Client Server Computing

- Although the Internet provides a basic communication service, the protocol software cannot initiate contact with, or accept contact from, a remote computer. Instead, two application programs must participate in any communication with one application initiates communication and the one accepts it.
- In network applications, a server application waits passively for contact after informing local protocol software that a specific type of message is expected, while a client application initiates communication actively by sending a matched type of message.

Characteristics of Clients Software

- is an arbitrary application program that becomes a client temporarily when remote access is needed, but also performs other computation locally.
- is invoked directly by a user, and executes only for one session.
- runs locally on a user's personal computer.
- actively initiates contact with a server.
- can access multiple services as needed, but actively contacts one remote server at a time.
- does not require special hardware or sophisticated operating system.

Characteristics of Servers Software

- is a special-purpose, privileged program dedicated to providing one service, but can handle multiple remote clients at the same time.
- is invoked automatically when a system boots, and continues to execute through many sessions.
- runs on a shared computer (i.e., not on a user's personal computer).
- waits passively for contact from arbitrary clients.
- accepts contact from arbitrary clients, but offers a single service.
- requires powerful hardware and a sophisticated OS.

Identifying A Particular Service

- Transport protocols assign each service a unique identifier.
- Both client and server specify the service identifier; protocol software uses the identifier to direct each incoming request to the correct server.
- In TCP/IP, TCP uses 16-bit integer values known as **protocol port numbers** to identify services.

Concurrent Server

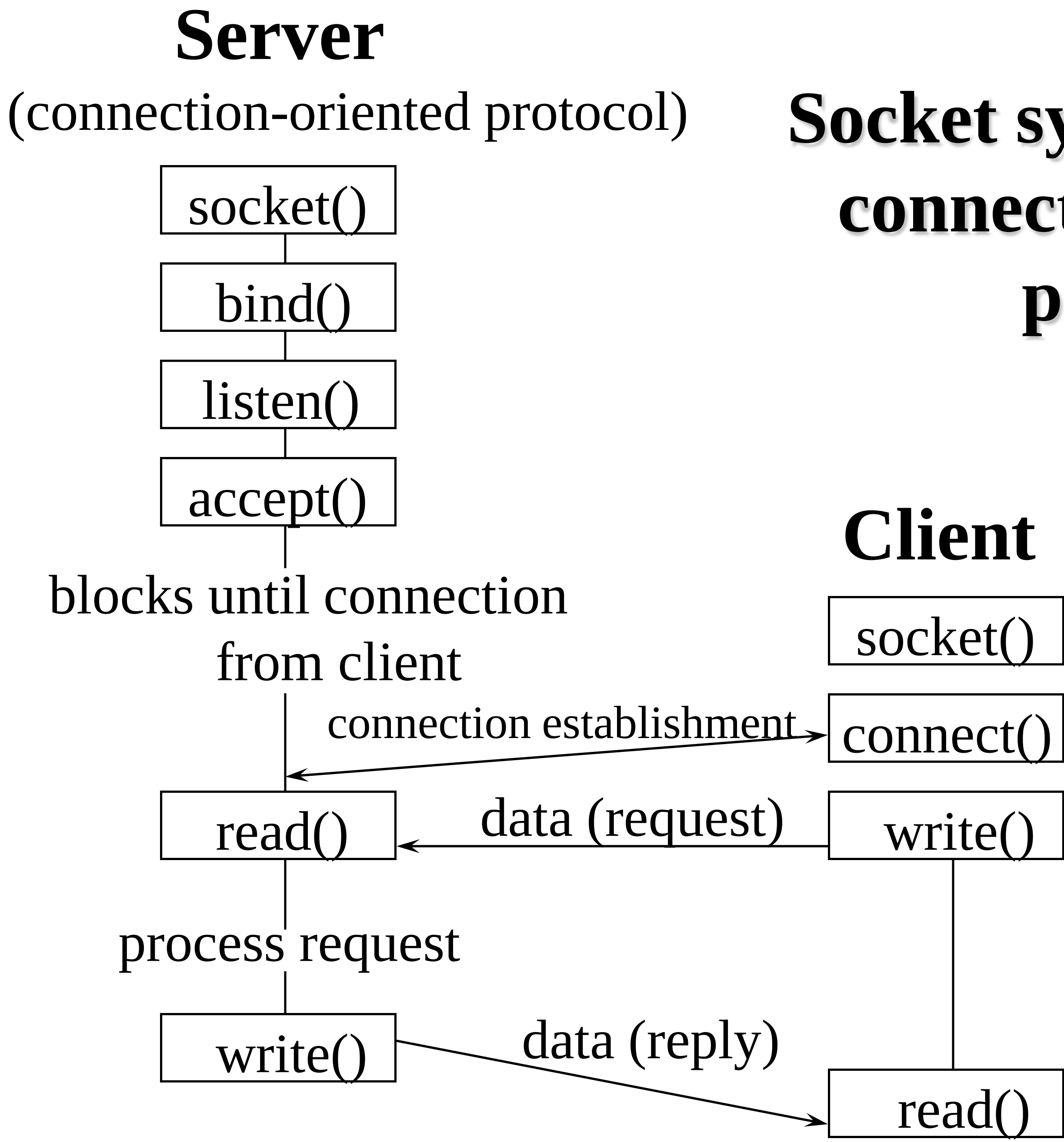
- Concurrent execution is fundamental to servers because concurrency permits multiple clients to obtain a given service without having to wait for the server to finish previous requests.
- In concurrent server designs, the server creates a new thread to handle each client.
- Transport protocols assign an identifier to each client as well as to each service.
- Protocol software on the server’s machine uses the combination of client and server identifiers to choose the correct copy of a concurrent server.

