Reti di Calcolatori - Laboratorio

Lezione 2

Gennaro Oliva

Struttura di un'applicazione server elementare

- Crea il socket
- Gli assegna un indirizzo
- Si mette in ascolto
- Accetta una nuova connessione
- ...
- Chiude il socket

- socket(...)
- bind(...)
- listen(...)
- connect(...)
- ...
- close(...)

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <time.h>
int main(int argc, char **argv)
{
  int          listenfd, connfd;
  struct sockaddr_in  servaddr;
  char        buff[4096];
  time_t        ticks;
  if ( ( listenfd = socket(AF_INET, SOCK_STREAM, 0) ) < 0 ) {
    perror("socket");
    exit(1);
    }
  servaddr.sin_family      = AF_INET;
  servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
  servaddr.sin_port        = htons(13);
  if ( bind(listenfd, (struct sockaddr *) &servaddr, sizeof(servaddr)) < 0 ) {
    perror("bind");
    exit(1);
  }
```

```
  if ( listen(listenfd, 1024) < 0 ) {
    perror("listen");
    exit(1);
  }
for ( ; ; ) {
    if ( ( connfd = accept(listenfd, (struct sockaddr *) NULL, NULL) ) < 0 ) {
      perror("accept");
      exit(1);
    }
    ticks = time(NULL);
    snprintf(buff, sizeof(buff), "%.24s\r\n", ctime(&ticks));
    if ( write(connfd, buff, strlen(buff)) != strlen(buff)) {
      perror("write");
      exit(1);
    }
    close(connfd);
  }
}
```

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <time.h>
int main(int argc, char **argv)
{
  int          listenfd, connfd;
  struct sockaddr_in  servaddr;
  char        buff[4096];
  time_t        ticks;
  if ( ( listenfd = socket(AF_INET, SOCK_STREAM, 0) ) < 0 ) {
    perror("socket");
    exit(1);
    }
  servaddr.sin_family      = AF_INET;
  servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
  servaddr.sin_port        = htons(13);
  if ( bind(listenfd, (struct sockaddr *) &servaddr, sizeof(servaddr)) < 0 ) {
    perror("bind");
    exit(1);
  }
```

Direttive al preprocessore

Direttive al preprocessore

- read,write,close
- socket, bind, listen, connect
- struct sockaddr_in
- exit
- time
- strlen

- <unistd.h>
- <sys/types.h> <sys/socket.h>
- <arpa/inet.h>
- <stdlib.h>
- <time.h>
- <string.h>

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <time.h>
int main(int argc, char **argv)
{
  int          listenfd, connfd;
  struct sockaddr_in  servaddr;
  char        buff[4096];
  time_t       ticks;
  if ( ( listenfd = socket(AF_INET, SOCK_STREAM, 0) ) < 0 ) {
    perror("socket");
    exit(1);
    }
  servaddr.sin_family      = AF_INET;
  servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
  servaddr.sin_port        = htons(13);
  if ( bind(listenfd, (struct sockaddr *) &servaddr, sizeof(servaddr)) < 0 ) {
    perror("bind");
    exit(1);
  }
```

Creazione della socket

Perror

void perror(const char *s);

- La funzione perror produce un messaggio sullo standard error che descrive l'ultimo errore avvenuto durante una system call o una funzione di libreria
- Se l'argomento passato non è NULL, viene stampato prima del messaggio d'errore seguito da ':'
- Di solito si passa il nome della routine invocata

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <time.h>
int main(int argc, char **argv)
{
  int          listenfd, connfd;
  struct sockaddr_in  servaddr;
  char        buff[4096];
  time_t        ticks;
  if ( ( listenfd = socket(AF_INET, SOCK_STREAM, 0) ) < 0 ) {
    perror("socket");
    exit(1);
    }
  servaddr.sin_family      = AF_INET;
  servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
  servaddr.sin_port        = htons(13);
  if ( bind(listenfd, (struct sockaddr *) &servaddr, sizeof(servaddr)) < 0 ) {
    perror("bind");
    exit(1);
  }
```

