Esercitazione [8]

Server multi-process/multi-thread

Leonardo Aniello - aniello@dis.uniroma1.it
Daniele Cono D'Elia - delia@dis.uniroma1.it
Federico Lombardi – lombardi@dis.uniroma1.it

Sistemi di Calcolo - Secondo modulo (SC2)
Programmazione dei Sistemi di Calcolo Multi-Nodo

Corso di Laurea in Ingegneria Informatica e Automatica

A.A. 2015-2016

Sommario

- Soluzione esercizio su EchoServer
- Obiettivi dell'esercitazione
- Network and host byte order
- Server multi-process
- Esercizio: completare server multi-process
- Server multi-thread
- Esercizio: completare server multi-thread

Esercizio su EchoServer

- Scenario: client/server seriale
  - Il client si connette ed invia un messaggio testuale
  - Il server risponde con lo stesso messaggio inviato dal client
    - Se il messaggio è «QUIT», il server chiude la connessione
- Esercizio: completare le parti di scambio messaggi e fase `accept()`
- Soluzione
  - Sia per il client che per il server, bisogna verificare che il valore di ritorno delle chiamate send/recv non sia negativo
    - Se lo è, va verificato se `errno` corrisponde ad un'interruzione…
  - I messaggi vanno confrontati rispetto ai comandi noti
    - Usare la `memcmp` ed uscire dal ciclo in caso di «QUIT»

Obiettivi Esercitazione [8]

- Capire la differenza tra network byte order e host byte order
- Gestire più connessioni in parallelo lato server
  - Usando un processo per ogni connessione
  - Usando un thread per ogni connessione

Network e host byte order

- Nello scambio di dati numerici tra macchine con architetture (potenzialmente) differenti, occorre verificare il **byte order**
  - Un dato numerico è rappresentato come una sequenza di byte
  - *Il primo byte di tale sequenza è il più significativo* (**Big Endian**) *o il meno significativo* (**Little Endian**)*?*
- Dati numerici scambiati tra macchine che usano byte order diversi (**host byte order**) vengono interpretati in maniera diversa
- La maggior parte dei protocolli di rete (inclusi IPv4 e TCP) usano Big Endian come **network byte order** nell'header

Network e host byte order
Funzioni di conversione per la porta

- Il numero di porta di una socket TCP può variare tra 0 e 65535
  - la definizione del tipo generico `uint16_t` può essere inclusa tramite **`<arpa/inet.h>`** o più in generale `<stdint.h>`
  - su Linux IA32 e x86_64 equivale ad un `unsigned short`
  - le porte nel range 0-1023 richiedono privilegi di root

➢ Funzione `htons()` (host-to-network-ushort)

- `uint16_t htons(uint16_t hostshort);`
- Converte un ushort da host byte order a network byte order

➢ Funzione `ntohs()` (network-to-host-ushort)

- `uint16_t ntohs(uint16_t netshort);`
- Converte un ushort da network byte order a host byte order

Network e host byte order
Funzioni di conversione per l'indirizzo (1/2)

- Un indirizzo IPv4 è rappresentato con `struct in_addr`
- Il campo `sin_addr` di `struct sockaddr_in` è infatti di tipo `struct in_addr`
  - *Reminder*: variabili di tipo `struct sockaddr_in` vengono usate nella `bind()` e nella `accept()` lato server e nella `connect()` lato client
- `struct in_addr` contiene il campo `s_addr` di tipo `in_addr_t` che rappresenta l'indirizzo in network byte order
- Entrambe le strutture sono definite in `<netinet/in.h>`

Network e host byte order
Funzioni di conversione per l'indirizzo (2/2)

- `in_addr_t inet_addr(const char *cp)`
  - Converte un indirizzo IPv4 dalla forma dotted (x.y.z.w) al network byte order
  - Il valore di ritorno viene di solito assegnato al campo `s_addr` di `struct in_addr`
- `const char *inet_ntop(int af, const void *src, char *dst, socklen_t n);`
  - Converte l'indirizzo di rete `src` della address family `af` in una stringa di lunghezza `n` e la copia in `dst`
  - Ritorna un puntatore a `dst`, oppure `NULL` in caso di errore
  - Le macro `AF_INET` e `INET_ADDRSTRLEN` possono essere usate rispettivamente per il primo e l'ultimo argomento

