Sockets programming Concepts

There are different types of sockets. What makes the difference between them is the topology of the communication and the types of addresses used. There are three types of addresses: the first one is represented by the Internet addresses (the corresponding sockets are called Internet sockets), paths to the local nodes for communicating between processes from the same station (Unix sockets) or addresses of the X.25 type (X.25 sockets, least used).

Socket – it represents an end of the communication, defining a form of the Inter Process Communication, for processes found on the same station, or on different stations of the same network. A pair of connected sockets represents a communication interface between two processes, an interface similar to the pipe interface in Unix.

Association – it defines in a unique way the connection established between two ends of communication, as being a set of parameters like (local_protocol, local_address, local_ port, remote_address, remote_port). The source_port and destination_port notions allow a unique definition of the communication between pairs of sockets and make a unique identification of a SAP (Service access point) of the transport level.

Socket descriptor – it is a file descriptor; the argument used in the library functions.

Binding – it is the operation that associates a socket address to a socket, so the socket can be accessed.

Port – it is an identifier used for making the difference between sockets found at the same address. There can be used addresses between 1024 – 49151 (registered ports) and addresses between 49152 – 65535, named also dynamic or ephemeral ports; the addresses between 1 – 1023 are reserved for the system (see the file etc/services in Unix).

The family of addresses – it represents the format of the addresses used for identifying the addresses used in sockets (usually the Internet and Unix addresses).

The socket address – it depends on the family of addresses used (for the domain of the family of Internet addresses, the socket address is made up by the Internet address and by the port address used at the host).
The Internet address – it is a 4-byte-long address, which identifies a node (host or router) inside a network.

The client – server model

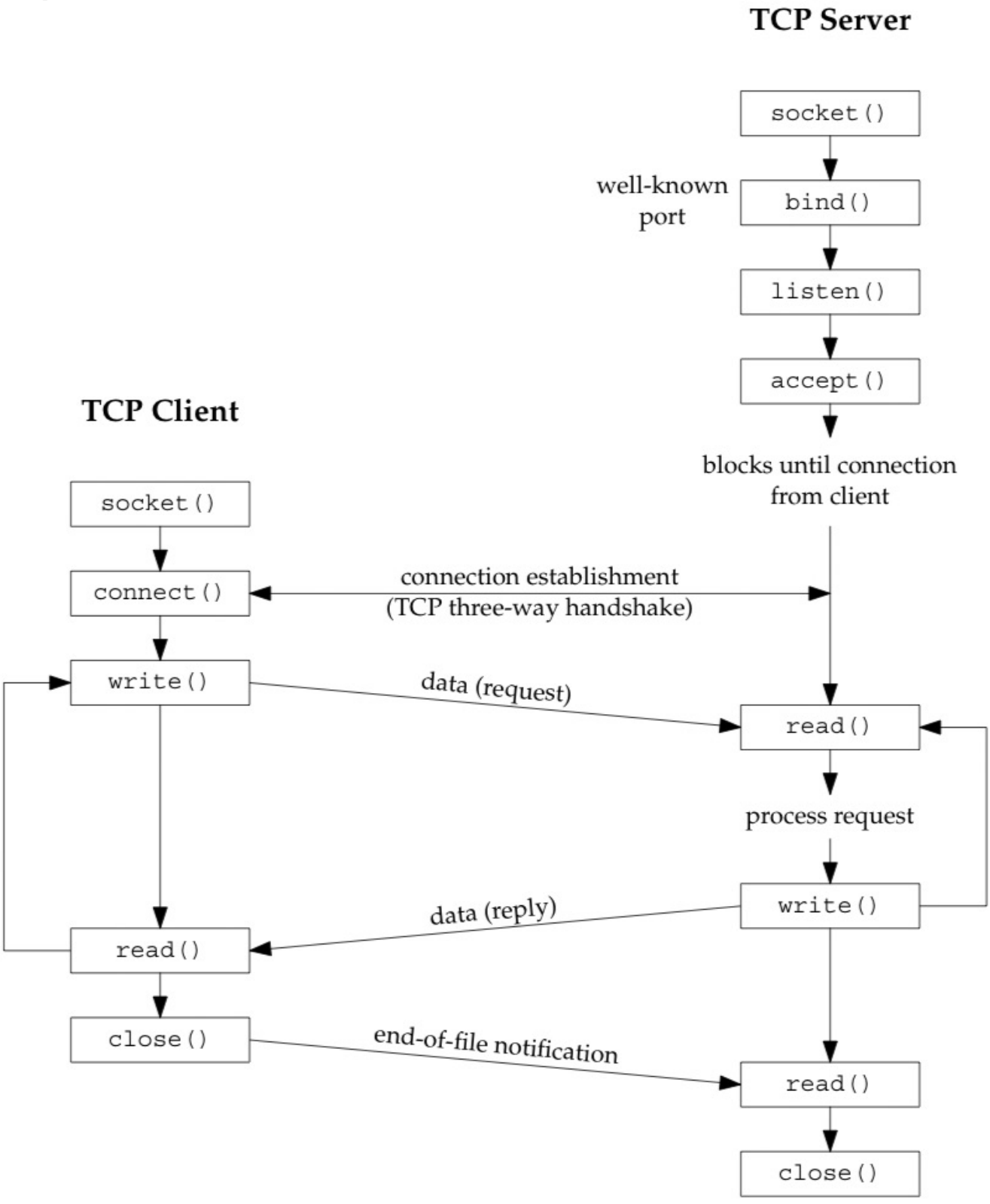
Any operation inside a network can be seen as a client process that communicates with a server process. The server process creates a socket, associates to the socket an address and launches a mechanism for listening for the connection requests of the clients. The client process creates a socket and asks for a connection to the server. After accepting the request by the server and after the connection has been established, a communication between the sockets can be established. This classical model can vary with the nature of the designed application; it can be symmetric or asymmetric, based on simplex or duplex communication. Using this client – server model, different applications have been made, such as: telnet (for the connection of a remote host using the port 23) for which there is a program at that host, program called telnetd, that will respond, ftp/ftpd or bootp/bootpd.

