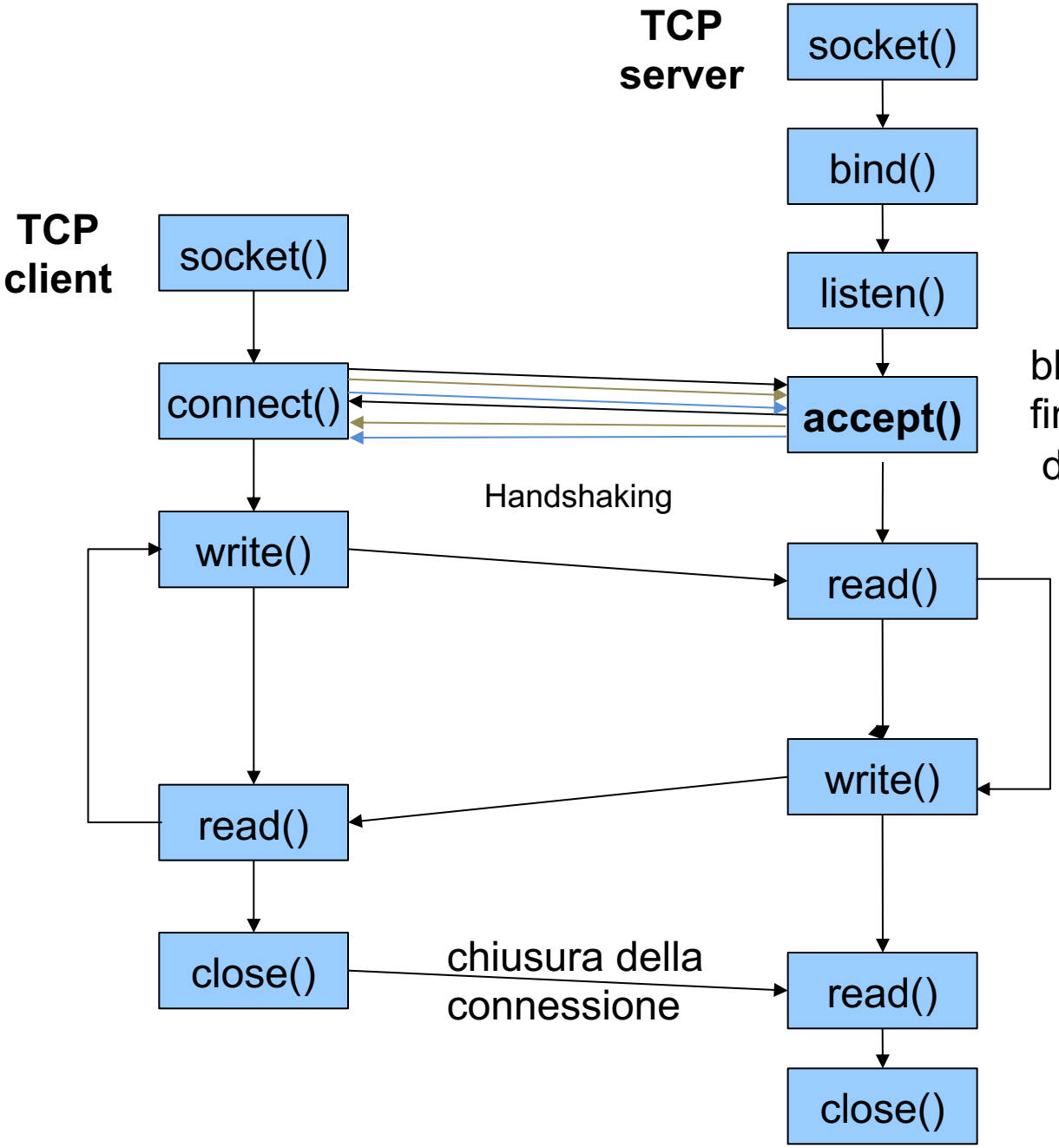
Laboratorio di Reti di Calcolatori

Lezione 5


TCP server
socket()
bind()
listen()
accept()
TCP client
socket()
connect()
Handshaking
write()
read()
write()
read()
close()
chiusura della connessione
read()
close()
blocca il processo fino alla connessione di un client

