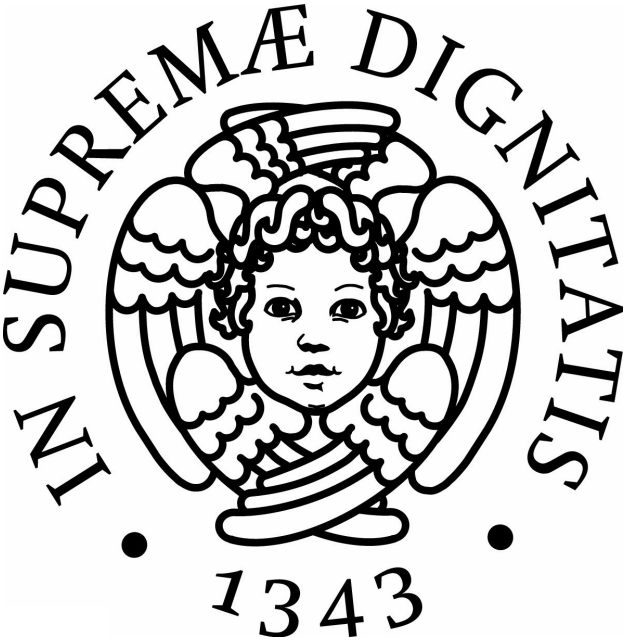

Università di Pisa
Facoltà di Scienze Matematiche Fisiche e Naturali
Corso di Laurea in Informatica

msg:
un semplice server per scambio messaggi di testo

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

Filippo Fontanelli

Prof. Pelagatti Susanna

Anno Accademico 2009/2010

1) Introduzione

Il progetto svolto si occupa dello sviluppo di un sistema client-server per lo scambio di messaggi di testo fra un insieme di client.

2) Organizzazione

Il sistema è costituito da due processi concorrenti e multithreaded forniti dal committente.

Il primo msgcli è il processo client (uno per ogni utente connesso) che si occupa di interagire con il server per inviare e ricevere messaggi per conto dell'utente.

Il secondo msgserv è il processo server che verifica se un utente può accedere al servizio. msgserv mantiene la traccia di tutti gli utenti connessi, riceve e spedisce i messaggi da e per i client. Inoltre tiene traccia in un file di log dei messaggi inviati correttamente il quale verra' successivamente analizzato da un script offline (logpro) per determinare il numero di caratteri inviati da ciascun client.

3) Principali scelte del progetto

Le scelte piu' significative sono state:

a) per quanto riguarda il client non vi sono state scelte particolari dato che la sua realizzazione non richiede particolari algoritmi.

b) Per quanto riguarda il *server* le implementazioni piu' critiche hanno riguardato la memorizzazione del file degli utenti autorizzati, la struttura dati condivisa tra writer e worker e la gestione dei thrade worker.

i) Struttura dati condivisa tra worker e writer

Il gruppo di thread worker invia messaggi al thread writer attraverso una coda opportunamente sincronizzata. La coda rappresenta una tra le più semplici soluzioni possibili, pur mantenendo alta la probabilità che le richieste di scrittura siano ordinate rispetto all'effettivo momento di invio del messaggio, anche se da specifiche non sono stati posti vincoli a riguardo.

ii) Tabella Hash degli utenti Autorizzati

Realizzati tramite le funzioni dellla libreria libmsg, la quale rispetta le caratteristiche presentate dal committente.

iii) Tabellla Hash dei Worker

Per i thread worker ho ceciso di utilizzare una second a tabella hash, dopo aver effettuato le seguenti considerazioni:

(1) Avrei potuto utilizzare una qualsiasi altra struttura di memorizzazione dati ma non avrei avuto la stessa efficienza nelle funzioni di ricerca ,invocate con una percentuale relativamente elevata dalla elaborazione dei messaggi ricevuti al client.

(2) *Un'altra soluzione che forse avrebbe privilegiato la velocita' di creazione dei singoli thread , sarebbe stata quella di utilizzare un threadpool*.

Ma considerato il poco tempo rimastomi a disposizione, e supponendo che gli utenti abilitati siano un numero nettamente maggiore rispetto agli utenti attivi ho deciso di optare per una seconda tabella Hash.
La struttura di tale tabella prevede di utilizzare il tid dei vari thread whorker come key e non utilizzare il campo payload.

4) Strutturazione del codice

Ho cercato di rendere il codice piu' chiaro possibile andando ad accumulare parti simile di codice e cercare di suddividere il codice moduli ottenendo cosi' diversi file sorgente e diversi file header.

