Laboratorio di Reti di Calcolatori

Esecuzione client-server: analisi e gestione degli errori.

Paolo D'Arco

Abstract

Scopo della lezione è mostrare come analizzare l'esecuzione di un client ed un server, evidenziando i problemi tipici che si presentano durante l'esecuzione (e.g., terminazione del server, crash dell'host o della rete ...) e presentare alcune contromisure che possono esser prese per scrivere applicazioni robuste.

1 Analisi dell'esecuzione: echo client ed echo server

Per semplicità mandiamo in esecuzione server e client sulla *stessa* macchina, utilizzando due terminali testuali diversi. Un client può far riferimento ad un server che gira sulla stessa macchina, indicando come indirizzo IP il valore 127.0.0.1 (indirizzo di *loopback*). Non c'è nessuna perdita di generalità. Se il server fosse in esecuzione su una macchina diversa della interrete, il client invece di 127.0.0.1 dovrebbe usare l'indirizzo IP dell'altra macchina.

Attraverso il comando *netstat* possiamo visualizzare lo stato delle connessioni stabilite e raccogliere informazioni circa l'esecuzione di client e server.

1.1 Esecuzione normale

Mandiamo in esecuzione il server sul primo terminale

```
macbook-di-paolo-darco:FIN pd$ ./echosrv 9999
```

e mandiamo in esecuzione il client sul secondo

```
macbook-di-paolo-darco:FIN pd$ ./echocli 127.0.0.1 9999
```

Usiamo ./ prima del nome del client/server con i parametri perchè per default il sistema operativo cerca i file eseguibili in apposite directory (specificate nella variabile di ambiente PATH). Per ragioni di sicurezza, la variabile PATH non contiene la path che indica la directory corrente (per esserne certi provate a visualizzarne il contenuto con il comando *echo* $PATH). Per cui occorre *esplicitamente* indicarla, altrimenti il loader non riesce a mandar in esecuzione i due programmi.

A questo punto, usando un terzo terminale, e invocando il comando *netstat* con le opzioni -a (all) -n (numeric) -p (protocol, tcp in questo caso), si ottiene una lista di tutte le connessioni attive [1].

[1] Attenzione, sto usando un Macbook, con sistema operativo Leopard 10.5 ... altri sistemi operativi possono produrre un output leggermente diverso e la stessa sintassi per le opzioni di netstat potrebbe essere leggermente diversa. Per sicurezza usate *man netstat* per informazioni sull'uso del comando.

```
macbook-di-paolo-darco:~ pd$ netstat -na -ptcp
Active Internet connections (including servers)
Proto Recv-Q Send-Q  Local Address          Foreign Address        (state)
tcp4       0      0  127.0.0.1.9999         127.0.0.1.49152        ESTABLISHED
tcp4       0      0  127.0.0.1.49152        127.0.0.1.9999         ESTABLISHED
tcp4       0      0  *.9999                 *.*                    LISTEN
tcp4       0      0  *.88                   *.*                    LISTEN
tcp6       0      0  *.88                   *.*                    LISTEN
tcp4       0      0  *.631                  *.*                    LISTEN
tcp6       0      0  *.631                  *.*                    LISTEN
tcp4       0      0  *.445                  *.*                    LISTEN
tcp4       0      0  *.139                  *.*                    LISTEN
tcp4       0      0  127.0.0.1.631          *.*                    LISTEN
tcp6       0      0  ::1.631                *.*                    LISTEN
tcp4       0      0  *.515                  *.*                    LISTEN
tcp6       0      0  *.515                  *.*                    LISTEN
```

Tre entry sono di nostro interesse. Le prime due entry sono i socket del nostro server (processo figlio) e del nostro client. La connessione viene riportata *due volte*. Nel primo caso, il processo locale è il server (processo figlio) 127.0.0.1.9999 che è connesso al client "remoto" 127.0.0.1.49152. Nel secondo caso, il processo locale è il client 127.0.0.1.49152., che è connesso al server (processo figlio) 127.0.0.1.9999. Se avessimo mandato in esecuzione il server su una macchina remota, nella lista sarebbe comparsa una sola entry, la seconda. Si noti che lo stato della connessione è ESTABLISHED. Il protocollo di handshake a tre passi ha avuto successo e la connessione è stata stabilita. Nota che la terza entry rappresenta il processo server (padre) che è nello stato LISTEN, in attesa cioè di ricevere nuove connessioni su una qualsiasi interfaccia.