Stream sockets vs. datagram sockets

The communication between client and server programs is made using the level protocols TCP ("Transmission Control Protocols"), which is a connection-oriented transport protocol or UDP ("User Datagram Protocol"), which is a connectionless-oriented transport protocol. The stream sockets are secure communication flows (with no errors), full duplex, based on the TCP protocol, which provides a sequential, errorless data transmission. The datagram sockets, also known under the name of connectionless sockets, use the IP addresses for routing, but the transport level protocol is the UDP. These sockets do not maintain an open connection during the communication, but they make the transfer packet by packet (the tftp - Trivial File Transfer Protocol, the bootp applications use the datagram sockets).

Elementary TCP Sockets

Here we are going to study about elementary socket functions required to write a complete TCP client and server.



This figure shows a timeline of the typical scenario that takes place between a TCP client and server.

Primitive	Meaning
Socket	Create a new communication endpoint
Bind	Attach a local address to a socket
Listen	Announce willingness to accept connections
Accept	Block caller until a connection request arrives
Connect	Actively attempt to establish a connection
Send	Send some data over the connection
Receive	Receive some data over the connection
Close	Release the connection

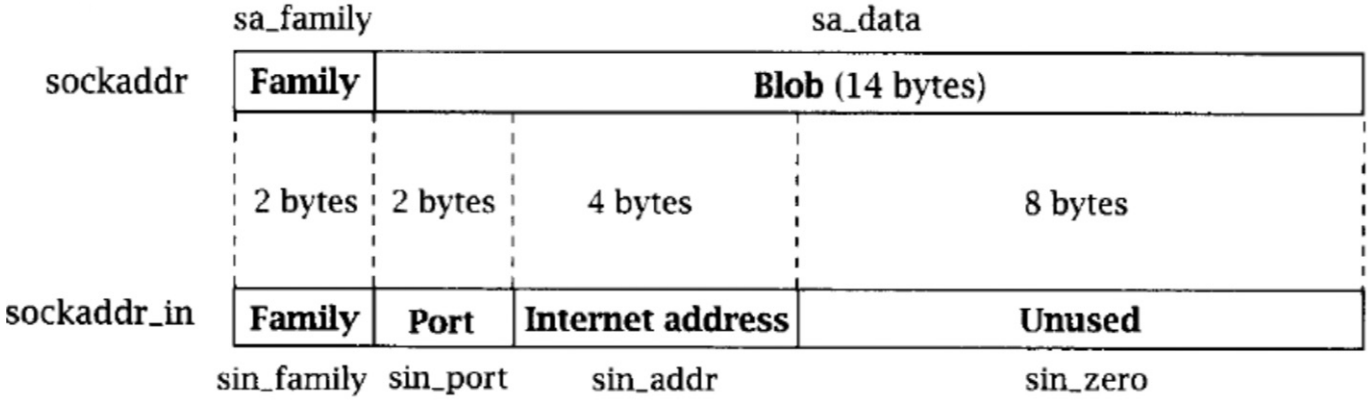
Specifying Addresses

Applications need to be able to specify Internet address and Port number.

Use Address Structure

1. Sockaddr: generic data type
2. in_addr : internet address
3. sockaddr_in: another view of Sockaddr

```
struct sockaddr_in{
unsigned short sin_family; /* Internet protocol (AF_INET) */
unsigned short sin_port;/* Address port (16 bits) */
struct in_addr sin_addr; /* Internet address (32 bits) */
char sin_zero[8];/* Not used */
}
```



This structure allows a simple reference to the elements of a socket address. Sin_zero will be set to 0, using bzero() or memset() functions. The pointer to the struct sockaddr_in maps the pointer to the structure struct sockaddr. We must highlight that sin_family corresponds to sa_family in the structure struct sockaddr, and it will be set to "AF_INET", value used for the TCP/IP protocol family.

The conversion functions