Server.c / .h *realizzazione di un package di funzioni invocate dal server durante il suo ciclo di vita un package contenente funzioni rappresentanti i vari moduli di elaborazione del server, implementando una funzione per ogni "stato/step" in cui il server puo' trovarsi.*

Utilityserver.c /.h : *realizzazione di un package di funzioni con il compito di fare da filtro tra le funzioni della comsock e l'esecuzione del server.*

Utilityclient.c /.h : *realizzazione di un package di funzioni con il compito di fare fa da filtro tra le funzioni della comsock e l'esecuzione del client.*

Queue.c / .h : *realizzazione della coda generica bloccante tra whorker e writer.La concorrenza su tale coda e' stata gestita tramite un semaforo pthread .*

5) Struttura dei programmi sviluppati

Sono stati sviluppati due programmi il client e il server .

La loro programmazione inizialmente e' stata asincrona dato che prima ho realizzato un semplice server che sfruttasse le funzioni della comsock (la prima ad essere stata sviluppata) per mettersi in ascolto su una socket in attesa di connessione.

Successivamente, ho sviluppato la prima bozza del client la quale effettuava una richiesta di connessione al server inviandogli un semplice messaggio.

A questo punto seguendo le specifiche e i test a disposizione, ho sviluppato il codice necessario per il protocollo di comunicazione tra il client e il server.

Dopo aver verificato la corretta esecuzione dello scambio di messaggi, ho cercato di modificare il codice in modo da ottenere una suddivisione in moduli del server, non resa necessaria per il client vista la sua semplicita', creando il file server.c/h e il file utilityserver.c /.h.

Per il client, ho ritenuto necessario creare un file utilityclient.c /.h con le medesime funzionalita' del file utilityserver.c / .h .

Avrei potuto utilizzare una singola classe per entrambi, ma ho deciso di non farlo per le seguenti motivazioni.

a) il protocollo di comunicazione tra server e client prevedeva un formato differente per i messaggi se inviati dal client o dal server.

b) concettualmente l'architettura client- server prevede due entita' distinte non necessariamente operanti sulla stessa macchina, ma tendenzialmente operanti su macchine differenti.

Quindi ho ritenuto incompatibile l'utilizzo di un solo file.

Dopo tale fase ,ho effettuato vari test da terminale con comandi mirati alla verifica di particolari condizioni limite e non, avvalendomi di un debug minimale dell'errore(effettuato principalmente con la stampa su stdout di messaggi riepilogativi) e con l'uso massiccio di valgrind per i test riguardanti la memoria anche se sfortunatamente non hanno portato i risultati sperati anche se ho appreso molte nozioni sul suo utilizzo.

E in fine, ho provveduto a occuparmi dei segnali e delle chiusure gentili, cercando la soluzione migliore, al termine di tale ricerca ho optato per un rafforzamento del protocollo di disconnessine tra il client e il server.

6) Il Server

Dopo aver effettuato le operazioni di parsing delle opzioni (in particolare il file di log e il file degli utenti autorizzati), vengono create le strutture dati destinate a contenere le autorizzazioni e i thread worker e solo dopo aver creato il canale socket , si procede alla creazione di un thread writer, di un thread handler(per la gestione dei segnali) e della struttura dati condivisa tra i worker e i writer.
Terminata tale fase di "preparazione del server", esso si mette in ascolto sulla socket aspettando nuove richieste di connessione.
Alla ricezione di una richiesta procede alla creazione di un thread worker.

6.1.I Thread Worker

Ogni thread ascolta le richieste provenienti dalla socket associata.
All'arrivo di ogni richiesta, segue una fase di interpretazione necessaria per la sua esecuzione, ogni messaggio correttamente spedito viene inviato al writer mediante l'apposita struttura dati condivisa.

In caso di richiesta di disconnessione o in presenza di errori sul canale di comunicazione , la gestione delle richieste del client associato termina.

6.2.Il Thread Writer

Il thread writer legge i messaggi da scrivere sul file di log dalla coda dei messaggi che come detto precedentemente esso condivide con i vari thread worker, bloccandosi in caso tutte le richieste siano state soddisfatte e non ce ne siano altre (coda vuota).

6.3.Gestione dei Segnali

Per la gestione e' stato necessario la specifica di un thread handler predisposto alla gestione dei segnali per aver una terminazione gentile del server. Il thread e' l'unico che non maschera i segnali di SIGINT e SIGTERM, bloccandosi in attesa di un loro eventuale arrivo.
All'arrivo, viene chiusa la socket principale e viene inviato un segnale di SIGUSR1 a tutti i worker, al main e al writer ,in modo da sbloccare il flusso di esecuzione eventualmente bloccato su funzioni bloccanti, e quindi ad eliminare la msgsock.

