Concurrent Programming in Unix

Short tutorial on socket programming

Giuseppe Lipari
http://feanor.sssup.it/~lipari

Scuola Superiore Sant'Anna

Outline

Client Server model

In client-server model:

- The server is initially blocked waiting for connection requests by the clients;
- communication is initiated by the client;
  - the client needs to know the address of the server, while the server accepts connections by any client;
- Once the communication channel has been established, the server knows the client address, and communication can be carried on from both sides (initiated by the server or by the client).

Sockets

- A socket is a communication channel between processes (IPC).
  - It is a general communication interface, that can be used for many protocols
  - We will only analyze the Internet protocols (TCP/IP)
  - Depending on the protocol, different primitives can be used for communication (read/write, recv, send).

Socket interface

- A socket is a file descriptor, that can be created with the following primitive:

```
#include <sys/socket.h>

int socket(int family, int type, int protocol);
```

Input arguments:

family Specifies the protocol family.

type It specifies the kind of connection tjhat will be established on the socket

protocol Protocol that will be used. Usually set equal to 0 (the default protocol), as the pair (family, type) usually identifies a default protocol.

- The socket function returns a file descriptor that can be used later to
  - establish a connection
  - perform the communication
- The usage is different in the server and in the clients.

Socket families and types

In this short tutorial we will only analyze the combination `family=AF_INET` and `type=SOCK_STREAM`, that correspondes to TCP/IP. However, there are other possibilities:

family	Description
`AF_INET`	IPv4 protocols
`AF_INET6`	IPv6 protocols
`AF_LOCAL`	Unix domain protocols
`AF_ROUTE`	Routing sockets
`AF_KEY`	Key sockets

Table: Protocol families

type	Description
`SOCK_STREAM`	stream socket
`SOCK_DGRAM`	datagram socket
`SOCK_RAW`	raw socket

