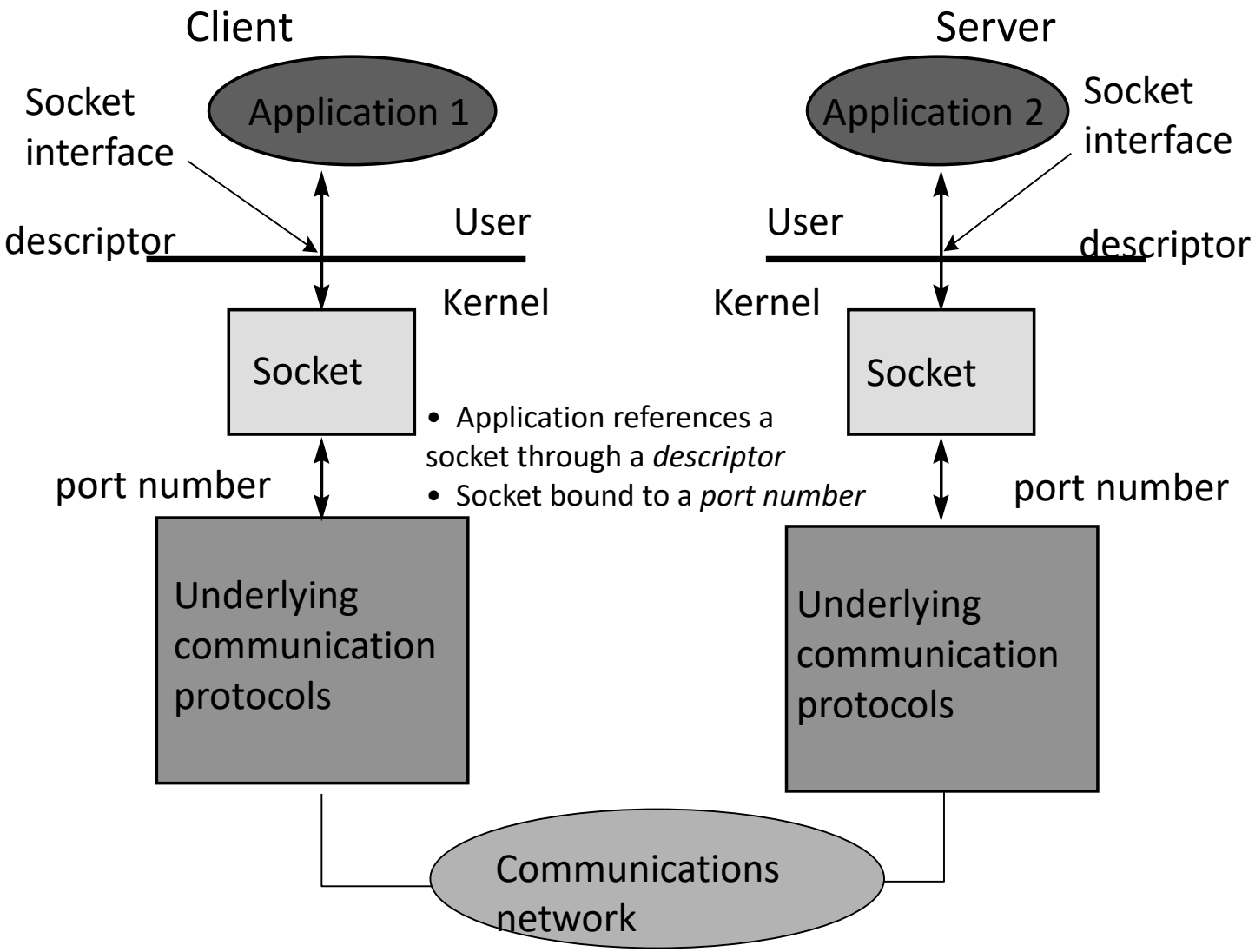
Lecture 9
Socket Programming

Socket API

- API (Application Programming Interface)
  - Provides a standard set of functions that can be called by applications
- Berkeley UNIX Sockets API
  - Abstraction for applications to send & receive data
  - Applications create sockets that “plug into” network
  - Applications write/read to/from sockets
  - Implemented in the kernel
  - Facilitates development of network applications
  - Hides details of underlying protocols & mechanisms
- Also in Windows, Linux, and other OS’s

Communications through Socket Interface



Stream mode of service

Connection-oriented

- First, setup connection between two peer application processes
- Then, reliable bidirectional in-sequence transfer of *byte stream* (boundaries not preserved in transfer)
- Multiple write/read between peer processes
- Finally, connection release
- Uses TCP

- Connectionless
- Immediate transfer of one block of information (boundaries preserved)
- No setup overhead & delay
- Destination address with each block
- Send/receive to/from multiple peer processes
- Best-effort service only
  - Possible out-of-order
  - Possible loss
- Uses UDP

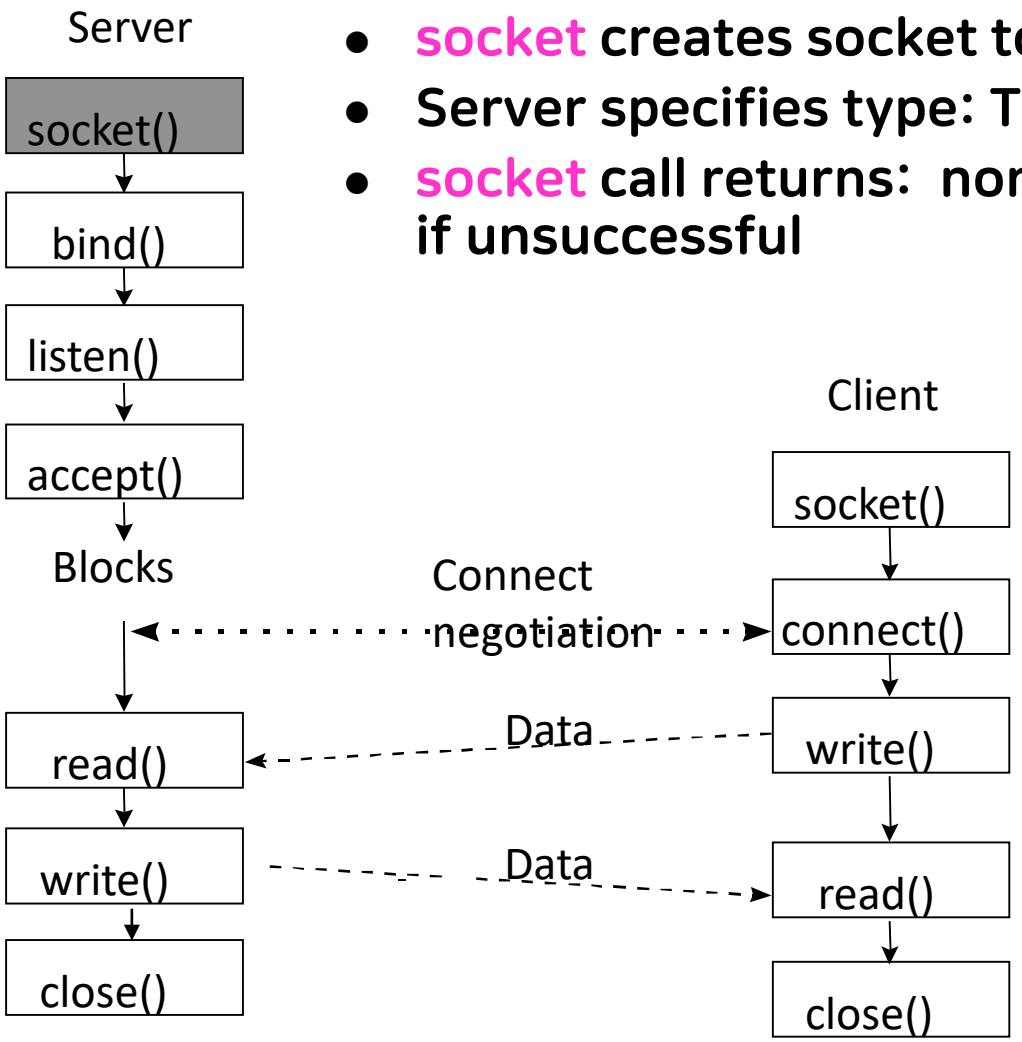
Client & Server Differences

- Server
  - Specifies well-known port # when creating socket
  - May have multiple IP addresses (net interfaces)
  - Waits passively for client requests
- Client
  - Assigned ephemeral port #
  - Initiates communications with server
  - Needs to know server's IP address & port #
    - DNS for URL & server well-known port #
  - Server learns client's address & port #

