OSOR

Introduzione alla programmazione distribuita

Obiettivi

- Introdurre i concetti di base su programmazione distribuita
  - Modello Client-Server
  - Interfaccia Socket
- Capire come funzionano le applicazioni di rete più comuni
  - Web
  - E-mail
  - File Transfer
- Progettare e realizzare una semplice applicazione distribuita (esercitazioni)

Cooperazione fra processi

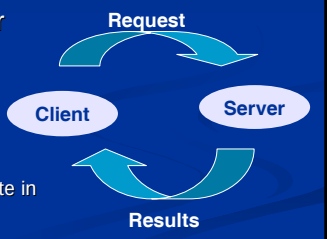
- Processi indipendenti
  - L'esecuzione di un processo non dipende dall'altro processo, e viceversa
- Processi Cooperanti
  - Sincronizzazione
  - Comunicazione (scambio di informazioni)

Comunicazione fra processi

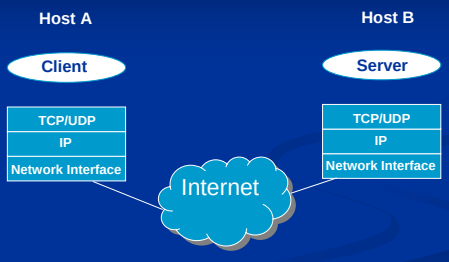
- Esecuzione sullo stesso calcolatore
  - Memoria condivisa
  - Scambio di messaggi
- Esecuzione in sistema distribuito
  - Client-Server
  - Remote Procedure Call
  - Remote Method Invocation
  - …

Client-Server

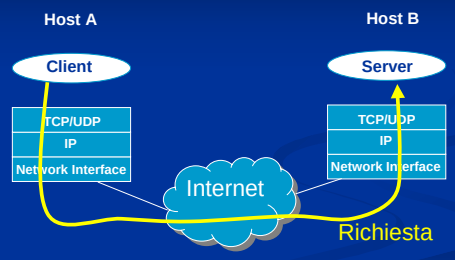
- Paradigma basato su scambio di msg
- Scambio di msg per
  - Richiesta di servizio
  - Invio dei risultati
- Paradigma generale
  - Ma usato principalmente in sistemi distribuiti



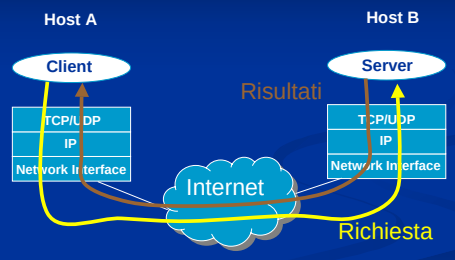
Client-Server in Sistemi Distribuiti



Client-Server in Sistemi Distribuiti



Client-Server in Sistemi Distribuiti



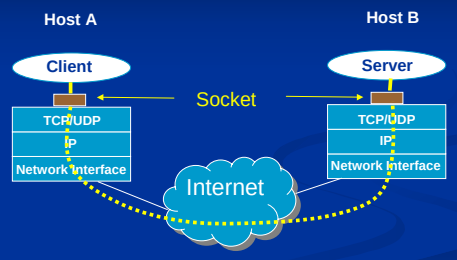
Socket

- Meccanismo di comunicazione tra processi
  - In genere su macchine differenti
- Interfaccia unica per operare con i vari protocolli di rete a disposizione
- I socket nascondono tutti i meccanismi di comunicazione di livello inferiore

Socket

- Estremità di canale di comunicazione identificata da un indirizzo
  - Socket: presa telefonica
  - Indirizzo: numero di telefono
- Indirizzo
  - Indirizzo dell'Host (Indirizzo IP)
  - Indirizzo del processo (Numero di porta)
- La comunicazione avviene tramite una coppia di socket

Comunicazione mediante socket



Supporto del SO

- Il SO implementa l'astrazione di socket
- System call per
  - Creare un socket
  - Associare indirizzo IP e porta al socket
  - Mettere in ascolto un processo su un socket (server)
  - Accettare una richiesta di servizio su un socket (server)
  - Aprire una connessione verso un socket remoto (client)
  - Inviare un messagio verso un socket remoto
  - Ricevere un messaggio da un socket
  - ....