Table: Socket type

TCP/IP connection scheme

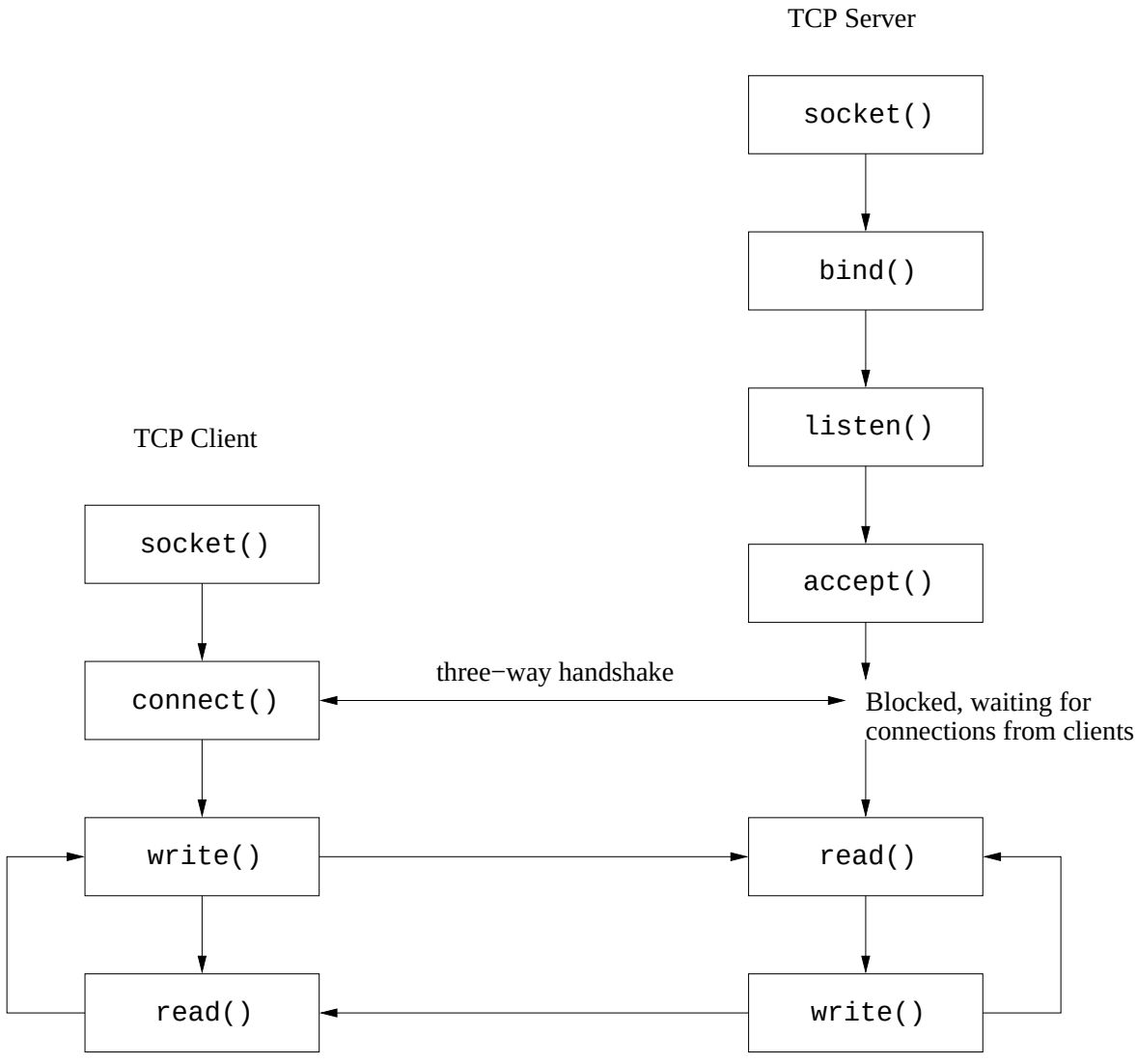

Figure: Connection scheme

Server sequence

This is the list of calls in the server:

- socket creation
- bind
- listen
- accept

Let's start from the bind

- It binds the socket to a specific internet address
- An internet address (also called end-point) consists of two components:
  - IP address
  - port number (16 bits, 0-65535)
- A node (computer) can have more than one IP address (for example it has multiple network cards because it is attached to more than one network)
- On any internet address, each application can receive connections on any port.

Internet addresses

- Ports in the range 0-1023 are reserved to specific applications (well-known ports)
  - The list of assigned ports is reported in RFC 1700, and updated by IANA (*Internet Assigned Numbers Authority*)
- Ports in the range 1024-49151 are freely usable by applications (registered ports)
  - Some of them are quite commonly used always by the same applications (e.g. Zope uses 8080 consistently)
  - IANA keeps track of these ports, but has no power on them
- Ports in the range 49152-65535 are assigned by the operating system to client processes (ephemeral ports).
  - They are dynamically associated to an application

Internet address specification

In header file `<netinet/in.h>`, it is possible to find the following data structures that must be filled by the application:

```
struct sockaddr_in {
  uint8_t         sin_len;      /* lenght. of struct */
  sa_family       sin_family;   /* AF_INET */
  in_port_t       sin_port;     /* port  (16 bit) */
  struct in_addr  sin_addr;     /* IPv4 address */
  char            sin_zero[8];  /* not used */
};
```

```
struct in_addr {
  in_addr_t   s_addr; /* IPv4 address (32 bit) */
};
```

Getting IP address

- Usually the IP address is unique for each node
- Should not be encoded in the program (otherwise the program cannot be ported on another computer without re-compiling)
- It is possible to implicitely let the OS set the IP address of the socket by setting the `s_addr` field of the **`struct`** `in_addr` to `INADDR_ANY`

Example:

```
struct sockaddr_in my_addr;
int myport = 50000;

bzero(&my_addr, sizeof(struct sockaddr_in));
my_addr.sin_family = AF_INET;
my_addr.sin_port = htons(myport);
my_addr.sin_addr.s_addr = INADDR_ANY;
base_sd = socket(AF_INET, SOCK_STREAM, 0);
bind(base_sd, (struct sockaddr *)(&my_addr),
     sizeof(struct sockaddr_in));
```

Listen

```
#include <sys/socket.h>

int listen(int sockfd, int backlog);
```

Input arguments:

sockfd — socket descriptor (returned by `socket()`)

backlog — maximum number of connection requests that can be buffered before being processed by the `accept()`.