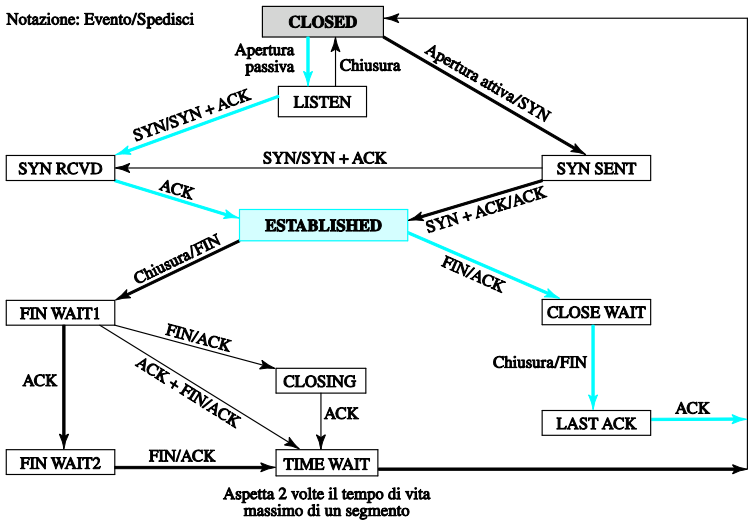


Figure 1: TCP: diagramma di stato

A questo punto, il client riceve input dalla tastiera, invia la linea ricevuta al server, riceve risposta e stampa a video.

```
macbook-di-paolo-darco:FIN pd$ ./echocli 127.0.0.1 9999
Ciao Server, come stai?
Ciao Server, come stai?
Arrivederci
Arrivederci
```

Interrompiamo ora l'esecuzione del client premendo CTRL D ed invochiamo subito il comando *netstat -na -ptcp*. La schermata risultante è

```
macbook-di-paolo-darco:~ pd$ netstat -na -ptcp
Active Internet connections (including servers)
Proto Recv-Q Send-Q  Local Address          Foreign Address        (state)
tcp4       0      0  *.9999                 *.*                    LISTEN
tcp4       0      0  *.88                   *.*                    LISTEN
tcp6       0      0  *.88                   *.*                    LISTEN
tcp4       0      0  *.631                  *.*                    LISTEN
tcp6       0      0  *.631                  *.*                    LISTEN
tcp4       0      0  *.445                  *.*                    LISTEN
tcp4       0      0  *.139                  *.*                    LISTEN
tcp4       0      0  127.0.0.1.631          *.*                    LISTEN
tcp6       0      0  ::1.631                *.*                    LISTEN
tcp4       0      0  *.515                  *.*                    LISTEN
tcp6       0      0  *.515                  *.*                    LISTEN
tcp4       0      0  127.0.0.1.49152        127.0.0.1.9999         TIME_WAIT
```

Cosa è successo? L'entry finale indica che il client ha una connessione aperta con il server (processo figlio) nello stato TIME_WAIT. (Il processo padre è sempre in attesa di nuove richieste). Quando l'utente digita CTRL D,

- il client esce dal ciclo while (*fgets()* restituisce NULL) e termina la propria esecuzione (chiama *exit(0)*).
- il kernel, quindi, chiude tutti i descrittori (tra essi il socket descriptor).
- ha inizio la procedura di chiusura in quattro passi (vedi diagramma di stato del protocollo TCP): il client spedisce il FIN al server.
- il server (processo figlio) risponde con ACK per FIN (che il client attende).
- il server (processo figlio) finisce nello stato CLOSE_WAIT.
- il client riceve ACK e finisce nello stato FIN_WAIT_2. La prima parte della procedura di chiusura è conclusa (vedi Figura 2).

D'altra parte, quando il server riceve FIN, sta eseguendo la *reti_readline*(), che ritorna il valore 0. Pertanto, il server

- esce dal ciclo while ed invoca *exit(0)*.