Socket Calls for Connection-Oriented Mode

Server does Passive Open

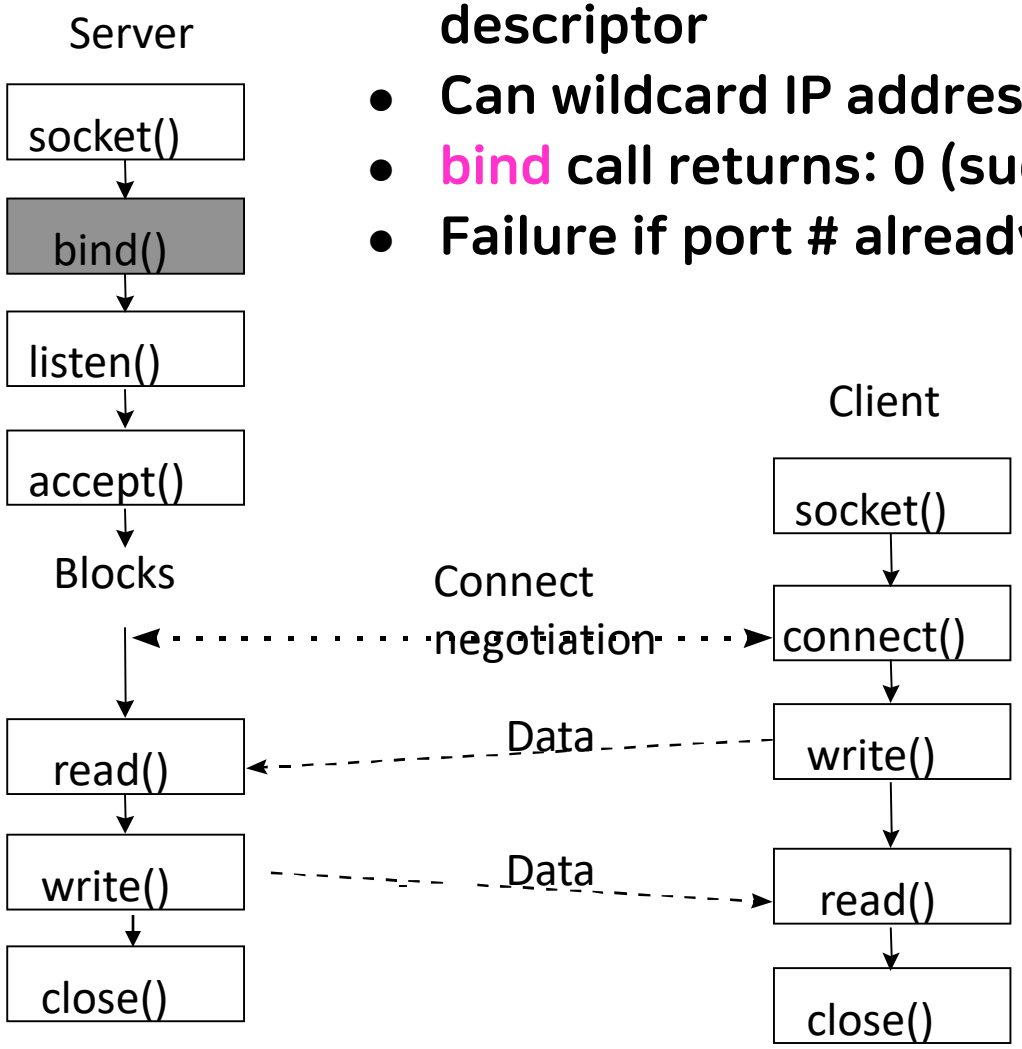
- **socket creates socket to *listen* for connection requests**
- **Server specifies type: TCP (stream)**
- **socket call returns: non-negative integer *descriptor*; or -1 if unsuccessful**



Socket Calls for Connection-Oriented Mode

Server does Passive Open

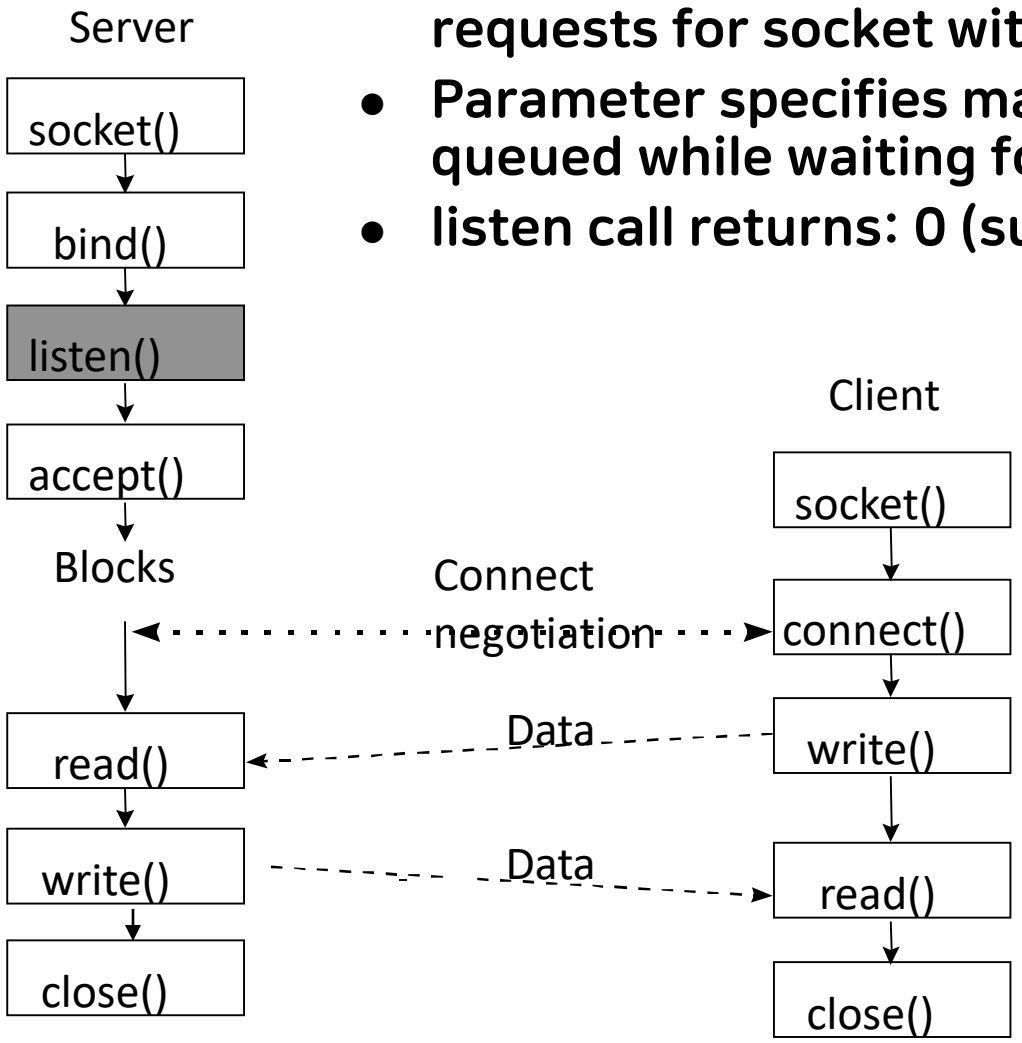
- **bind assigns local address & port # to socket with specified descriptor**
- **Can wildcard IP address for multiple net interfaces**
- **bind call returns: 0 (success); or -1 (failure)**
- **Failure if port # already in use or if reuse option not set**



