Relazione sul progetto di Sistemi Operativi e Laboratorio

A cura di Massimo Puddu

Indice

Introduzione

Lo scopo del progetto era quello di creare chat concorrenti tra utenti implementando un server multithreaded capace di gestire la connessione e le richieste concorrenti di un centinaio di utenti.
Ho lavorato tutto il tempo utilizzando le nozioni apprese durante il corso, spulciando varie volte il manuale per approfondimenti; nozioni che si sono rivelate molto utili per la realizzazione del server.

Script bash

Lo script bash controlla primariamente se è presente –help fra i vari argomenti passati, se questi lo comprendono si restituisce il messaggio di Usage. Dopo aver controllato che siano corretti si procede all'esecuzione solamente se tutti gli argomenti necessari sono presenti, in caso contrario termina con un messaggio di errore.
Dopo che lo script bash si è assicurato di avere tutti gli argomenti corretti, dopo aver controllato che si abbiano tutti i permessi per il file da aprire (se è un file), si procede all'apertura del file passato e cercando DirName escludendo infine le righe con DirName e con # come inizio.
Una volta trovata si cerca come potrebbe iniziare il path(con / ~ .), ottenuta questa si prende la lunghezza della stringa facendo, a questo punto, pather matching per eliminare tutto quello antecedente al mio path; in questo modo si avrà solo il path richiesto.
Non resta altro da fare se non suddividere il tutto in un array per eliminare eventuale spazi e semplificare il lavoro in maniera tale da aprire in seguito il file con cd dato che gli spazi verrano sostituiti con gli opportuni backslash. Dopodiché vengono controllati i permessi e l'ultimo argomento per decidere se eliminare oppure visualizzare la cartella, nel primo caso si utilizza un banale ls ordinato mentre, nel secondo caso, si usa la find per trovare i file creati da più di N minuti e si procede con l'opzione - delete per eliminarli.

Strutture dati principali

Per quanto riguarda la comunicazione tra listener e worker ho deciso di adottare due liste, una quando spedisce il listener al worker e l'altra quando il worker deve spedire al listener; questo per ottimizzare la comunicazione tra i due. La seconda lista fa capire al listener quando chiudere un fd o meno, mentre la prima fa capire che operazioni far eseguire al worker.

Invece, per memorizzare gli utenti registrati, ho deciso di utilizzare una tabella hash ad adiacenza impostata con una grandezza di 2000, numero scelto in base alle specifiche del server. Questa non ha liste ad adiacenza ma alberi in modo da velocizzare il piú possibile la ricerca anche nel peggiore dei casi. Il nodo della tabella ê strutturato affinché contenga il fd associato alla connessione nel caso in cui fosse momentaneamente online, in caso contrario il suo valore è -1, invece se si tratta di un gruppo ha valore -2. Inoltre qui sono contenute anche le history dei messaggi non inviati, il numero di messaggi giacenti. Contiene anche un campo group che funziona nel seguente modo: se l'utente in questione è un gruppo vengono memorizzati qua anche i suoi utenti, al contrario se si parla di un normale utente vengono memorizzati i gruppi di cui fa parte.

Ho preferito memorizzare qui la history perché nel caso avessi deciso di creare una struttura parallela alla tabella hash avrei avuto problemi di sincronizzazione perdendo così il vantaggio di velocità a cui avevo mirato fin dall'inizio.

La history è un array circolare gestito dal numero di messaggi giacenti controllati in modulo in modo da sapere la posizione dell'ultimo messaggio ricevuto.

Oltre alla tabella hash devono essere diretti anche gli utenti online, per fare ciò ho utilizzato, di nuovo, una struttura ad albero con inserimento in base al fd, viene anche associato il nome dell'utente al fd. Ho scelto questa implementazione per velocizzare il più possibile, è vero che sono più lento nell'inserimento

rispetto a una lista ma per ogni connessione c'è anche una disconnessione e quindi un log n; questo mi permette di recuperare il tempo perso nell'inserimento.
Tuttavia gli utenti online non saranno mai tanti per poter sfruttare appieno questa implementazione, ogni piccolo dettaglio può decidere una risposta immediata al client rispetto a una con un leggero ritardo.