Primitiva socket()

- Crea un socket
  - Restituisce il descrittore (valore intero non negativo)
  - In caso di errore restituisce -1 (setta la variabile *errno*)

int socket(int family, int type, int protocol) [man 2 socket]

- *family*: famiglia di protocolli da utilizzare
  - PF_INET: protocolli internet IPv4 [man 7 ip]
  - PF_UNIX: Unix domain protocol [man 7 unix]
- *type*: stile di comunicazione da che si vuole utilizzare
  - SOCK_STREAM: socket di tipo stream (TCP)
  - SOCK_DGRAM: socket di tipo datagram (UDP)
- protocol: settato a 0

```
sk = socket(PF_INET, SOCK_STREAM, 0);
```


Primitiva setsockopt()

- Manipola le opzioni associate con un socket

int setsockopt(int s, int level, int optname, const void* optval, socklen_t optlen);

  - level: stabilisce il livello a cui manipolare le opzioni
    - SOL_SOCKET: opzioni di livello socket
    - Numero del protocollo: /etc/protocol
  - optname: opzione da settare (man 7 socket per le opzioni di livello socket)
    - SO_REUSEADDR: permette di fare una bind su una certa porta anche se esistono delle connessioni established che usano quella porta (il restart del server)
  - optval e optlen: servono per accedere al valore della opzione

- Restituisce 0 in caso di successo, -1 in caso di errore (setta *errno*)
- Si mette tra la socket() e la bind()

Strutture Dati per Indirizzi

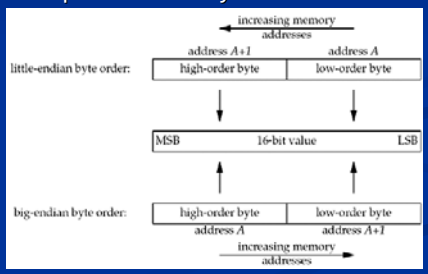
- **struct sockaddr { /* man 7 ip */**
  sa_family_t sa_family; /* AF_INET */
  char sa_data[14] /* address (protocol specific) */
  };

- **struct sockaddr_in { /* man 7 ip */**
  sa_family_t sin_family; /* AF_INET */
  u_int16_t sin_port; /* porta, 16 bit */
  struct in_addr sin_addr; /* indirizzo IP 32 bit */
  };

- **struct in_addr {**
  u_int32_t s_addr; /* indirizzo IP 32 bit */
  };

Formato di Rete

- Calcolatori diversi possono usare convenzioni diverse per ordinare i byte di una word

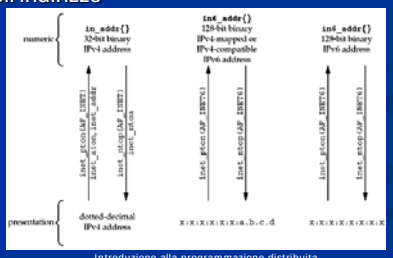


Formato di Rete

- L'indirizzo IP ed il numero di porta devono essere specificati nel formato di rete (*network order*, big endian) in modo da essere indipendenti dal formato usato dal calcolatore (*host order*)
  - uint32_t htonl(uint32_t hostlong);
  - uint16_t htons(uint16_t hostshort);
  - uint32_t ntohl(uint32_t netlong);
  - uint16_t ntohs(uint16_t netlong);

Formato di Rete

- Alcune funzioni consentono di passare dal formato *numeric* al formato *presentation* dell'indirizzo



Formato di Rete

- Formato *numeric* : valore binario nella struttura socket
  - int inet_pton(int af, const char* src, void* addr_ptr);
    - Restituisce 0 in caso di insuccesso
- Formato *presentation* : stringa
  - char* inet_ntop(int af, const void* addr_ptr, char* dest, size_t len);
    - len: deve valere almeno INET_ADDRSTRLEN
    - Restituisce un puntatore NULL in caso di errore

