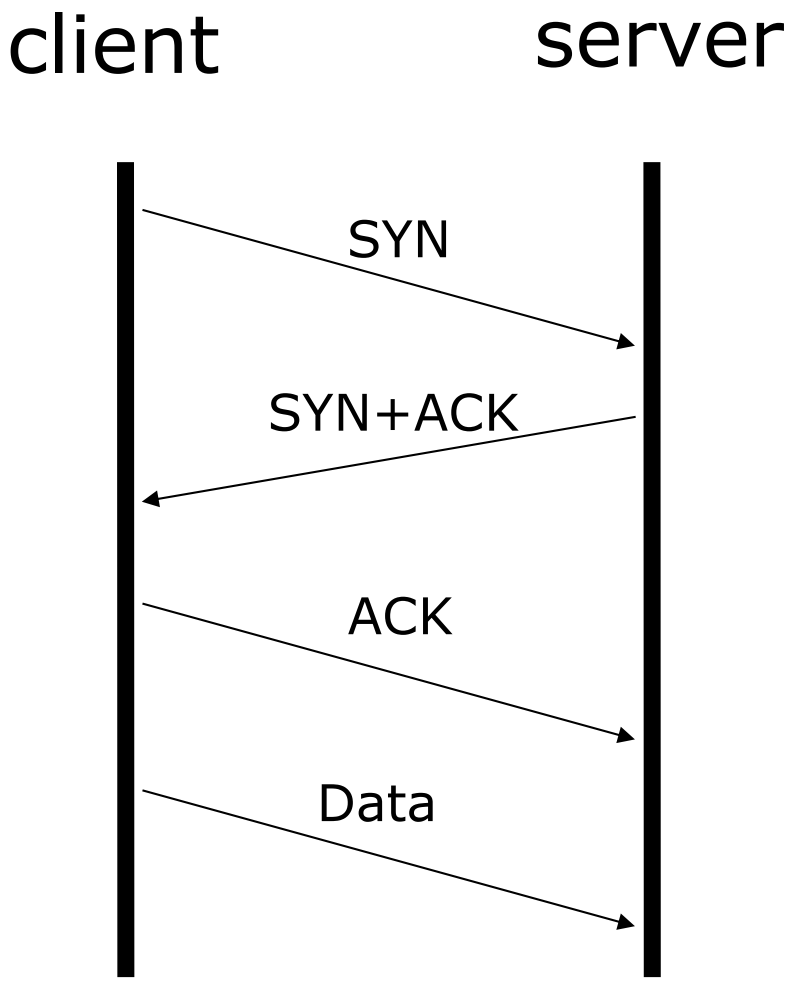
BASIC TCP SOCKET

Contents

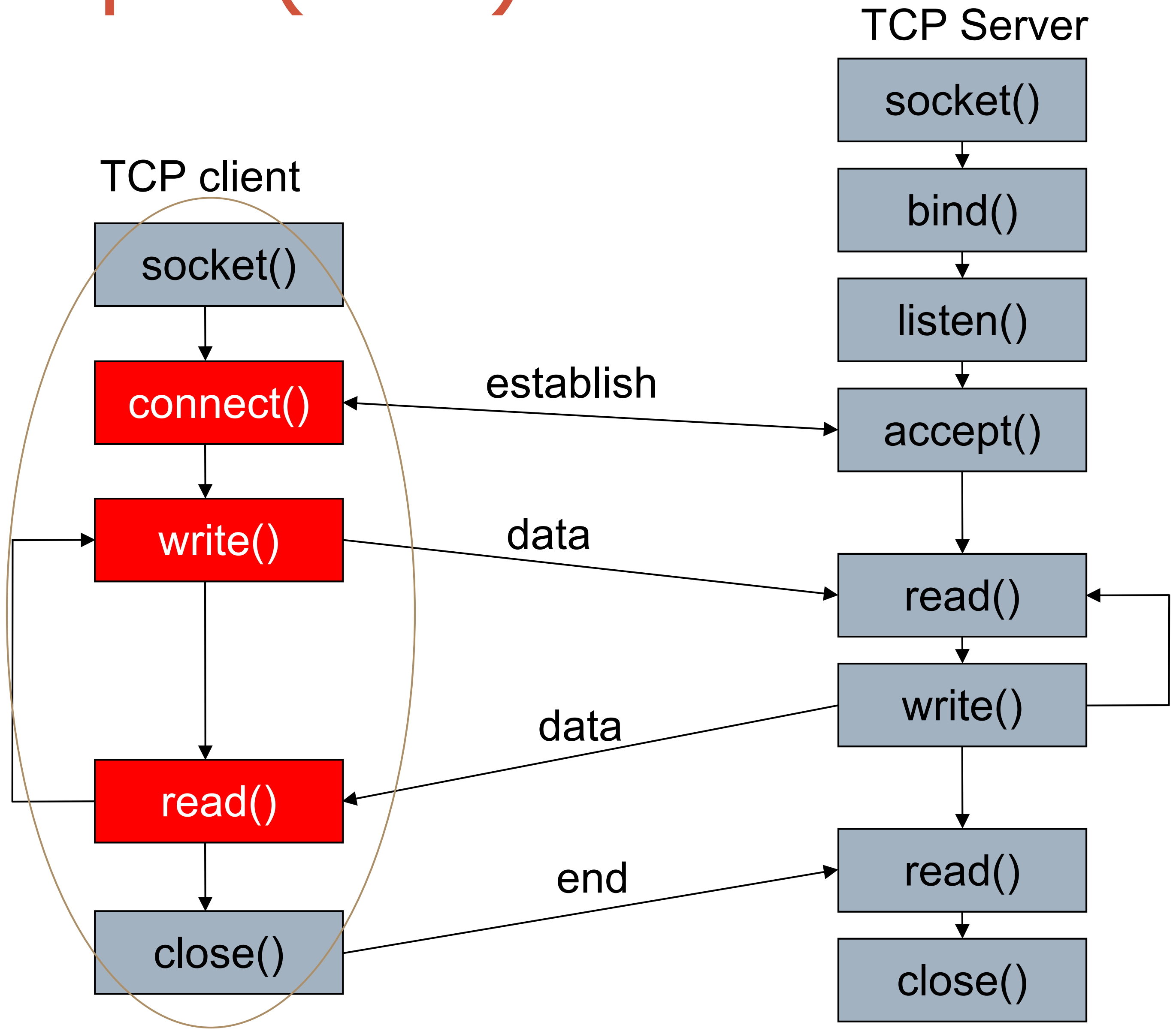
TCP CLIENT

TCP (Transmission Control Protocol)

- Provide reliable communication
- Data rate control
- Example
  - Mail
  - WEB
  - Image



Example (TCP)


TCP Server
socket()
bind()
listen()
accept()
read()
write()
read()
close()
TCP client
socket()
connect()
write()
read()
close()
establish
data
data
end

Algorithm for TCP Client in C

- Find the IP address and port number of TCP server
- Create a TCP socket
  - socket().
- Connect the socket to server, server must be up waiting for client connection
  - connect().
- Send/ receive data with server using the socket
  - recv(), send()
- Close the connection
  - closesocket().
- Why “clients” doesn’t need bind() ?

socket()

```
#include <sys/types.h>
#include <sys/socket.h>
int socket(int domain, int type, int protocol);
```

- Returns value
  - A new socket file descriptor (a socket “handle”) that you can use to read/receive data from/to
  - If error occurs, return -1 (remember **errno**)
- The *domain* is AF_INET, AF_INET6, or AF_UNSPEC, …
- The *type* argument can be:
  - SOCK_STREAM: socket for TCP connection
  - SOCK_DGRAM: Socket for datagram communication (UDP)
  - SOCK_SEQPACKET: Establishes a reliable, connection based, two way communication with maximum message size. (This is not available on most machines.)
- *protocol* is usually zero, means that the protocol is automatically chosen according to communication *type*.

socket() - example

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

int sockfd;

sockfd = socket(AF_INET, SOCK_STREAM, 0);

// create a TCP socket
```

connect()

- Connect a socket to a server

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

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

- *sockfd*
  - A descriptor identifying an unconnected socket.
- *serv_addr*
  - The address of the server to which the socket is to be connected.
  - IPv4 socket uses structure sockaddr_in
  - IPv6 socket uses structure sockaddr_in6
  - ➔ they are both based on sockaddr, need to cast them to sockaddr.
- *addrlen*
  - The length of the address.
- Return value
  - If no error occurs, **connect()** returns 0.
  - Otherwise, it returns -1

send()

- Send data on a connected socket

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

ssize_t send(int sockfd, const void *buf, size_t len, int flags);
```