The Socket API

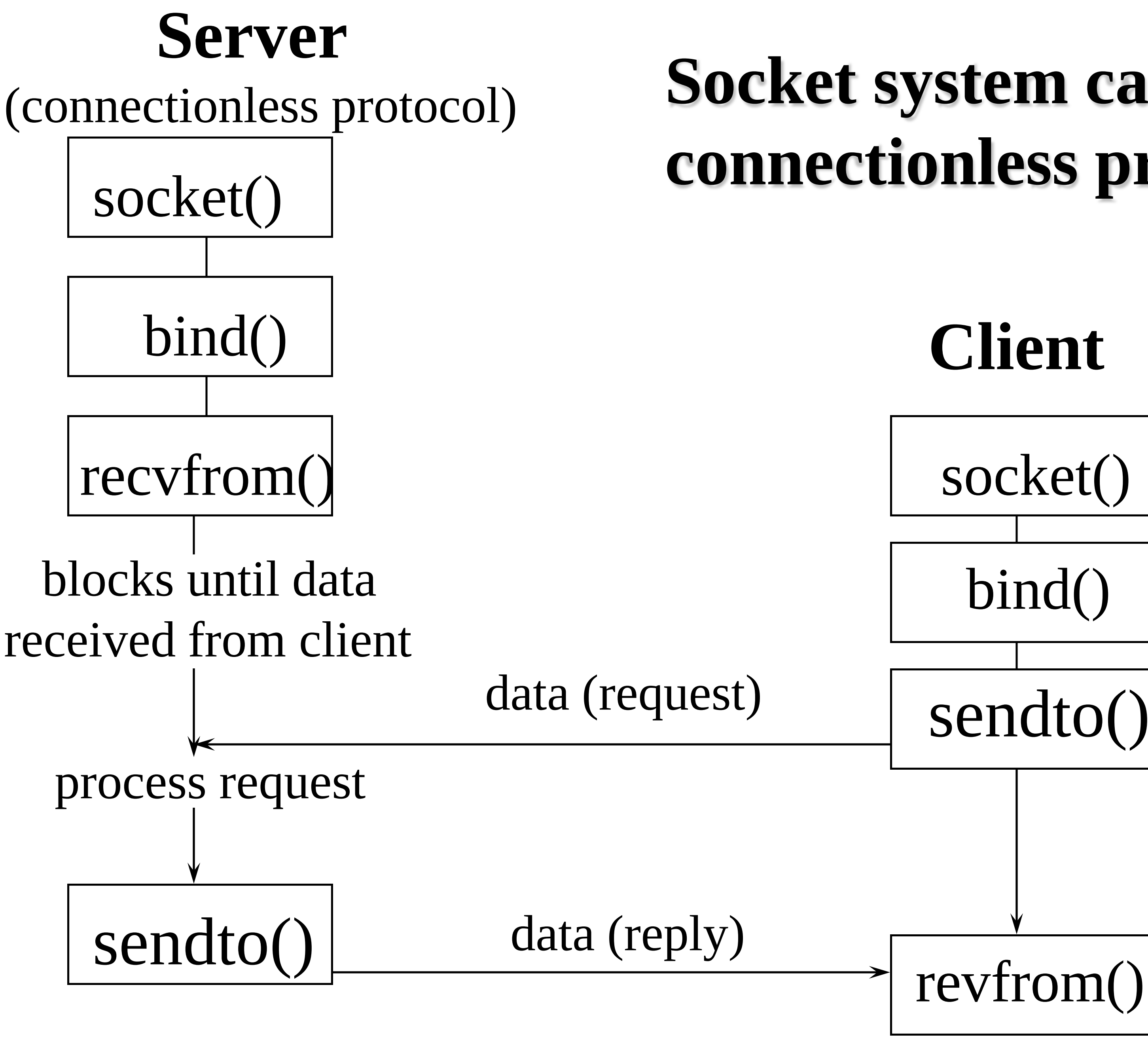
- The interface between an application program and the communication protocols in an operating system (OS) is known as the Application Program Interface or API.
- Sockets provide an implementation of the SAP (Service Access Point) abstraction at the Transport Layer in the TCP/IP protocol suite, which is part of the BSD Unix.
- A socket library can provide applications with a socket API on an operating system that does not provide native sockets (e.g. Windows 3.1). When an application calls one of the socket procedures, control passes to a library routine that makes one or more calls to the underlying OS to implement the socket function.