Due to the fact that in data transmission, in order to design portable applications it is necessary to convert the transmitted data between the hosts that use the little_endian or the big_endian representation, and also due to the fact that the data representation at the network level is unique (the protocols of the TCP/IP family use the representation at the network level called NetWork Byte Order), some functions are needed to make these conversions.
Some of the most-used conversion functions in the socket programming are:

- htons() - "Host to Network Short", host conversion to a short network;
- htonl() - "Host to Long Network", host conversion at a long network;
- ntohs() - "Network to Host Short", network conversion to a short host;
- ntohl() - "Network to Host Long", network conversion to a long host.

Being known the fact that sin_addr and sin_port are put in the IP packet, respectively in the UDP-TCP, they have to be in the Network_byte_order, and the sin_family field, used by the kernel of the operating system (to find out what type of address the structure contains), in the Host Byte Order, it is necessary to use the conversion functions.

A basic time-client in C :

Connect to IP of time-a-g.nist.gov to object file of program

ping time-a-g.nist.gov

```c
#include<stdio.h>
#include<sys/socket.h>
#include<arpa/inet.h>
#include<unistd.h>
int main(int argc, char *argv[])
{
  //create socket
  int net_socket;
  net_socket = socket(AF_INET, SOCK_STREAM, 0);
  //server address
  struct sockaddr_in server_address;
  server_address.sin_family = AF_INET;
   server_address.sin_port = htons(13);
   server_address.sin_addr.s_addr = inet_addr(argv[1]);
  //connect to server
  connect(net_socket, (struct sockaddr *) &server_address, sizeof( server_address));
  //recv the data
  char data[201];
  int n;
  n = recv(net_socket, data, 200, 0);
  printf("%s",data);
  //close the socket
  close(net_socket);
  return(0);
}
```

Create a socket

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

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

→ Header file needed

→ Proto type of the function

Returns: non-negative descriptor if OK, –1 on error

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

Protocol	Description
IPPROTO_TCP	TCP transport protocol
IPPROTO_UDP	UDP transport protocol
IPPROTO_SCTP	SCTP transport protocol

type	Description
SOCK_STREAM	stream socket
SOCK_DGRAM	datagram socket
SOCK_SEQPACKET	sequenced packet socket
SOCK_RAW	raw socket

socket () returns the descriptor of the new socket if no error occurs and -1 otherwise.