- sockfd
  - The file descriptor of the local socket from which data will be sent.
- buf
  - A buffer containing the data to be transmitted.
- len
  - The length of the data in buff.
- flags
  - Specifies the way in which the call is made.
  - Usually 0
- Return value
  - If no error occurs, send() returns the total number of characters sent
  - Otherwise, return -1

send (2)

- More about “flags”
  - MSG_OOB Send as “out of band” data. TCP supports this, and it's a way to tell the receiving system that this data has a higher priority than the normal data. The receiver will receive the signal SIGURG and it can then receive this data without first receiving all the rest of the normal data in the queue.
  - MSG_DONTROUTE Don't send this data over a router, just keep it local.
  - MSG_DONTWAIT If **send()** would block because outbound traffic is clogged, have it return EAGAIN. This is like a “enable non-blocking just for this send.”
  - MSG_NOSIGNAL If you **send()** to a remote host which is no longer **recv()**, you'll typically get the signal SIGPIPE. Adding this flag prevents that signal from being raised.

recv

- Receive data on a socket

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

ssize_t recv(int sockfd, void *buf, size_t len, int flags);
```

- sockfd
  - The file descriptor of the local socket where the data is receiving.
- buf
  - A buffer for the incoming data.
- len
  - The length of buf.
- flags
  - Specifies the way in which the call is made.
- Return value
  - If no error occurs, recv() returns the total number of characters received
  - Otherwise, return -1

close()

- Close a socket descriptor

```
#include <unistd.h>

int close(int sockfd);
```

- sockfd
  - The descriptor of the socket to be closed.
- Return value
  - If no error occurs, close() returns 0.
  - Otherwise, return -1 (and **errno** will be set accordingly)

EXAMPLE & EXERCISE

Analyzing a client

```
#define SERV_PORT 3000
int main(int argc, char **argv)
{ int sockfd;
  struct sockaddr_in servaddr;
 char sendline[MAXLINE], recvline[MAXLINE];
//Create a socket for the client
if ((sockfd = socket (AF_INET, SOCK_STREAM, 0)) <0) {
  perror("Problem in creating the socket");
  exit(2);
 }
 //Creation of the remote server socket information structure
 memset(&servaddr, 0, sizeof(servaddr));
 servaddr.sin_family = AF_INET;
 servaddr.sin_addr.s_addr= inet_addr(argv[1]);
 servaddr.sin_port =  htons(SERV_PORT); //convert to big-endian order
```

create a client socket

create a socket addr info pointing to server socket

Analyzing a client (cont.)

```
// Connect the client to the server socket
if (connect(sockfd, (struct sockaddr *) &servaddr, sizeof(servaddr))<0) {
 perror("Problem in connecting to the server");
 exit(3);
}

while (fgets(sendline, MAXLINE, stdin) != NULL) {
 send(sockfd, sendline, strlen(sendline), 0);