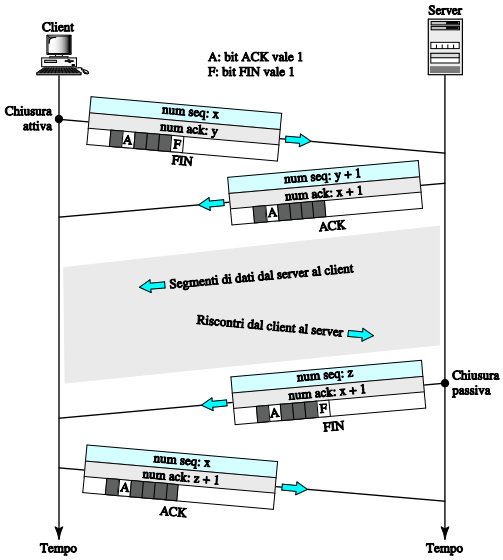


Figure 2: Chiusura in quattro passi

- il kernel, quindi, chiude tutti i descrittori (tra essi il socket descriptor) ed un FIN viene spedito al client.
- il client risponde con un ACK per il FIN e il server riceve l'ACK.

A questo punto la connessione è completamente chiusa: il server ha chiuso la connessione e terminata la propria esecuzione e il client entra nello stato TIME_WAIT. Dopo un certo periodo di tempo (2 Maximum Segment Lifetime, 2MSL in breve, massimo intervallo di tempo che un datagram IP può sopravvivere nella interrete) il client chiude definitivamente la connessione. Le ragioni per la transizione del client in questo stato sono due:

- l'ACK finale inviato dal client potrebbe non raggiungere il server. In questo caso, il server rispedirebbe FIN e il client deve ancora avere informazioni sulla connessione per rispondere nuovamente con l'ACK.
- evitare che pacchetti di una connessione chiusa siano ancora in giro e possano *riapparire* in una nuova connessione tra le due parti che usa *stessi* IP e *numeri di porta*. Questo fenomeno prende il nome di *reincarnazione*. L'attesa di 2MSL assicura che tutti i pacchetti persi per la interrete siano stati eliminati.

La schermata riporta esattamente questa situazione. Se riproviamo ad invocare il comando *netstat* nuovamente (dopo che i due MSL sono trascorsi) la schermata sarà la seguente:

```
macbook-di-paolo-darco:~ pd$ netstat -na -ptcp
Active Internet connections (including servers)
Proto Recv-Q Send-Q  Local Address          Foreign Address        (state)
```

```
tcp4       0      0  *.9999                 *.*                    LISTEN
tcp4       0      0  *.88                   *.*                    LISTEN
tcp6       0      0  *.88                   *.*                    LISTEN
tcp4       0      0  *.631                  *.*                    LISTEN
tcp6       0      0  *.631                  *.*                    LISTEN
tcp4       0      0  *.445                  *.*                    LISTEN
tcp4       0      0  *.139                  *.*                    LISTEN
tcp4       0      0  127.0.0.1.631          *.*                    LISTEN
tcp6       0      0  ::1.631                *.*                    LISTEN
tcp4       0      0  *.515                  *.*                    LISTEN
tcp6       0      0  *.515                  *.*                    LISTEN
```

E' rimasto soltanto il processo padre ad attendere nuove richieste.

1.2 Gestione degli Zombie

Un altro comando utile a raccogliere informazioni circa lo stato dei processi è il comando *ps*. (Ancora una volta, le schermate possono essere leggermente diverse al variare del sistema operativo).

```
macbook-di-paolo-darco:FIN pd$ ps
  PID TTY           TIME CMD
  150 ttys000    0:00.03 -bash
  332 ttys001    0:00.01 -bash
  345 ttys001    0:00.00 ./echosrv 9999
  347 ttys001    0:00.00 (echosrv)
  363 ttys002    0:00.01 -bash
```

Il processo con PID 345 è in esecuzione normalmente ed è il processo server. Il processo con PID 347 (echosrv) è il processo figlio con cui il server ha servito il client. La notazione indica un processo *defunto*, non più attivo, un cosiddetto *zombie*.

