Relazione Progetto
Laboratorio Sistemi Operativi
a.a. 2018/2019

"Out-of-band signaling"

Cristiana Angiuoni
matricola 546144

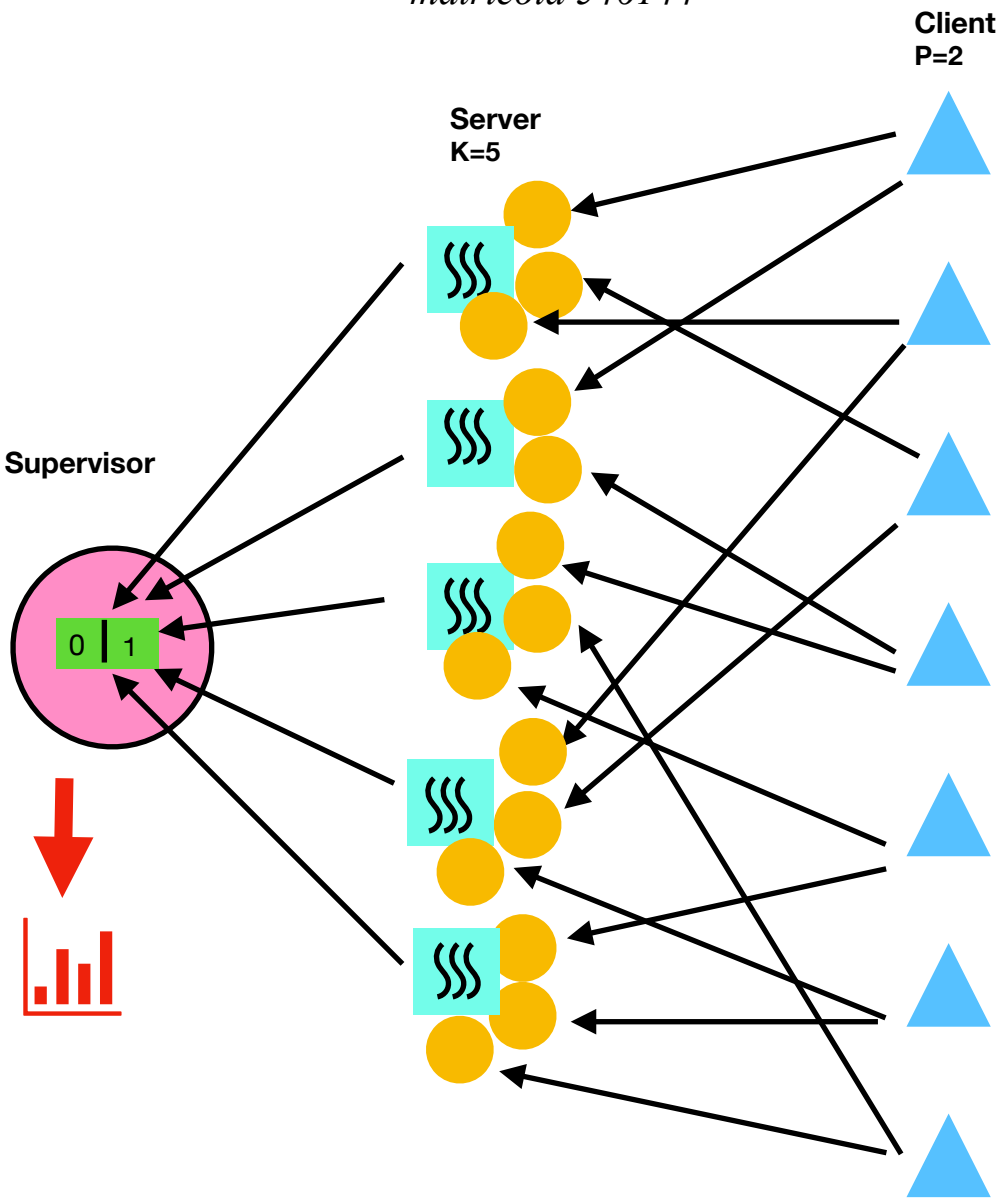

***Fig 1**. Schema generale di comunicazione tra le tre componenti: supervisore, server e client.*

= SOCKET

= PIPE

A → **B** = A invia msg a B

= SERVER MULTITHREAD : un thread per ogni connessione

Organizzazione delle cartelle:

