Relazione Progetto: Out-of-Band Signaling

Lapo Frati 489617

Università di Pisa

Sessione estiva a.a. 2013/2014

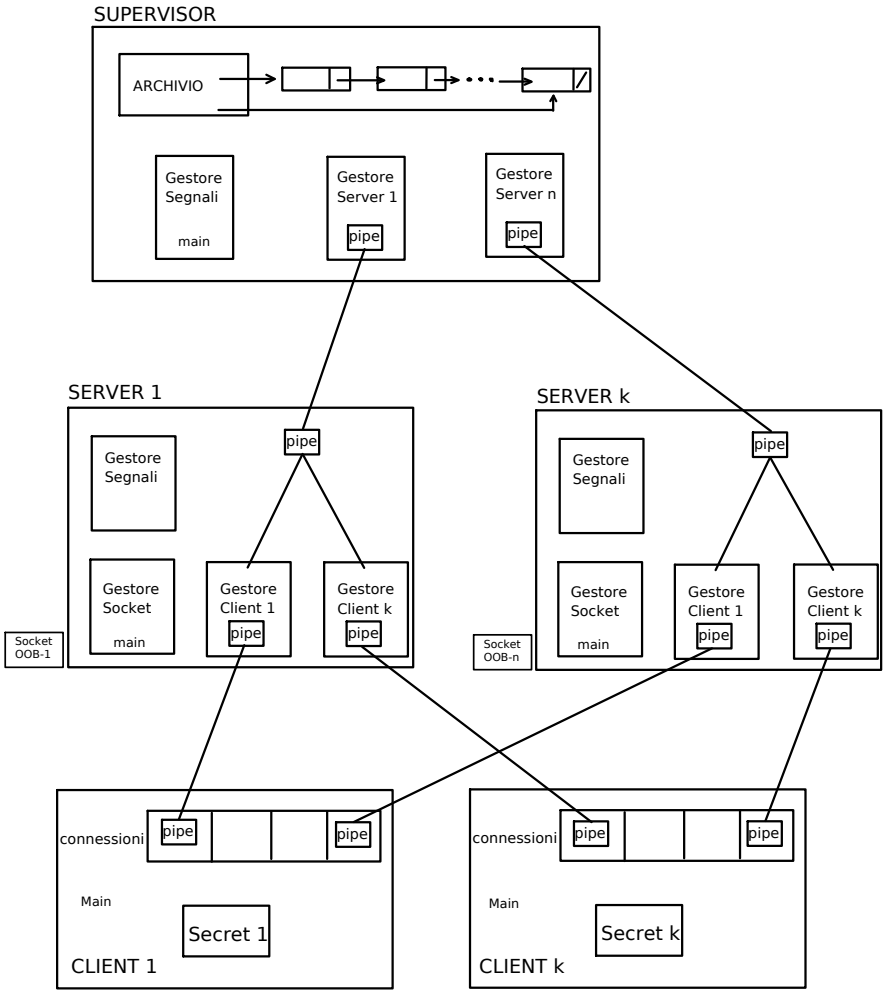
Contents

1 Introduzione

Il progetto implementa un sistema di comunicazioni criptate *out-of-band signaling*: i Client comunicano al Supervisor un loro $0 < secret \leq 3000$, nascondendolo nei tempi tra due messaggi consecutivi.

Le chiamate di sistema utilizzate sono conformi allo standard **POSIX.1-2001**

2 Schema Progetto



3 Client

Client è un programma singlethread, singleprocess.

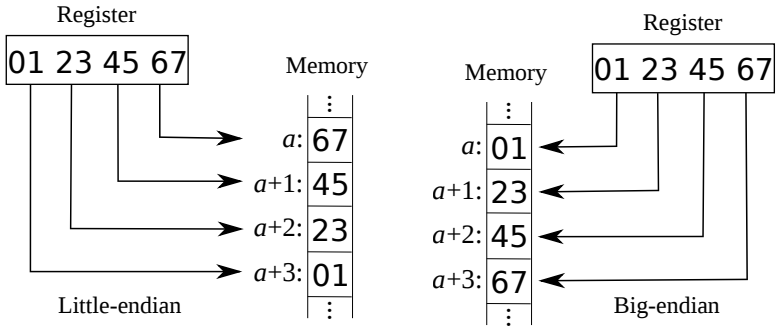
Il client inizializza il proprio socket alla family **AF_UNIX** come da specifica, dopodichè crea il proprio *secret* ed ID . Poichè il suo ID deve essere a 64 bit lo alloca come *uint64_t*. Per generare l'ID casualmente non è stato possibile utilizzare una semplice *rand*() poichè genera un risultato compreso tra 0 e RAND_MAX che è assicurato essere almeno $2^{15} - 1$ (15 bit random) e servono quindi almeno 5 rand() per generarne 64. Pertanto il numero casuale è stato realizzato nel seguente modo:

- *rand*() viene inizializzata usando come seme sia *time*() che il pid del client per una maggiore sicurezza
- Alloca un puntatore a *uint8_t* e viene fatto puntare all'ID, trattandolo come un array di 8 numeri di 8 bit.
- Per ogni elemento dell'array è stata effettuata una chiamata di *rand*()

Dopo aver generato l'ID casuale provvede a stamparlo come sarà stampato da server/supervisor; per fare ciò controllo se la macchina su cui sono è little o big endian:

- *little-endian*: rovescio l'ordine dei byte e poi lo stampo
- *big-endian*: lo stampo così come è

Come controllo se la macchina è big o little endian?
Creo una variabile *uint32_t* $v = 0x01234567$ cosicchè:



Pertanto, quando con un puntatore *uint8_t* accedo al valore del primo byte se questo vale 67 allora la macchina è *little endian* , in caso contrario la macchina è *big endian*.