Gli zombi sono un problema. Se un server riceve 30000 richieste e, al termine di ogni richiesta nel sistema resta un processo zombie, in breve tempo il sistema si appesantirà, i process control block necessari a creare nuovi processi si esauriranno e il sistema operativo collasserà. Gli zombie vanno pertanto rimossi.

Quando il processo figlio termina, il sistema operativo invia il *segnale* SIGCHLD al processo padre, per notificargli la terminazione del figlio. Al momento il segnale non viene *gestito* dal padre.

Il meccanismo dei segnali permette al sistema operativo di comunicare ai processi in esecuzione *eventi sincroni ed asincroni* che si verificano nel corso dell'esecuzione. Per molti segnali, i processi hanno la facoltà di decidere *cosa fare* all'atto della notifica. In particolare, un processo può cioè chiedere al sistema operativo di *ignorare* il segnale o di *attivare un gestore* che specifica il codice da eseguire alla ricezione del segnale.

Per default, la ricezione di SIGCHLD non implica nessuna azione. Il sistema operativo non elimina i figli defunti perchè il padre può avere necessita di controllare lo stato di terminazione dei figli o di contabilizzare le risorse che hanno utilizzato (e.g., memoria, cicli di CPU, etc...). Per rimuovere i processi zombie, occorre pertanto far si che quando al processo padre viene notificato il segnale SIGCHLD, il processo padre rimuova il figlio defunto e liberi tutte le sue risorse. A tal fine sono necessarie due azioni. Il processo padre deve:

- comunicare al sistema operativo che il segnale SIGCHLD deve essere gestito
- fornire al sistema operativo il gestore che deve essere applicato

La prima azione richiede l'uso della funzione *signal()*, che prende in input il numero del segnale e un puntatore a funzione. Nel nostro caso il segnale è rappresentato dalla costante SIGCHLD, e la funzione è *gestore()*, dichiarata prima del *main*(). La funzione *signal*() va invocata ovviamente prima della funzione *fork*().

```
void gestore(int signo);
int main(int argc, char **argv) {
...
        if (signal(SIGCHLD, gestore) < 0){
                printf("signal error \n");
                exit(1);
...
```

La funzione gestore è definita come segue:

```
void gestore(int signo) {
        pid_t pid;
        int stat;

        while ( (pid = waitpid(-1, &stat, WNOHANG)) > 0) {
                 printf("Figlio %d terminato, con lo stato di %d\n",pid,stat);
        }
}
```

Il cuore del gestore è la chiamata alla funzione *waitpid*().

```
#include <sys/wait.h>

pid_t waitpid(pid_t pid, int *stat_loc, int options);

valore di ritorno: -1 se errore
                   il pid del processo che ha terminato, altrimenti
```

Quando il primo argomento è -1, il processo invocante attende la terminazione di *uno qualsiasi* dei suoi figli (se ne ha più di uno). Il parametro *stat* contiene informazioni sullo stato di uscita del processo. Il flag WNOHANG specifica che la chiamata di sistema non deve portare il processo invocante all'arresto se non ci sono figli che hanno terminato.

Modificando il codice del server in questo modo, gli zombie vengono automaticamente rimossi dal sistema. Provate!

In realtà, al fine di avere un server robusto, un altro problema che deve essere risolto è il seguente. Il processo padre ha invocato la funzione *accept*() ed è in attesa (i.e., chiamata di sistema) di nuove richieste di servizio. L'arrivo del segnale SIGCHLD provoca *l'interruzione* della chiamata di sistema. Alcuni sistemi operativi *automaticamente* riavviano una chiamata di sistema interrotta. Altri invece no. Se il sistema operativo non riavvia la chiamata, il server viene interrotto e la variabile di sistema *errno* contiene il codice EINTR, che identifica il tipo di errore. Al fine di evitare questa situazione, occorre gestire l'errore, riavviando la chiamata. Un modo per farlo è il seguente.

```
for ( ; ; ) {
  cliaddr_len = sizeof(cliaddr);
  if( (connfd = accept(listenfd, (struct sockaddr *) &cliaddr, &cliaddr_len)) < 0) {
     if(errno == EINTR)
         continue;
     else
        {printf("accept error\n");exit(1);}
    }
   ...
}
```

Così facendo, la funzione *accept*() viene riavviata se l'errore è dovuto ad una interruzione.