Indirizzi

```
struct sockaddr_in addr_a;

memset(&addr_a, 0, sizeof(addr_a)); /* azzera la struttura*/

addr_a.sin_family = AF_INET;      /* IPv4 address */

addr_a.sin_port = htons(1234);    /* network ordered */

inet_pton(AF_INET, “192.168.1.1”, &addr_a.sin_addr.s_addr);
```


Primitiva bind()

- Collega un indirizzo locale al socket creato con la socket()
- Usata dal server per specificare l’indirizzo su cui il server accetta le richieste
  - Indirizzo IP
  - Numero di Porta
- Il client non esegue la bind()
  - la porta viene assegnata dal SO

Primitiva bind()

int bind(int sd, struct sockaddr* myaddr, int addrlen);

- sd: descrittore del socket
- myaddr: indirizzo della struttura dati che contiene l'indirizzo da associare al socket
  - A seconda della famiglia di protocolli usata dal socket, la struttura dati contenente gli indirizzi varia di formato. Occorre eseguire un casting del puntatore
- addrlen: dimensione della struttura myaddr
- Restituisce 0 in caso di successo, -1 in caso di errore (setta la variabile *errno*)

Primitiva bind()

```
sockaddr_in my_addr;
...
ret = bind(sd, (struct sockaddr *) &my_addr,
sizeof(my_addr));
```

man 2 bind per ulteriori dettagli

Primitiva listen()

- Mette il socket in attesa di eventuali connessioni.
- Usata dal server per dire che è disposto ad accettare richieste di connessione su un certo socket

int listen(int sd, int backlog);

- sd: descrittore di socket sul quale il server si mette in ascolto
- backlog: dimensione massima per la coda di connessioni pendenti (connessioni established in attesa della accept)
- Restituisce 0 in caso di successo; -1 in caso di errore (setta *errno*)

Primitiva accept()

- Usata dal server per accettare richieste di connessione
- Estrae la prima richiesta di connessione dalla coda delle connessioni pendenti relativa al (listening) socket
- Crea un nuovo socket (connected socket) e gli associa la connessione.
- Il listening socket è usato per accettare le richieste
- Il connected socket è usato per la comunicazione vera e propria con il client
  - In un server c'è sempre un solo socket in ascolto, e le varie connessioni vengono gestite dai socket creati dalla accept
- Il connected socket ha le stesse proprietà del listening socket

Primitiva accept()

int accept(int sd, struct sockaddr* addr, socklen_t* addrlen);

- sd: descrittore di socket creato con la socket()
  - listening socket
- addr: puntatore alla struttura che sarà riempito con l'indirizzo del client (IP e porta)
- addrlen: puntatore alla dimensione della struttura addr che viene restituita

- Restituisce il descrittore del connected socket; -1 in caso di errore (e setta errno)
- Se non ci sono connessioni completate la funzione è bloccante

Creazione della connessione

Primitiva connect()

- Usata dal client per stabilire una connessione con il server
  - usando il socket creato localmente

```
int connect(int sd, const struct sockaddr*
serv_addr, socklen_t addrlen);
```

  - sd: socket creato presso il cliente con la socket()
  - serv_addr: struttura contenente l'indirizzo IP ed il numero di porta del server da contattare
  - addrlen: dimensione della struttura serv_addr
- Restituisce 0 in caso di connessione; -1 in caso di errore (e setta errno)

Lato Server

```
#define SA struct sockaddr;
struct sockaddr_in my_addr, cl_addr;
int ret, len, sk, cn_sk;

sk = socket(PF_INET, SOCK_STREAM, 0);
memset(&my_addr, 0, sizeof(my_addr));
my_addr.sin_family = AF_INET;
my_addr.sin_addr.s_addr = htonl(INADDR_ANY);
my_addr.sin_port = htons(1234);

ret = bind(sk, (SA *) &my_addr, sizeof(my_addr));
ret = listen(sk, 10);

len = sizeof(cl_addr);
cn_sk = accept(sk, (SA *) &cl_addr, &len);
```

- Con INADDR_ANY il server si mette in ascolto su una qualsiasi delle sue interfacce di rete