Server concorrenti

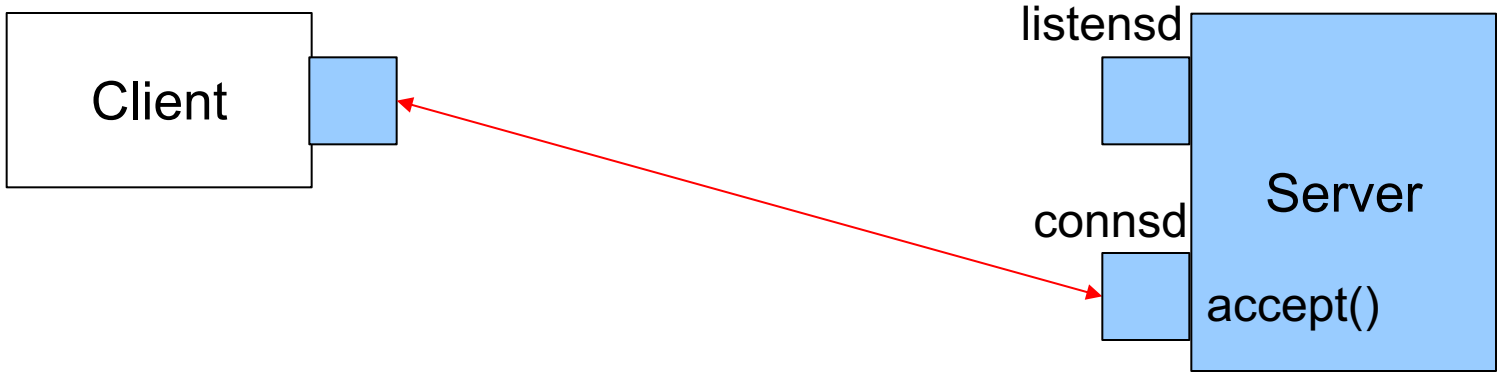
- Gestiscono più connessioni contemporaneamente
- Utilizzano una seconda istanza di se stessi per gestire le connessioni client
- Utilizzano la system call **fork**() per generare un processo figlio
- I processi server padre e figlio vengono eseguiti "contemporaneamente"

Server concorrenti

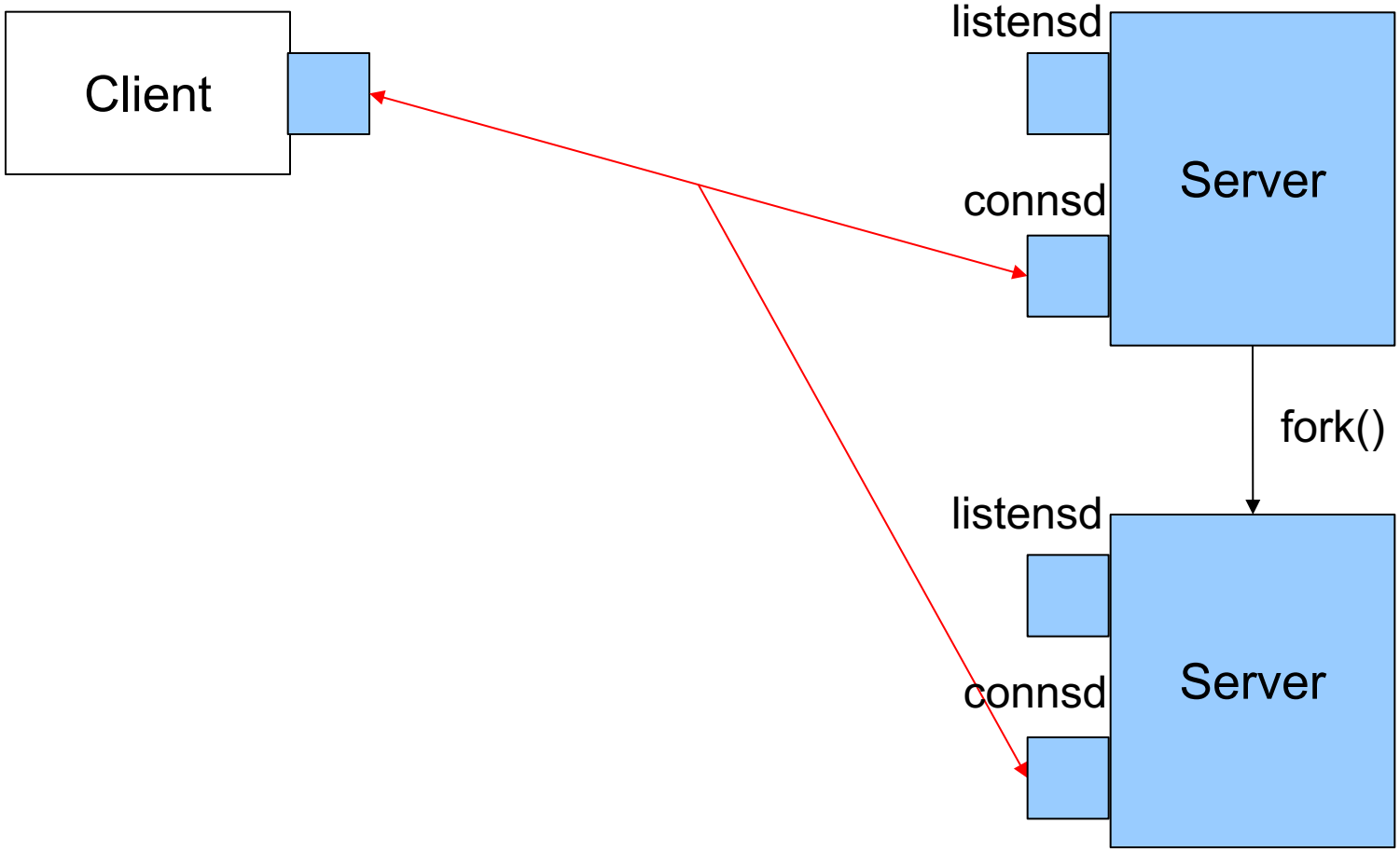
- Il processo figlio gestisce la connessione con un dato client
- Il processo padre può accettare nuove connessioni
- Ogni nuova connessione, genera un nuovo processo figlio che gestisce le richieste del client