- A socket may be thought of as a generalization of the BSD Unix file access mechanism (open-read-write-close) that provides an end-point for communication.
- When an application creates a socket, the application is given a small integer descriptor used to reference the socket. If a system uses the same descriptor space for sockets and other I/O, a single application can be used for network communication as well as for local data transfer.
- An application must supply many details for each socket by specifying many parameters and options (e.g. an application must choose a particular protocol, provide address of remote machine, specify whether it is a client or server, etc.)
- To avoid having a single socket function with separate parameters for each options, designers of the socket API chose to define many functions, each with a few parameters.

Socket system calls for connection-oriented protocol



Socket system calls for connectionless protocol



- Data communication between two hosts on the Internet require the five components of what is called an **association** to be initialized: {*protocol,local-addr, local-process, foreign-addr, foreign-process*}
- The different system calls for sockets provides values for one or more of these components.

Socket system call

- The first system call any process wishing to do network I/O has to call is the **socket** system call.
  - int *listenfd* = **socket** (int *family*, int *type*, int *protocol*)
  - Examples of Family include:
    - PF_UNIX
    - **PF_INET**
  - Examples of Type include
    - **SOCK_STREAM**
    - **SOCK_DGRAM**
    - SOCK_RAW
  - The protocol argument is typically zero, but may be specified to request an actual protocol like UDP, TCP, ICMP, etc.
  - Ideally, the three parameters should be orthogonal, but in reality, not all combinations are meaningful.