 if (recv(sockfd, recvline, MAXLINE,0) == 0){
  //error: server terminated prematurely
  perror("The server terminated prematurely");
  exit(4);
 }
```

Connect the client socket with remote server

Send and receive data from client socket

Echo Server and Echo Client

- Download the Echo server and Echo client from course website.
- http://users.soict.hust.edu.vn/linhtd
- Run Echo server first then run Echo client
  - ./echoServer
- Syntax:
  - $echoClient <IP address of the server>
- Server works on port 3000 on the Server side
- Working protocol
  - Echo server waits for connections from Echo client.
  - Echo client waits for messages from users and sends to server.
  - The server prints the received message and sends it back to Echo client.
  - Client prints also the received messages.

Exercise

- Write your own Echo client to communicate similarly to the existing Echo server.
- Your Echo client allows users to specify which days of week they are interested and send that day to server.
  - User can type a weekday: Monday, Friday …
  - User can also type: All, if he is interested in the schedule of the whole week.
- Test it with the existing Echo server.

TCP SERVER

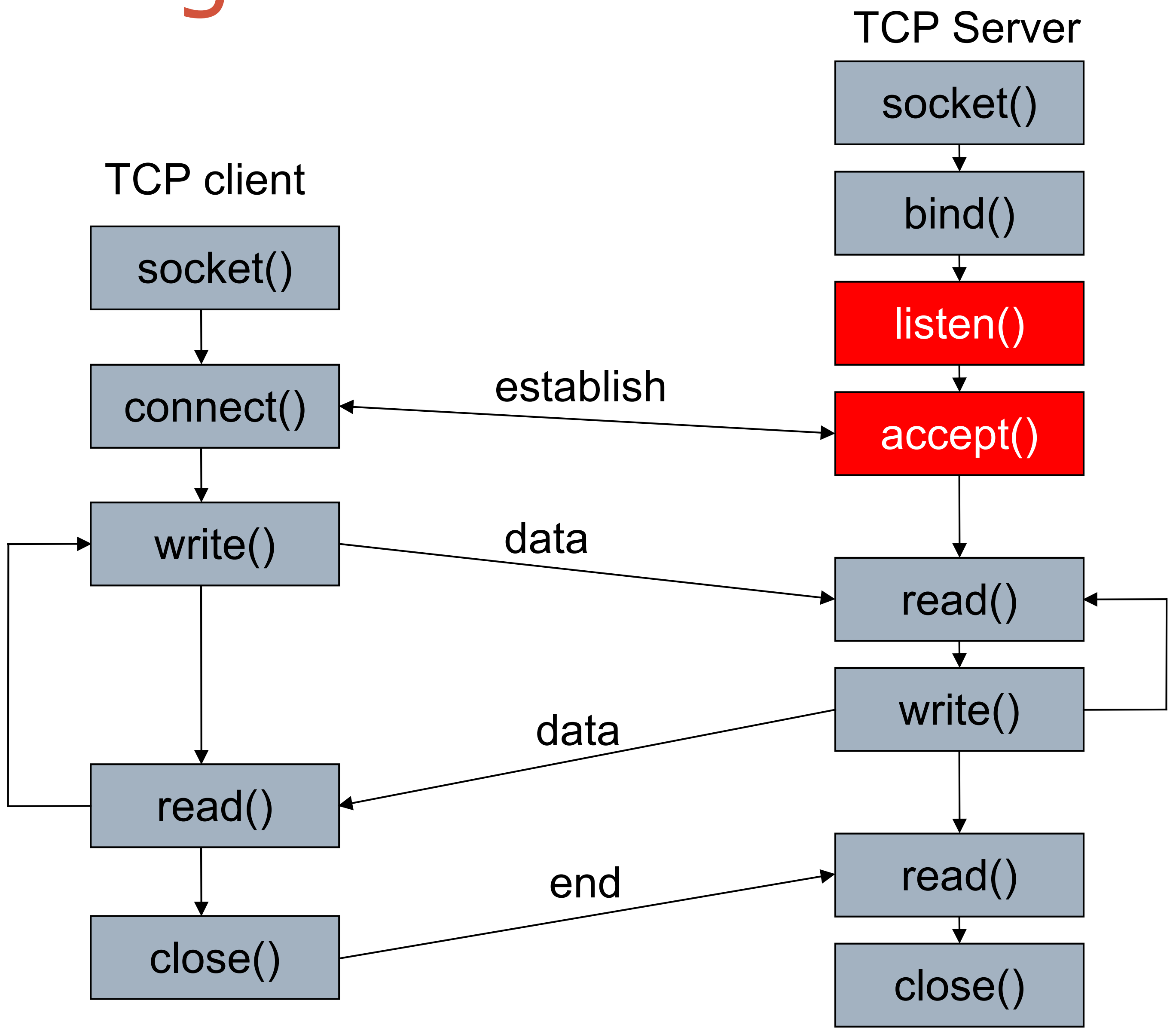
Algorithm for TCP Server in C

- Create a TCP *server socket*
  - socket()
- Bind the *server socket* to server IP and Port number (this is the port to which clients will connect)
  - bind()
- Ready to wait for connection from client
  - listen()
- Accept a new connection from client
  - returns a *file descriptor* that represents the client which is connected
  - accept()
- Send/ receive data with client using the *client socket*
  - recv(), send()
- Close the connection with client
  - closesocket()

Socket Mode

- Types of server sockets
  - *Iterating server*:
    - Only one socket is opened at a time and a single server process handle client requests one after the other.
  - *Forking server*:
    - After an accept, a child process is forked off to handle the connection. Multiple connections can be handled simultanously
  - *Concurrent single server*:
    - a single process simultaneously wait on multiple open sockets, and the process is woken up only when new data arrives to one of the sockets.

Iterating Server



bind()

- Associate a socket with an IP address and port number

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

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

- Where
  - *sockfd* : is the file descriptor of the socket to be bound with the address in *my_addr*
  - *my_addr* : is a pointer to the structure of the address to be assigned to the socket *sockfd* .
  - *addrlen* : is the size of **my_addr*
- Return value
  - 0 if no errors
  - -1 if has errors

listen()

- Establish a socket to listen for incoming connection.

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

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

- sockfd
    - The file descriptor of the unconnected socket that is waiting for connections from client.
- backlog
    - The maximum number of pending connections.
- Use after bind() with a port number
- Return value
    - 0 if no errors
    - - if has errors

accept()

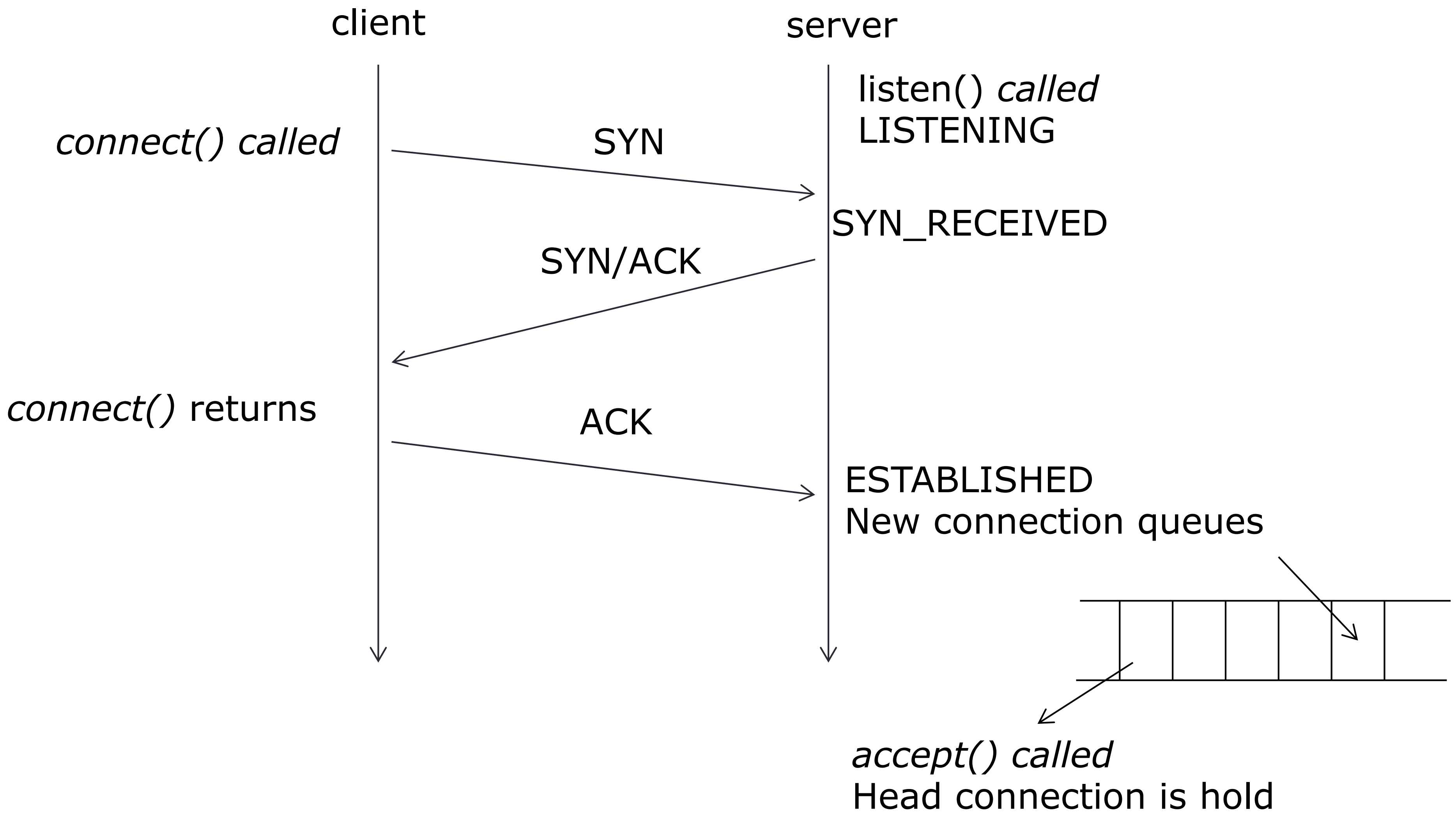
- Accept an incoming connection on a listening socket

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

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

- sockfd
  - The file descriptor of the socket which receives the connection request to be accepted after a listen().
- addr
  - An optional reference pointer to the address of the client socket on the other end of the connection
  - The format of the addr is determined by the address family
- addrlen
  - A optional pointer to an integer which contains the length of the address addr.
- Return value
  - Newly connected socket file descriptor if no errors
  - - if has errors

Process connections



send(), recv()

- Similar to in TCP client

send()

