghezza della lista data inizialmente) più un thread dispatcher, un thread writer e un thread che ha la funzione di signal handler.

Tutti i thread sono attivati mentre la maschera del main non permette la ricezione di segnali, permettendo l'utilizzo in un momento successivo di un signal handler.

Come da specifiche, il server termina solo per la ricezione di segnali appositi.

Writer, thread di scrittura file di log

Il writer si occupa della lettura del buffer e della scrittura, secondo una sintassi prestabilita, all'interno del file di log passato al server come secondo argomento. È il primo thread ad essere attivato, e parte con la memoria già “pronta”, cioè con tutte le strutture dati inizializzate.

Quando viene creato, apre il file e successivamente esegue un ciclo infinito, all'interno del quale fa una lettura del buffer, che lo sospende se questi è vuoto. Ad ogni lettura corrisponde una scrittura.

È previsto un cleanup, ovverosia una funzione da eseguirsi alla terminazione del thread (che avverrà dopo l'esecuzione nel signal handler di una chiamata per la sua cancellazione) che si occupa di svuotare completamente il buffer e di chiudere il file utilizzato come registro.

Dispatcher, thread di accettazione utente

Il dispatcher è il secondo thread a partire, e per prima cosa crea la socket AF_UNIX del server, utilizzando createServerChannel della libreria comsock.h.

Successivamente effettua un ciclo continuo in cui accetta le connessioni degli utenti. Il dispatcher, dopo l'accettazione, esegue il protocollo di connessione, in cui a un MSG_CONNECT che contiene

il nome utente di colui che desidera connettersi, fa seguire un MSG_OK o un MSG_ERROR.

Una volta che la connessione è stabilita (e che dunque si è controllato che il nome utente fosse disponibile), si inserisce il file descriptor della socket su cui avverrà la connessione all'interno della struttura socket lock, il cui elemento sarà poi passato come argomento al thread worker che gestirà le richieste dell'utente; si aggiorna inoltre la lista degli utenti connessi.

Il cleanup del dispatcher prevede unicamente la chiusura della socket.

Worker, thread di gestione dei messaggi

Il worker, come gli altri thread, esegue un ciclo infinito in cui riceve i messaggi. Subito dopo la ricezione, i messaggi vengono standardizzati (ad esempio, ricavando il destinatario se è un MSG_TO_ONE) in modo che la struttura message_t sia standard fino al momento della chiamata della procedura sendClient, la quale formatta e scrive nel buffer.

L'uscita dal worker può avvenire in diverse maniere, ma a meno di ricezione di segnali, avviene per chiusura della socket da parte del client o per invio di messaggi di uscita.

In questo caso, prima della terminazione del thread viene chiamata la procedura disconnectUser, che si occupa di rimandare al client il messaggio di uscita (che pone termine al protocollo di uscita) e di liberare le strutture dati globali da ogni traccia dell'utente.

Gestione dei Segnali

La gestione dei segnali, nel server, avviene attraverso un thread che funge da signal handler. Questo thread è il main, ed è l'unico a non avere tutti i segnali nella maschera. La gestione si occupa (fatto salvo il caso in cui si riceve un errore del tipo SIGSEGV o SIGBUS) di:

a) Notificare la terminazione a tutti i client
b) Terminare tutti i thread
c) Liberare tutta la memoria

La fase (a) viene eseguita contemporaneamente a una parte consistente della (b) all'interno di una funzione, denominata cancelWorkers, che scorre la tabella hash, impostando tutti gli utenti come disconnessi (in modo da evitare che vengano inviati altri messaggi) ma, soprattutto, scorre la socket lock e chiude tutti i socket, causando il ritorno dalla funzione receiveMessage su cui erano (probabilmente) sospesi i worker e dunque la loro terminazione.

Il resto dei thread vengono terminati mandando un segnale di cancellazione: l'ultimo è quello inviato al writer, che può così evitare di accedere in mutua esclusione alle sue strutture dati globali per effettuare il cleanup.

Infine, tutte le strutture vengono liberate con le funzioni di libreria messe a disposizione o con quelle create ad hoc.

Struttura del Client

Ogni client è specializzato in due thread, uno che si occupa della lettura da standard input e dell'invio, l'altro che riceve i messaggi e li stampa su standard output.

La gestione dei segnali avviene attraverso gestori.

Output: thread per la ricezione dei messaggi

Viene creato una volta che il protocollo di connessione è già stato eseguito con successo, e prende come parametro il descrittore della socket su cui i due processi comunicano. Esegue fino a che la

variabile stop (che segnala l'uscita) è uguale a zero, un ciclo continuo di lettura e di stampe su standard output. Inoltre, se riceve un messaggio di uscita dal server imposta 1 la variabile exit_received, che segnala appunto l'evento della ricezione di un messaggio di uscita dal server.