Server Concorrenti

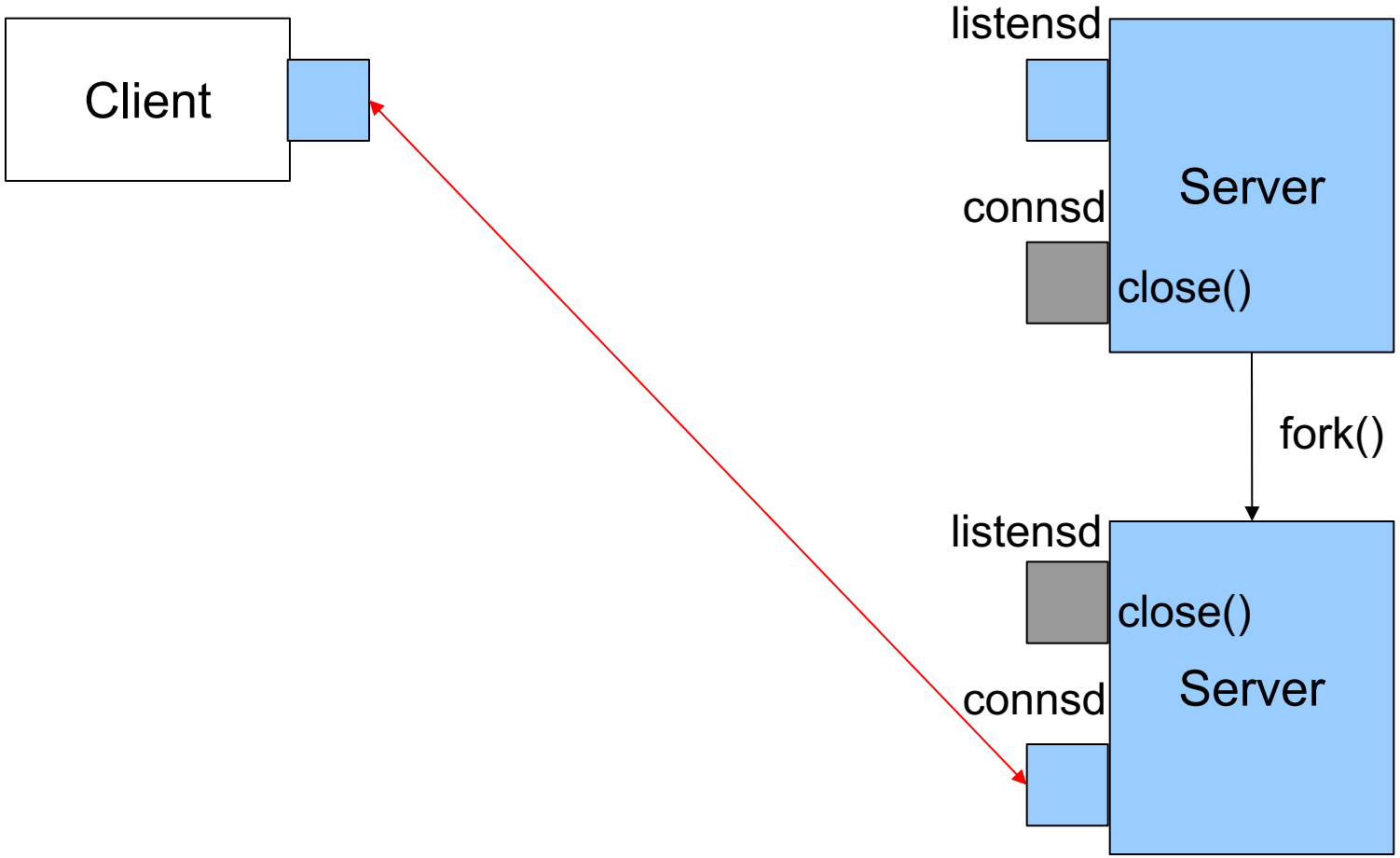
Server Concorrenti



Server Concorrenti



Server Concorrenti



fork

- pid_t **fork**(void);
- Crea un nuovo processo figlio copia esatta del processo chiamante (padre)
- Eredita i descrittori del processo padre
- Restituisce un diverso valore al padre e al figlio:
  - al padre restituisce il pid del figlio
  - al figlio restituisce 0

Server Concorrente

- Socket(...); Bind(...); Listen(...);
- while (1) {
- connsd = Accept(listensd, NULL, NULL);
- if ((pid = **fork**()) == 0) { /* processo figlio */
- close(listensd); /* chiude listensd interagisce con il client
  tramite la connessione con connsd */
- ...
- exit(0); /* Terminazione del figlio */
- }/* il processo padre chiude connsd e ripete il ciclo */
- close(connsd);
- }

connsd – e' il socket della connessione con il client

listensd – e' il socket in attesa di connessioni

Server Concorrente

- 1 # include <sys/ types .h> /* predefined types */
- 2 # include <unistd .h> /* include unix standard library */
- 3 # include <arpa / inet .h> /* IP addresses conversion utililites */
- 4 # include <sys/ socket .h> /* socket library */
- 5 # include <stdio .h> /* include standard I/O library */
- 6 # include <time .h>

Server Concorrente

```
if ( ( list_fd = socket(AF_INET, SOCK_STREAM, 0) ) < 0 ) {
  perror("socket");
  exit(1);