- **build:** contiene i *file.o* e gli eseguibili.
- **lib:** contiene le librerie statiche create per supervisor, server e client.
- **header:** contiene i *file.h*, i quali contengono le dichiarazioni di funzione e le definizioni dei tipi di dato astratti utilizzate nei sorgenti.
- **test:** contiene *test.sh* e il file *misura.sh*

1. Introduzione ai file:

- **SUPERVISOR**:

FILE DEL SUPERVISOR	CONTENUTO
`supervisor.c`	Main e invocazione delle funzioni di avvio del supervisore, gestione segnali e creazione dei processi figli server.
`support_supervisor.c`	Implementazione delle funzioni usate dal supervisore per gestire i segnali e la connessione con i server.
`supervisor.h`	Dichiarazione delle funzioni e dei tipi di dato astratti usati dal supervisor. (con specifica)

- **SERVER:**

FILE DELSERVER	CONTENUTO
`server.c`	Main e invocazione delle funzioni di avvio del server
`support_server.c`	Implementazione delle funzioni usate dal server per accettare connessioni e creare i thread workers
`worker.c`	Implementazione della funzione svolta da ogni thread worker per il calcolo della stima del secret.
`support_client.h`	Definizione delle funzioni e delle struc usate dal server (con specifica)
`worker.h`	Dichiarazione delle funzioni e dei tipi di dati astratti usati dai workers.

- **CLIENT:**

FILE DEL CLIENT	CONTENUTO
`client.c`	Main e invocazione delle funzioni di avvio dei client
`support_client.c`	Implementazione delle funzioni usate dai client per connettersi ai server e inviare l'ID
`client.h`	Dichiarazione delle funzioni e dei tipi di dati astratti usati dal client (con specifica).

- **`Utils.h`** : contiene le macro per gestione degli errori, differenziando quelle utilizzate per gestire gli errori nei processi e quelle per gestirli nei thread.
- **`Test.sh:`** script che manda in esecuzione il supervisor (stampando "`Eseguo il supervisore`") e ridirige lo standard output nel file `output_supe.log`. Poi manda in esecuzione i client a coppie (ridirigendo lo standard output sul file `output_client.log`), stampa "`Eseguo i 20 client`" e invia 6 segnali SIGINT singoli a distanza di 10 secondi l'uno dall'altro. Per capire come evolve l'esecuzione ad ogni iterazione stampa l'indice del segnale e le stime parziali dovute alla ricezione di SIGINT. Quando arriva il doppio SIGINT viene lanciato il file `misura.sh` a cui passo i due file precedentemente scritti, dopo aver stampato "`Terminazione supervisor`".
- **`Misura.sh:`** riceve in input i file `output_super.log` e `output_client.log` e, ridirigendo lo standard input sui file, la *read* li scorre riga per riga. Ogni volta che viene letta la parola "BASED" e "CLIENT SECRET" memorizza rispettivamente le stime nell'array *stime* e i secret nell'array *secret*. Dopo ciò confronta i secret veri con quelli stimati e calcola la differenza tra essi in valore assoluto. Tenendo conto della tolleranza di 25 unità, stampa la percentuale di stime correttamente calcolate e l'errore medio sulla base del numero totale di stime. Se la differenza è maggiore della tolleranza verrà stampato il secret sbagliato e e il relativo id del client, altrimenti stampa solo la percentuale di correttezza e l'errore medio.