1.3 Server busy

Esiste un'altra situazione, molto simile all'interruzione della chiamata di sistema appena descritta, che richiede semplicemente di richiamare la funzione *accept*(). Questo errore è stato riscontrato su server sovraccarichi di richieste (alcuni web server). Può accadere quanto segue:

- Il processo di handshake in tre passi client-server ha avuto successo. Tuttavia, il server non ha ancora prelevato (tramite la *accept*()) la entry del client dalla coda delle connessioni complete (vedi la Figura 3)
- Il client, stanco di attendere, interrompe la connessione (lo strato TCP del client invia allo strato TCP del server un pacchetto RST)
- Quando il server esegue la *accept*(), la funzione restituisce l'errore ECONNABORTED, che indica che la connessione è stata abortita.

In questo caso, piuttosto che terminare, il server può richiamare *accept*() per la prossima connessione. In alcuni sistemi operativi, l'intero processo è gestito dal kernel, i.e., il server non riceve dalla *accept*() nessun errore e non si accorge di nulla.

1.4 Terminazione del server

Un'altra situazione da gestire è l'eventuale terminazione improvvisa del server (processo figlio). Tale situazione può essere facilmente riprodotta interrompendo volontariamente il server (processo figlio) tramite il comando *kill*. Le azioni che vengono intraprese sono le seguenti:

- Il sistema operativo chiude tutti i descrittori (tra cui il socket descriptor). La chiusura implica l'invio di FIN al client, che risponde con ACK.
- Il segnale SIGCHLD viene notificato al server (processo padre) che gestisce, come abbiamo discusso in precedenza, correttamente la terminazione.
- Al client non accade niente. E' bloccato nella *fgets*(), in attesa di una linea dal terminale.

Utilizzando il comando *netstat* a questo punto, riusciamo a visualizzare lo stato del client

```
macbook-di-paolo-darco:~ pd$ netstat -na -ptcp
Active Internet connections (including servers)
Proto Recv-Q Send-Q  Local Address          Foreign Address        (state)
tcp4       0      0  127.0.0.1.50007        127.0.0.1.9999         CLOSE_WAIT
tcp4       0      0  *.9999                 *.*                    LISTEN
...
tcp4       0      0  127.0.0.1.9999         127.0.0.1.50007        FIN_WAIT_2
```

Rimuovendo un po' di entry relative ad altre connessioni aperte, le righe di nostro interesse sono le prime due e l'ultima. In particolare, l'ultima riga mostra che il server (processo figlio) si trova nello stato FIN_WAIT_2, mentre il client (prima riga) si trova nello stato CLOSE_WAIT.

Se a questo punto diamo in input una nuova linea al client, il client scriverà sul socket invocando (*reti_writen*()) (l'operazione è lecita perchè il client *non* ha chiuso il socket) e, subito dopo, tenterà di leggere attraverso *reti_readline*(). Tuttavia, *reti_readline*() ritorna immediatamente EOF (valore 0), a causa del FIN inviato in precedenza dal server (processo figlio) al momento della chiusura del socket descriptor. Poichè il client non si aspetta di ricevere EOF, termina con errore e chiude tutti i descrittori.

```
macbook-di-paolo-darco:FIN pd$ ./echocli 127.0.0.1 9999
bravissimo
bravissimo
posso scrivere ancora
echocli.c: server terminated prematurely
macbook-di-paolo-darco:FIN pd$
```

Un paio di osservazioni.

- TCP permette la mezza chiusura (*half close*). Il lato che chiude non invierà più dati. Vedremo casi in cui è utile.
- La ricezione del FIN *non* dà informazioni al client circa la terminazione del server (processo figlio).

In realtà quando il client scrive sul socket, il sistema operativo del server invia in risposta un datagram RST che indica che l'altro lato della connessione non c' è più. Nel nostro esempio, il client invoca immediatamente la *readline*(), riceve 0, e termina. E' un evento dovuto alla temporizzazione delle operazioni. Se avesse invece ricevuto prima il datagram RST, la *readline*() sarebbe terminata con errore ECONNRESET.

