Università di Pisa
Dipartimento di informatica
Corso di Laurea in Informatica

Progetto di Laboratorio di Sistemi Operativi

a.a. 2018-19(#2)

Alunno: **Gabriele Sergi**
Matricola: **532362**

Sezioni

1. Include.h

2. Client.c

3. Supervisor.c

4. Server.c
    a. Gestione connessione (thread)

5. Altro
    a. Makefile
    b. Run.sh
    c. Misure.sh

Questo progetto è diviso in tre sezioni principali che racchiudono il cuore del codice, che sono i file .c quali, supervisor, server e client.

Un file .h di supporto per l'esecuzione che racchiude delle funzioni condivise da client e supervisor-server. Inoltre racchiude le strutture principali, gli include e altre funzioni di supporto in modo da rendere più leggibile il codice.

Makefile e misure.sh come da specifica e run.sh che è di supporto al makefile.

1.Include.h

Le strutture contenute da include.h sono le seguenti:

- *Server* utilizzata dal client per avere a disposizione tutte le informazioni disponibili di un server;
- *Figli* utilizzata dal supervisor per mantenere le informazioni dei processi avviati con la fork;
- *StimeFinali* struttura utilizzata come messaggio scambiato tra server e supervisor tramite pipe anonime;
- *List* usata dal server per tener traccia dei thread avviati;

Tra le funzioni comprese in questo file abbiamo quelle per l'apertura e la gestione della connessione:

- *OpenConnect*: Passatoli una stringa, crea una socket con quel nome, la mette in ascolto e ne restituisce il fd della stessa;
- *DirectConnect*: Passatoli una stringa rappresentante il nome di una socket già esistente, cerca di connettersi ad essa ritornando il fd in caso di successo;

Poi ci sono le funzioni utilizzate per l'invio dei messaggi, quali writen e readn prese dalle lezioni del corso;

In più una funzione per l'invio del messaggio da parte del client:

- *SendMsg*: prende in input un puntatore e un fd, e cerca di inviare il messaggio tramite writen. Ritorna TRUE se l'oprazione ha successo;

Una funzione di supporto al client per la creazione di una struttura che si connette ai server e ne tiene traccia:

- *InitServerArr*: prende due parametri *p* e *k*. Dove *p* è il numero di server a cui connettersi e *k* il numero complessivo di server a disposizione. Crea un array di booleani che rappresentano i k server, e un array di server che sarà il risultato della funzione. Parte un ciclo che ad ogni iterazione si ha la certezza di connettersi ad un server al quale non ci siamo connessi in precedenza grazie all'array di booleani, e che ritroveremo nell'array di server che verrà restituito al termine del ciclo;

Infine le ultime funzioni sono per la formattazione dei byte per la comunicazione:

- IsLilEnd: restituisce TRUE o FALSE a seconda se la macchina su cui stiamo lavorando utilizza un ordinamento dei byte little endian;
- Htonlx: funziona come le funzioni della famiglia di htonl, solo che lavora con numeri a 64 bit;
- Ntohlx: similmente a prima funziona come le funzioni della famiglia di ntohl, con numeri a 64 bit;

2.Client.c

Il client è molto semplice, prende da linea di comando i tre parametri p, k e w ed inizializza le variabili che serviranno per la l'invio dei messaggi e l'attesa.

Inizia inizializzando il seme per la sunzione *rand()* con il prodotto del proprio id e il tempo preso in quell'istatante.

Genera il suo id con una funzione che assegna randomicamente ogni bit di un numero a 64 bit. Poi lo converte in NBO in modo da essere pronto per l'invio futuro.

Genera il secret ed imposta di conseguenza la struttura *timespec* della libreria *time.h* da utilizzare poi nella nanosleep.

Inizializza i server con la funzione vista in precedenza *initServerArr();*

Infine comincia il ciclo in cui ogni volta invia ad un server tra quelli connessi, scelto randomicamente il messaggio contenente il proprio ID tradotto in NBO e poi attende *secret* millisecondi prima di inviare il prossimo messaggio.

3.Supervisor.c

Anche il supervisor molto semplice.

Ha il compito di generare i k server, con k preso da linea di comando. Quindi alloca un array di *figli* (struttura contenuta in include.h) e fa partire un ciclo dove inizialmente apre una pipe anonima chiudendola dal lato scrittura, e passandola come argomento alla *execl()* che lancia il server. Execl() che prende come argomento anche il numero identificativo del server che si vedrà meglio dopo.