- **`Makefile:`** permette di compilare tutti i file e creare tre librerie statiche divise per moduli concettuali: *libclient.a, libserver.a, libsupervisor.a*. Inoltre, come richiesto dalla specifica, contiene i PHONY TARGET: all, cleanall e test. Si trova nella directory principale, insieme a tutti i file sorgenti.

Compilazione ed esecuzione: il codice è stato scritto e compilato in c99 ed è stato testato con Valgrind per evitare memory leak. Fare *make* e *make test* per mandarlo in esecuzione.

2. Componenti principali:

- **SUPERVISOR:**

rappresentato dalla struttura :

```
typedef struct superv{
     int pipe[2];
     pid_t* serv;
}superv_t;
```

E' il processo principale che verrà mandato in esecuzione per primo e, dopo aver configurato le strutture per la *gestione dei segnali*, crea una *pipe anonima* e, attraverso la chiamata *fork( )* crea e sovrascrive il codice dei k server figli, tramite la funzione *execl( )* a cui sono passati due parametri per mandare in esecuzione ogni server figlio:

- l'indice che rappresenterà quel server (assegnato dal supervisore)
- valore del *file descriptor* della pipe anonima (lato scrittura) condivisa da ogni figlio, attraverso cui comunicare con il padre, dove ognuno manderà la propria migliore stima calcolata.

Quando riceverà i segnali *SIGINT* si preoccuperà di stampare le stime ricevute, salvando solo la minima stima (in alternativa il MCD) calcolata fino a quel momento dai diversi server per ogni client in un array di struct di size *NCLIENT* semplice da gestire. Sulla base del test richiesto dalla specifica del progetto la variabile globale *NCLIENT* è settata a 20, ma nel caso in cui si volesse modificare il numero iniziale di client sarà sufficiente modificare la variabile globale. Visto il numero ridotto di client non è stato necessario utilizzare una struttura dati complessa, utile invece se fossero in numero superiore o una struttura dinamica se il numero di client fosse stato variabile.

1.2 SERVER:

rappresentato dalla struttura:

```
typedef struct server{
    int fdsock;
    int i;
    int fdpipe;
}server_t;
```

Si tratta di un *server multithread*, che esegue un ciclo infinito in cui accetta richieste di connessione dai client e per ognuna di esse, viene creato un thread worker in *modalità detach*, affinché si dedichi solo a un client alla volta. Dopo aver calcolato la stima del secret del client, in base al tempo di ricezione di un messaggio e il successivo, ogni *thread worker* invia sulla pipe del supervisor un buffer con 3 informazioni: *id_client, stima, id_server.* Per ottenere una stima più accurata possibile ogni *worker* controlla che il proprio calcolo sia inferiore a 3000, perché in caso contrario sarà già cosciente del fatto che il secret vero del client è almeno la metà. A quel punto divide finché non ottiene una stima migliore della precedente, essendo un multiplo. Se il server riceve un solo messaggio dal client non invia niente al supervisor, perchè il fatto che il numero di messaggi inviati dal client sia w>3p garantisce che al supervisore arriveranno comunque delle stime. Alla fine,

ogni worker dealloca la memoria, chiude il fd della socket e termina, mentre il server continua a rimanere in attesa di ulteriori richieste da parte di altri client.

1.3 CLIENT:

rappresentato dalla struttura:

```
typedef struct client{
    uint32_t id
    int secret
}client_t;
```

Il client genera in modo pseudo-casuale il suo ID univoco di 32 bit, attraverso una funzione che combina i bit in XOR e AND, utilizzando una maschera, partendo da un seme unico dato dalla funzione *srand( ),* sulla base dei millisecondi, in modo che anche i due client lanciati in coppia a distanza di un millisecondo abbiano ID diverso e univoco. Dopo essersi connesso a *p* server scelti random tra i k attivi tramite socket, invia il proprio ID a ciascuno di essi, ogni volta selezionando un server casuale diverso grazie alla *rand()*.

