OUT-OF-BAND SIGNALING

Università di Pisa
Dipartimento di Informatica
Sistemi Operativi e Laboratorio
A.A 2019/2020

Matteo Montalbetti
MAT.559881

Introduzione

"Lo studente dovrà realizzare un sistema per l'out-of-band signaling. Si tratta di una tecnica di comunicazione in cui due entità si scambiano informazioni senza trasmettersele direttamente, ma utilizzando segnalazione collaterale: per esempio, il numero di errori "artificiali", o la lunghezza di un pacchetto pieno di dati inutili, o anche il momento esatto delle comunicazioni. Nel nostro caso, vogliamo realizzare un sistema client-server, in cui i client possiedono un codice segreto (che chiameremo secret) e vogliono comunicarlo a un server centrale, senza però trasmetterlo. Lo scopo e rendere difficile intercettare il secret a chi stia catturando i dati in transito. [...] "

Ho deciso di impostare la relazione seguendo la struttura del testo del progetto. La presentazione di esso, infatti, prevede una descrizione accurata di ogni modulo che dovrà essere realizzato.
Riprendo la stessa divisione per specificare, modulo per modulo, alcuni aspetti e/o scelte che ho ritenuto necessario indicare.

Client *("client.c")*

La prima particolarità che si incontra leggendone il codice riguarda sicuramente la generazione dell'intero a 64 bit. Per dichiarare un intero di questo tipo, in C, è presente il tipo *uint64_t*.
Nasce però subito un problema: l'*id* doveva essere generato in maniera pseudo-casuale dal client, ma il metodo *rand()* lavora con interi di 32 bit. Ho ritenuto allora necessario dichiarare due variabili di tipo *uint32_t* le quali, una volta generate attraverso *rand() e poi* combinate, mi hanno permesso di ottenere un intero di tipo *uint64_t* e quindi di 64 bit.
Ma perchè abbiamo usato proprio quel tipo e non, ad esempio, *long long int*? Il progetto richiede l'invio dell'*id* (ai server) in *network byte order*; la conversione viene fatta tramite la funzione *htonl*, la quale lavora con variabili di tipo *uint32_t*.

La formula utilizzata per la combinazione è la seguente:

[client.c, riga 143]

final_id = (uint64_t) id_one << 32 | id_ two

Un altro punto che trovo particolarmente interessante è l'inizializzazione del generatore pseudo-casuale. Sebbene per la maggior parte delle volte un generatore viene inizializzato attraverso un banale *time(NULL)*, nella nostra situazione questo avrebbe portato ad alcuni problemi. Durante lo sviluppo del progetto è richiesto di creare uno script *test*, il quale, ad un certo punto, lancia 20 client, 2 alla volta. Quei due client lanciati insieme sono praticamente eseguiti simultaneamente e pertanto *time(NULL)* risulterebbe il medesimo per entrambi. Questo avrebbe voluto dire che quei due client avrebbero generato, nello stesso ordine, gli stessi numeri casuali, ed avrebbero quindi avuto stesso *id* e stesso *secret*. Ciò non ci piace! Ho deciso quindi di far dipendere il *seed* del generatore non solo da un fattore tempistico, ma anche dall'id del processo, differente per ogni client.

Server *("server.c")*

Il server comunica in maniera bidirezionale sia con i *client*, tramite *socket*, sia con il *supervisor*, tramite *pipe anonime*.
Responsabilità del server è il meccanismo di comunicazione *server-client*, che ho deciso di implementare attraverso il *multithreading*. Il processo principale, lanciato dal *supervisor,* funge quindi da *dispatcher*, attendendo connessioni da parte dei *client*. All'arrivo di una nuova connessione, il *dispatcher* procede al lancio di un thread *worker*, il quale si occuperà di quella precisa connessione con quel determinato *client*.