Socket Calls for Connection-Oriented Mode

Server does Passive Open

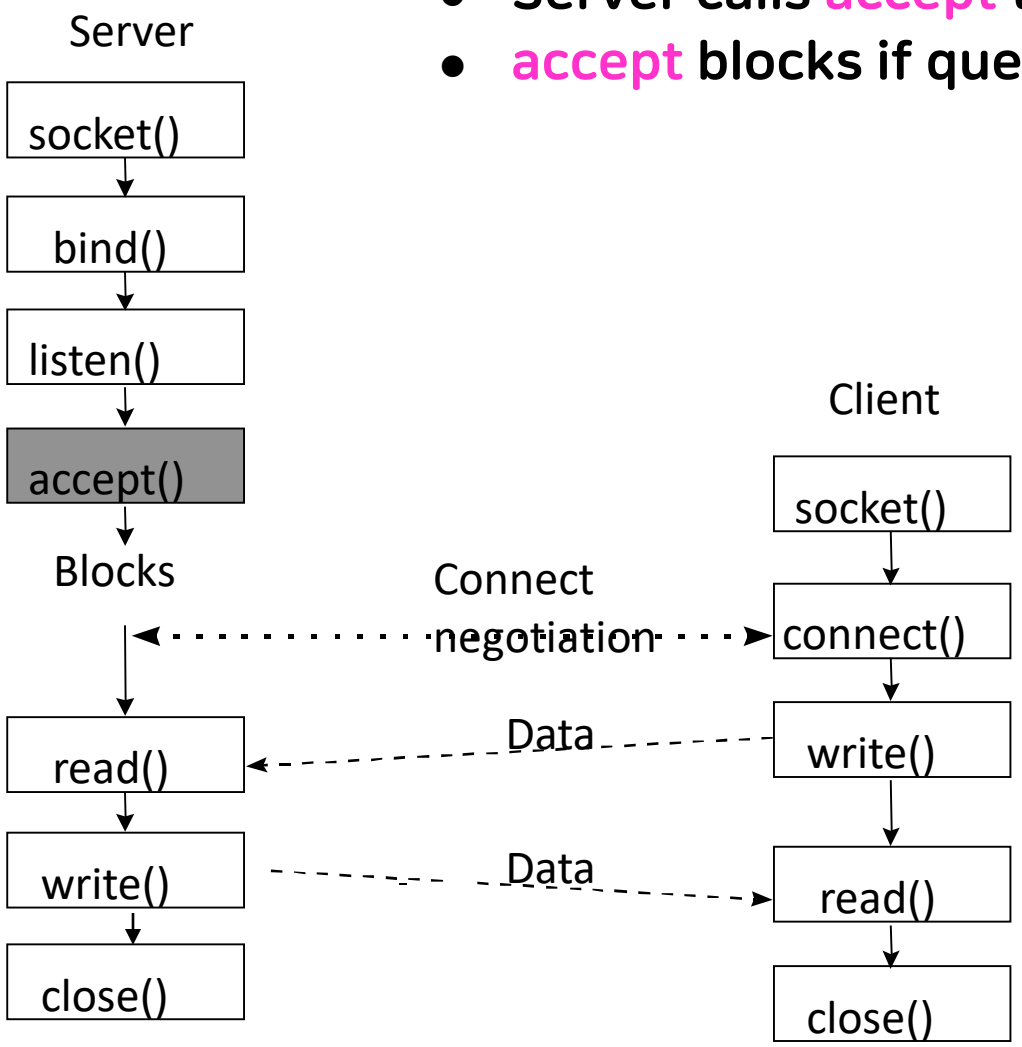
- listen indicates to TCP readiness to receive connection requests for socket with given descriptor
- Parameter specifies max number of requests that may be queued while waiting for server to accept them
- listen call returns: 0 (success); or -1 (failure)



Socket Calls for Connection-Oriented Mode

Server does Passive Open

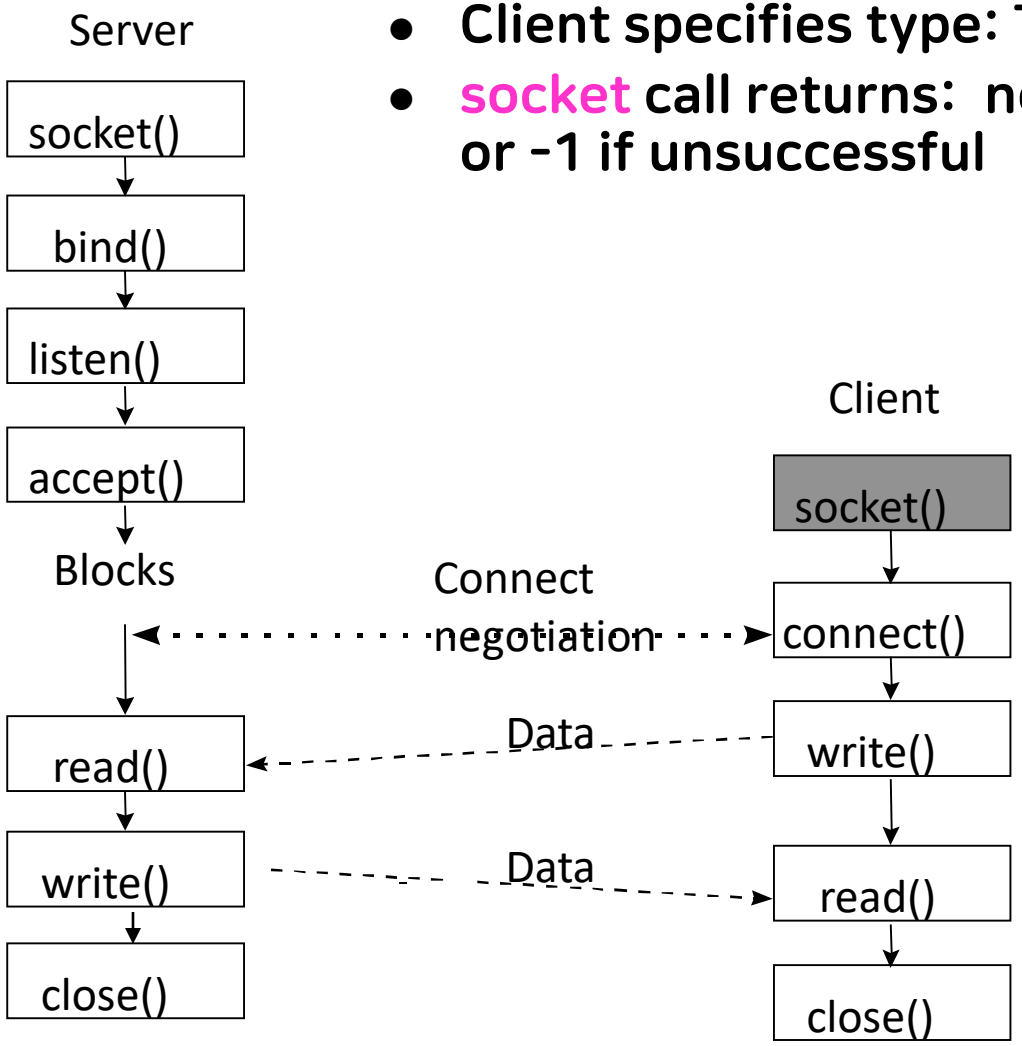
- Server calls accept to accept incoming requests
- accept blocks if queue is empty



Socket Calls for Connection-Oriented Mode

Client does Active Open

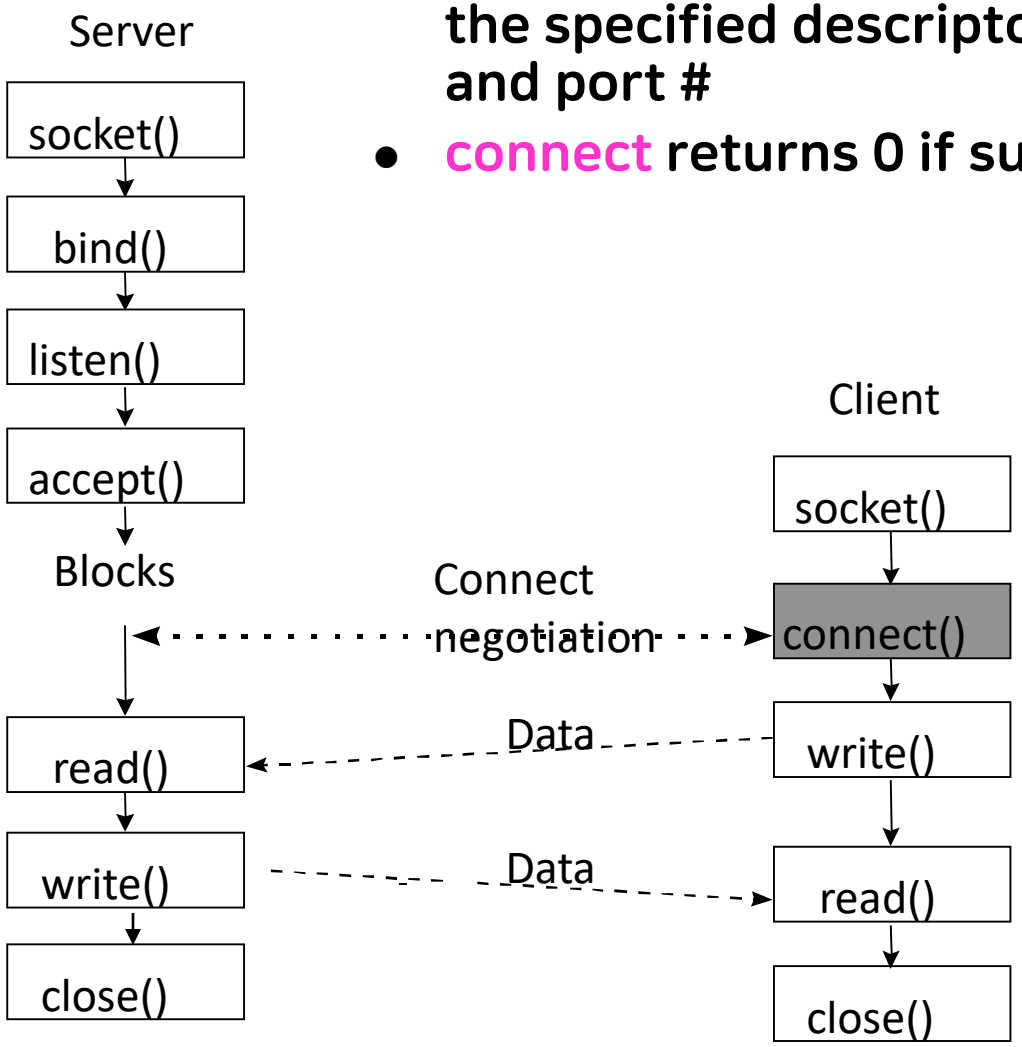
- socket creates socket to connect to server
- Client specifies type: TCP (stream)
- socket call returns: non-negative integer *descriptor*; or -1 if unsuccessful