Come vedremo tra qualche lezione, il problema di fondo è che il client lavora *contemporamente con due descrittori*, *stdin* e *sockfd* e, quando il server spedisce FIN sul socket, il client è "impegnato" a leggere da stdin, bloccato nella chiamata a *fgets*(). Occorre un modo per notificare *subito* al client che il server (processo figlio) ha chiuso la connessione.

Tornando al nostro esempio, e procedendo nell'analisi, poniamoci la seguente domanda: cosa accadrebbe se il client, invece di terminare a seguito del valore riportato dalla *readline*(), ignorasse l'errore e *scrivesse nuovamente* sul socket (per esempio, il nostro client effettua due *reti_writen*() di seguito)? Ebbene, il client riceve il segnale SIGPIPE. L'azione di default associata al segnale è terminare il processo. Pertanto, il client per evitare di terminare deve *catturare* il segnale e gestirlo

opportunamente o *ignorarlo*. Se il client ignora il segnale SIGPIPE, l'operazione di scrittura genera l'errore EPIPE.

In conclusione: come comportarsi? Se l'applicazione che stiamo progettando (in particolare, il nostro client) non deve terminare, allora si può:

1. Ignorare (usando signal(SIGPIPE, SIG_IGN)) il segnale SIGPIPE.
2. Controllare l'errore EPIPE sulle operazioni di scrittura e, in caso di errore, non scrivere più nulla.

1.5 Host del server non raggiungibile

Altra situazione che può verificarsi è che l'host su cui è in esecuzione il server risulti irraggiungibile. Tale condizione può dipendere dalla rete o dall'host. E' facilmente simulabile mandando in esecuzione il client ed il server su macchine diverse, dando in input una prima linea al client, aspettando l'eco, disconnettendo dalla rete la macchina del server, e dando al client in input una seconda linea. Accade quanto segue

- Quando l'host viene disconnesso, non viene inviato alcun messaggio al client.
- Digitando la seconda riga, il client scrive sul socket (*reti_writen*()) e si blocca in attesa di ricevere il secondo eco sul socket (*reti_readline*()).

TCP continuerà a trasmettere la seconda riga fino al raggiungimento di un numero massimo di volte; dopodichè, il client riceverà un messaggio d'errore che può essere di tre tipi:

- ETIMEOUT, se l'host è caduto e nel tempo massimo consentito non è giunto l'ACK atteso
- EHOSTUNREACH oppure ENETUNREACH se risponde qualche *router* intermedio che si è accorto del problema ed ha spedito un pacchetto ICMP (protocollo di controllo usato da IP, i dettagli li riceverete nel corso di teoria)

1.6 Shutdown e reboot del server

Restano da analizzare due situazioni. Nella prima il server va in crash durante l'esecuzione ma immediatamente viene effettuato il reboot. Ripensando alla simulazione precedente, è come se, dopo aver disconnesso l'host dalla rete, facessimo lo shutdown del server (che il client non vede), il reboot e lo riconnettessimo alla rete.

Il client, inviata la seconda riga al server, resta bloccato in attesa di ricevere il secondo eco sul socket (*reti_readline*()). Come in precedenza, TCP (lato client) continuerà a trasmettere la seconda riga. Questa volta però, ad un certo punto, quando il server ha completato il reboot, TCP (lato server), avendo perso tutte le informazioni relative alle connessioni precedenti, *non riconosce* i segmenti delle vecchie connessioni ed invia in risposta un RST. Pertanto, il client, alla ricezione dell'RST, esce dalla *readline*() con errore ECONNRESET.

Se l'applicazione che stiamo scrivendo ha necessità di sapere se l'host del server va in crash occorre usare altre tecniche (che sono disponibili e daremo qualche cenno nelle ultime lezioni di questo corso).

La seconda situazione si verifica, invece, quando client e server sono connessi ed il server subisce uno shutdown (i.e., l'operatore dell'host avvia la procedura di shutdown dell'host). In un sistema Unix-like, la procedura di shutdown comporta la notifica del segnale SIGTERM a tutti i processi e dopo pochi secondi del segnale SIGKILL, che termina i processi. Pertanto i processi chiudono i descrittori aperti. TCP (lato client) riceve RST ed il processo client esce dalla *readline*() con errore ECONNRESET.