The listen serves two purpouses:

- Transforms the socket into passive socket (i.e. we want to receive connections on this socket)
- Specifies the lenght of the request queue. It must be set equal to the maximum number of requests that can arrive before two consecutive calls to `accept`, otherwise some request can be lost (they will go into a timeout)

Accept

```
#include <sys/socket.h>

int accept(int sockfd, struct sockaddr *cliaddr,
           socklen_t *addrlen);
```

Input arguments:

sockfd passive socket deswcriptor on which the server will block waiting for incoming connection requests

cliaddr if the function returns succesfully, contains the internet address (IP + port) of the client that has sent the request

addrlen when the function returns, it contains the lenght in bytes of the structure cliaddr

Return value:

- The function will return a new socket descriptor that will be used for communication with the client
- In case or error, returns -1

- The new socket descriptor is active, and is dedicated to communication with the specific client that made the request. It lives until the request has been completed.

Client connection

The client is more simple: it only needs to perform a connect

```
#include <sys/socket.h>

int connect(int sockfd, const struct sockaddr *saddr,
            socklen_t addrlen);
```

Input arguments:

sockfd the socket descriptor that will be used for communication (active)

saddr pointer to a structure that contains the Internet address (IP + port) of the server

addrlen lenght in bytes of the structure given as second paramter

Return value:

- 0 if the connection was correctly established
- -1 in case of error

- The client does not need to call the bind, as its Internet address is automatically specified by the OS (node IP address + ephemeral port).

Connection errors

The client send a SYN packet when the connect is invoked

- If no server is listening on the specified port, the OS of the destination host will *probably* send a RST packet in response to the SYN packet sent by the client, which results in a `ECONNREFUSED` error
- If the destination host cannot be reached, the intermediate routers will send back a set of ICMP packets, and the client gets a `EHOSTUNREACH` or a `ENETUNREACH` error.
- Finally, if the destination host does not respond to the SYN (for some reason), after a timeout (typically 75 seconds), the error `ETIMEDOUT` is returned.

Closing the connection

```
#include <unistd.h>

int close(int fd);
```

- The same function used for files
- The function returns immediately
- on internet sockets, the remote connection must be closed, and this may take some time;
  - all queued data are sent
  - then, the socket is internally marked as closed, but cannot be re-used until the remote host agrees on the closure
  - if the socket descriptor is shared among different processes, a reference counter is decreased, and only when the last process close the socket, the closing procedure is initiated

Getting host name

In most cases, we need to refer to hosts by names. The DNS protocol is used through the following function

```
#include <netdb.h>

struct hostent *gethostbyname(const char *name);
```

Warning! The function returns a pointer to a unique address in memory where the structur is stored. Therefore, after the following code, s1 and s2 point to the same address in memory that contains the address of the second host:

```
struct hostent *s1,*s2;
...

s1 = gethostbyname("www.sssup.it");
s2 = gethostbyname("www.ing.unipi.it");
```

Example: the client

A simple example: a server that transforms all characters of a string into capital letters.

```
struct sockaddr_in s_addr;
struct hostent     *server;

int  sd, s_port;
char msg[100];

int main(int argc, char *argv[])
{
  if (argc < 3) user_err("usage: client <servername> <port>");

  s_port = atoi(argv[2]);

  sd = socket(AF_INET, SOCK_STREAM, 0);

  bzero(&s_addr, sizeof(struct sockaddr_in));
  s_addr.sin_family = AF_INET;
  s_addr.sin_port = htons(s_port);
  server = gethostbyname(argv[1]);
  if (server == 0) sys_err("server not found!");
```