Socket Calls for Connection-Oriented Mode

Client does Active Open

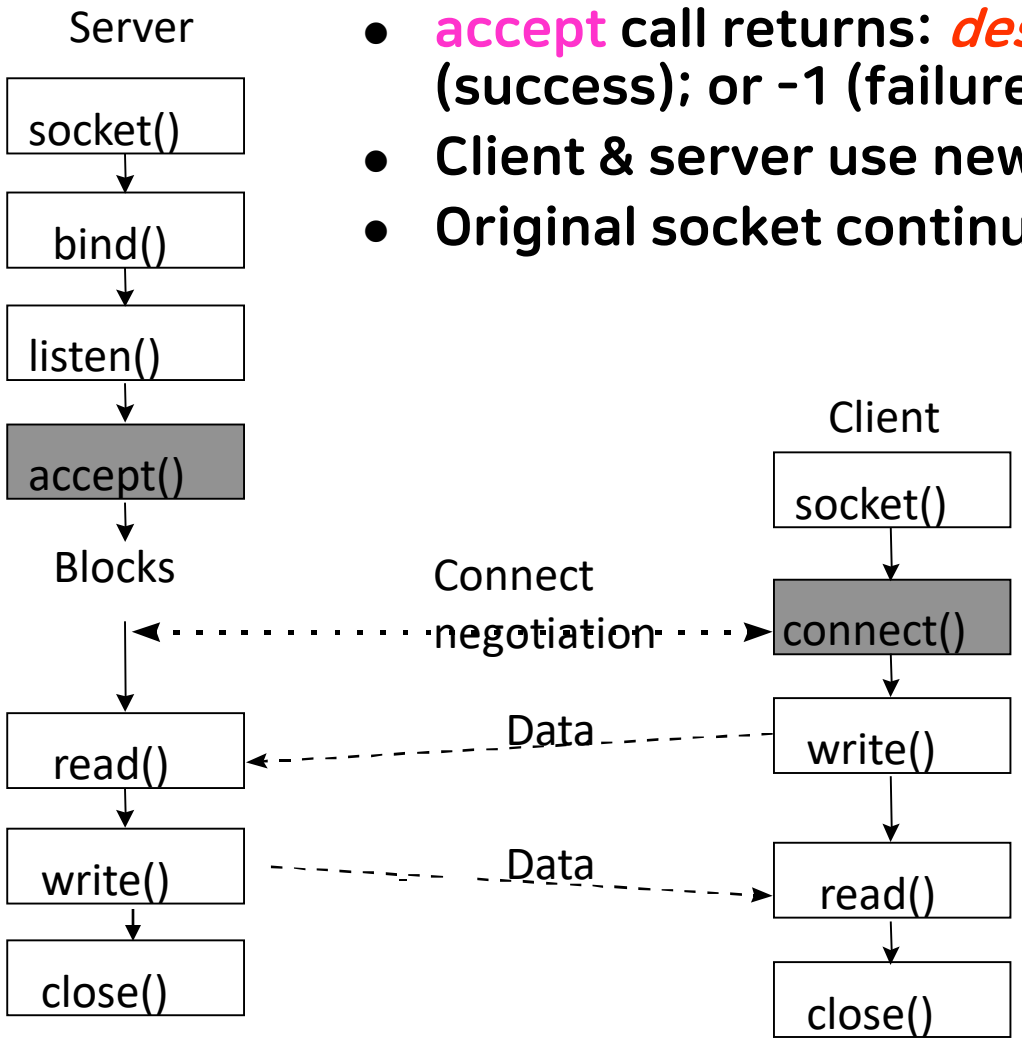
- **connect establishes a connection on the local socket with the specified descriptor to the specified remote address and port #**
- **connect returns 0 if successful; -1 if unsuccessful**



Note: `connect` initiates TCP three-way handshake

Socket Calls for Connection-Oriented Mode

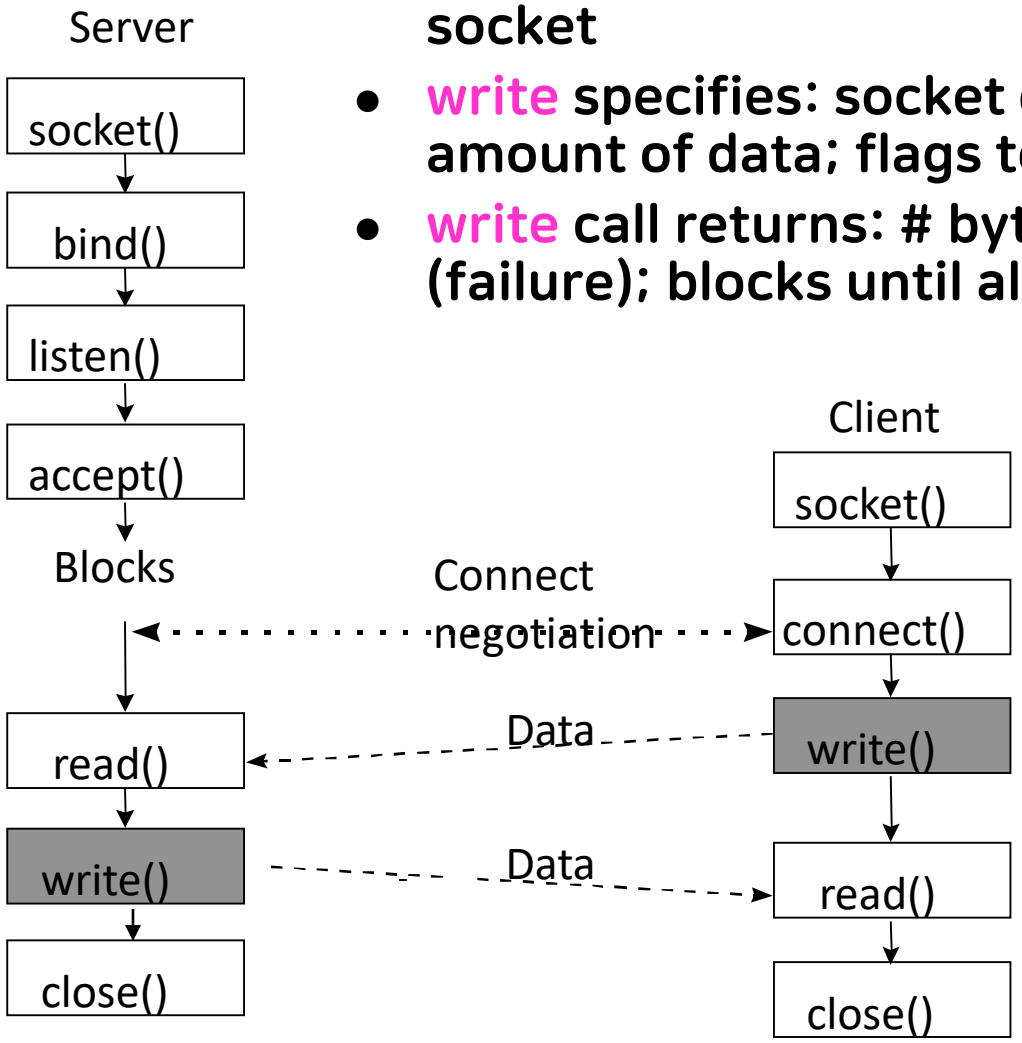
- accept wakes with incoming connection request
- accept fills client address & port # into address structure
- accept call returns: *descriptor of new connection socket* (success); or -1 (failure)
- Client & server use new socket for data transfer
- Original socket continues to listen for new requests