Entrambe le strutture ad albero sono avl, dove il codice è stato preso e modificato per l'esigenza da http://www.geeksforgeeks.org/avl-tree-set-1-insertion/
Anche la funzione per il calcolo della tabella hash è stata presa da https://codereview.stackexchange.com/questions/85556/simple-string-hashing-algorithm-implementation

Parsing file config

Per memorizzare i dati presenti nel config ho utilizzato una struttura dati (struct configuration) composta da un array di stringhe e un array di int.
Nella funzione get config leggo riga per riga dal file di configurazione prendendo quelle interessate, infine ordino i dati per sapere come saranno posizionati gli indici e inserire i questi all'interno della mia struttura (il suo inserimento nella struttura non era dei migliori, ma non ho trovato un modo per inserire il tutto con un for). Il parsing è stato testato modificando il file config facendo in modo di eliminare sia una riga es.UnixPath che il simbolo dell'uguale o lasciando solo spazi bianchi sia dopo che prima le righe interessate.

Attivazione server

Il server inizia scaricando i dati dalle funzioni di parsing nel modo descritto in precedenza e, una volta presi, inizia a impostare i dati del server: prima attiva i thread, inizializza le mutex e infine attiva il listener e i worker in modalià join. Si fa partire anche un thread per la gestione dei segnali. Elimino, nel caso ci fosse, un socket con lo stesso nome e le cartelle di download di chatty.

Gestione dei segnali

Ci sono 6 segnali da gestire, il thread apposito si occupa di 5 di questi attraverso una sigwait, nel caso riceva SIGQUIT, SIGINT, SIGTERM o SIGSEGV si cambia il valore del flag in modo che il server inizi a impostare la sua chiusura. Se ricevo SIGUSR1 invece si stampano le statistiche. L'ultimo caso, cioè quello non presente tra i segnali della sigwait è SIGPIPE; se viene ricevuto, il programma continua normalmente come se nulla fosse. In seguito si annulla l'operazione di invio di un messaggio a un client a cui il programma stava spedendo.

Listener

Ho utilizzato la select in modo da sapere quando un fd è pronto per essere letto evitando di avere thread in attesa attiva. Oltre che spedire ai thread si può anche ricevere.
In base a quello che mi viene inviato decido se sia necessario chiudere il fd o reinserirlo nel fd_set. Nel caso in cui il fd non è presente nel set viene aggiunto, in seguito viene rimosso per spedirlo a uno dei thread worker a cui viene detto di controllare se si può connettere. Se era già nel set invece viene rimosso ma nell'invio dico al worker di non controllare se si può connettere.
Se è necessario chiudere il listener allora si chiudono tutti i fd aperti per la select e il programma termina.

Worker

La gestione del worker inizia molto semplicemente facendolo stare in attesa sulla wait di qualche richiesta dal listener. Appena la lista non è più vuota, il worker inizia a lavorare attraverso le 3, se necessario 4, fasi seguenti:

1. Scansione della richiesta
2. Elaborazione preliminare
3. Esecuzione della stessa
4. Rimbalzo dell'operazione

Durante la prima fase il programma legge all'interno della lista in cui il listener o altri worker possono aver scritto. All'interno di questa lista ho utilizzato un tipo enumeratore per aiutarmi a capire cosa fare in vari casi.
Nel secondo punto in base al tipo enumeratore di prima, capisco se devo leggere dal socket o passare direttamente alla terza parte.
Nella terza fase in base alla richiesta del client (register, sendmessage_op etc.) si decide cosa fare e se eseguire l'ultimo punto. Decido di eseguirlo solo se ho ritenuto l'operazione troppo costosa per i thread come per esempio mandare un messaggio a tutti gli utenti o post file o anche la register. Dopo essere entrato nello specifico di questi quattro punti, illustrerò perché ho deciso di suddividerle.