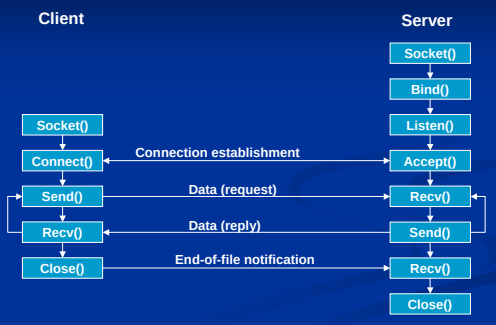
Lato Cliente

```
#define SA struct sockaddr;
struct sockaddr_in srv_addr;
int ret, sk;

sk = socket(PF_INET, SOCK_STREAM, 0);
memset(&srv_addr, 0, sizeof(srv_addr));
srv_addr.sin_family = AF_INET;
srv_addr.sin_port = htons(1234);
ret = inet_pton(AF_INET, "192.168.1.1", &srv_addr.sin_addr);

ret = connect(sk, (SA *) &srv_addr, sizeof(srv_addr));
```

Interazioni Client-Server



Primitiva send()

- Usata per spedire dati attraverso il socket

  ssize_t send(int sd, const void* buf, size_t len, int flags);

  - sd: descrittore del socket usato per la comunicazione
  - buf: buffer contenente il messaggio da spedire
  - len: lunghezza del messaggio
  - flags: definisce il comportamento della send
- Restituisce il numero di caratteri spediti; -1 in caso di errore

Invio dati

```
int ret, sk_a;
char msg[1024];
...
strcpy(msg, “something to send”);
ret = send(sk_a, (void *) msg, strlen(msg), 0);
if(ret == -1 || ret < strlen(msg)){  /* error */
...
}
```

Primitiva Receive()

- Usata per ricevere dati da un certo socket

  ssize_t recv(int sd, void* buf, size_t len, int flags);
  - sd: socket dal quale ricevere i dati
  - buf: buffer dove mettere i dati ricevuti
  - len: dimensione del buffer
  - flags: definisce il comportamento della recv
- Restituisce il numero di byte ricevuti; -1 in caso di errore
- È bloccante

Ricezione dati

```
int ret, len, sk_a;
char msg[1024];
...
ret = recv(sk_a, (void *) msg, len, MSG_WAITALL);
/* non ritorna finchè non ha letto l'intera lungh. del msg */

ret = recv(sk_a, (void *) msg, len, 0);
/* len is the size of the incoming message (<= sizeof(msg)) */

if( (ret == -1) || (ret<len) ) { /* error */
  ...
}
```

Numero di caratteri che si vogliono leggere

Primitiva close()

- Marca come closed il socket
- Il socket non può più essere usato per inviare o ricevere dati

  int close(int sd)
  - sd è il descrittore del socket che si vuole chiudere
- Restituisce 0 se tutto è andato bene; -1 altrimenti

Tipologie di server

- Server iterativo
  - viene servita una richiesta alla volta
- Server concorrente:
  - Per ogni richiesta da parte di un client (accettata dalla *accept*) il server
    - Crea un processo figlio (primitiva fork())
    - Crea un thread
    - Attiva un thread da un pool creato in anticipo
  - Il processo/thread si incarica di gestire il client in questione

Server concorrente multiprocesso

```
For(;;) {
    consd = accept(listensd, …);          /* probably blocks */
    if((pid = fork()) = = 0) {
        close(listensd);          /* child closes listening socket */
        doit(consd);              /* process the request */
        close(consd);             /* done with this client */
        Exit(0);                  /* child terminates */
    }
    close(consd);                 /* parent closes conn. socket */
}
```

- La close(consd) è obbligatoria
  - decrementa il numero di riferimenti al socket descriptor.
  - La sequenza di chiusura non viene innescata fintanto che il numero di riferimenti non si annulla

Server concorrente multiprocesso

```
#include <sys/types.h>
#include <unistd.h>
...
int sock, cl_sock, ret;
struct sockaddr_in srv_addr, cl_addr;
pid_t child_pid;
sock = socket(PF_INET, SOCK_STREAM,0);
if(sock==-1){ /*errore*/}
/*inizializzazione srv_addr*/
bind(sock, &srv_addr, sizeof(srv_addr));
listen(sock,QUEUE_SIZE);
while(1){
    cl_sock = accept(sock, &cl_addr, sizeof(cl_addr));
    if(cl_sock==-1){ /*errore*/}
    /* gestione cl_addr */
    child_pid = fork();
    if(child_pid==0) /* sono nel processo figlio*/
        gestisci_richiesta(cl_sock, sock, ...); /*funzione per la gestione delle
    richieste per il det servizio */
    else /* sono nel processo padre*/
        close(cl_sock);
}
```