Socket Calls for Connection-Oriented Mode

Data Transfer

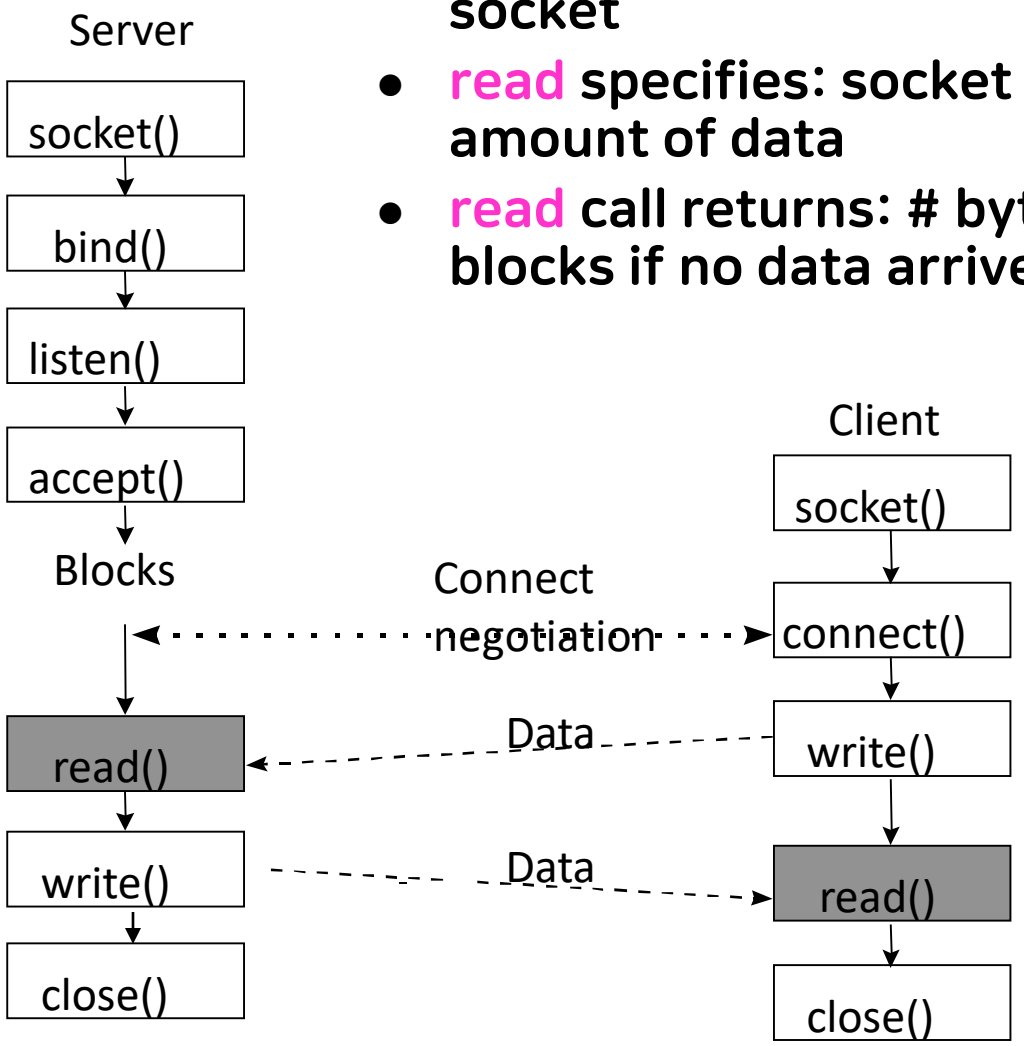
- Client or server call write to transmit data into a connected socket
- write specifies: socket descriptor; pointer to a buffer; amount of data; flags to control transmission behavior
- write call returns: # bytes transferred (success); or -1 (failure); blocks until all data transferred



Socket Calls for Connection-Oriented Mode

Data Transfer

- Client or server call read to receive data from a connected socket
- read specifies: socket descriptor; pointer to a buffer; amount of data
- read call returns: # bytes read (success); or -1 (failure); blocks if no data arrives

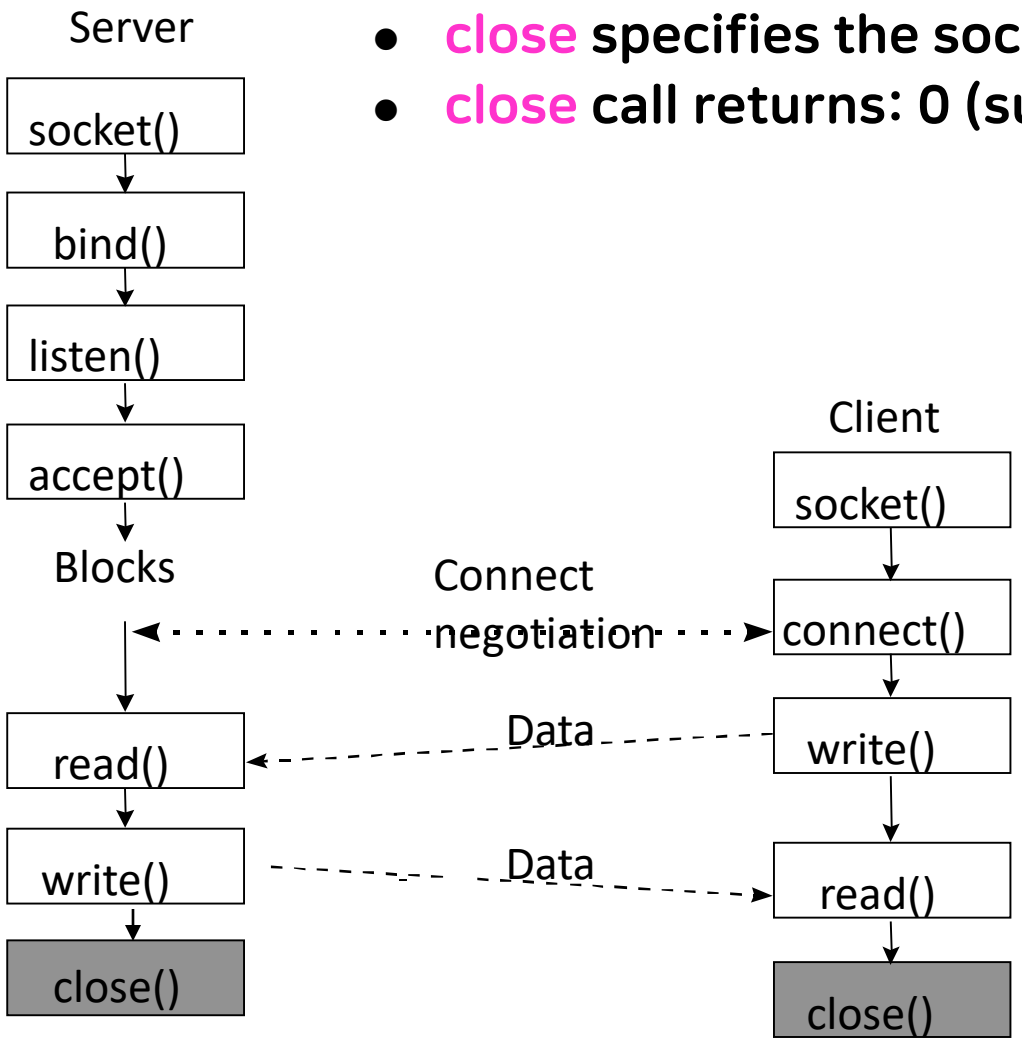


Note: `write` and `read` can be called multiple times to transfer byte streams in both directions

Socket Calls for Connection-Oriented Mode

Connection Termination

- Client or server call close when socket is no longer needed
- close specifies the socket descriptor
- close call returns: 0 (success); or -1 (failure)



Note: `close` initiates TCP graceful close sequence

Example: TCP Echo Server

```
/* A simple echo server using TCP */
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

#define SERVER_TCP_PORT           3000
#define BUFLEN                                  256

int main(int argc, char **argv)
{
      int   n, bytes_to_read;
      int   sd, new_sd, client_len, port;
      struct sockaddr_in server, client;
      char  *bp, buf[BUFLEN];

      switch(argc) {
      case 1:
            port = SERVER_TCP_PORT;
            break;
      case 2:
            port = atoi(argv[1]);
            break;
      default:
            fprintf(stderr, "Usage: %s [port]\n", argv[0]);
            exit(1);
      }

      /* Create a stream socket */
      if ((sd = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
            fprintf(stderr, "Can't create a socket\n");
            exit(1);
      }
```

```
      /* Bind an address to the socket */
      bzero((char *)&server, sizeof(struct sockaddr_in));
      server.sin_family = AF_INET;
      server.sin_port = htons(port);
      server.sin_addr.s_addr = htonl(INADDR_ANY);
      if (bind(sd, (struct sockaddr *)&server,
      sizeof(server)) == -1) {
            fprintf(stderr, "Can't bind name to socket\n");
            exit(1);
      }

      /* queue up to 5 connect requests */
      listen(sd, 5);

      while (1) {
            client_len = sizeof(client);
            if ((new_sd = accept(sd, (struct sockaddr *)&client,
            &client_len)) == -1) {
                  fprintf(stderr, "Can't accept client\n");
                  exit(1);
            }

            bp = buf;
            bytes_to_read = BUFLEN;
            while ((n = read(new_sd, bp, bytes_to_read)) > 0) {
                  bp += n;
                  bytes_to_read -= n;
            }
            printf("Rec'd: %s\n", buf);

            write(new_sd, buf, BUFLEN);
            printf("Sent: %s\n", buf);
            close(new_sd);
      }
      close(sd);
      return(0);
}
```