Scansione delle richieste

La scansione va ad allacciarsi direttamente con la quarta fase, la lista per rispedire le operazioni è la stessa. Per distinguerle uso il tipo enumeratore. Infatti, nel caso fosse spedito da un'altro thread, esso avrebbe un valore che rappresenta il tipo di operazione che aveva compiuto in modo che le altre fasi sappiano come agire diversamente dal normale. Se invece la richiesta viene dal listener può avere solo due valori: uno indica che la connessione di un nuovo client non deve far superare il numero massimo di connessioni nello stesso momento, in caso contrario passo direttamente alla lettera del socket.

Elaborazione preliminare

In questa fase si decide se accettare una connessione da un client o meno. Nel caso il listener mi inviasse la richiesta controllo se non eccede il numero massimo di connessioni, se si tratta di una nuova. Da adesso si controlla se message_t contiene qualcosa, se non contiene niente allora si procede con la lettura dal socket perché vuol dire che la richiesta non viene dai worker. Se la read restituisce un errore allora si spedisce al listener un messaggio che gli comunica di chiudere il file descriptor.
Da qui in poi a prescindere dalla sua provenienza il comportamento è deciso dalla terza parte.

Esecuzione della stessa e Rimbalzo operazioni

Da qui si utilizza uno switch per aiutarmi a rendere il codice più leggibile. Se l'operazione è stata rimbalzata, ossia due worker si suddividono l'operazione, si potrebbe evitare di fare qualche write dato che è possibile che un op_ok sia stato inviato da un'altro thread. Solitamente la prima cosa che faccio è accertarmi che l'operazione vada a buon fine. Se questo avviene mando op_ok e poi continuo a eseguire l'operazione. Da qui in poi l'esecuzione dipende pura- mente dalla richiesta dell'utente, per cui ora andrò ad approfondirle. Alla fine dell'esecuzione della richiesta rispedisco il file descriptor al master set nel listener. Il listener deciderà poi se chiuderlo o riaggiungerlo. Ovviamente prima di rispedire il fd al listener si controlla che non sia un'operazione che stava rimbalzando.

Libreria connections.h

L'obbiettivo è quello di gestire efficientemente lo scambio di messaggi tra client e server.

Contenuto:

- OpenConnection: controlla se il client si puó connettersi al socket del server, nel caso non possa momentaneamente riprova n volte ogni x secondi;
- ReadHeader: effettua una read per leggere header di message_t;
- ReadData: effettua una read sul campo su message_data_hdr_t per sapere la lunghezza del messaggio, nel caso ce ne sia uno da leggere faccio una read anche sul buf di message_data_t;
- SendHeader: effettua una write sull header di message_t;
- SendData: effettua due write separate se la lunghezza del messaggio è maggiore di 0 altrimenti mi limito a mandare message_data_hdr_t;
- SendRequest: fa in sostatanza un sendHeader e SendData a seconda della richiesta;
- ReadMsg: Fa in sostanza una readHeader e una ReadDat, se l'operazione letta da readHeader lo richiede.

Operazioni da gestire in worker.h

Case: Register_op, Connect_op e Usrlist_op

Il programma controlla se nella tabella hash è presente un utente con lo stesso nome; se non è presente ve lo si aggiunge.
Ho deciso di far rimbalzare la register come connect in quanto poi dovró anche spedire la usrlist perché altrimenti ci sarebbe stato un overhead non indifferente rispetto alle altre 2 operazioni dato che le includeva entrambe. Per farlo rimbalzare mi basta cambiare registor_op in connection_op nel campo op di message_t e inserirlo in lista. Adesso questo andrà direttamente alla connection op dove lo inserirà nella lista utenti online per poi concludere con usrlist. Per la usrlist visito ricorsivamente l'albero degli utenti online e memorizzo in un char* questi utenti in modo da inviare successivamente i dati al client.