Altro punto interessante del *server* riguarda la gestione dei segnali. Ho reputato necessario installare due gestori, per due diversi segnali. In primis, i *server* sembrano non avere una condizione di terminazione, ed effettivamente così è. Questi rimangono in esecuzione fino a quando il *supervisor* resta attivo, ed è quindi la terminazione di quest'ultimo ad implicare la terminazione dei vari *server*.
Diventa quindi compito del *supervisor* terminare i vari *server*, essendo processo padre di essi; questo meccanismo è stato implementato attraverso i segnali. Nel momento in cui il *supervisor* si trova in condizione di terminare, durante le operazioni di pulizia della memoria, invia un segnale (attraverso la *kill*) *SIGTERM* a tutti i server, i quali reagiscono alla ricezione di questo terminando, dopo aver liberato in maniera corretta lo spazio di memoria da loro utilizzato.
La seconda operazione inerente ai segnali è in realtà un ripristino: il *supervisor*, prima di fare la *fork*, installa il proprio gestore da eseguire in caso di *SIGINT*. Questo fa sì che i vari server, processi figli di *supervisor*, ereditano la seguente modifica; ho quindi resettato il comportamento di *server* in caso di *SIGINT* installando come gestore *SIG_IGN*.

Ultimo aspetto importante: come fanno i thread a stimare il *secret*? Dal momento che il *client* invierà, ad intervalli temporali "multipli di *secret*", messaggi ai vari server (e quindi ai vari

thread), questi ultimi, misurando il tempo trascorso tra i vari messaggi ricevuti, riescono a stimare il *secret*. Queste stime potranno essere: il *secret* (se il client ha inviato per due volte consecutive il messaggio allo stesso server) o un multiplo di esso. Per questo motivo, ogni thread gestisce una variabile in cui memorizza la stima **minore** fino a quel momento.
Successivamente, tra le varie stime dei vari thread che verranno poi comunicate al *supervisor*, è considerata migliore nuovamente quella **minore**.

Supervisor *("supervisor.c")*

Il *supervisor* ha bisogno di una struttura dati in cui memorizzare le varie stime di ogni client. Ho deciso di utilizzare, ed implementare, una *linked-list,* strutturata come segue:

[ID[0] , ESTIMATE[0] , COUNTER[0]] → … → [ID[N] , ESTIMATE[N] , COUNTER[N]]

Ho quindi definito le struct necessarie alla creazione di questa e alcuni metodi che ho ritenuto utile implementare: aggiunta di un elemento, stampa della lista (sia sullo *stdout* che sullo *stderr*), member, aggiornamento di una stima e infine l'eliminazione (con pulizia della memoria) della struttura.

Come ho già detto, la comunicazione *supervisor-server* avviene tramite l'utilizzo di *pipe anonime*. Per gestire la presenza di più *pipe* ho deciso utilizzare una *select*.

L'ultima cosa da indicare è il criterio di valutazione delle stime: *supervisor* riceverà più stime dai vari server. Se una stima è riferita ad un client non ancora registrato nella tabella di *supervisor* si procede ad una semplice aggiunta. In caso contrario, bisogna capire quale tra la nuova stima e la precedente deve essere considerata migliore; per fare ciò si segue lo stesso ragionamento utilizzato nell'implementazione del *server*: "vince" la stima minore.

Devo però aggiungere un'ulteriore spiegazione; riprendendo il commento scritto nel file *supervisor.c, riga 538:*

"... Testando però un determinato numero di volte il mio progetto, ho potuto osservare che tutte le volte in cui una stima non viene fatta correttamente l'errore è di circa il doppio. Se ad esempio il SECRET fosse 1600, in caso di errore verrà stimato 3200. Ho deciso allora di inserire un controllo: se la stima più recente è minore di quella già presente, la tabella si aggiornerà sicuramente; se però la nuova stima (pur essendo minore della precedente) è maggiore di 3000, allora vado ad inserire la metà di essa. Posso farlo perché il SECRET non potrà mai essere maggiore di 3000: se la migliore stima dovesse essere tale, è per forza errata!

Rimane però un problema. Che succede se, ad esempio, il SECRET dovesse essere 400 e arriva una stima di 800? In questo caso non posso inserire la metà, perché 800 potrebbe benissimo essere un altro SECRET!