For protocol we can use 0 for default value the default protocol is TCP

NB: Not all family are compatible with all type of connection

	AF_INET	AF_INET6	AF_LOCAL	AF_ROUTE	AF_KEY
SOCK_STREAM	✓	✓	✓	✗	✗
SOCK_DGRAM	✓	✓	✓	✗	✗
SOCK_SEQPACKET	✓	✓	✓	✗	✗
SOCK_RAW	✓	✓	✗	✓	✓

connect Function

Size of the struct in bytes

Pointer to a struct sockaddr

```
#include <sys/socket.h>
int connect(int sockfd, const struct sockaddr *servaddr, socklen_t addrlen);
```

Returns: 0 if OK, –1 on error

Tries to open a connection to the server

Times out after 75 seconds if no response

In the case of a TCP socket, the connect function initiates TCP's three-way hand- shake (Section 2.6). The function returns only when the connection is established or an error occurs. There are several different error returns possible.

1. If the client TCP receives no response to its SYN segment, ETIMEDOUT is returned. 4.4BSD, for example, sends one SYN when connect is called, another 6 seconds later, and another 24 seconds later (p. 828 of TCPv2). If no response is received after a total of 75 seconds, the error is returned.

2. If the server's response to the client's SYN is a reset (RST), this indicates that no process is waiting for connections on the server host at the port specified (i.e., the server process is probably not running). This is a hard error and the error ECONNREFUSED is returned to the client as soon as the RST is received.

An RST is a type of TCP segment that is sent by TCP when something is wrong. Three conditions that generate an RST are: when a SYN arrives for a port that has no listening server (what we just described), when TCP wants to abort an existing connection, and when TCP receives a segment for a connection that does not exist. (TCPv1 [pp. 246–250] contains additional information.)

3. If the client's SYN elicits an ICMP "destination unreachable" from some inter- mediate router, this is considered a soft error. The client kernel saves the mes- sage but keeps sending SYNs with the same time between each SYN as in the first scenario. If no response is received after some fixed amount of time (75 sec- onds for 4.4BSD), the saved ICMP error is returned to the process as either EHOSTUNREACH or ENETUNREACH. It is also possible that the remote system is not reachable by any route in the local system's forwarding table, or that the connect call returns without waiting at all.

recv Function

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

ssize_t recv(int socket, void *buffer, size_t length, int flags);
```

flags

Specifies the type of message reception. Values of this argument are formed by logically OR'ing zero or more of the following values:

MSG_PEEK Peeks at an incoming message. The data is treated as unread and the next recv() or similar function shall still return this data.

MSG_OOB Requests out-of-band data. The significance and semantics of out-of-band data are protocol-specific.

MSG_WAITALL On SOCK_STREAM sockets this requests that the function block until the full amount of data can be returned. The function may return the smaller amount of data if the socket is a message-based socket, if a signal is caught, if the connection is terminated, if MSG_PEEK was specified, or if an error is pending for the socket.

close Function

When finished using a socket, the socket should be closed

```
int close(int sockfd);
```

0 if successful, -1 if error

sockfd → the file descriptor (socket being closed)

Closing a socket

closes a connection (for stream socket)

frees up the port used by the socket

Designing a basic server and client for time

Server.c

```
#include<stdio.h>
#include<sys/socket.h>
#include <arpa/inet.h>
#include <unistd.h>
#include<stdlib.h>
#include<time.h>
#include<string.h>
int main(int argc, char *argv[])
{
    int n;
    char data[201];
    int net_socket;
    net_socket = socket(AF_INET, SOCK_STREAM, 0);
    struct sockaddr_in server_address, client_address;
    server_address.sin_family = AF_INET;
    server_address.sin_port = htons(5600);
    server_address.sin_addr.s_addr = inet_addr(argv[1]);
    bind(net_socket, (struct sockaddr*)&server_address,sizeof(server_address));
    listen(net_socket,10);
    time_t tick;
    char str[100];
    int fd;
    int c;
    while(1)
    {
       fd=accept(net_socket,(struct sockaddr*)&client_address,&c);
       printf("Accepted");
       tick=time(NULL);
       snprintf(str,sizeof(str),"%s",ctime(&tick));
       printf("%s",str);
       write(fd,str,strlen(str));
    }
    return(0);
}
```

Client.c

```c
#include<stdio.h>

#include<sys/socket.h>

#include <arpa/inet.h>
#include <unistd.h>
#include<stdlib.h>
int main(int argc, char *argv[])
{
    int n;
    char data[201];
    //create a socket
    int net_socket;
    net_socket = socket(AF_INET, SOCK_STREAM, 0);