Case: CreateGroup e AddGroup

Per registrare i gruppi utilizzo la stessa tabella hash degli utenti; non c'è infatti alcun modo di fare una netta distinzione tra i due (es. posttxt_op) basandoci sui vari test forniti.
Nel caso si tratti di un gruppo, nel campo connessione dell'utente, al posto di ITSUSER (con valore -1) avrà ITSGROUP (con valore -2). Nel campo group dell'utente vengono memorizzati gli utenti che ne fanno parte. Il creatore del gruppo all'interno della sezione group avrà un intero di valore ITSGROUPOWNER che indica il creatore del gruppo. Dopo averlo registrato si ricerca l'utente che ha creato il gruppo nella tabella hash. Invece nel campo group dell'utente registrato inserisco i gruppi a cui questo è registrato. Qui finisce l'operazione CreateGroup la quale differisce dalla AddGroup solo perché non crea un nuovo gruppo ma aggiorna il campo group del gruppo e dell'utente.

Case: DelGroup

La DelGroup funziona nel seguente modo: cerca se il gruppo è registrato, se trova un utente controlla che sia un gruppo. Se lo è procedo; in caso contrario abortisco l'operazione.

Nel primo caso mi assicuro che la persona che ha richiesto la cancellazione sia il creatore del gruppo, se lo è cancello l'intero gruppo. Per farlo devo aggiornare il campo gruppo di ogni persona presente. Quest'operazione è abbastanza costosa per cui la faccio rimbalzare. Il thread che prende la richiesta rimbalzata non cercherà a sua volta di cancellare il gruppo ma si limiterà ad aggiornare la propria casella utente eliminandolo.
Nel caso chi ha fatto la DelGroup non fosse l'owner mi limito ad aggiornare la casella utente e elimino il suo nome dagli iscritti del gruppo.

Case: PostTXT_OP, FileMessage, OP_FiletoDownload, TXTFILE, e OPSENDFILE

Questa parte è un po' particolare perché, come si può intuire dal titolo, è una specie di "coltellino svizzero": quando le operazioni rimbalzano molto probabilmente andranno a finire qua.
Come prima cosa controllo se il messaggio non supera i byte massimi consentiti, successivamente se l'utente esiste, controllo che il messaggio non sia destinato a un gruppo. Ci sono dei comportamenti diversi da questa fase in poi:

1. Se un gruppo vuole mandare un messaggio a tutti i suoi utenti il programma deve far rimbalzare l'operazione modificando il receiver n volte per ogni utente iscritto al gruppo in modo da mandare n richieste. A questo punto l'esecuzione continua su altri thread come descritto all'inizio. Dopo il passaggio 1 il rimbalzo continua nella fase due.
2. La richiesta al gruppo viene trattata singolarmente utente per utente; mi comporto dunque come se fosse un invio tra singoli utenti ma salto il passaggio di controllo della lunghezza messaggio. Memorizzo il messaggio al receiver e memorizzo il suo fd; se è online mi occupo di spedire il messaggio normalmente. Se la richiesta era un posttxt o un postfile imposto nell'header il messaggio che si aspetta il client.
3. Se il messaggio proveniva da un getprevmsg sto invece attento a non memorizzare di nuovo il messaggio e poi procedo come al punto due. Un particolare appunto qua va fatto alle statistiche: a causa di getprevmsg considero un messaggio non mandato se e solo se l'utente era offline o l'invio è stato interrotto da un epipe. Possiamo affermare che le mie statistiche sono date dall'istanza in cui invio il messaggio e non dal messaggio stesso.

Getprevmsg è stato quello più problematico ma ne parlerò piú avanti.