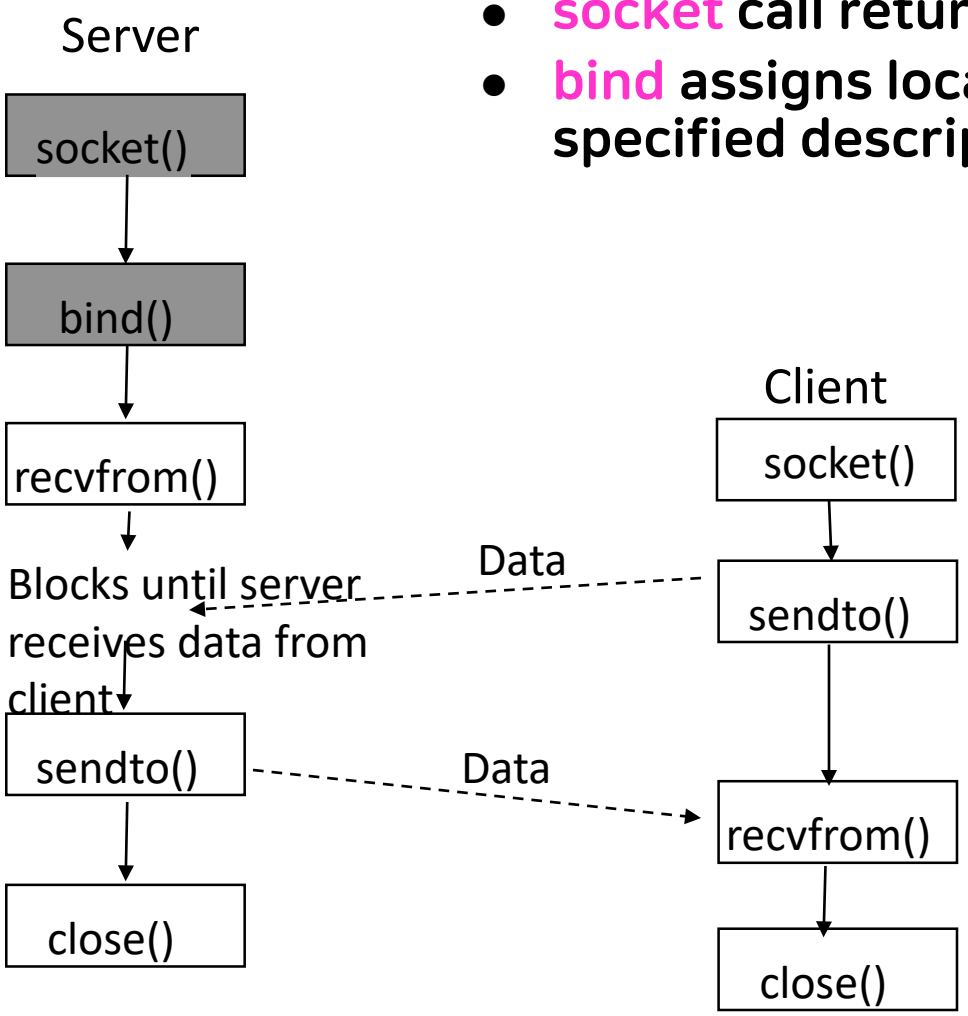
Example: TCP Echo Client

```
/* A simple TCP client */
#include <stdio.h>
#include <netdb.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

#define SERVER_TCP_PORT           3000
#define BUFLEN                                  256

int main(int argc, char **argv)
{
      int   n, bytes_to_read;
      int   sd, port;
      struct hostent *hp;
      struct sockaddr_in server;
      char  *host, *bp, rbuf[BUFLEN], sbuf[BUFLEN];

      switch(argc) {
      case 2:
            host = argv[1];
            port = SERVER_TCP_PORT;
            break;
      case 3:
            host = argv[1];
            port = atoi(argv[2]);
            break;
      default:
            fprintf(stderr, "Usage: %s host [port]\n", argv[0]);
            exit(1);
      }

      /* Create a stream socket */
      if ((sd = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
            fprintf(stderr, "Can't create a socket\n");
            exit(1);
      }
      bzero((char *)&server, sizeof(struct sockaddr_in));
      server.sin_family = AF_INET;
      server.sin_port = htons(port);
      if ((hp = gethostbyname(host)) == NULL) {
            fprintf(stderr, "Can't get server's address\n");
            exit(1);
      }
      bcopy(hp->h_addr, (char *)&server.sin_addr, hp->h_length);

      /* Connecting to the server */
      if (connect(sd, (struct sockaddr *)&server,
      sizeof(server)) == -1) {
            fprintf(stderr, "Can't connect\n");
            exit(1);
      }
      printf("Connected: server's address is %s\n", hp->h_name);

      printf("Transmit:\n");
      gets(sbuf);                                write(sd, sbuf, BUFLEN);

      printf("Receive:\n");
      bp = rbuf;
      bytes_to_read = BUFLEN;
      while ((n = read(sd, bp, bytes_to_read)) > 0) {
            bp += n;
            bytes_to_read -= n;
      }
      printf("%s\n", rbuf);

      close(sd);
      return(0);
}
```

Socket Calls for Connection-Less Mode

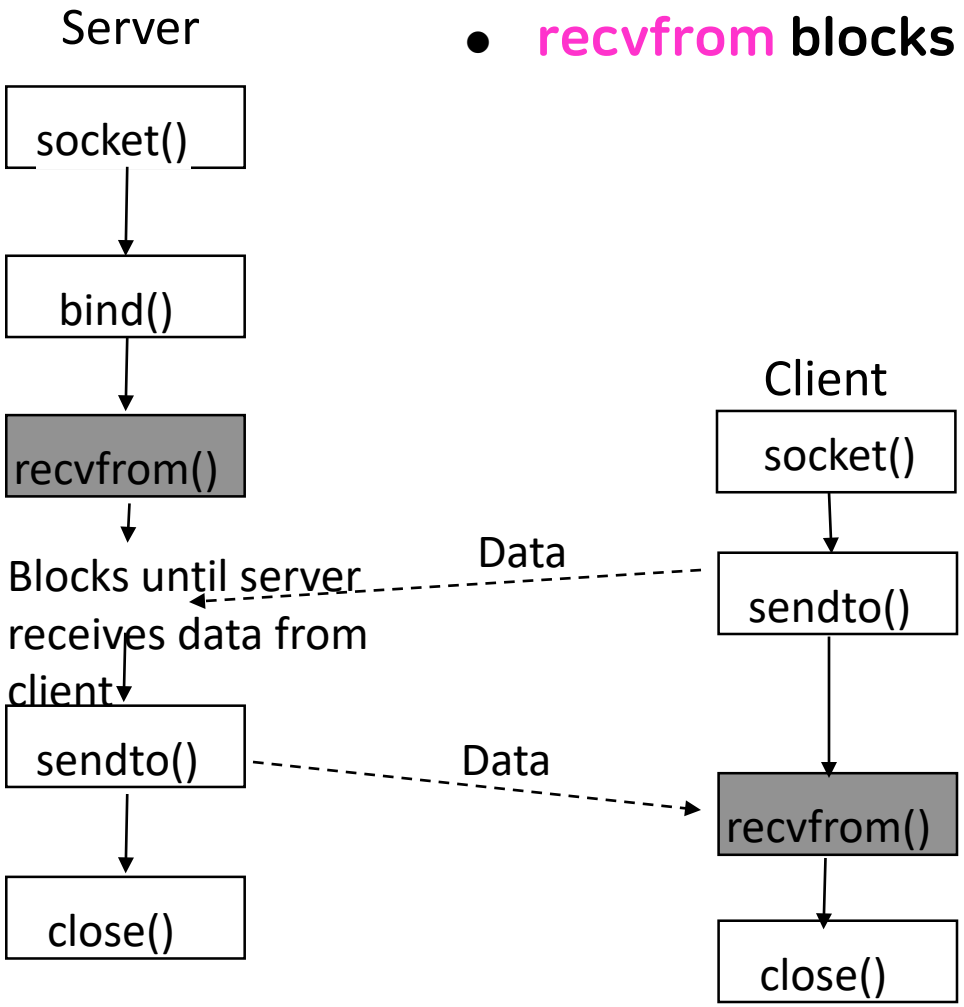
Server started

- socket creates socket of type UDP (datagram)
- socket call returns: *descriptor*; or -1 if unsuccessful
- bind assigns local address & port # to socket with specified descriptor; Can wildcard IP address