    //connect to a server
    //where we want to connect to
    struct sockaddr_in server_address;
    server_address.sin_family = AF_INET;
    server_address.sin_port = htons(5600);
    server_address.sin_addr.s_addr = inet_addr(argv[1]);

    connect( net_socket, ( struct sockaddr * ) &server_address, sizeof( server_address));

    n=recv(net_socket, data,200,0);
    printf("%d\n",n);
    printf("%s",data);
    close(net_socket);
    return(0);
}
```

bind function

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

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

Returns: 0 if OK, –1 on error

The bind function assigns a local protocol address to a socket.

The first argument is the socket id created prior to call of bind function. The second argument is a pointer to a protocol-specific address, and the third argument is the size of this address structure.

Calling bind lets us specify a port number, an IP address, both, or neither.

Most of the time bird function does the following task

- Servers bind their well-known port when they start.
- A process can bind a specific IP address to its socket.

Normally, a TCP client does not bind an IP address to its socket. The kernel chooses the source IP address when the socket is connected, based on the outgoing interface that is used, which in turn is based on the route required to reach the server

Calling bind lets us specify the IP address, the port, both, or neither.

Process specifies		Result
IP address	port	
Wildcard	0	Kernel chooses IP address and port
Wildcard	nonzero	Kernel chooses IP address, process specifies port
Local IP address	0	Process specifies IP address, kernel chooses port
Local IP address	nonzero	Process specifies IP address and port

With IPv4, the wildcard address is specified by the constant INADDR_ANY, whose value is normally 0. This tells the kernel to choose the IP address.

```
struct sockaddr_in   servaddr;
servaddr.sin_addr.s_addr = htonl(INADDR_ANY);     /* wildcard */
```

While this works with IPv4, where an IP address is a 32-bit value that can be repre- sented as a simple numeric constant (0 in this case), we cannot use this technique with IPv6, since the 128-bit IPv6 address is stored in a structure.

```
struct sockaddr_in6    serv;
serv.sin6_addr = in6addr_any;     /* wildcard */
```

The system allocates and initializes the in6addr_any variable to the constant IN6ADDR_ANY_INIT. The <netinet/in.h> header contains the extern declaration for in6addr_any.

listen Function

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

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

Returns: 0 if OK, –1 on error

The listen function is called only by a TCP server and it performs two actions:

- When a socket is created by the socket function, it is assumed to be an active socket, that is, a client socket that will issue a connect. The call to listen moves the socket from the CLOSED state to the LISTEN state.
- The second argument to this function specifies the maximum number of connec- tions the kernel should queue for this socket.

To understand the backlog argument, we must realize that for a given listening socket, the kernel maintains two queues:

- An incomplete connection queue, which contains an entry for each SYN that has arrived from a client for which the server is awaiting completion of the TCP three-way handshake.
- A completed connection queue, which contains an entry for each client with whom the TCP three-way handshake has completed.

accept Function

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

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

Returns: non-negative descriptor if OK, –1 on error

accept is called by a TCP server to return the next completed connection from the front of the completed connection queue. If the completed connection queue is empty, the process is put to sleep

The cliaddr and addrlen arguments are used to return the protocol address of the connected peer process (the client). addrlen is a value-result argument. Before the call, we set the integer value referenced by *addrlen to the size of the socket address structure pointed to by cliaddr; on return, this integer value contains the actual number of bytes stored by the kernel in the socket address structure.

If accept is successful, its return value is a brand-new descriptor automatically created by the kernel.

Fork function

There are two typical uses of fork:
1. A process makes a copy of itself so that one copy can handle one operation while the other copy does another task.
2. A process wants to execute another program. Since the only way to create a new process is by calling fork, the process first calls fork to make a copy of itself, and then one of the copies (typically the child process) calls exec (described next) to replace itself with the new program.

If the call to fork() is executed successfully, Unix will

- make two identical copies of address spaces, one for the parent and the other for the child.
- Both processes will start their execution at the next statement following the fork() call.

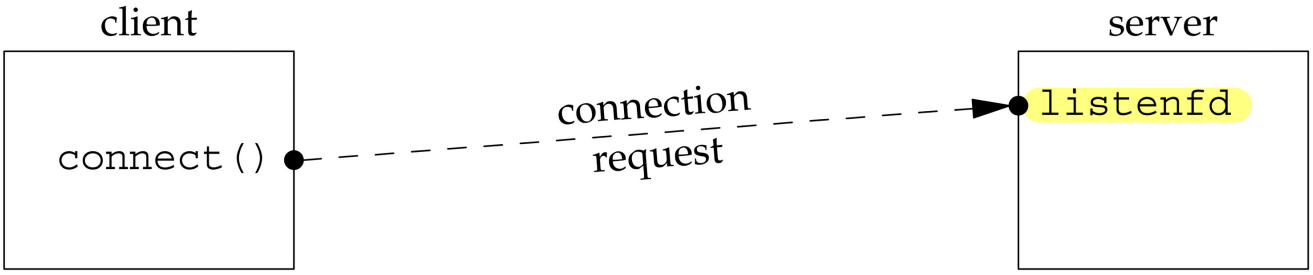
```
#include <unistd.h>