Indirizzo Server

INADDR_ANY

servaddr.sin_addr.s_addr = htonl(INADDR_ANY);

- INADDR_ANY viene utilizzato come indirizzo del server
- L'applicazione accetterà connessioni da qualsiasi indirizzo associato al server
- Se avessimo utilizzato 127.0.0.1 avremmo potuto eseguire soltanto connessioni dalla macchina su cui gira il server

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <time.h>
int main(int argc, char **argv)
{
  int          listenfd, connfd;
  struct sockaddr_in  servaddr;
  char        buff[4096];
  time_t        ticks;
  if ( ( listenfd = socket(AF_INET, SOCK_STREAM, 0) ) < 0 ) {
    perror("socket");
    exit(1);
    }
  servaddr.sin_family      = AF_INET;
  servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
  servaddr.sin_port        = htons(13);
  if ( bind(listenfd, (struct sockaddr *) &servaddr, sizeof(servaddr)) < 0 ) {
    perror("bind");
    exit(1);
  }
```

Assegnazione Indirizzo

Bind

int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);

- Assegna l'indirizzo addr al socket sockfd
- addr è un sockaddr di tipo generico
- Nei socket TCP fallisce se la porta è in uso
- addrlen è sizeof del secondo argomento
- Restituisce 0 oppure -1

```
 if ( listen(listenfd, 1024) < 0 ) {
   perror("listen");
   exit(1);
 }
for ( ; ; ) {
    if ( ( connfd = accept(listenfd, (struct sockaddr *) NULL, NULL) ) < 0 ) {
      perror("accept");
      exit(1);
    }
    ticks = time(NULL);
    snprintf(buff, sizeof(buff), "%.24s\r\n", ctime(&ticks));
    if ( write(connfd, buff, strlen(buff)) != strlen(buff)) {
      perror("write");
      exit(1);
    }
    close(connfd);
  }
}
```

Messa in ascolto

Listen

int listen(int sockfd, int lunghezza_coda);

- Mette il socket in modalita' di ascolto in attesa di nuove connessioni
- Il secondo argomento specifica quante connessioni possono essere in attesa di essere accettate
- Restituisce 0 oppure -1

```
  if ( listen(listenfd, 1024) < 0 ) {
    perror("listen");
    exit(1);
  }
for ( ; ; ) {
    if ( ( connfd = accept(listenfd, (struct sockaddr *) NULL, NULL) ) < 0 ) {
      perror("accept");
      exit(1);
    }
    ticks = time(NULL);
    snprintf(buff, sizeof(buff), "%.24s\r\n", ctime(&ticks));
    if ( write(connfd, buff, strlen(buff)) != strlen(buff)) {
      perror("write");
      exit(1);
    }
    close(connfd);
  }
}
```

Accettazione nuova connessione

Accept

int accept(int sockfd, struct sockaddr *clientaddr, socklen_t *addr_dim);

- Il secondo e terzo argomento servono ad identificare il client possono essere NULL
- Restituisce un nuovo descrittore o -1
- Il nuovo socket e' associato alla nuova connessione
- Il vecchio socket resta in ascolto

```
  if ( listen(listenfd, 1024) < 0 ) {
    perror("listen");
    exit(1);
  }
for ( ; ; ) {
    if ( ( connfd = accept(listenfd, (struct sockaddr *) NULL, NULL) ) < 0 ) {
      perror("accept");
      exit(1);
    }
    ticks = time(NULL);
    snprintf(buff, sizeof(buff), "%.24s\r\n", ctime(&ticks));
    if ( write(connfd, buff, strlen(buff)) != strlen(buff)) {
      perror("write");
      exit(1);
    }
    close(connfd);
  }
}
```

soddisfacimento richiesta

Write

ssize_t write(int fd, const void *buf, size_t count);

- si usa per scrivere su un socket
- write restituisce il numero di byte scritti
- Può accadere che si scrivono meno bytes di quelli richiesti
- Sono necessarie chiamate successive
- FullWrite scrive esattamente count byte s iterando opportunamente le scritture

FullWrite

