Laboratorio di
Reti Informatiche

Corso di Laurea Triennale in Ingegneria Informatica

A.A. 2018/2019

Ing. Carlo Vallati

carlo.vallati@unipi.it

Esercizi

Programmazione con i socket

Programma di oggi

- Esercizi di programmazione distribuita

Leggere gli argomenti


```
int main(int argc, char* argv[]){
  /* … */
}
int main(int argc, char** argv){
  /* … */
}
```

- argc è il numero degli argomenti passati **+ 1**
- argv è un array di **stringhe**
  - Quindi un puntatore a puntatori di caratteri
  - La prima stringa rappresenta il comando
  - Se ci servono valori numerici bisogna convertire le stringhe (`atoi()`, `atol()`, …)

Leggere gli argomenti


```
int main(int argc, char* argv[]){
  int i;
  printf("Ci sono %d argomenti:\n", argc);
  for (i = 0; i < argc; ++i) {
    printf("%s\n", argv[i]);
  }
  return 0;
}
```

```
$ ./mio_prog a1 a2 a3
Ci sono 4 argomenti:
./mio_prog
a1
a2
a3
```

Esercizio 1: Hello Server

- Implementare un semplice **server TCP mono-processo** che fornisce un messaggio "Hello!" ai client che si collegano
  - Leggere la porta da linea di comando (opzionale)
- Implementare il relativo client
  - Il client si connette, riceve il messaggio, lo stampa ed esce
  - Leggere indirizzo e porta del server da linea di comando
- Note:
  - Gestire gli errori
  - Server e client conoscono la dimensione del messaggio

Esercizio 2: Echo Server

- Implementare un **server TCP mono-processo** che re-invia al mittente un messaggio ricevuto
- Implementare un client che di continuo:
  - Legge una stringa da tastiera
  - Se la stringa è "Bye" interrompe la connessione ed esce
  - Altrimenti invia la stringa, riceve la risposta e la stampa
- Il server continua a fare echo al client finché la connessione non si interrompe
- Server e client leggono/scrivono sempre 20 byte
  - Occhio al terminatore di stringa!
- Provare a connettersi con un secondo client mentre il primo viene servito. Cosa succede?

Leggere una riga intera


```
#define BUFFER_SIZE 1024
…
int main(int argc, char* argv[]){
  char buffer[BUFFER_SIZE];
  …
  fgets(buffer, BUFFER_SIZE, stdin);
}
```

Esercizio 3: Echo Server

a) Rendere multi-processo il server dell'esercizio 2 usando la primitiva fork()

b) Rimuovere il limite dei 20 byte: il client invia la dimensione esatta della stringa e il server legge la dimensione esatta di byte.
   - Come fa il server a sapere in anticipo quanti byte leggere?

Inviare la dimensione


```
uint16_t lmsg;
…
// Invio al server la quantita di dati
len = strlen(buffer);
lmsg = htons(len);
// Invio il numero di dati
ret = send(sd, (void*) &lmsg, sizeof(uint16_t), 0);
// Invio I dati
ret = send(sd, (void*) buffer, len, 0);
```

```
// Ricevo la quantita di dati
ret = recv(new_sd, (void*)&lmsg, sizeof(uint16_t), 0);
len = ntohs(lmsg); // Rinconverto in formato host

// Ricevo I dati
ret = recv(new_sd, (void*)buffer, len, 0);
```

Socket non bloccante


```
#include <unistd.h>
#include <fcntl.h>

int fcntl(int fd, int cmd, ...);
```

man fcntl

Serve per manipolare un descrittore di file o di socket

- **fd** è il descrittore
- **cmd** è il comando da passare
  - **F_GETFL**: Ottieni i flag di stato
  - **F_SETFL**: Imposta i flag di stato
- In base al comando può esserci un terzo argomento
- Il valore restituito dipende dal comando passato, -1 e **errno** su errore

Socket non bloccante

I flag vengono passati tutti insieme, quindi bisogna salvare quelli impostati e riscriverli insieme ai nuovi

```
int sd, ret;

sd = socket(…);

/* Ottieni i flag attuali */
ret = fcntl(sd, F_GETFL, NULL);

/* Setta il flag non bloccante, mantenendo gli altri */
fcntl(sd, F_SETFL, ret | O_NONBLOCK);
```

Stampare l'ora


```
time_t rawtime;
/* Ottieni l'ora in formato POSIX */
time(&rawtime);

/* Stampa l'ora */
// ctime() trasforma l'ora in stringa
printf("%s\n", ctime(&rawtime));
```