Server concorrente multi-threaded

```
For(;;) {
    consd = accept(listensd, ...);          /* probably blocks */
    if ( pthread_create( &tid, NULL, doit, (void*)consd ) ) {
                exit(0);            /* error on thread creation */
    }
}
```

- Chiusura della connessione
  - in questo caso la close(consd) viene fatta dal thread gestore (all'interno della funzione gestione_richiesta)

Server concorrente multi-threaded

```
#include <sys/types.h>
#include <unistd.h>
...
void* gestisci_richiesta( void* socket ) { /*funzione per la gestione delle richieste */ }
...
int sock, cl_sock, ret;
struct sockaddr_in srv_addr, cl_addr;
pthread_t tid;
sock = socket(PF_INET, SOCK_STREAM,0);
if(sock==-1){ /*errore*/}
/*inizializzazione srv_addr*/
bind(sock, &srv_addr, sizeof(srv_addr));
listen(sock,QUEUE_SIZE);
while(1){
    cl_sock = accept(sock, &cl_addr, sizeof(cl_addr));
    if(cl_sock==-1){ /*errore*/}
    /* gestione cl_addr */
    if ( pthread_create( &tid, NULL, gestisci_richiesta, (void*)cl_sock ) ) {
        exit(0);                /* errore */
    }
}
```


Includes

- Headers da includere
  - #include <unistd.h>
  - #include <sys/types.h>
  - #include <sys/socket.h>
  - #include <arpa/inet.h>
- Per il server multi-threaded aggiungere
  - #include <pthread.h>

Appendice

- Richiami di Programmazione C
- Differenze principali C/C++

Definizioni di variabili

- Le variabili possono essere definite solo all'inizio di un blocco

Stile C++

```
int main(void) {
  int a=5, i, b;
  a=func(1000);
  int b=f(a);
  ...
  for(i=0; a<100; i++) {
    b=f(a);
    int c=0;
    ...
  }
}
```

Stile C

```
int main(void) {
  int a=5, i, b;
  int b=f(a);
  a=func(1000);
  ...
  for(i=0; a<100; i++) {
    int c=0;
    b=f(a);
    ...
  }
}
```


Strutture

- Le strutture devono essere sempre riferite con la parola chiave `struct`

Stile C++

```
struct Complesso{
  double re;
  double im;
}

int main(void)
{
  int a=5;
  Complesso c;
  ...
}
```

Stile C

```
struct Complesso{
  double re;
  double im;
}

int main(void)
{
  int a=5;
  struct Complesso c;
  ...
}
```

Gestione memoria dinamica

```
#include <stdlib.h>
```

- Allocazione

```
int main(void) {
 int mem_size=5;
 void *ptr;
 ptr  = malloc(mem_size);
 if(ptr == NULL){
   /* gestione condizione
      di errore */
 }
…
}
```

- Deallocazione

```
int main(void) {
 int mem_size=5;
 void *ptr;
 ptr = malloc(mem_size);
 if(ptr == NULL){
    /* gestione condizione di
    errore */
 }
 …
 free(ptr);
}
```


Operazioni di I/O

```
#include <stdlib.h>
```

- Output a video:

```
int printf(char* format, arg1, …);
```

```
char* str="ciao \n";
printf(str);
printf("str=%s",str);
printf("ciao ciao\n");
int i = 5;
printf("i=%d\n",i);
```

- Input da tastiera:

```
int scanf(char* format, …);
```

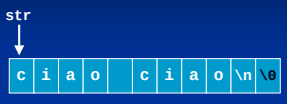
```
int i;
scanf("%d", &i);
/* l'utente digita un valore, es. 10*/
printf("%d", i);   /* stampa 10 */
```


Gestione stringhe

```
#include <string.h>
```

- Lunghezza:

```
int main(void)
{
char *str="ciao ciao\n";
int len;
len = strlen(str);
…
}
```



11 bytes allocati in memoria ma len=10!!