- The socket system call just fills in one element of the five-tuple we've looked at - the protocol. The remaining are filled in by the other calls as shown in the figure.

	protocol	*local_addr, local_process*	*foreign_addr, foreign_process*
Connection-Oriented Server	socket()	bind()	accept()
Connection-oriented Client	socket()	connect()	
Connectionless Server	socket()	bind()	recvfrom()
Connectionless Client	socket()	bind()	sendto()

Bind System Call

- The bind system call assigns an address to an unnamed socket. Example
  - int **bind**(int *listenfd*, struct sockaddr_in **myaddr*, int *addrlen*)
  - What is ***bind*** used for ?
    - *Servers* (both connection oriented and connectionless) NEED to register their well-known address to be able to accept connection requests.
    - A *client* can register a specific address for itself.
    - A *connectionless client* NEEDS to assure that it is bound to some unique address, so that the server has a valid return address to send its responses to.

- The ***bind*** system call provides the values for the *local_addr* and *local_process* elements in the five_tuple in an association.
- An address for the Internet domain sockets is a combination of a hostname and a port number, as shown below:
  - struct sockaddr_in {
    - short sin_family ; /*typically AF_INET*/
    - u_short sin_port; /* 16 bit port number, ***network byte ordered*** */
    - struct in_addr sin_addr ; /* 32 bit netid/hostid, ***network byte ordered*** */
    - char sin_zero[8]; /* unused*/
    - }

Connect/Listen/Accept System Calls

- Connect
  - A **client** process ***connect***s a socket descriptor after a ***socket*** system call to establish a connection with the server.
  - int **connect**(int *listenfd*, struct sockaddr_in **servaddr*, int *addrlen*)
  - For a *connection-oriented client*, the **connect** (along with an **accept** at the server side) assigns all four addresses and process components of the association.

- ***Listen***
  - The listen system call is used by a connection-oriented server to indicate it is willing to receive connections, e.g., **listen**(*listenfd*, *qlength)*, where the system will enqueue up to qlength requests for connections.
- ***Accept***
  - After the *server* executes a *listen*, it waits for connection requests from client(s) in the ***accept*** system call, e.g., *connfd* = **accept**(*listenfd*, *peer*, *addrlen*)
    - **accept** returns a new socket descriptor, which has all five components of the association specified - three (*protocol*, *local addr*, *local_process*) are inherited from the existing *listenfd* (which has its foreign address and process components unspecified, and hence can be re-used to accept another request. This scenario is typical for concurrent servers.)

Sending and Receiving Data

- Here’s how you might read from a socket:
  - num_read = **read**(listenfd, buff_ptr, num_bytes)
- And here’s how you read from an open file descriptor in Unix:
  - num_read = **read**(fildes, buff_ptr, numbytes)
- There are other ways (with different parameters) to send and receive data: read, readv, recv, recvfrom, recvmsg to receive data through a socket; and write, writev, send, sendto, sendmsg to send data through a socket.

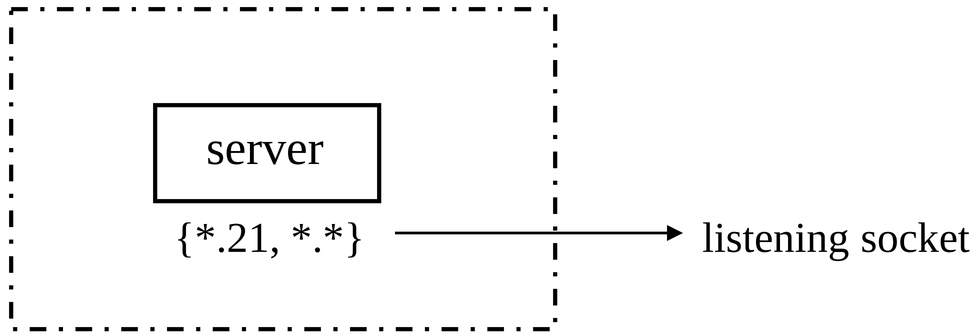


Figure 1. TCP concurrent server with a passive open on port 21.