Esercizio 4: Time Server

- Implementare un semplice server UDP che periodicamente invia l'ora ai client che si registrano
  - Il server periodicamente controlla se c'è una richiesta di registrazione, se c'è registra il client, poi invia a tutti i client registrati un pacchetto UDP contenente la data e l'ora
- Il client invia la richiesta e poi aspetta il dato, stampandola ogni volta che arriva
- Non usare fork() o select()

Stampare data e ora


```
time_t rawtime;
struct tm * timeinfo;
/* Ottieni l'ora in formato POSIX */
time(&rawtime);
timeinfo = localtime (&rawtime);
/* La struttura timeinfo non va deallocata (è allocata
staticamente dal sistema operativo, viene sovrascritta
ad ogni invocazione */

/* Stampa l'ora */
// asctime() trasforma l'ora in stringa
printf("%s\n", asctime(timeinfo));
```

struct tm


```
struct tm{
    int     tm_sec   // Seconds [0,60].
    int     tm_min   // Minutes [0,59].
    int     tm_hour  // Hour [0,23].
    int     tm_mday  // Day of month [1,31].
    int     tm_mon   // Month of year [0,11].
    int     tm_year  // Years since 1900.
    int     tm_wday  // Day of week [0,6] (Sunday =0).
    int     tm_yday  // Day of year [0,365].
    int     tm_isdst // Daylight Savings flag.
    };
```

Inviare una struttura su un socket

38

37

I am trying to pass whole structure from client to server or vice-versa. Let us assume my structure as follows

```
struct temp {
  int a;
  char b;
}
```

I am using **sendto** and sending the address of the structure variable and receiving it on the other side using the **recvfrom** function. But I am not able to get the original data sent on the receiving end. In sendto function I am saving the received data into variable of type struct temp.

```
n = sendto(sock, &pkt, sizeof(struct temp), 0, &server, length);
n = recvfrom(sock, &pkt, sizeof(struct temp), 0, (struct sockaddr *)&from,&fromlen
```

Where pkt is the variable of type struct temp.

Eventhough I am receiving 8bytes of data but if I try to print it is simply showing garbage values. Any help for a fix on it ?

59

This is a very bad idea. Binary data should always be sent in a way that:

- Handles different endianness
- Handles different padding
- Handles differences in the byte-sizes of intrinsic types

Don't ever write a whole struct in a binary way, not to a file, not to a socket.

Always write each field separately, and read them the same way.