1.7 Formato dei dati

L'esempio client-server considerato sino ad ora rappresenta una situazione particolarmente semplice: il server *non* esamina mai i dati che il client invia. Legge fino al newline e reinvia la riga letta al client. Normalmente non è così. I dati solitamente vengono esaminati ed *elaborati*. Occorre, pertanto, fare attenzione al *formato dei di dati* che client e server si scambiano.

Consideriamo un esempio per capire le insidie nascoste in questa operazione. Modifichiamo il nostro echo server come segue: nella stringa ricevuta dal client, il server si aspetta di trovare due interi, separati da uno spazio bianco. Il server legge i due interi, ne fa la somma e la restituisce al client.

Se rivediamo la struttura del nostro client e del nostro server, ci accorgiamo che l'unica parte che occorre modificare è la funzione *server_echo*, che codifica l'elaborazione del server. Riscriviamola e chiamiamola *server_somma*().

```
void server_somma(int sockfd) {
1.  int      i, arg1, arg2;
2.  ssize_t      n;
3.  char      sendline[MAXLINE], rcvline[MAXLINE];
4.  char c;

5.   for ( ; ; ) {
6.        if ( (n = reti_readline(sockfd, rcvline, MAXLINE)) == 0)
7.             return;         /* connection closed by other end */

8.          if( sscanf(rcvline, "%d %d", &arg1, &arg2) == 2 )
9.            sprintf(sendline, "%d\n", arg1 + arg2);
10.        else
11.          sprintf(sendline, "input error\n");
12.        n = strlen(sendline);
13.        reti_writen(sockfd, sendline, n);
       }
}
```

Il server, usando la funzione *sscanf*(), legge dalla stringa ricevuta dal client i due parametri, li trasforma in interi, e li memorizza in $arg1$ e $arg2$ (linea 8.). Successivamente, usando la funzione *sprintf*(), trasforma la somma intera $arg1 + arg2$ in formato stringa (linea 9.) e la invia al client (linea 13.).

Client e server funzionano correttamente, indipendentemente dalle specifiche rappresentazioni dei dati delle due macchine sottostanti. Continuano a scambiarsi stringhe.

In realtà, pensando all'elaborazione che fanno, potremmo pensare di scrivere client e server in modo diverso. Potremmo definire un file header con due strutture:

```
struct args {
int arg1;
int arg2;
};
```

e

```
struct result {
int sum;
};
```

e far si che il client invii al server una struttura *args* e il server, in risposta, una struttura *result*. Per esempio, la funzione server_somma diventerebbe:

```
void server_somma(int sockfd) {
1.  ssize_t      n;
2.  struct args args;
3.  struct result result;

4.   for ( ; ; ) {
5.        if ( (n = reti_readline(sockfd, &args, sizeof(args))) == 0)
6.             return;         /* connection closed by other end */

7.     result.sum = args.arg1 + args.arg2;
8.          reti_writen(sockfd, &result, sizeof(result));
        }
}
```

e, similmente, la funzione eseguita dal client diventerebbe:

```
void client_echo(FILE *fp, int sockfd) {
1.  char    sendline[MAXLINE];
2.  struct args args;
3.  struct result result;

4.  while (fgets(sendline, MAXLINE, fp) != NULL) {

5.  if( sscanf(sendline, "%d %d", &args.arg1, &args.arg2) != 2 ){
6.   printf("Invalid input: %s", sendline);
7.   continue;
     }

9.  reti_writen(sockfd, &args, sizeof(args));

10.   if (reti_readn(sockfd, &result, sizeof(result)) == 0
11.     { printf("%s: server terminated prematurely",__FILE__);
         exit(-1); }
12.   printf("%d\n", result.sum);
   }
}
```

Sfortunatamente, se client e server sono eseguiti su macchine aventi la stessa architettura, l'applicazione funziona correttamente, ma se le architetture sono distinte non funziona più: i dati spediti dal client vengono *interpretati* diversamente dal server!