Questo tipo di situazione rimane l'unico caso (dico questo basandomi sui test fatti fino ad ora) in cui si verifica un errore nelle stime; … "

Misurazione *("misura.sh")*

Tramite lo script *test.sh* lo *stdout* e lo *stderr* dei vari programmi vengono rediretti in due file di testo: *server* e *supervisor* "scrivono" nel file di testo *supervisor.txt*, mentre *client* "scrive" all'interno del file *client.txt*.

Tramite lo script *misura.sh* ho gestito i file di testo appena citati per arrivare a fare alcune misurazioni richieste dal progetto; di seguito elenco i comandi *bash* che ho utilizzato per tale scopo.

- ***tail -n*** → estrae da un file di testo le ultime -n righe;
- ***sed /pattern/d*** → elimina da un file di testo tutte le righe contenenti *pattern*;
- ***echo*** → stampa sullo *stdout;*
- ***cut -d ' ' -f n*** → è una specie di tokenizer; recupera la parola in posizione n (dividendo la stringa in base allo spazio);
- ***bc*** → calcolatrice per lavorare con numeri decimali (importante per restituire l'errore medio);
- ***operatori per il confronto, operatori aritmetici***

Per ulteriori informazioni e una maggior precisione, consultare i commenti inseriti nel file *misura.sh.*

Ho deciso di inserire, come ultima operazione dello script, la stessa procedura di pulizia fatta da *make clean*, in modo tale da non dover ogni volta lanciare quest'ultimo.

Makefile e Test *("makefile* e *test.sh")*

Ho compilato il *makefile* partendo da un modello preimpostato, fornito l'anno scorso agli studenti.
Ho semplicemente seguito il pattern suggerito, personalizzandolo in base alla mia situazione, andando così a creare le regole relative agli eseguibili *client*, *server* e *supervisor*.
Ho dovuto inserire alcune dipendenze aggiuntive; vedremo nella sezione *EXTRA* quali sono queste dipendenze.

Ho infine aggiunto il target *clean*, con il rispettivo comando di pulizia, e il target *test*, il quale lancia lo script *test.sh*.

Per quanto riguarda *test.sh*, lo script non fa niente di più, niente di meno, rispetto a ciò che chiede il progetto.

Extra

Durante la realizzazione del progetto ho deciso di creare tre file esterni, per raccogliere alcune funzioni che ho definito, così da avere un codice più pulito ed una migliore organizzazione. La spiegazione accurata di ogni funzione è contenuta all'interno dei vari file sotto forma di commenti.

I file creati sono i seguenti:

- ***pipes.c*** *(pipes.h)* → mi serviva una struttura dati che potesse tenere traccia di tutti i descrittori dei file relativi alle pipe create. La creazione, gestione ed eliminazione di questo array è responsabilità di questo file;
- ***err-managements.c*** *(err-managements.h)* → ogni funzione di sistema (e non) prevede una notifica dell'errore, spesso basata sul valore che questa ritorna. Sarebbe necessario, o comunque preferibile, ogni qual volta una di questa funzioni viene invocata, fare dei controlli sull'esito di essa, in modo da definire il comportamento da tenere in caso di errore. Ho quindi ridefinito alcune funzioni in questo file, in modo tale da avere un codice più pulito in *supervisor, client* e *server*;
- ***aux-functions.c*** *(aux-functions.h)* → durante il corso del progetto ho ritenuto necessario creare alcune funzioni di supporto per l'esecuzione di alcune operazioni. Per tenere nei vari file il codice più pulito ho raccolto queste in un unico foglio c.

Infine, ho cercato di gestire la memoria quanto più in maniera corretta; testando il progetto attraverso il comando di debug *valgrind*, il risultato è stato positivo. Inoltre, per quanto riguarda il rischio di *"processi zombie"*, il comportamento del progetto sembra essere corretto. *(Il test è stato fatto lanciando ./supervisor 8 e ./client 5 8 16)*

Vista la sintesi richiesta da questa relazione, ho cercato di commentare, nel modo più chiaro ed esaustivo possibile, ogni passaggio fatto durante lo sviluppo del progetto. Ogni file (dove possibile) è stato, in modo più o meno dettagliato, commentato.