Example continued

```
  bcopy((char*)server->h_addr,
        (char*)&s_addr.sin_addr.s_addr,
        server->h_length);

  if (connect(sd, CAST_ADDR(&s_addr), sizeof(struct sockaddr_in)) < 0)
    sys_err("connect failed!");

  sprintf(msg, "I am the client with PID n %d\n", getpid());
  printf("Client: sending message!\n");
  write(sd, msg, strlen(msg));
  read(sd, msg, 100);
  printf("client: Message received back: %s", msg);
  close(sd);
}
```

Example: the server

Let's start with an utility function that sets up the passive socket.

```
int init_sd(int myport)
{
  struct sockaddr_in my_addr;
  int sd;

  bzero(&my_addr, sizeof(struct sockaddr_in));
  my_addr.sin_family = AF_INET;
  my_addr.sin_port = htons(myport);
  my_addr.sin_addr.s_addr = INADDR_ANY;

  sd = socket(AF_INET, SOCK_STREAM, 0);
  if (bind(sd, CAST_ADDR(&my_addr), sizeof(my_addr)) < 0)
    sys_err("bind failed!");

  if (listen(sd, 5) < 0)
    sys_err("listen failed!");
  printf("Server listening on port %d\n", myport);

  return sd;
}
```

Example: the server

Now the main cycle: notice that we move the actual service into a separate service routine `do_service()`.

```
int main(int argc, char *argv[])
{
  struct sockaddr_in c_add;
  int base_sd, curr_sd;
  int addrlen;
  int myport;
  int err = 0;

  if (argc < 2) user_err("usage: server <port>");
  myport = atoi(argv[1]);

  base_sd = init_sd(myport);

  while (!err) {
    curr_sd = accept(base_sd, CAST_ADDR(&c_add), &addrlen);
    if (curr_sd < 0) sys_err("accept failed!");

    do_service(curr_sd);
    close(curr_sd);
  }
  close(base_sd);
}
```

Example: the service routine

```
void do_service(int sd)
{
  int i, l=1;
  char msg[BUFFERSIZE];
  char ris[BUFFERSIZE];

  do {
    l = read(sd, msg, BUFFERSIZE - 1);
    if (l == 0) break;
    msg[l] = 0;
    printf("Server: received %s\n", msg);
    for (i=0; i<l; i++) msg[i] = toupper(msg[i]);
    printf("Server: sending %s\n", msg);
    write(sd, msg, l);
  } while (l!=0);
}
```

Outline

Why concurrency in servers

In the previous example, we have shown a sequential server.

- It serves requests sequentially, in order of arrival (FIFO)
- A client has to wait for all preceding requests and for its request to be served before getting the response
- Problems:
  - A short request by a client may have to wait for longer requests to be completed
  - The server can be blocked on I/O while serving a request; this is inefficient!
- A solution is to have concurrent servers:
  - Multi-process: one process per client (dynamically created, or “pre-forked”);
  - Multi-thread: one thread per client (dynamically created, or pre-created).

Multi-process servers

```
int main(int argc, char *argv[])
{
  struct sockaddr_in c_add;
  int base_sd, curr_sd;
  int addrlen;
  int myport;
  int err = 0;
  int ch=0;

  if (argc < 2) user_err("usage: server <port>");
  myport = atoi(argv[1]);

  base_sd = init_sd(myport);

  signal(SIGCHLD, sig_child);

  while (!err) {
    if ( (curr_sd = accept(base_sd, CAST_ADDR(&c_add), &addrlen)) < 0) {
      if (errno == EINTR)
        continue;
      else sys_err("accept failed!");
    }
    ch = fork();
    if (ch == 0) {
      do_service(curr_sd);
      close(curr_sd);
      exit(0);
    }
    close (curr_sd);
  }
  close(base_sd);
}
```

Multi-process servers - II

```
void sig_child(int signo)
{
  pit_t pid;
  int stat;

  while ((pid = waitpid(-1, &stat, WNOHANG)) > 0)
    printf("Il figlio %d ha terminato\n", pid);
}
```

Multi-process servers - III

- When a request arrives, the parent forks a new child and goes back on doing a new accept;
- When the child terminated, a signal `SIGCHLD` is sent to the parent
  - The handler is called which invokes a `waitpid` to retrieve the children return codes and avoid zombies
- If the signal interrupts the parent while blocked on the accept, the parent in unblocked and the accept returns with error.
- To understand what kind of error, we look at `errno` for error `EINTR`. In such a case, the parent blocks again on the accept.