You need to have functions like

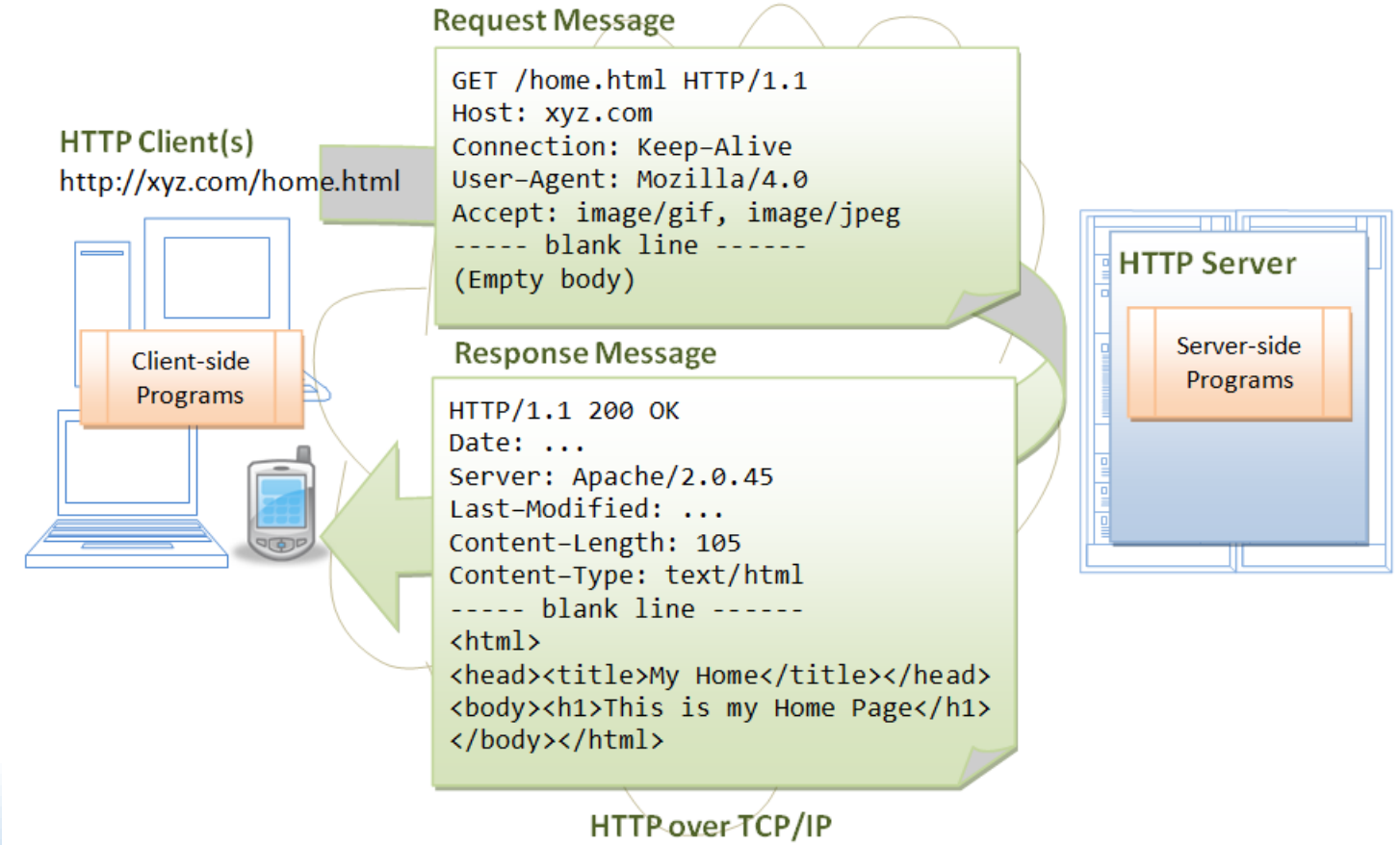
```
unsigned char * serialize_int(unsigned char *buffer, int value)
{
  /* Write big-endian int value into buffer; assumes 32-bit int and 8-bit char. */
  buffer[0] = value >> 24;
  buffer[1] = value >> 16;
  buffer[2] = value >> 8;
  buffer[3] = value;
  return buffer + 4;
}

unsigned char * serialize_char(unsigned char *buffer, char value)
{
  buffer[0] = value;
  return buffer + 1;
}

unsigned char * serialize_temp(unsigned char *buffer, struct temp *value)
{
  buffer = serialize_int(buffer, value->a);
  buffer = serialize_char(buffer, value->b);
  return buffer;
}
```

Altrimenti usare la htonl()

Text Protocols vs Binary Protocols

- Molti protocolli a livello applicativo inviano messaggi di testo (text protocols) invece di inviare direttamente i dati delle strutture dati (binary protocols)
- I dati da inviare sono tradotti in stringhe e inviati nel socket come testo
- I campi del messaggio vengono descritti usando delle parole chiave
- Il protocollo definisce una lunghezza massima dei messaggi

Text Protocols



Text Protocol: convertire la struttura in una stringa

- Nel caso della nostra struttura questa potrebbe essere convertita in testo:

```
struct temp{
    int a;
    char b;
};
struct temp t;
sprintf(buffer,"%d %c",t.a, t.b);
```

Text Protocol: convertire la struttura in una stringa

- Al ricevente parsare il messaggio:

```
struct temp t;
sscanf(buffer,"%d %c",&t.a, &t.b);
```

Binary Protocol

- Si definiscono dei messaggi con una struttura prefissata, composti da campi che rappresentano l'informazione da scambiare
- Ciascun campo ha una lunghezza e un tipo associate
- Alcuni protocolli di tipo binario potrebbero contenere campi a lunghezza variabile, in quel caso comunque il protocollo definisce una lunghezza massima dei messaggi

Binary Protocol: Invio

- La struttura deve essere progettata già con dei tipi che possano essere trasferiti:

```
struct temp{
    uint32_t a;
    uint8_t b;
};
struct temp t;
…
// Convertire il formato prima dell'invio
t.a = htonl(t.a);
```

Esercizio 5: Time Server TCP

- Modificare il time server in modo che utilizzi il protocollo TCP
- I client rimangono connessi e periodicamente invia la richiesta e poi aspettano il dato, stampandola ogni volta che arriva
- Inviare attraverso il socket solamente l'ora in hh:mm:ss
- Usare la select() per gestire richieste multiple da parte dei client