pid_t fork(void);
```

Returns: 0 in child, process ID of child in parent, –1 on error

Concurrent Servers

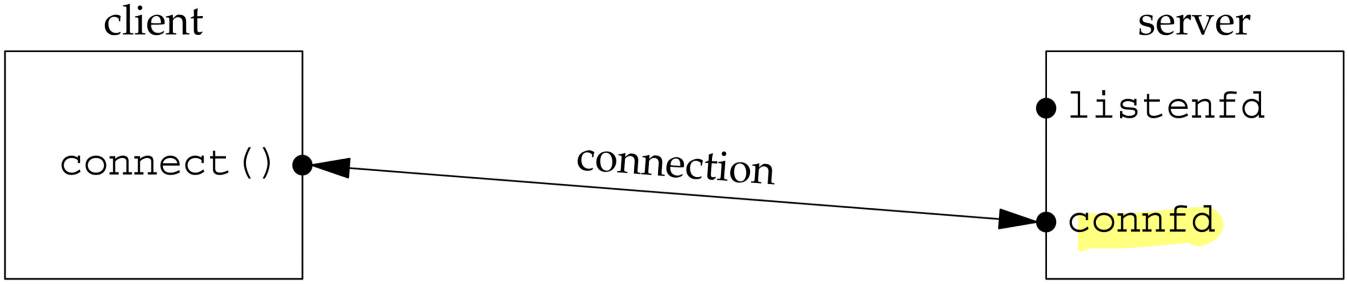
The simplest way to write a concurrent server under Unix is to fork a child process to handle each client.

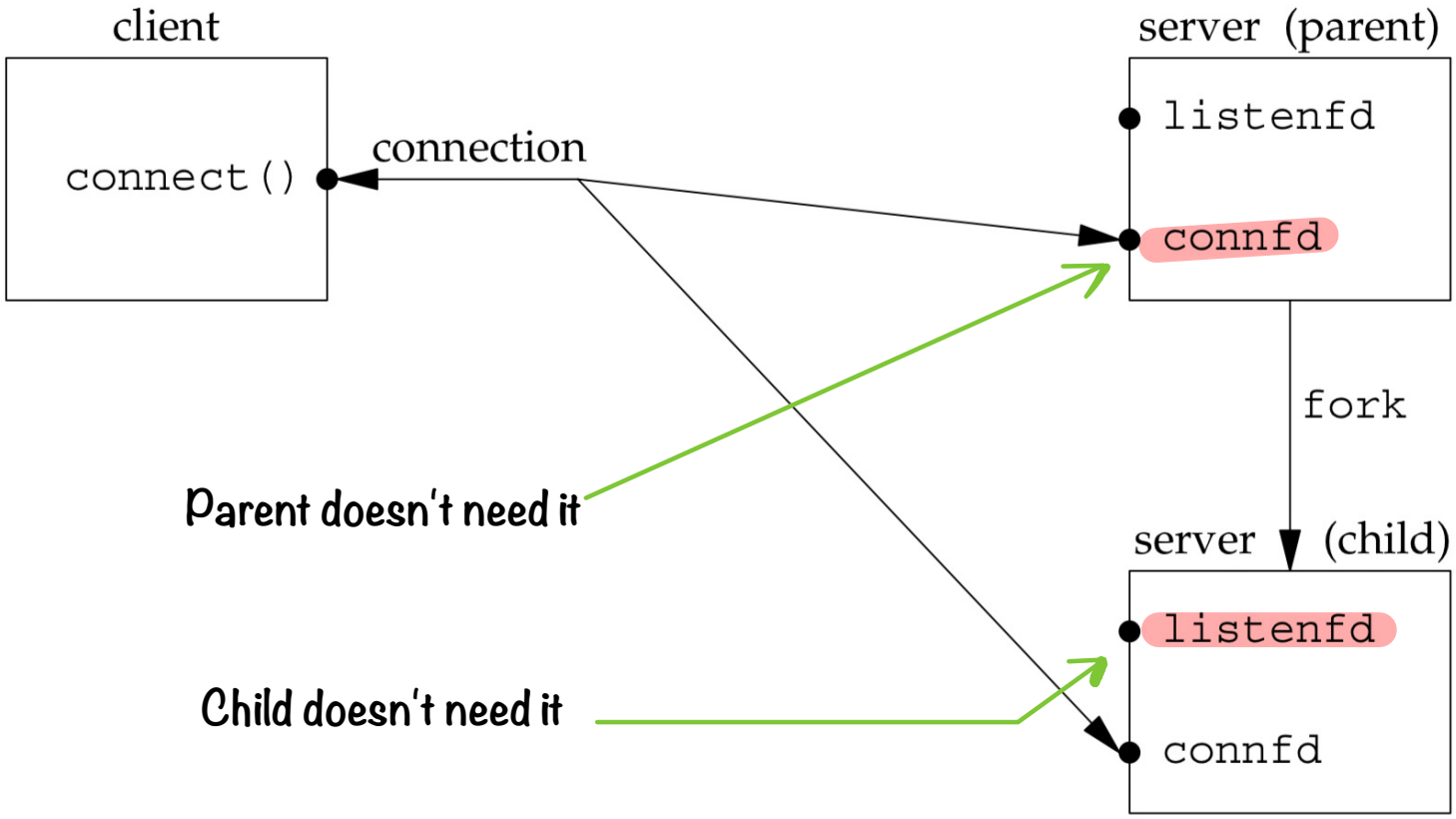
When a connection is established, accept returns, the server calls fork, and the child process services the client and the parent process waits for another connection. The parent closes the connected socket since the child handles the new client.