Case: PostFile_Op

Per risolvere postfile in modo efficiente ho adottato una strategia un po' particolare. Innanzitutto provo ad aprire il file, se non esiste lo creo e ci scrivo. In seguito il programma invia il messaggio al receiver per informarlo che un utente vuole inviargli un file facendo rimbalzare operazione con OPSENDFILE. Quando un utente cerca di inviare un file con il nome già esistente, per evitare problemi di race condition, creo un file temporaneo attraverso la funzione mkstemp. Successivamente il programma cambia i diritti al file temporaneo rinominandolo in modo da sovrascrivere il vecchio file. Cosí facendo non ci saranno problemi perché chi aveva anche aperto il vecchio file può ancora accedervi e tutti gli utenti che lo richiederanno successivamente apriranno la nuova versione.

Case: GetFile_OP

Getfile funziona in maniera molto basilare: cerco il file richiesto all'interno della mia cartella e memorizzo il suo contenuto all'interno del char* in message_t, successivamente lo faccio spedire con un altro thread.

Case: GetPrevMsg_OP

Il funzionamento di GetPrevMsg: cerco l'utente, copio la history in una fittizzia, il programma fa rimbalzare ogni singola richiesta ad altri worker che si occuperanno di spedire effettivamente i messaggi.
I messaggi andranno tutti nel "coltellino svizzero".
A causa di questa implementazione potrebbe verificarsi un problema: se nel mezzo di getprevmsg qualcuno invia un nuovo messaggio all'utente esso non riceverà il suo messaggio come nuovo ma lo riceverà insieme a quelli predisposti per la history; gli arriverà quindi a parte un messaggio appartenente alla vecchia history.

Case: Unregister_Op

Primariamente controllo che la persona richiedente la deregistrazione sia effettivamente la stessa da deregistrare visto che non consento che chiunque possa cancellare chiunque.
Procedo togliendo l'utente dalle persone online e lo cancello dalla tabella hash ma non è finita così, devo anche disiscriverlo da tutti i gruppi di cui faceva parte. Questa operazione è stata eseguita facendo rimbalzare questa richiesta ad ogni gruppo come delgroup_op. Il file descriptor associato all'utente viene ovviamente chiuso al posto di essere reinserito nel master set.

Case: Default

In questo caso il client ha inviato una operazione non consentita. Viene pertanto restituito al client un generico messaggio di fallimento.

Chiusura del server

Ogni volta che ricevo un segnale che faccia parte della maschera di sigwait, eccetto sigusr1, chiudo il server. La fase di chiusura fa in modo che cambi il valore del flag il quale determina la chiusura del ciclo del listener e dei worker. In questo modo i worker eseguiranno le loro ultime operazioni senza disturbi per poi fermarsi (nel caso di sigsegv probabilmente si chiuderà prima l'intero programma). Infine essi liberano la memoria rimanente del loro thread e quelli bloccati su una wait vengono svegliati per poter chiudere. Anche il listener chiude tutti i fd ancora aperti per poi terminare. A questo punto il main libera tutte le strutture dati condivise e termino definitivamente. Con la funzione atexit inoltre elimino la cartella per il download dei file e elimino il socket tuttavia non tocco il file delle statistiche in modo da poterle visualizzare anche a server chiuso.

Considerazioni finali

Il progetto è risultato piuttosto complicato, portandomi a dover approfondire diversi argomenti. Mi sono spesso ritrovato bloccato in diverse scelte implementative o, per esempio, per implementare una select che potesse soddisfare al meglio i requisiti del progetto.
Il codice compila e funziona perfettamente sotto MacOs Sierra e Xubuntu 16.04 installati sulle macchine da me in possesso.

La maggior parte dei define e include si trovano all'interno del file Header.h.

Concludo questo progetto con estremo piaciere e soddisfazione, aver visto la creazione del server e vederlo funzionare aggiunta dopo aggiunta è una grande emozione, trattandosi oltre tutto del primo progetto serio che mi sono trovato ad affrontare nella mia vita. Il corso inoltre è stato uno dei più interessanti che ho affrontato fin'ora, aiutandomi ad espandere le mie conoscenze considerevolmente.