Socket Calls for Connection-Less Mode

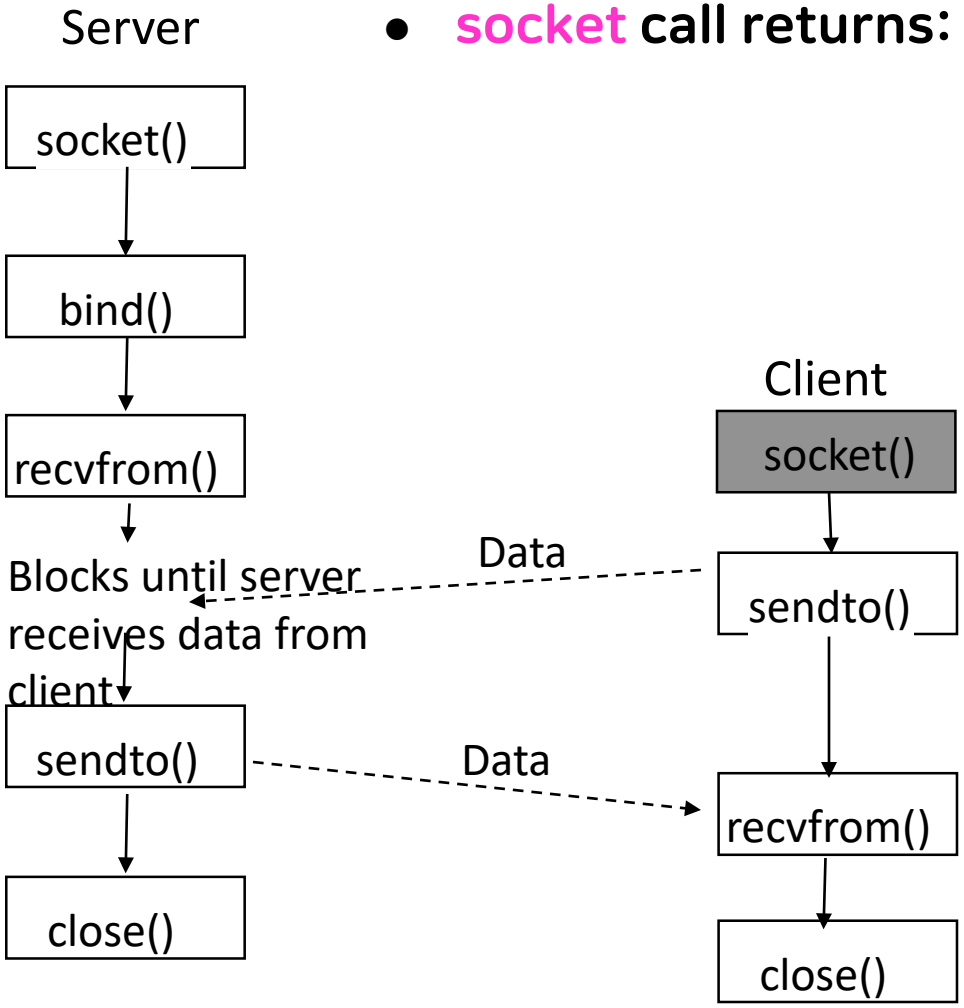
- recvfrom copies bytes received in specified socket into a specified location
- recvfrom blocks until data arrives



Socket Calls for Connection-Less Mode

Client started

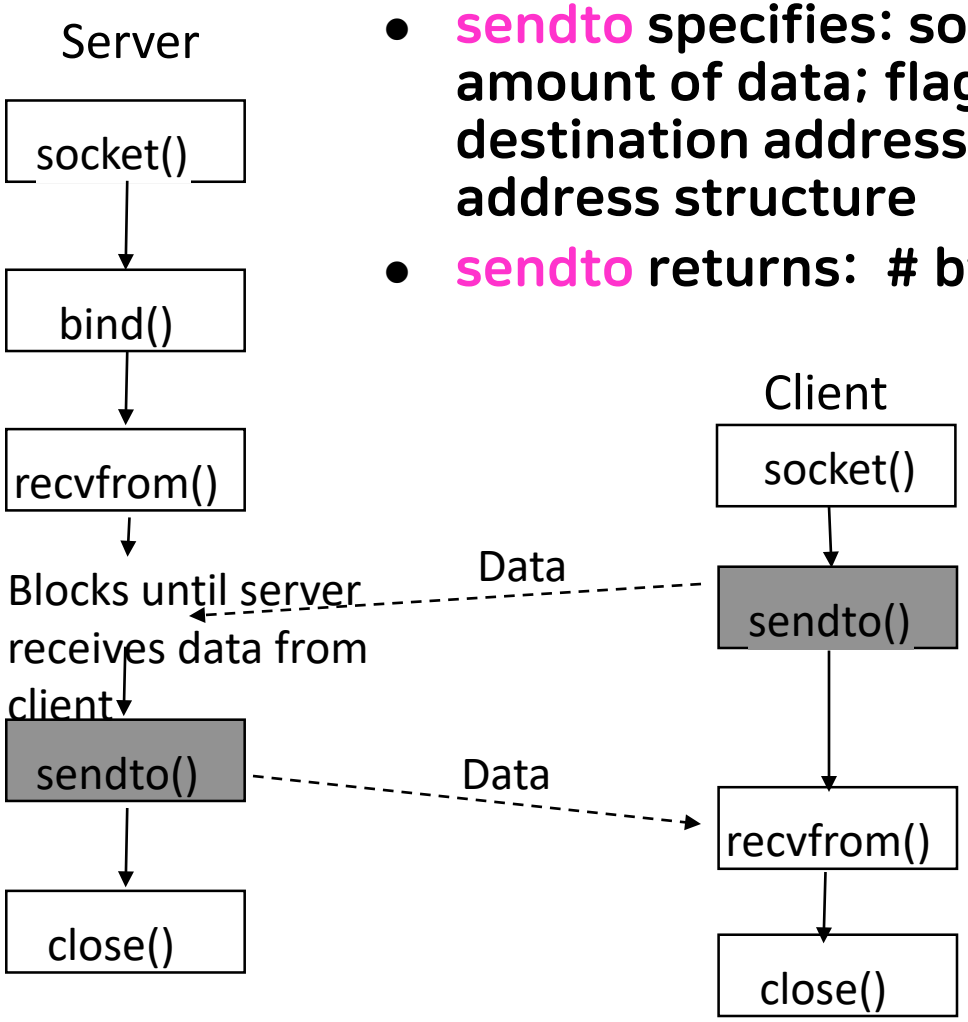
- **socket creates socket of type UDP (datagram)**
- **socket call returns: *descriptor*; or -1 if unsuccessful**



Socket Calls for Connection-Less Mode

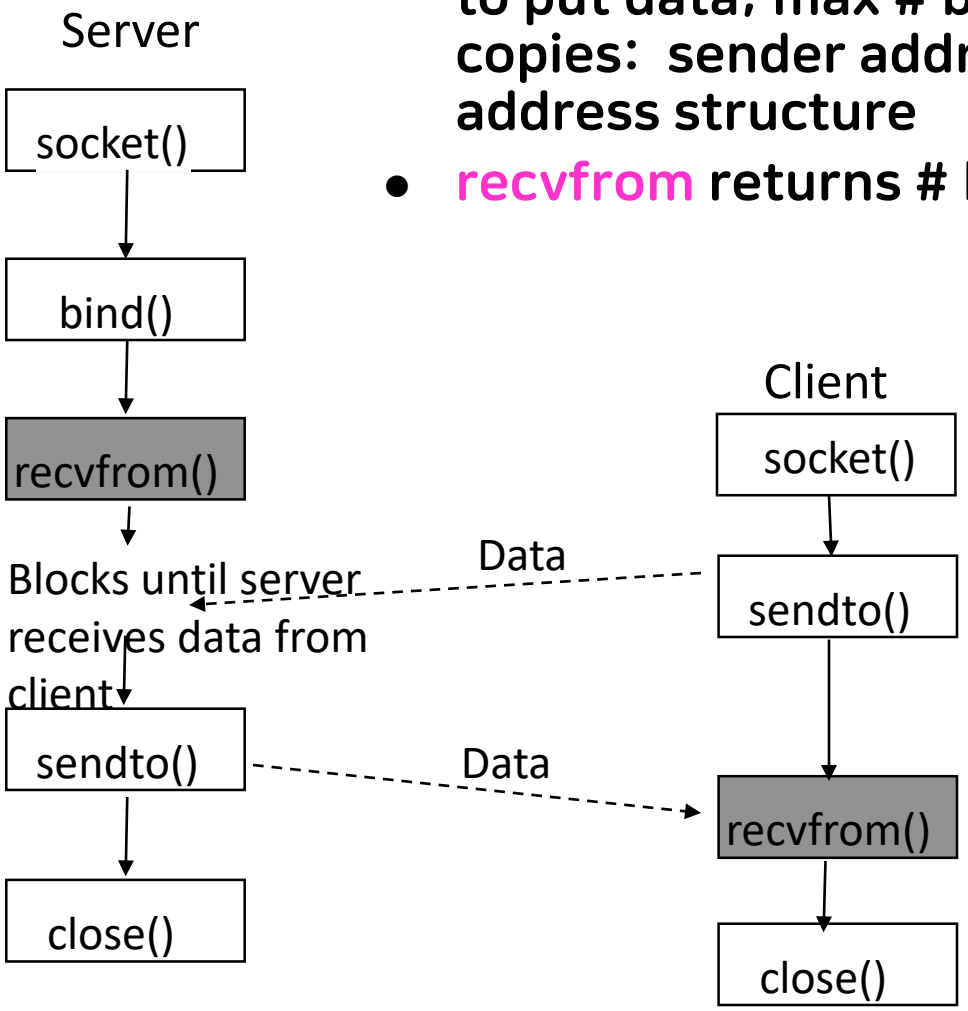
Client started

- **sendto transfer bytes in buffer to specified socket**
- **sendto specifies: socket descriptor; pointer to a buffer; amount of data; flags to control transmission behavior; destination address & port #; length of destination address structure**
- **sendto returns: # bytes sent; or -1 if unsuccessful**