This figure shows the status of the client and server while the server is blocked in the call to accept and the connection request arrives from the client.

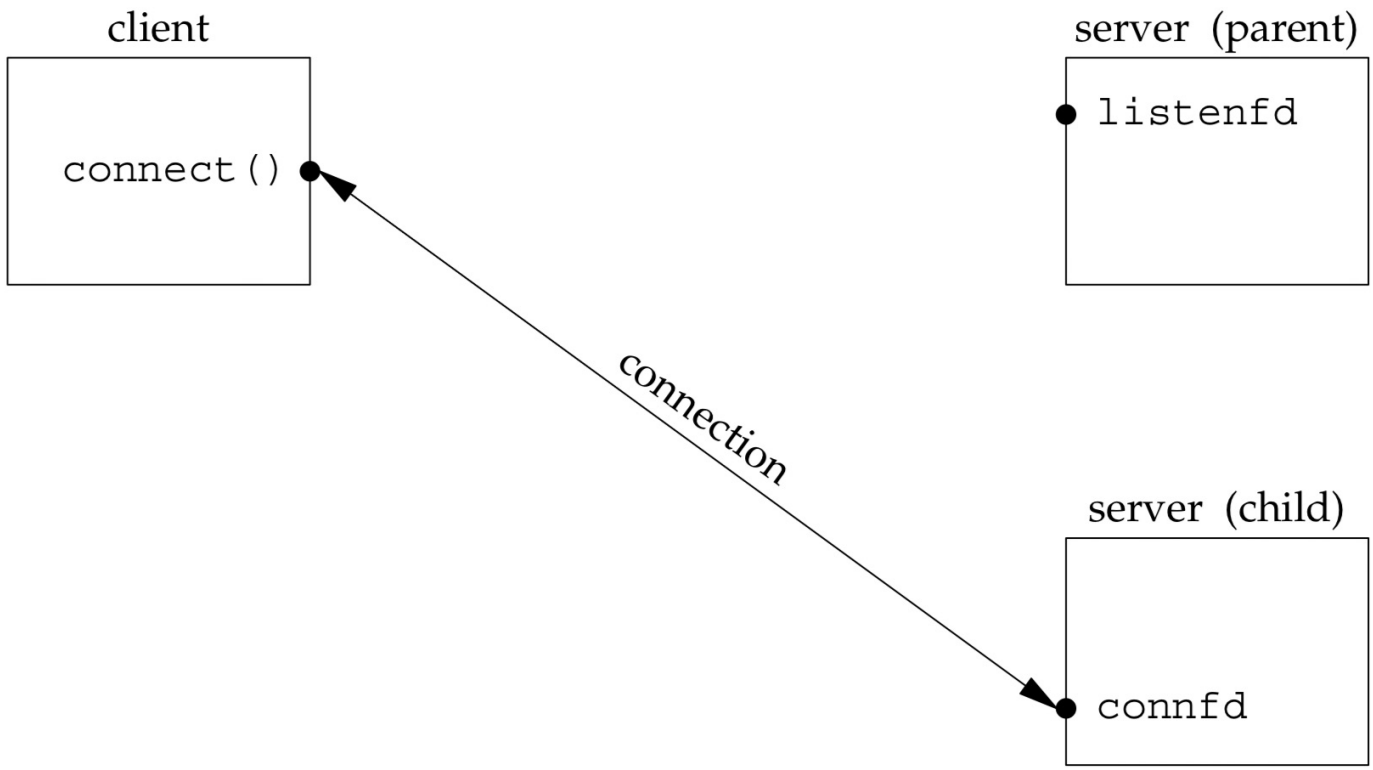
Immediately after accept returns the connection is accepted by the kernel and a new socket, connfd, is created. This is a connected socket and data can now be read and written across the connection.





Notice that both descriptors, listenfd and connfd, are shared (duplicated) between the parent and child.

The next step is for the parent to close the connected socket and the child to close the listening socket.



Concurrent time server.c