Parallelismo lato server

- Nelle scorse esercitazioni abbiamo visto server «seriali»:
  - Viene servita una connessione alla volta
  - Connessioni che arrivano nel mentre vengono messe in coda…
  - …e verranno processate sequenzialmente al termine della connessione attualmente servita
  - Questo comporta dei tempi di attesa crescenti all'aumentare del numero di connessioni in coda!
  - ➢ La soluzione consiste nel disaccoppiare l'accettazione delle connessioni dalla loro elaborazione
  - ➢ Una volta accettata, una connessione viene elaborata in un processo o thread dedicato, così il server può subito rimettersi in attesa di altre connessioni da accettare

Server multi-process

- Per ogni connessione accettata, viene lanciato un nuovo processo figlio tramite `fork()`
  - Il figlio deve chiudere il descrittore della socket tramite la quale il server accetta le connessioni
  - Analogamente, il server deve chiudere il descrittore della socket relativa alla connessione accettata
  - Una volta completata l'elaborazione della connessione, il processo figlio esce
- Elevato overhead legato alla creazione di nuovo processo per ogni connessione
- Complessa gestione di eventuali strutture dati condivise (tramite file, pipe, memoria condivisa oppure anche socket)

Server multi-process

```
while (1) {
        int client = accept(server, ……);
        <gestione errori>

        pid_t pid = fork();
        if (pid == -1) {
                <gestione errori>
        } else if (pid == 0) {
                close(server);
                <elaborazione connessione client>
                exit(0);
        } else {
                close(client);
        }
}
```

Esercizio: EchoServer multi-process

- Completare il codice dell'EchoServer in modalità multi-process
- Sorgenti
  - `Makefile`
  - Client: `client.c`
  - EchoServer seriale per confronto: `base.c`
  - EchoServer multi-process da completare: `multiprocess.c`
- Suggerimento: seguire i blocchi di commenti inseriti nel codice
- Altro suggerimento:
  Per monitorare a runtime il numero di istanze di processi attivi in un certo momento, lanciare da terminale il comando:

```
ps -e -O ppid|head -1;ps -e -O ppid|grep multiprocess
```

Server multi-thread

- Per ogni connessione accettata, viene lanciato un nuovo thread tramite `pthread_create()`
  - Oltre ai parametri application-specific, il nuovo thread avrà bisogno del descrittore della socket relativa alla connessione appena accettata
  - A differenza del server multi-process, non è necessario chiudere alcun descrittore di socket (perché???)
  - Una volta completata l'elaborazione della connessione, il thread termina
  - Il main thread può voler fare detach dei thread creati
- Minore overhead rispetto al server multi-process
- Gestione più semplice di eventuali strutture dati condivise
- Un crash in un thread causa un crash in tutto il processo!

Server multi-thread

```
while (1) {
      int client = accept(server, ……);
      <gestione errori>

      pthread_t t;
      t_args = ……
      <includere client in t_args>
      pthread_create(&t,NULL,handler,(void*)t_args);
      <gestione errori>
      pthread_detach(t);
}
```

Esercizio proposto: EchoServer multi-thread

- Completare il codice dell'EchoServer in modalità multi-thread
- Sorgenti
  - `Makefile`
  - Client: `client.c`
  - EchoServer seriale per confronto: `base.c`
  - EchoServer multi-thread da completare: `multithread.c`
- Suggerimento: seguire i blocchi di commenti inseriti nel codice
- Altro suggerimento:
  Per monitorare a runtime il numero di istanze di processi/thread attivi in un certo momento, lanciare da terminale il comando:

```
ps -e -T | head -1 ; ps -e -T |grep multithread
```