Tesi progetto laboratorio 2

Leonardo Sinceri

2021/2022

Contents

1 Introduzione

Il progetto consiste sull'invio di informazioni sensibili attraverso la rete, lo scopo è infatti quello di ridurre al minimo la possibilità che un terzo interlocutore (*man in the middle*) possa leggere il traffico di dati in transito e risalire alle informazioni. La tecnica utilizzata per evitare questa situazione è l'*out-of-band signaling*.

1.1 Out-Of-Band Signaling

L'*out-of-band signaling* è una tecnica di trasmissione in cui i dati non si trovano nel corpo del messaggio, ma nella "segnalazione collaterale". Un esempio di out-of-band signaling potrebbe essere l'invio di byte inutili e nascondere l'informazione effettiva nella lunghezza complessiva del messaggio, oppure come nel nostro caso le informazioni sono nascoste nella temporizzazione dell'arrivo dei messaggi.

1.2 Funzionamento generale

Nel progetto saranno presenti dei client, dei server e un supervisor.

Il compito dei client è generare casualmente una coppia id e numero segreto, inviando i dati generati al server. Ovviamente il messaggio non presenterà il numero segreto, ma conterrà solamente l'id del client che lo sta trasmettendo. Quest'ultimo cercarà di inviare messaggi ai server ogni x millisecondi, basati sul proprio numero segreto.

I server dovranno quindi leggere i messaggi inviati dai client e memorizzare il tempo in cui li hanno ricevuti. Alla fine ogni server dovrà comunicare al supervisor la propria stima migliore del secret per ogni client. Una stima esatta è una stima che dista al più 25 dal codice segreto originale: $|x - y| < 25$.

Il supervisor a fine progamma stampa la sua stima migliore per ogni client basata sulle stime ricevute dai vari server.

Il supervisor e i server devono funzionare in modo tale da permettere ai client non sviluppati dallo studente di funzionare correttamente, senza causare errori. Vale anche il viceversa: i client sviluppati dallo studente devono funzionare in modo appopriato con supervisor e server non propri.

2 Stuttura del progetto

2.1 Client

Il programma *client* dovrà accettare 3 parametri da linea di comando:

1. S: il numero totale di server presenti.
2. K: il numero di server a cui il client si deve connettere ($1 \leq K < S$).
3. M: il numero di messaggi da inviare ($M > 3 \times K$).

Dopo aver controllato che i parametri siano stati passati correttamente, il client dovrà generare e stampare sul proprio *stdout* un *client_id* (un numero da 64 bit) e un *secret* (un intero da 32 bit), si connetterà a K server casuali tra gli S attivi e invierà M messaggi scegliendo casualmente per ogni iterazione un server tra i K a cui si è già connesso. Al termine di ogni iterazione aspetta *secret* millisecondi e ripete. Alla fine di questo procedimento il client chiuderà le connessioni aperte per poi terminare.

2.2 Server

Al programma *server* deve essere associato un *server_id* unico (da 1 a S) da parte del supervisor ed ho deciso di implementare questo passsaggio di informazioni semplicemente sfruttando i parametri di ingresso del programma server. Quindi il server riceve come parametro un numero intero, che sarà il proprio *server_id*.

Il testo della prova mette a disposizione due scelte implementative per le connessioni: o utilizzare i domini *AF_UNIX* oppure *AF_INET*. La differenza è che i domini *AF_UNIX* sono associati ad un file speciale e generalmente solo i processi di una stessa macchina si possono connettere, mentre quelli *AF_INET* vengono associati ad un *IP* ed una porta, quindi anche processi esterni dalla macchina si possono connettere. Dato che il testo richiedeva che anche i client di altre persone potessero funzionare ho trovato più sensato utilizzare il dominio AF_INET.

Il server è composto da un thread che si occupa di accettare i client e da 3 thread che hanno il compito di gestire i client a loro assegnati. Quindi dopo aver aperto la socket con dominio *AF_INET*, aver fatto la *bind* della socket al proprio *IP* e porta $9000 + i$ e comunicato al kernel che quella è una socket in recezione tramite la *listen*, il thread addetto ad accettare client aspetta che arrivino richieste tramite la *select*. i client che vengono accettati vengono spartiti in modo equo nei 3 thread competenti. I thread che si occupano di gestire i client aspettano che ci sia un messaggio disponibile in uno qualunque dei client associati tramite la *select*. Se il messaggio è vuoto allora il client ha chiuso la connessione, altrimenti il messaggio contiene un numero e viene salvato il tempo di recezione.