Dopo aver selezionato i server a cui connettersi ed aver completato la connessione (può effettuare tre tentativi con attese crescenti prima di fallire) il client comincia un ciclo di invii di messaggi ed attese. Ad ogni iterazione seleziona casualmente uno fra i server a cui si è collegato, invia il proprio ID e poi attende secret millisecondi tramite *nanosleep*()(se questa viene interrotta prematuramente viene fatta ripartire con il tempo rimanente).

Il client termina quando ha completato gli invii previsti o se riceve una SIGPIPE in seguito alla write su una pipe il cui corrispondente server è stato chiuso

4 Supervisor

Supervisor è un programma multithread, multiprocess.

4.1 Main Thread

Il supervisor dopo una fase di inizializzazione genera un'array di unnamend pipes con cui comunicherà con i server suoi figli.

Provvede a bloccare i segnali SIGINT e SIGTERM (blocco che sarà ereditato dai server) e comincia un ciclo di execlp con cui genera i server, passando come argomenti il loro Server-ID (un numero progressivo che li identifica) ed il file-descriptor con cui invieranno i loro dati al supervisor.

Finito di lanciare i server, tramite *pthread_setdetachstate*() e *pthread_setstacksize*(), prepara la creazione di thread detached (non saranno conservate le strutture necessarie per un join postuma) e diminuisce lo stack dei thread (su Linux è 2MB di default e poichè il thread esegue operazioni molto semplici comportava un inutile spreco di spazio).

Dopodichè entra in un ciclo in cui, per ogni server, crea un thread che eseguirà "addetto_connessioni" (vedi dopo).

Infine inizia un loop per la gestione dei segnali SIGINT e SIGCHLD, mettendosi in attesa di essi tramite *sigwait*():

- SIGINT: Alla prima ricezione di SIGINT stampa su stderr l'archivio stime. A questo punto usa una *sigtimedwait*() per controllare se, entro un secondo, riceve un nuovo SIGINT. Nel caso in cui riceva il secondo SIGINT effettua una seconda stampa (stavolta su stdout) e termina

il programma, avviando la chiusura dei server tramite le funzioni di cleanup

- SIGCHLD: Poichè i server vengono terminati dal supervisor, se questi riceve un SIGCHLD significa che qualcosa è andato storto e provvede a terminare correttamente il programma, chiudendo i server tramite le funzioni di cleanup

4.2 Thread: addetto connessioni

Ogni thread che esegue "addetto_connessioni" è assegnato ad una specifica pipe (ricevuta come argomento) dalla quale riceve i messaggi di un Server.

Effettua un ciclo in cui legge, tramite *readall*() (funzione per leggere descritta in Advanced Unix Programming 2nd Edition pag. 97, probabilmente fin troppo sicura in questo contesto poichè i messaggi sono di piccole dimensioni), le struct msg ,inviategli dal suo server, contenenti la stima ed il client-ID a cui quella stima si riferisce.

Con i dati ricevuti aggiorna una lista globale in cui sono salvate per ogni client le migliori stime disponibili, ottenute selezionando ogni volta la minima (per un'analisi di questa strategia e possibili altre tecniche vedere l'altro pdf: "Possibili Strategie di Server e Supervisor").

L'accesso alla lista delle stime è controllato da un lock, per evitare incosistenze durante l'aggiornamento della lista e cosicchè nel caso si ricevano più richieste di stampa a distanza ravvicinata, queste non si sovrappongano.

Infine, se l'addetto_connessioni legge un EOF sulla sua pipe, provvede a chiudere tutto e terminarsi.

5 Server

Server è un programma multithread, singleprocess.

5.1 Main Thread

Una volta avviato il server, dopo una fase iniziale di inizializzazione, provvede a creare un thread che gestisca i segnali ed una socket (AF_UNIX) cui i client possono connettersi per comunicare con lui. Ad ogni accept, crea un thread che esegue "addetto_connessioni" e gli assegna la pipe con il client che si è collegato. (Nota: per i thread creati valgono le stesse considerazioni fatte nel supervisor riguardo detached-state e stack-size)

5.2 Thread: gestore segnali

Il gestore segnali non fa che attendere, tramite *sigwait*(), un'eventuale SIGTERM inviatogli dal supervisor. Nel caso lo riceva, provvede a terminare chiudendo e ripulendo tramite le funzioni di cleanup.

5.3 Thread: addetto connessioni

L'addetto_connessioni non fa altro che eseguire un ciclo in cui effettua *readall*() sulla pipe di cui si occupa.

Alla prima ricezione di un messaggio salva il tempo e verifica l'endianess della macchina con la stessa tecnica usata dal client:

- *little-endian*: salvo e stampo l'ID dopo aver roversciato l'ordine dei suoi byte
- *big-endian*: salvo e stampo l'ID così come l'ho ricevuto

In seguito, ogniqualvolta riceve un messaggio, controlla tramite *gettimeofday*() quando è arrivato e lo confronta con il tempo di arrivo del precedente messaggio. Con questa differenza, stima il secret del client con cui sta comunicando e se è la più piccola fin'ora ricevuta la salva, per inviarla in seguito al supervisor insieme all'ID del client corrispondente. Il ciclo prosegue fintanto che il valore *nread*, risultato di readall, è > 0. Quando esce controlla se:

- $nread = 0$: il client ha terminato i suoi invii, invia al supervisor la mia migliore stima e provvede a chiudere la pipe prima di terminare il thread.
- $nread < 0$: si è verificato un errore.

Infine chiude la pipe da cui riceveva i dati dal client, elimina il msg usato per inviare i suoi risultati al supervisor e conclude chiamando *pthread_exit*().