

Laboratorio di Reti Informatiche

Corso di Laurea Triennale in Ingegneria Informatica
A.A. 2016/2017

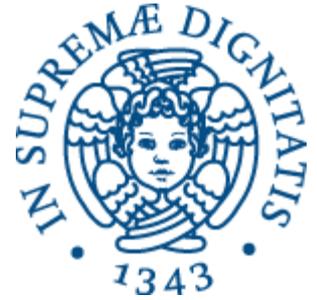
Ing. Niccolò Iardella
niccolo.iardella@unifi.it



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
```



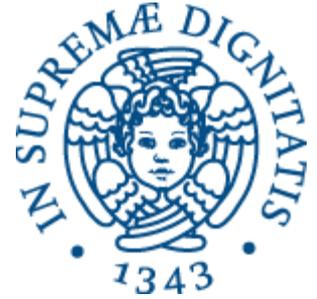
Esercizio 1: Hello Server

- Implementare un semplice **server TCP mono-processo** che fornisce un messaggio "Hello!" ai client che si collegano
 - Leggere la porta da linea di comando
- 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?



Esercizio 3: Echo Server

- a) Rendere multi-processo il server dell'esercizio 2 usando la primitiva `fork()`
- b) Rendere multiplexing il server dell'esercizio 2 usando la primitiva `select()`
- c) 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?



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);
```



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 l'ora
- Il client invia la richiesta e poi aspetta l'ora, stampandola ogni volta che arriva
- Non usare `fork()` o `select()`



Stampare l'ora

```
/* Ottieni l'ora in formato POSIX */  
time_t rawtime;  
time(&rawtime);  
  
...  
  
/* Stampa l'ora */  
// ctime() trasforma l'ora in stringa  
printf("%s\n", ctime(&rawtime));
```