```c
#include<stdio.h>

#include<sys/socket.h>

#include <arpa/inet.h>
#include <unistd.h>
#include<stdlib.h>
#include<time.h>
#include<string.h>
int main(int argc, char *argv[])
{
    int n;
    char data[201];
    pid_t pid;
    //create a socket
    int net_socket;
    net_socket = socket(AF_INET, SOCK_STREAM, 0);

    //connect to a server
    //where we want to connect to
    struct sockaddr_in server_address, client_address;
    server_address.sin_family = AF_INET;
    server_address.sin_port = htons(4600);
    server_address.sin_addr.s_addr = inet_addr(argv[1]);

    bind(net_socket, (struct sockaddr*)&server_address,sizeof(server_address));

    listen(net_socket,10);
    time_t tick;
    char str[100];
    int fd;
    int c;
    while(1)
    {
       fd=accept(net_socket,(struct sockaddr*)&client_address,&c);
       printf("Accepted");
      if ((pid = fork())==0)
      {
        close(net_socket);
        while(1)
        {
          tick=time(NULL);
          snprintf(str,sizeof(str),"%s",ctime(&tick));
          write(fd,str,strlen(str));
        }
      }
      else
      {
        close(fd);
      }
    }

    return(0);
}
```

Concurrent time client.c

```c
#include<stdio.h>

#include<sys/socket.h>

#include <arpa/inet.h>
#include <unistd.h>
#include<stdlib.h>
int main(int argc, char *argv[])
{
    int n;
    char data[201];
    //create a socket
    int net_socket;
    net_socket = socket(AF_INET, SOCK_STREAM, 0);

    //connect to a server
    //where we want to connect to
    struct sockaddr_in server_address;
    server_address.sin_family = AF_INET;
    server_address.sin_port = htons(4600);
    server_address.sin_addr.s_addr = inet_addr(argv[1]);

    connect( net_socket, ( struct sockaddr * ) &server_address, sizeof( server_address));
    while(1)
    {
      n=recv(net_socket, data,200,0);
      printf("%d\n",n);
      printf("%s",data);
    }
    close(net_socket);
    return(0);
}
```