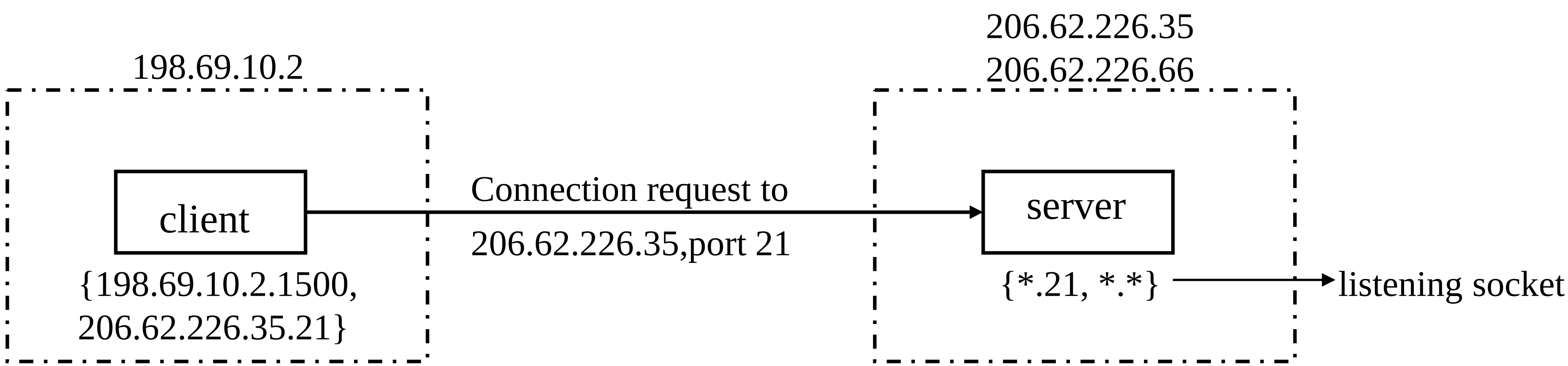


Figure 2. Connection request from client to concurrent server.