L'esempio mostra due potenziali problemi:

- Implementazioni diverse memorizzano i numeri in formati diversi (little-endian e big-endian).
- Implementazioni diverse "impacchettano" le strutture dati in modo diverso.

Inoltre, implementazioni diverse possono memorizzare gli stessi tipi di dati in modo diverso. Per esempio, abbiamo memorizzato gli argomenti come *int*. Se li memorizzassimo come *long*, potrebbe accadere che una implementazione rappresenti i *long* con 32 bit mentre l'altra con 64 bit. Ancora, le computazioni sarebbero inconsistenti.

Pertanto, esistono due modi per evitare il problema:

- Passare tutti i dati numerici come stringhe (è quanto abbiamo fatto nel primo caso). Non inviare mai strutture binarie su un socket.
- Client e server definiscono in anticipo esplicitamente il formato binario dei tipi di dati che si scambieranno e successivamente usano i tipi concordati. Per esempio, i pacchetti che implementano chiamate a procedura remota (RPC) solitamente utilizzano una rappresentazione dei dati intermedia, detta XDR (external data representation), attraverso cui i dati viaggiano su rete, per poi essere codificati/decodificati alle estremità.

1.8 Conclusioni.

In sintesi, nella lezione di oggi ci siamo convinti che:

- Scrivere applicazioni client-server *efficienti e robuste* non è banale. Occorre capire come il sistema operativo gestisce le risorse allocate ai processi e occorre tener conto di diversi eventi che possono verificarsi nel corso dell'esecuzione, dovuti a bug del software, a malfunzionamenti del server, degli host o della rete.
- Comprendendo bene come i protocolli sottostanti, nel caso specifico TCP, reagiscono agli eventi che accadono, è possibile scrivere client e server che gestiscono in modo opportuno le condizioni di errore.
- Occorre fare molta attenzione al formato dei dati che client e server si scambiano. In particolare, conviene trasformare i dati in formato stringa e far si che client e server si scambino stringhe. Lo scambio non accorto di strutture binarie su socket può essere fonte di problemi.

1.9 Esercizi

Esercizio 1. Si realizzi un'applicazione client-server che funziona come segue. Il client si connette al server, che risponde immediatamente inviando la stringa "Server mini-smtp, v1.0. Inviare comando". Il client, a questo punto, può inviare i seguenti comandi:

1. "BYE"
2. "MAIL FROM: <nome>"
3. "MAIL TO: <nome>"
4. "DATA"
5. "."

Alla ricezione di "BYE" il server chiude la connessione.

Alla ricezione di "MAIL FROM: <nome>" il server memorizza il nome del mittente, risponde con "OK, mittente ricevuto" e si mette in attesa del comando successivo che deve essere "MAIL TO: <nome>".

Alla ricezione di "MAIL TO: <nome>" il server memorizza il nome del destinatario, risponde con "OK, destinatario ricevuto" e si mette in attesa del prossimo comando che deve essere "DATA".

Alla ricezione del comando "DATA" il server risponde con "Pronto a ricevere i dati", apre in scrittura un file di testo con nome "<mittente>-<destinatario>" e si mette in attesa di linee di dati.

A questo punto il client invia linee di dati che il server riceve e scrive nel file.

Quando il client spedisce una linea che contiene solo il punto ("."), il server risponde con "OK, mail ricevuta", chiude il file di testo e si mette in attesa di un nuovo comando (che potrà essere di nuovo "MAIL FROM: <nome>" per spedire un nuovo messaggio, oppure "BYE" per chiudere la connessione).

Sia il client che il server visualizzano i messaggi ricevuti.

Gestire opportunamente le condizioni di errore.

Suggerimenti per risolvere l'esercizio. La funzione *sscanf* può essere utile per estrarre dalle linee di comando del client il nome del mittente e del destinatario, e.g.,

char mittente[MAX];
sscanf(linea-ricevuta, "MAIL FROM: %s", mittente);

mentre la funzione *sprintf* può essere utile per costruire il nome del file da scrivere, e.g.,

```
sprintf(nomefile, "%s-%s", mittente, destinatario);
```