Questi due elementi sono salvati sulla struttura figli insieme al pid del processo avviato.

Questo ci permette di far partire un altro ciclo che ci permette di leggere da tutte le pipe, che sono state settate tutte in lettura non bloccante in modo da poter scorre alla pipe successiva in caso di assenza di messaggi. Nel caso ci sia un messaggio, quest'ultimo conterrà la stima di un client quindi il supervisor controllerà la lista che conterrà le stime finali e sostituirà il valore se la stima è migliore di quella che già si possiede oppure ne inserirà un nuovo elemento.

Per la gestione dei segnali ho pensato ad un handler che viene avviato nel momento in cui si riceve un SIG_INT. Incremento di uno la variabile di tipo volatile sig_atomic_t che tiene il conto dei segnali arrivati, stampo le stime calcolate sino a quel momento e chiamo la funzione *alarm()* che manda un segnale SIGALARM dopo un secondo che azzera la variabile volatile.

Se prima di quel secondo arriva un altro SIGINT, comincio la cleanup del supervisor ed invio ai server i segnali si SIGTERM per farli terminare.

4.Server.c

Il server viene lanciato dal supervisor che li passa due argomenti, il numero identificativo del server e il FD della pipe anonima con il quale è connesso al supervisor.

Inizialmente il server genera il suo socket name con il formato OOB-server-n, dove n viene passato dal supervisor, ed usa la funzione *openConnect()* per aprire il socket con quel nome.

Dopodichè si mette in ascolto sul fd della socket per accettare connessioni.

All'arrivo di una richiesta di connessione, creo un elemento di tipo list (contenuta in include.h) e genero un thread che andrà a gestire le richieste del client connesso.

All'arrivo di un segnale SIGTERM, il server comincia a fare la cleanup facendo le join dei thread nella lista di tipo list e liberando la memoria allocata.

a.Gestione connessione (thread)

La connessione con il client è gestita da un thread che non fa altro che aspettare un messaggio sul FD passatoli dal server.

Nel momento dell'arrivo del primo messaggio vengono settati l'id del client e tutte le variabili che permettono il calcolo delle differenze dei tempi tra i vari messaggi.

Dal secondo messaggio in poi si comincia il calcolo della stima che consiste nella sottrazione dei tempi dal messaggio precedente con quello attuale. Si prende il minimo tra le stime calcolate.

Una volta chiusa la connessione da parte del client, il thread invia un oggetto di tipo stimeFinali sulla pipe anonima in modo da farlo arrivare al supervisor.

5.Altro

a.Makefile

Il mio makefile contiene quattro target:

- All, per la generazione degli eseguibili;
- Clean, per la rimozione dei file .o e dei resti del server;
- CleanAll, che rimuove anche i file di log sul quale si fanno le analisi;
- Test, che esegue un il test del codice;

b.Run.sh

E’ un script bash che lancia il supervisor, e poi ogni due secondi i client.

Lo standard out del supervisor viene copiato nel file di log logSuper.

Lo standard out del client viene copiato nel file di log logClient.

Questi due file mi serviranno per fare le analisi del codice da parte dello script misure.sh

c.Misure.sh

Lo script misure.sh è dedicato alle analisi dei file di log del supervisor e dei client.

Parsa inizialmente il file di log del client per ricavare gli id e i secret di ognuno, e i memorizza in due array distinti, dove il secret contenuto nell’array secret[i] corrispenderà a quello dell’id contenuto nell’array id[i].

Successivamente viene parsato il file di log del supervisor ignorando i messaggi dei server e ricavando informazioni solo da quelli del supervisor. Viene creato un nuovo array con le stime finali di ogni client.

Mi ritrovo con una struttura di questo tipo:

| ID | | secret | | stima |

0

1

.

.

A questo punto faccio il confronto tra il secret e la stima di quel client e verifico che questa sia corretta.

Per il calcolo della media dell’errore, sommo tutti gli errori e li divido per la lunghezza dell’array ID.

CONCLUSIONI

Sviluppare questo progetto è stato molto interessante, in quanto è il secondo progetto di medie dimensioni sviluppato nella mia vita, e mi ha permesso di capire come organizzare al meglio qualcosa di così grande. Cominciando dall’analisi preliminare sino alla programmazione in sé.