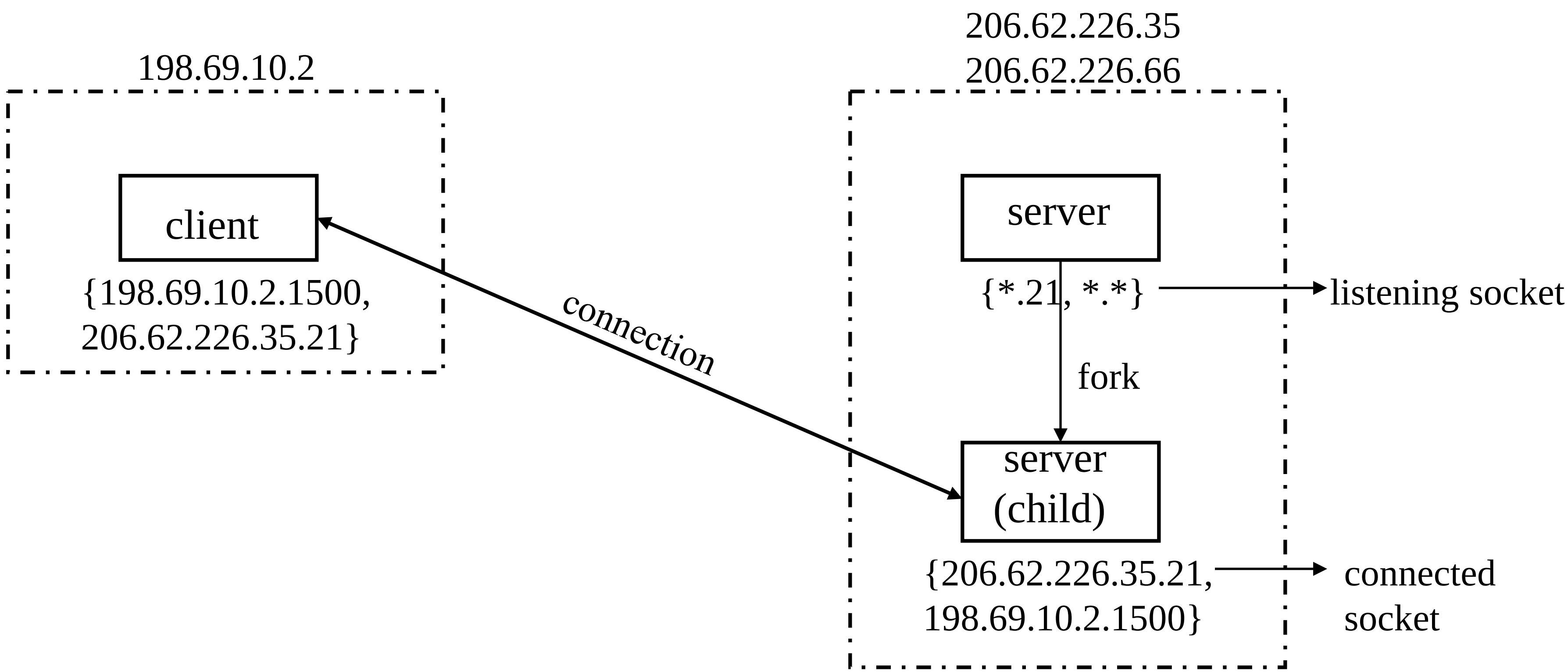


Figure 3. Concurrent server has child handle client.

Connection-oriented Concurrent Server

- int listenfd, connfd;

- if ((listenfd = socket(. . .)) < 0)
-     err_sys("socket error");
- if(bind(listenfd, . . .) < 0)
-     err_sys("bind error");
- if(listen(listenfd, 5) < 0)
-     err_sys("listen error");

- for (; ;) {
-     connfd = accept(listenfd, . . .); /*blocks */
-     if (connfd < 0)
-         err_sys("accept error");
-     if (fork() == 0) {
-         close(listenfd);   /* child */
-         doit(connfd);
-         close(connfd);
-         exit(0);
-     }
-     close(connfd);   /* parent */
- }

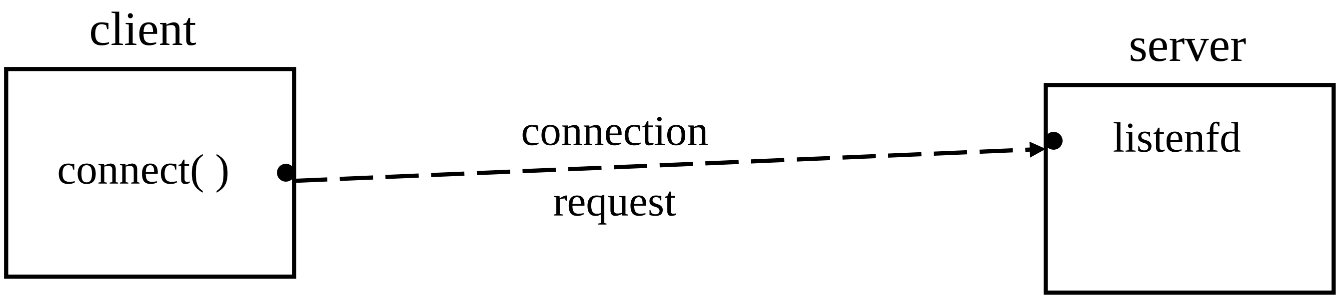


Figure 4. Status of client-server before call to *accept*.

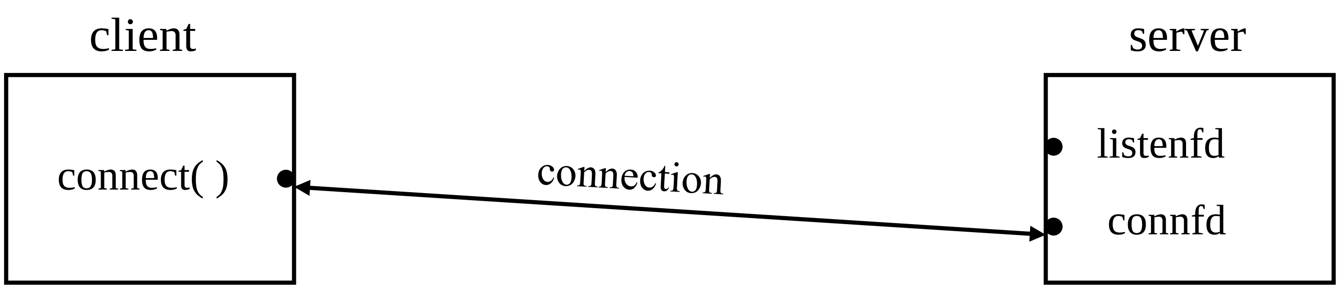


Figure 5. Status of client-server after return from *accept*.