```
#include <unistd.h>
ssize_t FullWrite(int fd, const void *buf, size_t count)
{
    size_t nleft;
    ssize_t nwritten;
    nleft = count;
    while (nleft > 0) {             /* repeat until no left */
        if ( (nwritten = write(fd, buf, nleft)) < 0) {
            if (errno == EINTR) {   /* if interrupted by system call */
                continue;           /* repeat the loop */
            } else {
                return(nwritten);   /* otherwise exit with error */
            }
        }
        nleft -= nwritten;          /* set left to write */
        buf +=nwritten;             /* set pointer */
    }
    return (nleft);
}
```

Read

ssize_t read(int fd, void *buf, size_t count);

- Si usa per leggere da un socket
- La read blocca l'esecuzione qualora non ci siano dati da leggere ed il processo resta in attesa di dati
- E' normale ottenere meno bytes di quelli richiesti
- Ottenere 0 bytes significa che il socket e' vuoto ed e' stato chiuso

FullRead

```
#include <unistd.h>
ssize_t FullRead(int fd, void *buf, size_t count)
{
    size_t nleft;
    ssize_t nread;
    nleft = count;
    while (nleft > 0) {             /* repeat until no left */
        if ( (nread = read(fd, buf, nleft)) < 0) {
            if (errno == EINTR) {   /* if interrupted by system call */
                continue;           /* repeat the loop */
            } else {
                return(nread);      /* otherwise exit */
            }
        } else if (nread == 0) {    /* EOF */
            break;                  /* break loop here */
        }
        nleft -= nread;             /* set left to read */
        buf +=nread;                /* set pointer */
    }
    return (nleft);
}
```

Funzioni Wrapper

- Nei programmi reali e' necessario verificare la condizione di uscita di ogni chiamata a funzione
- Spesso gli errori determinano la necessità di terminare l'esecuzione
- Per migliorare la leggibilità del codice si possono definire delle funzioni wrapper che chiamano la funzione, ne verificano l'uscita e terminano l'esecuzione in caso di errore

Esempio di Funzione Wrapper per socket

- La procedura Socket è un wrapper per socket

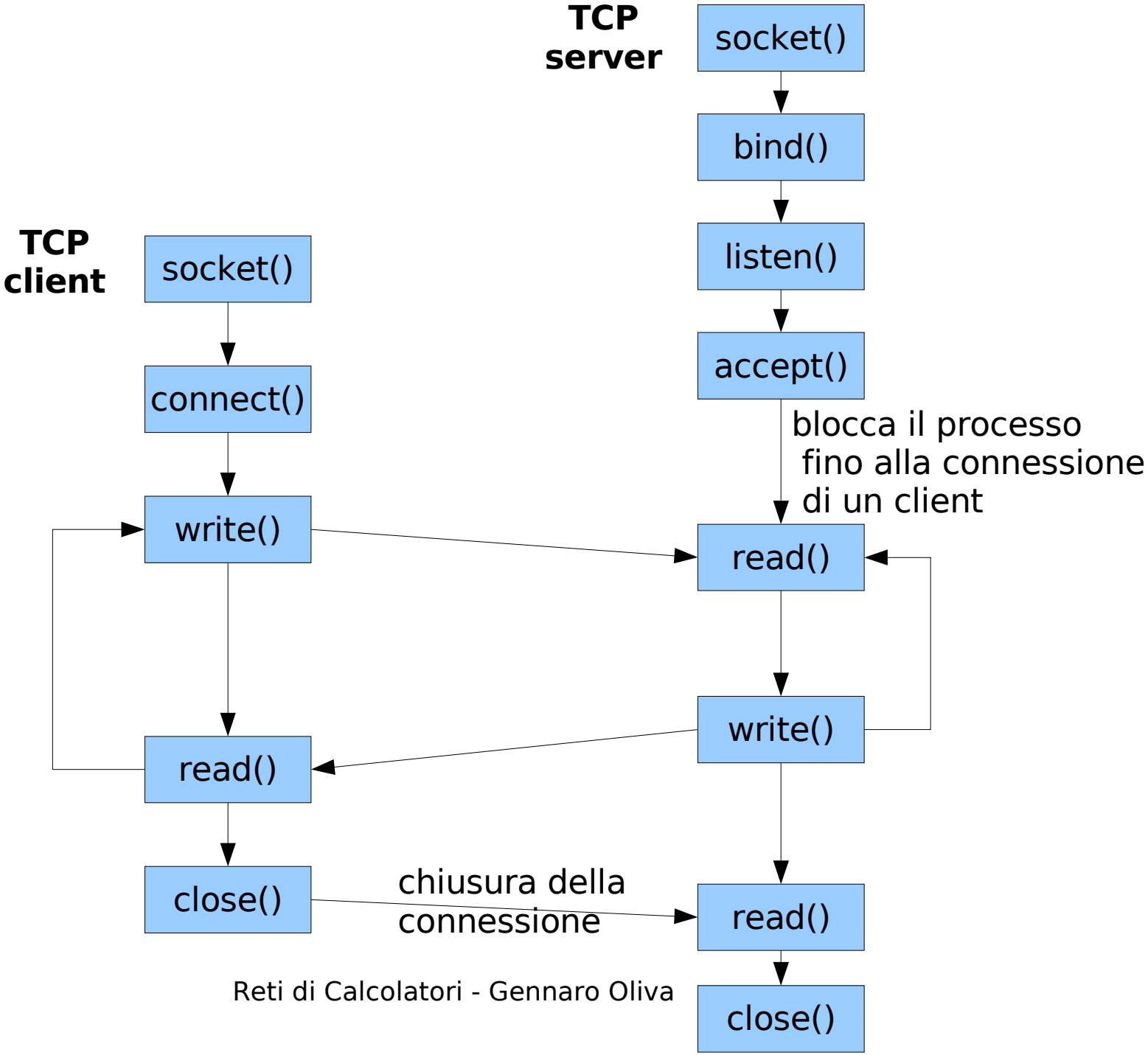
```
int Socket(int family, int type, int protocol)
{
        int    n;
        if ( (n = socket(family, type, protocol)) < 0) {
             perror("socket");
            exit(1);
          }
        return(n);
}
```