```
char sendBuff[2048];
int  dataLength, nLeft, idx;

// Fill sendbuff with 2048 bytes of data
nLeft = dataLength;
idx = 0;

while (nLeft > 0){
    // Assume s is a valid, connected stream socket
    ret = send(s, &sendBuff[idx], nLeft, 0);
    if (ret == -1)
    {
        // Error handler
    }
    nLeft -= ret;
    idx += ret;
}
```

recv()

```
char    recvBuff[1024];
int     ret, nLeft, idx;
nLeft = dataLength; //length of the data needs to be
                    //received
idx = 0;

while (nLeft > 0)
{
    ret = recv(s, &recvBuff[idx], nLeft, 0);
    if (ret == -1)
    {
        // Error handler
    }
    idx += ret;
    nLeft -= ret;
}
```

close()

- Close a socket descriptor

```
#include <unistd.h>

int close(int s);
```

- s
  - A descriptor identifying the socket to be closed.
- Return value
  - If no error occurs, close() returns 0.
  - Otherwise, return -1 (and **errno** will be set accordingly)

Analyzing a TCP server

```
int listenfd, connfd, n;
pid_t childpid;
 socklen_t clilen;
 char buf[MAXLINE];
 struct sockaddr_in cliaddr, servaddr;
 listenfd = socket (AF_INET, SOCK_STREAM, 0);

 servaddr.sin_family = AF_INET;
 servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
 servaddr.sin_port = htons(SERV_PORT);

 bind (listenfd, (struct sockaddr *) &servaddr, sizeof(servaddr));

 listen (listenfd, LISTENQ);
 printf("%s\n","Server running...waiting for connections.");
```

creation of server socket

Preparation of the socket address struct

Bind the socket to the port in address

Listen for connection to the socket

Analyzing a TCP server

```
for ( ; ; ) {
 clilen = sizeof(cliaddr);
 connfd = accept (listenfd, (struct sockaddr *) &cliaddr, &clilen);
 printf("%s\n","Received request...");
 while ( (n = recv(connfd, buf, MAXLINE,0)) > 0)  {
  printf("%s","String received from and resent to the client:");
  puts(buf);
  send(connfd, buf, n, 0);
 }
if (n < 0) {
 perror("Read error");
 exit(1);
}
close(connfd); // close the file descriptor.
}
close (listenfd); //close listening socket
```

Accept a connection request → return a File Descriptor (FD)

Send and receive data from the FD

Exercise 1

- Write your own Echo server to communicate with the existing Echo client
- Get Address IP and port of the client and sent them back to client.
- Leave your server to work with the client you have created in previous lab.

Exercise 2

- Make a simple file transfer application
- Revise the TCP server and TCP client so that
  - User at client side can choose a file to send to server
  - Client send the file to server.
  - On the server, the file should be readable.

- Hint:
  - Read file gradually on Client side and send to server small messages.
  - Use fread and fwrite
  - For testing, send an image file from client to server.

Exercise 3

- Go back with application Study Schedule Management (SSM) program that we did in Lecture 1.
- Now we modify SSM program so that users can consult study schedule over Internet from everywhere (store schedules in a server).
  - User interface will be on client side
  - Schedule storage and processing will be on server side.
- Login:
  - client takes username, password from user and sends to server.
  - Server verifies username and password if they match.
- Read schedule:
  - Client provides interface for user to enter a week day (or ALL)
  - Server extracts schedule of the day (or busy schedule) and sends back to student
  - Client displays the schedule

Exercise 3

- Which structure we should use for sending schedule from server to client?
  - Case 1: Server send the struct correspond to the searched schedule to the client
    - Run server and client on two different machines.
    - Can client read correctly the schedule? Why?
  - Case 2:
    - Before sending the searched schedule to client, server convert the struct containing schedule to a string. Then it send the string to client.
    - Run server and client on two different machines.
    - Can client read correctly the schedule? Why?

Conclusion:

- Do not send compound datatype.

- Do not send pointer based data