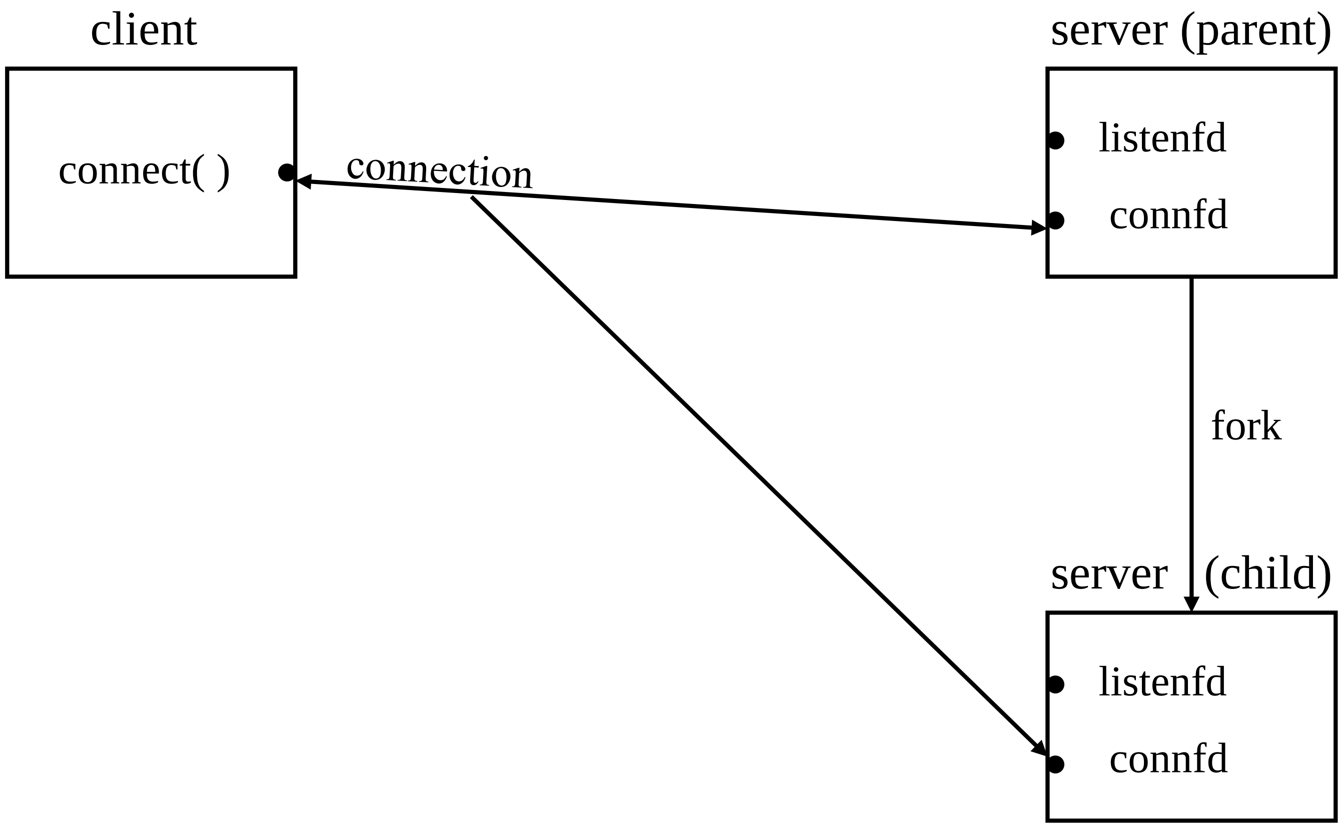


Figure 6. Status of client-server after *fork* returns.