```
  if ( listen(listenfd, 1024) < 0 ) {
    perror("listen");
    exit(1);
  }
for ( ; ; ) {
    if ( ( connfd = accept(listenfd, (struct sockaddr *) NULL, NULL) ) < 0 ) {
      perror("accept");
      exit(1);
    }
    ticks = time(NULL);
    snprintf(buff, sizeof(buff), "%.24s\r\n", ctime(&ticks));
    if ( write(connfd, buff, strlen(buff)) != strlen(buff)) {
      perror("write");
      exit(1);
    }
    close(connfd);
  }
}
```

chiusura connessione

Close

- Per terminare una connessione TCP si utilizza la system call close
- Una volta invocato Il descrittore del socket non e' piu' utilizzabile dal processo per operazioni di lettura o scrittura
- Il sottosistema di rete invia i dati in coda e successivamente chiude la connessione


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

Server ricorsivi

- 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 ricorsivi

- 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

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

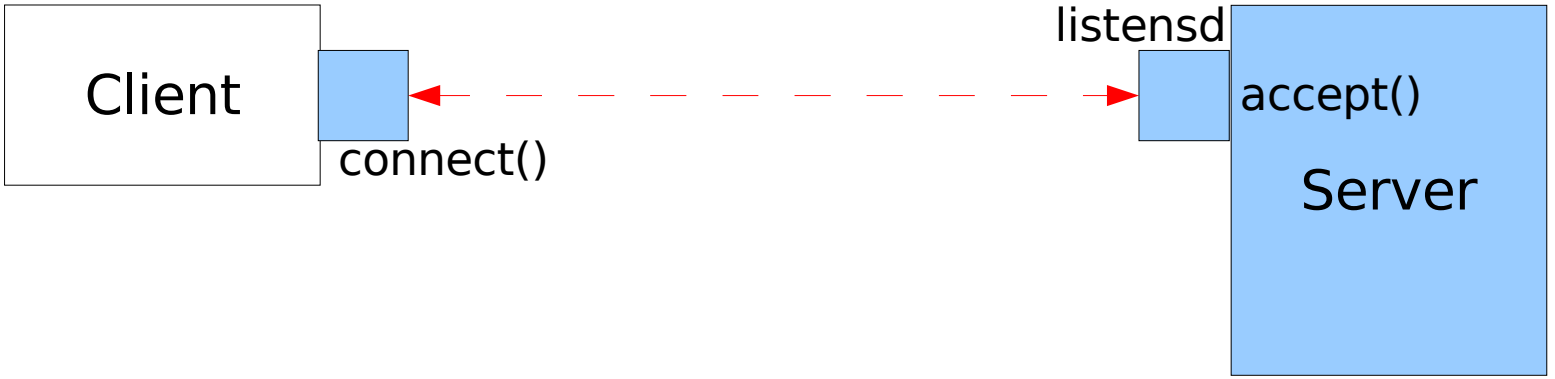
Server Concorrente