Dopo l'uscita dal ciclo, si controlla se il messaggio di uscita ricevuto è seguito a una richiesta inviata dal client, altrimenti segnaliamo all'altro thread di terminare ponendo la variabile stop a 1 ma, soprattutto, inviando al processo un segnale di terminazione che ne causi l'uscita dalla fgets su cui probabilmente input è sospeso.

Input: thread per l'invio dei messaggi

Input viene eseguito come funzione all'interno del main, dopo che questi ha creato correttamente il thread output. Viene terminato dalla ricezione di un segnale o dall'invio/ricezione di un MSG_EXIT.

La lettura da standard input avviene senza limiti nel numero di caratteri: ogni ciclo di lettura legge 256 caratteri, su cui controlla la validità e a cui eventualmente vengono aggiunti quelli del ciclo successivo (e così via) se la stringa letta non contiene il carattere newline, che segnala la fine del messaggio. Quando il messaggio è finito, ne viene riconosciuto il tipo da una funzione apposita e viene inviato al server per l'elaborazione.

Input termina quando l'utente invia un messaggio di uscita oppure quando se ne riceve uno dal server, con le modalità illustrate nel paragrafo riguardante il thread Output.

Terminazione del Client

La terminazione del client può avvenire in diverse maniere:

1. **Invio MSG_EXIT:** in tal caso il thread input imposta la variabile globale exit_send a 1, per segnalare questo evento. Input esce dal suo ciclo infinito e libera la memoria allocata. A questo punto output attende a sua volta un MSG_EXIT da parte del server, che segnala la corretta esecuzione del protocollo di disconnessione. Il client chiude la socket e termina.
2. **Chiusura stdin:** ancora una volta sarà input a rendersi conto di dover dare inizio alla procedura di terminazione. Input esce dal ciclo e, visto che non sarà stato né inviato né ricevuto un messaggio di terminazione, fa partire il protocollo di disconnessione inviando MSG_EXIT al server. Da questo punto in poi, ci riconduciamo al caso trattato in (1).
3. **Terminazione del Server:** nel caso standard, in cui cioè il server termina correttamente, colui che si rende conto della terminazione del server è il thread output, che riceverà un MSG_EXIT e reagirà uscendo dal ciclo e segnalando questa situazione al thread che gestisce l'input dell'utente attraverso un segnale al processo. In generale, però, se la terminazione del server avviene in maniera “brutale” (un esempio potrebbe essere la ricezione di SIGKILL) questo caso viene gestito ogni qualvolta c'è una chiamata alla funzione sendMessage, che rileva la chiusura della socket.
4. **Segnale di terminazione:** l'handler dei segnali di terminazione verifica se sono già stati inviati o ricevuti messaggi di uscita, poiché in tal caso la procedura di disconnessione è già cominciata. Se ciò non è accaduto, invia un MSG_EXIT al server e pone exit_sent a 1, avviando il protocollo di disconnessione.

Problemi Noti

Al momento l'unico problema potenziale noto, che però non si è mai verificato in sede di test, è la possibile starvation che potrebbe verificarsi nei thread che richiedono modifiche alla struttura socket lock: difatti prima che venga accolta, tutti gli utenti che hanno richiesto l'accesso a una socket devono aver terminato e liberato la struttura, che, nel caso il server sia molto congestionato, potrebbe far attendere indefinitamente un thread che dovesse richiedere una cancellazione o un inserimento.

Testing

Test aggiuntivo

È stato generato per verificare il comportamento del server in situazioni più “impegnative”: deve infatti gestire un numero molto più elevato di utenti (98) in contemporanea, che mandano un numero di messaggi casuale, con caratteri casuali e di lunghezza casuale. Il test consiste nel verificare se il server riesce a portare a termine correttamente tutte le richieste dei client, che quindi devono aver terminato correttamente. Il test può essere eseguito semplicemente con make test34.

Il testing è stato effettuato su Olivia, su un Intel i3 Quad-Core da 2.26ghz e su diverse macchine dell'aula H e dell'aula M, senza che il server o il client abbiano mai mostrato problemi.

In particolare, il testing è stato condotto in tre maniere parallele:

1. Sono stati eseguiti mille test consecutivi (eliminando la ri-compilazione per questioni di performance) su Olivia e su Fujim4, ottenendo il superamento continuo di tutti e tre i test standard e di quello aggiuntivo.
2. Si è simulato il comportamento dei tre test rilasciati dal docente mandando in esecuzione il server con valgrind, senza ottenere alcun memory leak.[1]
3. Si è simulato “manualmente” il comportamento di server e client in situazioni possibili, come la terminazione del server con client connessi, con client connessi bloccati etc.

1 Salvo i 5 blocchi che, sulle macchine del CLI, rimangono allocati ogni qual volta si fa terminare un thread con la pthread_cancel.