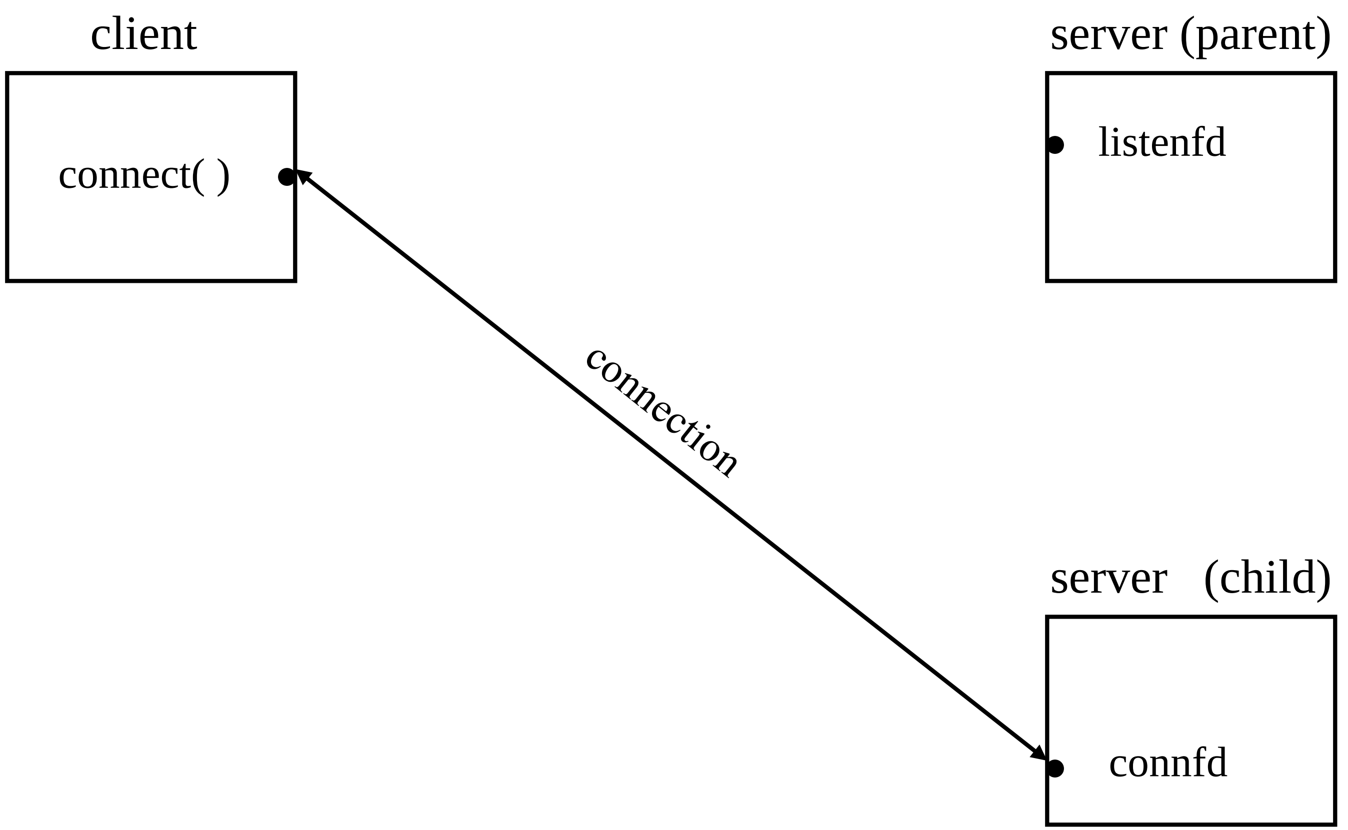


Figure 7. Status of client-server after parent and child *close* appropriate sockets.