2.3 Supervisor

Il programma *supervisor* deve accettare 1 argomento in input:

1. *S*: il numero totale di server da dover lanciare.

il supervisor quindi è il primo programma a partire, provvederà lui a far partire *S* processi distinti di server e a farli terminare. Solo quando i server saranno attivi allora i client funzionano correttamente (altrimenti durante la *connect* si genererà un errore).

Dopo aver controllato che l'input sia giusto il supervisor crea *S* processi distinti che sono la copia di se stesso, ma subito dopo il programma dei nuovi processi verrà sostituito con il programma del server tramite la *execl*. Una volta lanciati i server, il supervisor attende messaggi da parte di quest'ultimi e terminerà solo al ricevimento di un doppio *SIGINT*.

3 Implementazione

3.1 Come comunicano supervisor e server?

Come canale di comunicazione tra supervisor e server è stato richiesto un segmento di memoria condivisa, che ho scelto di gestire con la tecnica produttore-consumatore.

L'idea che ho avuto è stata quella di creare un struttura (*concurrent_queue_t*) condivisa tra i processi che permetta di leggere e scrivere su un array circolare gestendo la mutua esclusione. La struttura contiene una mutex e due semafori (unnamed) per la sincronizzazione. La mutex garantisce l'accesso all'array in modo esclusivo, il semaforo read, inizializzato con valore 0, garantisce la lettura solo in caso ci siano valori all'interno dell'array, altrimenti il thread che prova a leggere viene bloccato su *wait(read)*. Il semaforo write, inizializzato con valore *ARRAY_CAPACITY*, garantisce la scrittura solo in caso l'array abbia delle posizioni libere, altrimenti il thread che prova a scrivere viene bloccato su *wait(write)*. Dopo la lettura il semaforo write viene segnalato tramite *post(write)* per comunicare la presenza di un posto disponibile, allo stesso modo, dopo la scrittura il semaforo read viene segnalato della presenza di un elemento da leggere *post(read)*.

3.2 Come vengono chiusi supervisor e server?

La richiesta di terminazione del supervisor è l'invio di un doppio *SIGINT* in meno di un secondo. Una volta ricevuta questa richiesta di terminazione il supervisor avviserà tutti i server e aspetterà che quet'ultimi terminino per poi procedere a deallocare le risorse utilizzate. Per la terminazione dei server ho utilizzato la stessa tecnica utilizzata dal supervisor, ovvero la gestione del segnale *SIGINT*.

Quindi sia il supervisor che i server hanno un thread aggiuntivo il quale è l'unico in grado di ricevere i segnali, ciò avviene grazie all'utilizzo di apposite maschere. Il thread gestore dei segnali avrà nella maschera il bit *SIGINT* settato ad *true* mentre tutti gli altri thread avranno un maschera vuota, questo comporta che nessun segnale sarà gestito da loro. La sola differenza tra supervisor e server, riguardante la teminazione, sta nel numero di segnali da aspettare, infatti come detto precedentemente il supervisor dovrà aspettare un doppio *SIGINT* in meno di un secondo mentre il server ne aspetterà uno solo.

3.3 Come viene stimato il secret?

In ogni server rivceverà da 0 a N messaggi da parte dello stesso client. Si formano quindi tre casi distinti:

1. Nel caso in cui il server non riceva alcun messaggio da parte del client non può supporre nulla riguardo al secret e non viene nemmeno inviato l'id del client, quindi non viene inviato nulla al supervisor.

2. Invece nel caso in cui il server riceva un solo messaggio da parte del client, sarebbe noto soltanto i'id e non potrebbe fare nessuna supposizione sul secret. Quindi viene inviato il valore massimo del secret al supervisor per quel client.

3. Il caso più comune invece è quando il server riceve due o più messaggi e la stima del secret migliore è il ritardo di tempo minimo tra due messaggi consegutivi ($\min\{X_i - X_{i+1} : \forall\, i = 1, ..., N-1\}$).

Il supervisor invece riceve le migliori stime per ogni server e per scegliere la migliore seleziona quella con valore più basso.

4 Conclusione

Data la grandezza del progetto per mantenere ordine nella workstation ho creto molteplici cartelle. Le cartelle *client*, *server* e *supervisor* ovviamente contengono solo il codice (librerie e main) dei rispettivi. Le strutture condivise da due o da tutti i programmi sono state riposte nella cartella *shared*. Gli eseguibili e i file oggetto vengono creati nella cartella *executable*. Infine nella cartella *output* sono presenti dei file di testo contenti l'output dei supervisor, servers e clients lanciati da *make test*. Inoltre su *output* sono contenuti i file temporanei creati dallo script *misura*, il quale controlla quanti secret sono stati stimati correttamente e qual'è l'errore medio delle stime fatte.