Multi-thread server

```
void *body(void *arg)
{
  int sd = (int) arg;
  int i,l;

  pthread_detach(pthread_self());
  do_service(sd);
  close(sd);
}

int main(int argc, char *argv[])
{
  pthread_t tid;
  struct sockaddr_in c_add;
  int base_sd, curr_sd;
  int addrlen;
  int myport;
  int err = 0;

  if (argc < 2) user_err("usage: server <port>");
  myport = atoi(argv[1]);

  base_sd = init_sd(myport);

  while (!err) {
    curr_sd = accept(base_sd, CAST_ADDR(&c_add), &addrlen);
    if (curr_sd < 0) sys_err("accept failed!");
    pthread_create(&tid, 0, body, (void *)curr_sd);
  }
}
```

Multi-thread server

- To avoid zombie threads, we use `pthread_detach` to detach the thread, so that the main does not need to wait with `pthead_join`

Multi-thread server

- To avoid zombie threads, we use `pthread_detach` to detach the thread, so that the main does not need to wait with `pthead_join`
- In some cases, however, the overhead of creating a new thread for each request could be considered too high

Multi-thread server

- To avoid zombie threads, we use `pthread_detach` to detach the thread, so that the main does not need to wait with `pthead_join`
- In some cases, however, the overhead of creating a new thread for each request could be considered too high
- Another technique consists in creating in advance a certain number of threads ready to serve requests

Multi-thread server

- To avoid zombie threads, we use `pthread_detach` to detach the thread, so that the main does not need to wait with `pthead_join`
- In some cases, however, the overhead of creating a new thread for each request could be considered too high
- Another technique consists in creating in advance a certain number of threads ready to serve requests
- Such threads could call the `accept` themselves

Multi-thread server

- To avoid zombie threads, we use `pthread_detach` to detach the thread, so that the main does not need to wait with `pthead_join`
- In some cases, however, the overhead of creating a new thread for each request could be considered too high
- Another technique consists in creating in advance a certain number of threads ready to serve requests
- Such threads could call the `accept` themselves
- However, it is not possible to block more than one thread on the same passive socket descriptor with `accept`!
- This is a limitation of Unix

Multi-thread server

- To avoid zombie threads, we use `pthread_detach` to detach the thread, so that the main does not need to wait with `pthead_join`
- In some cases, however, the overhead of creating a new thread for each request could be considered too high
- Another technique consists in creating in advance a certain number of threads ready to serve requests
- Such threads could call the `accept` themselves
- However, it is not possible to block more than one thread on the same passive socket descriptor with `accept`!
- This is a limitation of Unix
- We will use an additional mutex.

Pre-created threads

```
pthread_t tid[MAXNTHREAD];
pthread_mutex_t m_acc;

int init_sd(int myport);
void do_service(int sd);

void *body(void *arg)
{
  struct sockaddr_in c_add;
  int addrlen;
  int i,l;
  int base_sd = (int) arg;
  int sd;

  while (1) {
    pthread_mutex_lock(&m_acc);
    sd = accept(base_sd, CAST_ADDR(&c_add), &addrlen);
    pthread_mutex_unlock(&m_acc);

    do_service(sd);
    close(sd);
  }
}
```

Pre-created threads

```
int main(int argc, char *argv[])
{
  int i;
  int base_sd;
  int myport;

  if (argc < 2) user_err("usage: server <port>");
  myport = atoi(argv[1]);

  base_sd = init_sd(myport);
  pthread_mutex_init(&m_acc, 0);

  for (i=0 ; i<MAXNTHREAD; i++)
    pthread_create(&tid[i], 0, body, (void *)base_sd);

  pause();
}
```