  }
 serv_add.sin_family      = AF_INET;
 serv_add.sin_addr.s_addr = htonl(INADDR_ANY);
 serv_add.sin_port        = htons(13);
 if ( bind(list_fd, (struct sockaddr *) &serv_add, sizeof(serv_add)) < 0 ) {
  perror("bind");
  exit(1);
 }
 if ( listen(list_fd, 1024) < 0 ) {
  perror("listen");
  exit(1);
 }
```

Server Concorrente

```
8 int main (int argc , char * argv [])
9 {
10    int list_fd , conn_fd ;
11    int i;
12    struct sockaddr_in serv_add , client ;
13    char buffer [ MAXLINE ];
14    socklen_t len;
15    time_t timeval ;
16    pid_t pid;
17    int logging =0;
      … socket, bind, listen
19    /* write daytime to client */
20    while (1) {
21          len = sizeof ( client );
            /*accept*/
            …
```

Server Concorrente

```
27          /* fork to handle connection */
28          if ( (pid = fork ()) < 0 ){
29                  perror (" fork error ");
30                  exit ( -1);
31          }
32          if (pid == 0) { /* child */
33                  close ( list_fd );
34                  timeval = time ( NULL );
35                  snprintf (buffer , sizeof ( buffer ), " %.24 s\r\n", ctime (&
timeval ));
36                  if ( ( write ( conn_fd , buffer , strlen ( buffer ))) < 0 ) {
37                          perror (" write error ");
38                          exit ( -1);
39                  }
```