Inoltre e' stato necessario ignorare il sengnale SIGPIPE.

7) Il Client

Dopo aver effettuato le operazioni di parsing delle opzioni (in particolare il nome dell'utente), il client tenta per 5 volte con un intervallo di un secondo di connettersi al server aprendo una connessine con la sua socket, dopo di che precede con l'invio di un messaggi di CONNECT e il test della risposta ricevuta dal server.
Dopo chi la connessione e' avvenuta con successo, il client crea due thread paralleli che si occupano rispettivamente di inviare e ricevere messaggi.
E un thread handler per la gestione dei segnali.

7.1.Il Thread Sender

Compito di tale thread e' di ricevere comandi da standard input e crare un opportuno messaggio di richiesta che sara' subito inviato al server(invocando le funzioni dell'utilityclient).

Il thraed termina nel caso in cui si riceva da standard input un messaggio di EXIT, oppure in caso di errori sul canale di comunicazione segnalatigli dal thread receiver.

7.2.Il Thread Receiver

Compito di tale thread e' di ricevere messaggi dal canale di comunicazione e di stamparli su standard output correttamente formattati.

Il thread termina quando si verifica un errore sul canale di comunicazione oppure quando riceve un messaggio MSG_NO dal server.

7.3.Gestione dei Segnali

La terminazione gentile ha richiesto un attenta politica di gestione dei segnali. I thread hanno accesso ad una variabile globale termina_sign eventualmente modificata da uno di essi, per avvertire gli altri di una imminente terminazione. Questo tipo di trattamento per la terminazione gentile del programma non era sufficiente, in quanto nei thread sono presenti delle funzioni bloccanti. A questo proposito, i thread sender e receiver, in caso di pronta terminazione, possono inviarsi a vicenda un segnale SIGUSR1 per eventualmente sbloccare il flusso di esecuzione del thread da una possibile funzione bloccante.
E' presente inoltre un thread aggiuntivo che si mette in ascolto di eventuali segnali SIGTERM o SIGINT. All'arrivo di uno dei segnali, viene opportunamente settata la variabile condivisa term ed eventualmente sbloccato il thread sender.
Tutti i thread ignorano il segnale SIGPIPE.

7.4.Protocollo di Terminazione

Inizialmente credevo che la soluzione migliore fosse che il server chiudesse il socket alla ricezione del messaggio di EXIT dal client, ma successivamente per la chiusura gentile del client ho deciso che il server rispondesse a tale messaggio con un messaggio di MSG_NO, interpretato dal client(receiver) come il seganle di fine comunicazione sulla socket e di conseguenza a chiuder la socket. Gestendo separatamente gli alti casi di terminazine.

La terminazione del client prevede i seguenti casi:

1) Terminazione esplicita del client tramite un MSG_EXIT

   In questo caso il thread sender invia al server un messaggio MSG_EXIT e si pone in attesa della terminazione del thread receiver, la quale avverra' alla ricezione di un MSG_NO dal server(in risposta al messaggio di EXIT).
   Ricevuto tale messaggio il thread receiver procede con la chiusura del canale di comunicazione e termina, cosi da riattivare il thread sender che procedera' con la fase di terminazione dell'intero client.

2) Terminazione dovuta ad un errore del canale di comunicazione

   In tal caso il thread receiver provvedera' a chiudere il canale di comunicazione e tramite i segnai SIGUSR1 e SIGINT a sbloccare e terminare, tramite una variabile condivisa serrata opportunamente, rispettivamente il thread sender e il thread handler.

3) Terminazione dovuta a un segnale di SIGINT o SIGTERM

   In questo caso e' il thread handler a sbloccare e terminare, tramite una variabile condivisa serrata opportunamente, rispettivamente il thread sender e il thread receiver.

8) Conclusioni

Le chiare specifiche iniziali nascondono la complessità e la grandezza di tutto il progetto. L'implementazione delle strutture dati, un'efficiente programmazione concorrente e di sistema, gli innumerevoli casi anomali da trattare discostano il progetto da una semplice banalità. Ho cercato di rendere il sistema il più consistente possibile, pur consapevole che molti miglioramenti possono essere apportati.

9) README

Come da specifiche e' stato ralizzato un Makefile che provvede alla compilazione di tutto Il codice necessario per l'esecuzione dei test, inoltre all'interno dei file eseguibili sono previsti degli #ifdef DEBUG che permettono se richiesto di visualizzare su standard output dei messaggi riepilogativi sull'esecuzione dei due processi.