Socket Calls for Connection-Less Mode

- recvfrom wakes when data arrives
- recvfrom specifies: socket descriptor; pointer to a buffer to put data; max # bytes to put in buffer; control flags; copies: sender address & port #; length of sender address structure
- recvfrom returns # bytes received or -1 (failure)

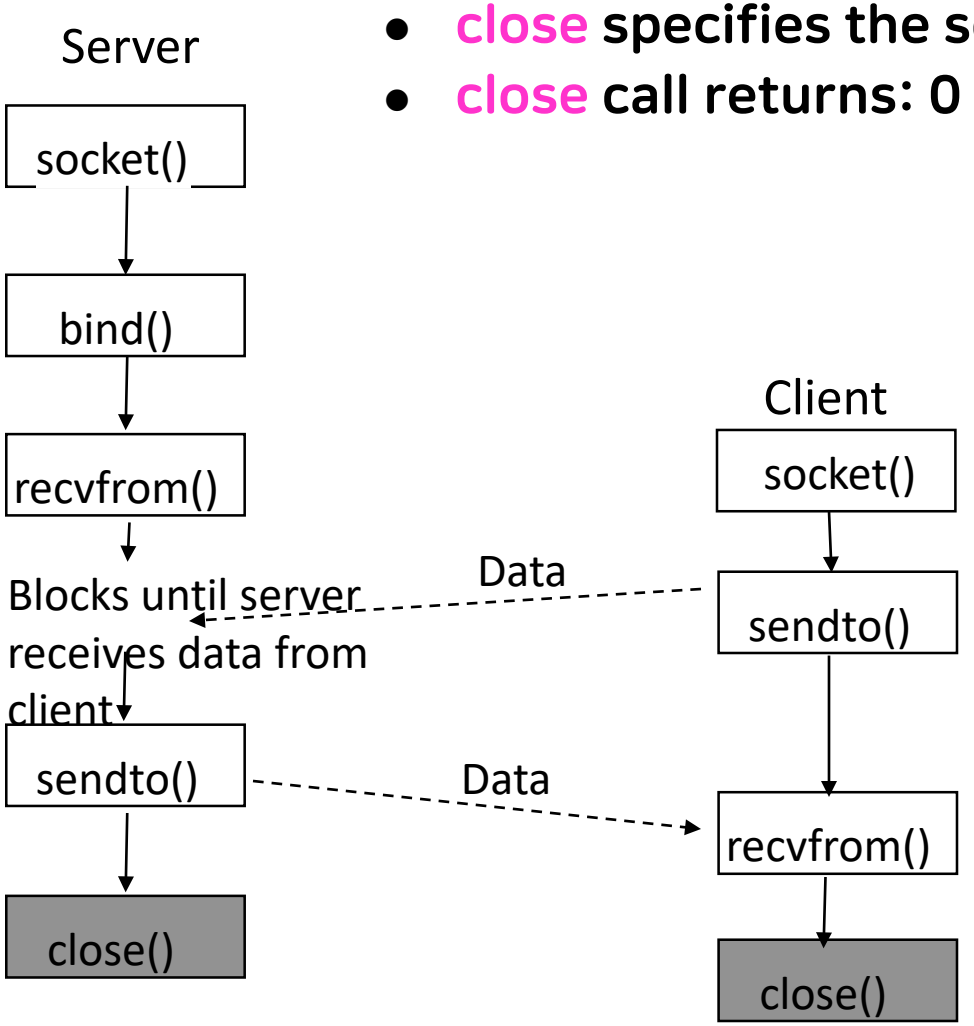


Note: receivefrom returns data from at most one send, i.e. from one datagram

Socket Calls for Connection-Less Mode

Socket Close

- Client or server call close when socket is no longer needed
- close specifies the socket descriptor
- close call returns: 0 (success); or -1 (failure)



Example: UDP Echo Server

```
/* Echo server using UDP */
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

#define SERVER_UDP_PORT                 5000
#define MAXLEN                          4096

int main(int argc, char **argv)
{
     int  sd, client_len, port, n;
     char           buf[MAXLEN];
     struct         sockaddr_in         server, client;

     switch(argc) {
     case 1:
          port = SERVER_UDP_PORT;
          break;
     case 2:
          port = atoi(argv[1]);
          break;
     default:
          fprintf(stderr, "Usage: %s [port]\n", argv[0]);
          exit(1);
     }

     /* Create a datagram socket */
     if ((sd = socket(AF_INET, SOCK_DGRAM, 0)) == -1) {
          fprintf(stderr, "Can't create a socket\n");
          exit(1);
     }

     /* Bind an address to the socket */
     bzero((char *)&server, sizeof(server));
     server.sin_family = AF_INET;
     server.sin_port = htons(port);
     server.sin_addr.s_addr = htonl(INADDR_ANY);
     if (bind(sd, (struct sockaddr *)&server,
     sizeof(server)) == -1) {
          fprintf(stderr, "Can't bind name to socket\n");
          exit(1);
     }

     while (1) {
          client_len = sizeof(client);
          if ((n = recvfrom(sd, buf, MAXLEN, 0,
          (struct sockaddr *)&client, &client_len)) < 0)
     {
                fprintf(stderr, "Can't receive
     datagram\n");
                exit(1);
          }

          if (sendto(sd, buf, n, 0,
          (struct sockaddr *)&client, client_len) != n) {
                fprintf(stderr, "Can't send datagram\n");
                exit(1);
          }
     }
     close(sd);
     return(0);
}
```

Example: UDP Echo Client

```
#include <stdio.h>
#include <string.h>
#include <sys/time.h>
#include <netdb.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#define SERVER_UDP_PORT         5000
#define MAXLEN                  4096
#define DEFLEN                  64

long delay(struct timeval t1, struct timeval t2)
{
        long d;
        d = (t2.tv_sec - t1.tv_sec) * 1000;
        d += ((t2.tv_usec - t1.tv_usec + 500) / 1000);
        return(d);
}
int main(int argc, char **argv)
{
        int     data_size = DEFLEN, port = SERVER_UDP_PORT;
        int     i, j, sd, server_len;
        char    *pname, *host, rbuf[MAXLEN], sbuf[MAXLEN];
        struct  hostent         *hp;
        struct  sockaddr_in     server;
        struct  timeval         start, end;
        unsigned long address;

        pname = argv[0];
        argc--;
        argv++;
        if (argc > 0 && (strcmp(*argv, "-s") == 0)) {
                if (--argc > 0 && (data_size = atoi(*++argv))) {
                        argc--;
                        argv++;
                }
                else {
                        fprintf(stderr,
                        "Usage: %s [-s data_size] host [port]\n", pname);
                        exit(1);
                }
        }
        if (argc > 0) {
                host = *argv;
                if (--argc > 0)
                        port = atoi(*++argv);
        }
        else {
                fprintf(stderr,
                "Usage: %s [-s data_size] host [port]\n", pname);
                exit(1);
        }

        if ((sd = socket(AF_INET, SOCK_DGRAM, 0)) == -1) {
                fprintf(stderr, "Can't create a socket\n");
                exit(1);
        }
        bzero((char *)&server, sizeof(server));
        server.sin_family = AF_INET;
        server.sin_port = htons(port);
        if ((hp = gethostbyname(host)) == NULL) {
                fprintf(stderr, "Can't get server's IP address\n");
                exit(1);
        }
        bcopy(hp->h_addr, (char *) &server.sin_addr, hp->h_length);

        if (data_size > MAXLEN) {
                fprintf(stderr, "Data is too big\n");
                exit(1);
        }
        for (i = 0; i < data_size; i++) {
                j = (i < 26) ? i : i % 26;
                sbuf[i] = 'a' + j;
        }
        gettimeofday(&start, NULL); /* start delay measurement */
        server_len = sizeof(server);
        if (sendto(sd, sbuf, data_size, 0, (struct sockaddr *)
                &server, server_len) == -1) {
                fprintf(stderr, "sendto error\n");
                exit(1);
        }
        if (recvfrom(sd, rbuf, MAXLEN, 0, (struct sockaddr *)
                &server, &server_len) < 0) {
                fprintf(stderr, "recvfrom error\n");
                exit(1);
        }
        gettimeofday(&end, NULL); /* end delay measurement */
        if (strncmp(sbuf, rbuf, data_size) != 0)
                printf("Data is corrupted\n");
        close(sd);
        return(0);
}
```