- Confronto:

```
int main(void)
{
char *str1="ciao", *str2="bye";
int i = strcmp(str1, str2);
…
}
```

i<0: str1 alfabeticamente minore di str2
i>0: str1 alfabeticamente maggiore di str2
i=0: str1 uguale a str2

Gestione stringhe

#include <string.h>

- Copia:

```
int main(void)
{
        char str1[100];
        strncpy(str1, "ciao\n", sizeof(str1)-1);
        str1[99] = '\0';
        ...
}
```

- Concatenazione:

```
int main(void)
{
        char str1[100];
        char *str2 = "bye\n"
        strncpy(str1, "ciao\n", sizeof(str1)-1);
        str1[99] = '\0';
        strncat(str1, str2, sizeof(str1)-strlen(str1)-1);
        str1[99] = '\0';
}
```


Gestione files

#include <stdio.h>

- Apertura:

```
int main(void)
{
    FILE *fp;
    fp = fopen("/tmp/prova.txt", "r");
    if(fp == NULL){
    /* gestione errore */
    }
    ...
}
```

Path relativo o assoluto

Modalita' di apertura:
"r" : read-only
"w" : write-only
"r+" : read and write
"a" : append
"a+" : append and read

Nota: Aperture in write/append di files inesistenti causano creazione del file (a patto di avere permessi sufficienti sulle directory del path)

I/O formattato su/da file

- Lettura: fscanf(FILE* fp, char* format, ...)

```
int main(void)
{
        int ret, n;
        FILE *fp;
        fp = fopen("/tmp/prova.txt", "r");
        ret = fscanf(fp, "%d", &n);
}
```

- Scrittura: fprintf(FILE* fp, char* format, ...)

```
int main(void)
{
        int ret;
        char *str="ciao\n";
        FILE *fp;
        fp = fopen("/tmp/prova.txt", "w");
        ret = fprintf(fp, "%s", str);
}
```

I/O su/da file binario

- Lettura:

```
#include <stdio.h>
size_t fread(void *ptr, size_t size, size_t nitems, FILE *stream);

int main(void)
{
        int ret;
        char str[1024];
        FILE *fp;
        fp = fopen("/tmp/prova.txt", "r");
        ret = fread(str, 1, sizeof(str)-1, fp);
        str[1023]='\0';
}
```


I/O su/da file binario

- Scrittura:

```
#include <stdio.h>
size_t fwrite(void *ptr, size_t size, size_t nitems, FILE *stream);

int main(void)
{
        int ret;
        char *str="ciao ciao";
        FILE *fp;
        fp = fopen("/tmp/prova.txt", "w");
        ret = fwrite(str, 1, strlen(str), fp);
}
```


Gestione files

- Dimensione:

```
#include <stdio.h>
#include <sys/stat.h>
int main(void) {
        int ret, size;
        struct stat info;
        ret = stat("/tmp/prova.txt", &info);
        size = info.st_size;
}
```

- Chiusura:

```
int main(void) {
        FILE *fp;
        fp = fopen("/tmp/prova.txt", "r");
        fclose(fp);
}
```

Includes

- Headers da includere
  - #include <stdlib.h> malloc(), free(), system()
  - #include <stdio.h> printf(), fopen(), fclose(), ...
  - #include <string.h> strlen(), strncpy(), ...
  - #include <types.h>
  - #include <sys/stat.h> stat()
  - #include <unistd.h>

Compilazione C

- Il file sorgente è identificato dall'estensione `c` (`nomefile.c`)
- **`gcc`** è il compilatore GNU per programmi scritti in C o C++
  - `gcc -c prova.c` (compila il file sorgente e genera un file oggetto con lo stesso nome: "prova.o")
  - `gcc -c -o pippo.o prova.c` (per specificare il nome del file oggetto)
  - `gcc -Wall -o prova prova.c` (compilazione e generazione del file eseguibile "prova". `-Wall` indica al compilatore di mostrare a video tutti i messaggi di Warning.)
- Esecuzione di un programma:
  - `./prova` (se il file eseguibile si trova nella directory corrente)