Server Concorrente

```
40          if ( logging ) {
41                  inet_ntop ( AF_INET , & client . sin_addr , buffer , sizeof ( buffer ));
42                  printf (" Request from host %s, port %d\n", buffer ,
43                  ntohs ( client . sin_port ));
44          }
45      close ( conn_fd );
46      exit (0);
47 } else { /* parent */
48      close ( conn_fd );
49      }
50 }
51 /* normal exit , never reached */
52 exit (0);
53 }
```

Esercizi

- Completare il server concorrente server_c_incomplete.c

Terminazione di un processo figlio

- Quando un processo figlio termina
  - viene inviato il segnale SIGCHLD al padre
  - il processo diventa “zombie”
- Gli zombie sono processi che hanno terminato l'esecuzione ma restano presenti nella tabella dei processi
- In genere possono essere identificati dall'output del comando ps per la presenza di una Z nella colonna di stato

Segnali

- Comunicazione asincrona tra processi
- Insieme fissato di segnali a cui corrispondono delle azioni di default (man 7 signal)
- E' possibile fare in modo che quando il destinatario riceve un segnale venga eseguita una procedura specifica (handler)

handler

- Un handler (gestore) è una funzione del tipo:

```
void funzione(int num_segnale) {

printf("%d", num_segnale);

}
```

- Una volta che l'handler termina, l'esecuzione del processo riprende dal punto in cui era stato interrotto

Catturare un segnale

- signal(SIGINT, handit)
- imposta la funzione handit come handler del segnale SIGINT
- E' anche possibile ignorare un segnale
  - signal(SIGINT, SIG_IGN)
- oppure ritornare alla reazione di default
  - signal(SIGINT, SIG_DFL)

Ignorare terminazione dei figli

- In Linux è possibile attraverso la chiamata signal(SIGCHLD, SIG_IGN)
- fare in modo che i processi figli non restino nella condizione di zombie una volta terminati
- Questo **non è conforme** allo standard POSIX

Opzioni del Socket

- Ogni socket aperto ha delle proprietà che ne determinano alcuni comportamenti
- Le opzioni del socket consentono di modificare tali proprietà
- Ogni opzione ha un valore di default
  - Alcune opzioni sono binarie (on o off)
  - Altre hanno un valore (int o anche strutture più complesse)

Livello delle Opzioni

- Le opzioni sono divise in vari livelli
  - **SOL_SOCKET livello socket**
  - IPPROTO_IP livello IP
  - IPPROTO_IPV6 livello IP per la versione 6
  - IPPROTO_ICMPV6 livello messaggi di controllo
  - IPPROTO_TCP livello TCP
- Ogni livello ha varie opzioni

Opzioni di Livello Socket

- SO_BROADCAST permette il broadcast
- SO_DEBUG abilita le informazioni di debug
- SO_DONTROUT Esalta il lookup nella tavola di routing
- SO_ERROR legge l'errore corrente
- SO_KEEPALIVE controlla che la connessione sia attiva
- SO_LINGER controlla la chiusura della connessione
- SO_RCVBUF grandezza del buffer in ricezione
- SO_SNDBUF grandezza buffer in spedizione
- SO_RCVLOWAT soglia per il buffer in ricezione
- SO_SNDLOWAT soglia per il buffer in spedizione
- SO_RCVTIMEO timeout per la ricezione
- SO_SNDTIMEO timeout per la spedizione
- **SO_REUSEADDR permette riutilizzo indirizzi locali**
- SO_REUSEPORT permette riutilizzo porte locali
- SO_TYPE il tipo di socket
- SO_USELOOPBACK per i socket di routing (copia i pacchetti)

Funzioni gestsockopt e setsockopt

- #include <sys/socket.h>

- int **getsockopt**(int sd,int level, int optname, void* optval, socklen_t optlen);

- int **setsockopt**(int sd,int level, int optname, const void* optval, socklen_t* optlen)

- Es.: dopo socket() e prima di bind()

```
int enable = 1;
setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &enable, sizeof(int))
```

Esercizi

- Scrivere un server daytime concorrente che conti il numero di client serviti ed invii questo numero aggiornato ad ogni nuovo client che si connette (un client si intende servito quando ha terminato la connessione con il server)
- Es.: “Sei il client #n, la data locale è: …..”