```
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);
}
```

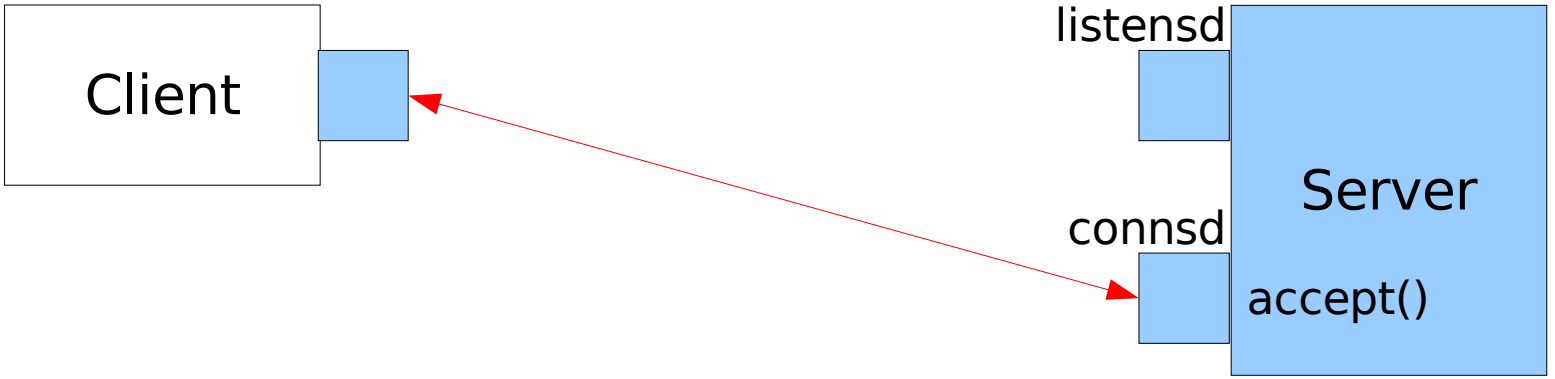
connsd – e' il socket della connessione con il client

listensd – e' il socket in attesa di connessioni

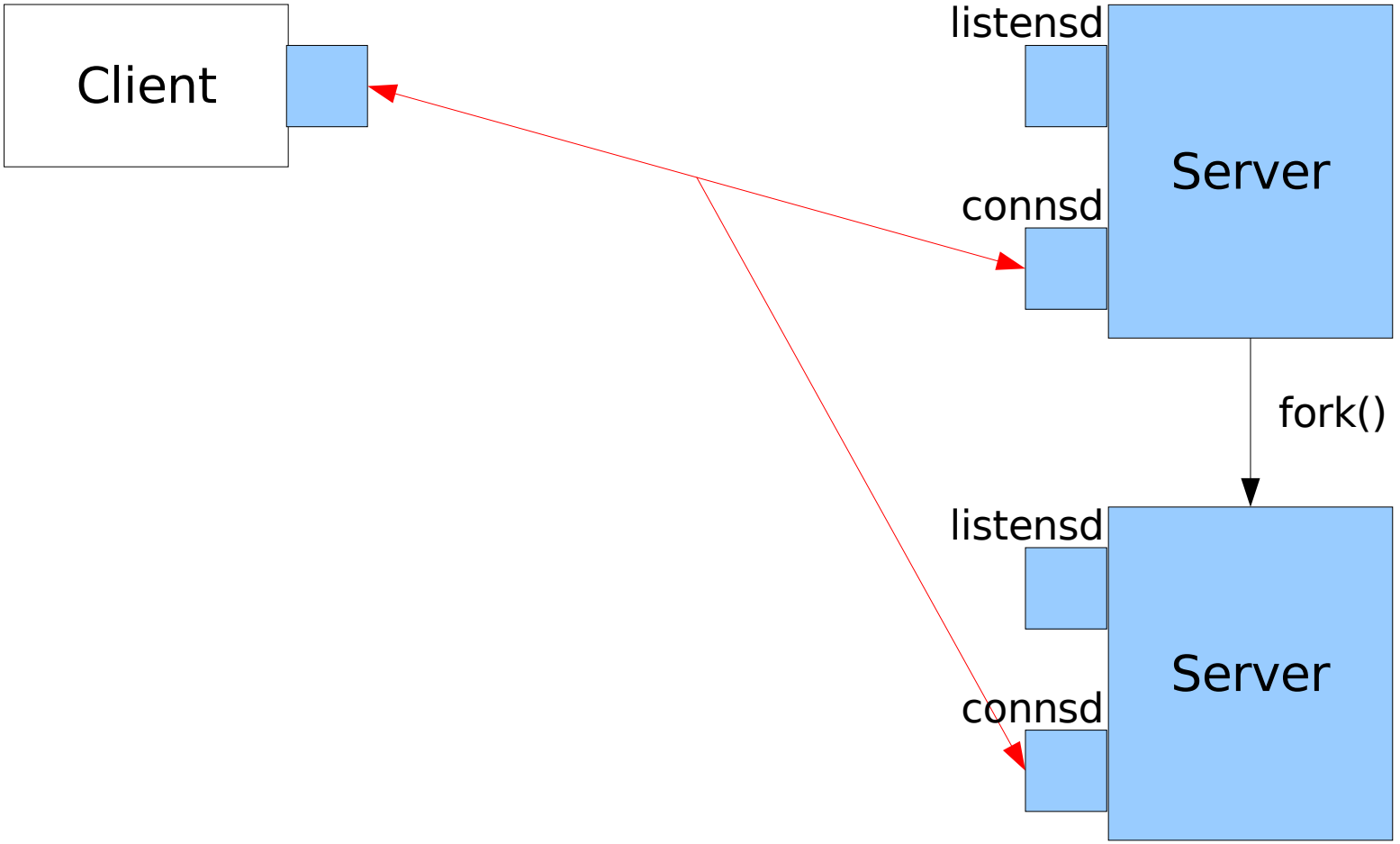
Server Concorrenti



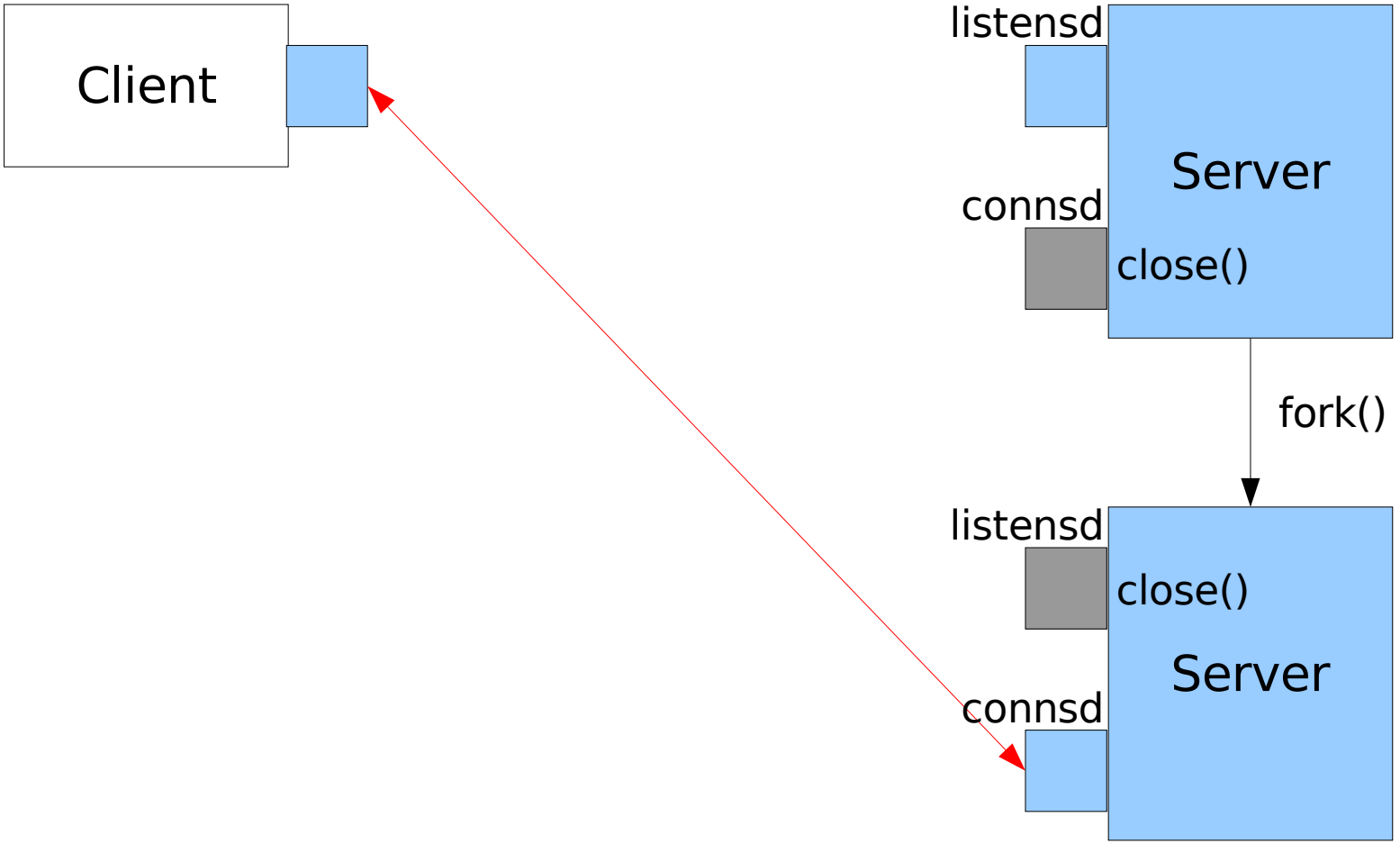
Server Concorrenti



Server Concorrenti



Server Concorrenti



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

- Con Linux è possibile attraverso la chiamata

  signal(SIGCHLD, SIG_IGN)

- fare in modo che i processi figli non restino nella condizione di zombi una volta terminati
- Questo non è conforme allo standard POSIX

Esercizi

- Esercizio1: Realizzare una libreria di wrapper per le chiamate alle procedure per le socket:
  - socket, connect, bind, listen, accept, fork
- Esercizio2: Utilizzando le funzioni wrapper
  - riscrivere il client daytime presentato nella prima lezione
  - scrivere un server daytime e testarlo