2. Comunicazione Supervisor-Server:

2.2 Comunicazione tramite pipe anonima condivisa: ognuno dei k server creati dal supervisore scrive sul canale condiviso e il supervisore si limita a leggere in un ciclo infinito. Non si è ritenuto necessario utilizzare un meccanismo di mutua esclusione, in quanto la chiamata di sistema *write( )* risulta essere atomica, visto che il numero di byte scritti (*id_client* + *stima* + *id_server*) è minore della capacità della pipe. Per questo motivo non possono avvenire interleaving dei dati scritti da processi diversi sulla stessa pipe. Inoltre, siccome una *read( )* eseguita su una pipe vuota è bloccante, il supervisore rimane in attesa che qualcuno scriva sulla pipe, finché c'è almeno un descrittore aperto in scrittura.

3. Comunicazione Server-Client:

3.2 Comunicazione tramite socket: ogni client cicla *p* volte per connettersi ai *p* server scelti random tra i *k* attivi. Ad ogni iterazione la chiamata di sistema *socket()* restituisce il file descriptor su cui instaurare la comunicazione con quel server, il quale viene inserito in un array di fd. A questo punto il client invierà il proprio ID ad ognuno di essi, scegliendo a caso uno dei fd dell'array. Dopo aver mandato *w* messaggi, chiude tutti i file descriptor, cioè le socket lato client ed esce.

4. Terminazione:

4.1 Terminazione dei server da parte del supervisore: il supervisor mantiene un array di pid dei *k* figli che utilizzerà per la loro terminazione, tramite l'invio del segnale *SIGTERM* gestito con *signal_handler()*.

4.2 Terminazione del supervisor: il supervisore terminerà non appena riceverà un doppio segnale *SIGINT* a distanza di meno di un secondo (contato grazie alla funzione *alarm(1)*. Una volta terminato il supervisor, termina il programma preoccupandosi di far terminare i figli e di conseguenza tutti i thread.

5. Gestione dei segnali:

- **SIGINT:** gestito con un flag volatile *sig_atomic_t sig_print* che, grazie al gestore *sigint_gest( ),* viene settato in caso di ricezione del segnale SIGINT singolo e permette la stampa delle stime parziali su stderr, inoltre viene mandato un allarme di un secondo. La funzione è breve e semplice come la buona prassi impone debba essere un gestore di segnali. Non fa altro che settare i flag, poi a livello applicativo viene invocata la funzione di stampa su stderr delle stime finali, tramite *print_signal()*.

- **SIGALRM:** viene mandato per capire se è passato un secondo dal segnale di SIGINT grazie ad *alarm(1)* ed è gestito dalla funzione sigalarm_gest che si limita a resettare un flag *volatile sig_atomic sig_ararm*, precedentemente settato da sigint_gest prima di mandare l'allarme. In questo modo se sig_alarm=1 non si deve terminare, perché è passato più di un secondo, altrimenti viene settato il flag *volatile sig_atomic sig_exit* e quando viene testata tale variabile nel ciclo principale del supervisore avviene la stampa su stdout.

- **SIGTERM:** inviato tramite *kill( )*, chiamata in *kill_chils( ),* registrata grazie alla atexit() per essere eseguita alla fine del programma. Essa permette di uccidere i k server figli, i cui pid sono memorizzati in un array mantenuto dal supervisor. Il segnale *SIGTERM è* l'unico che i server gestiscono con un handler*, mentre SIGINT e SIGALRM vengono ignorati. Per la gestione di SIGTERM si usa una variabile sig_atomic_t sig_term che viene settata dal signal_handler* in caso di ricezione del segnale e testata nel ciclo di read dalla pipe. *Se arriva SIGTERM* viene liberata la memoria e si esce. Lo stesso vale per i thread worker che ereditano dal server la maschera dei segnali.