CS 111 week 4
Project 1B: Compressed Network Communication

Discussion 1B

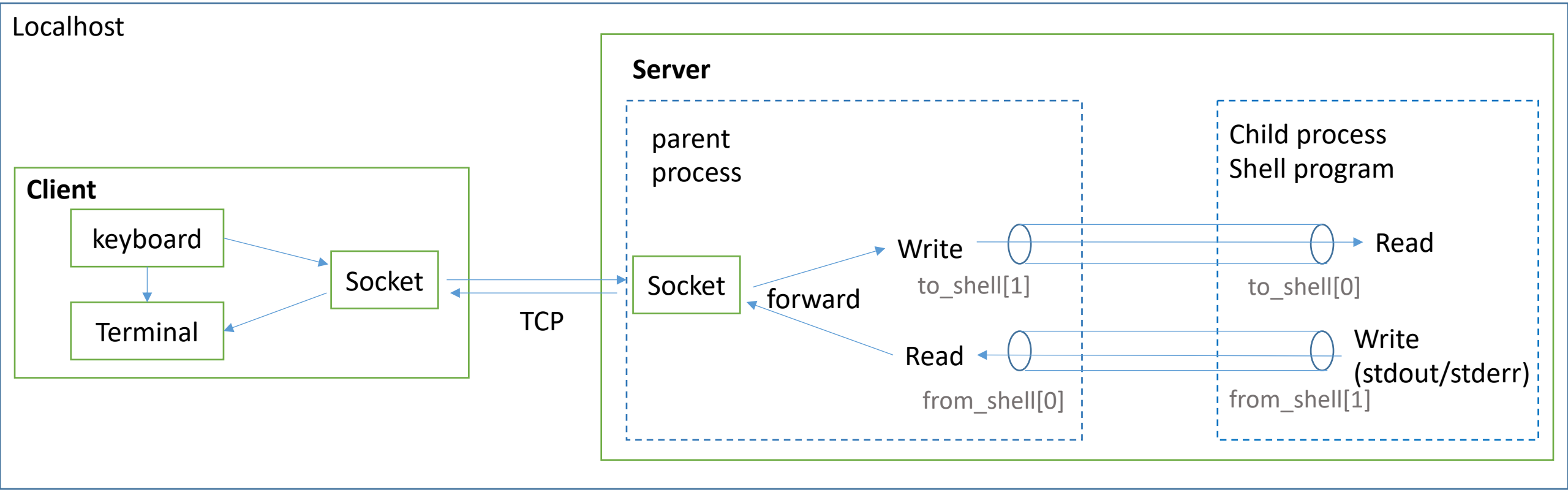
Tianxiang Li

Project Overview

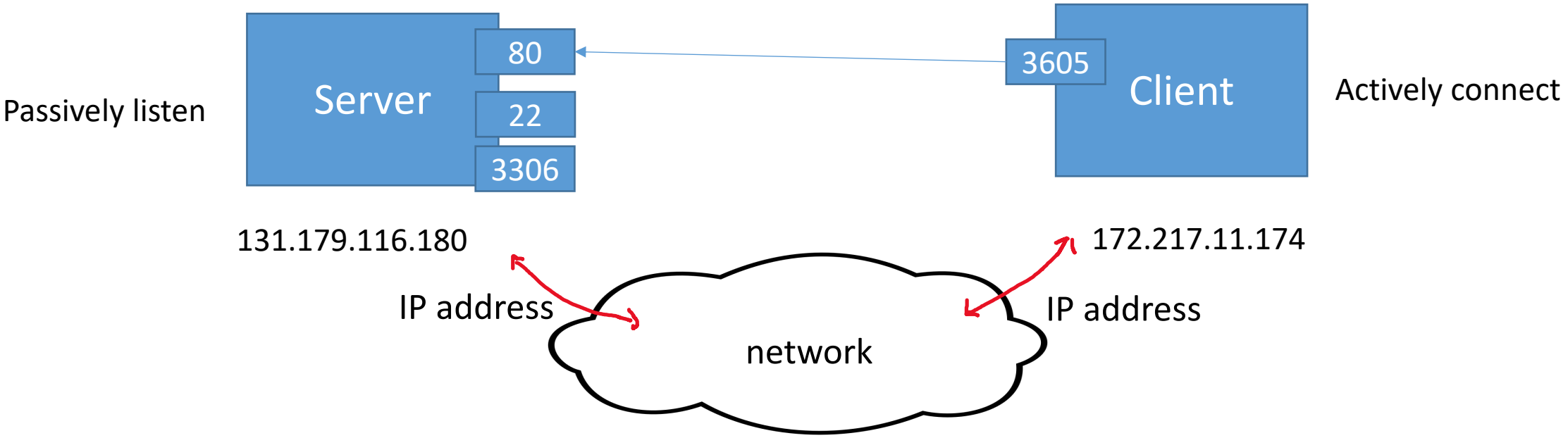
1. Part A: extend the project in Lab1A with network connection.

2. PartB: Compress the network connections

Project Overview

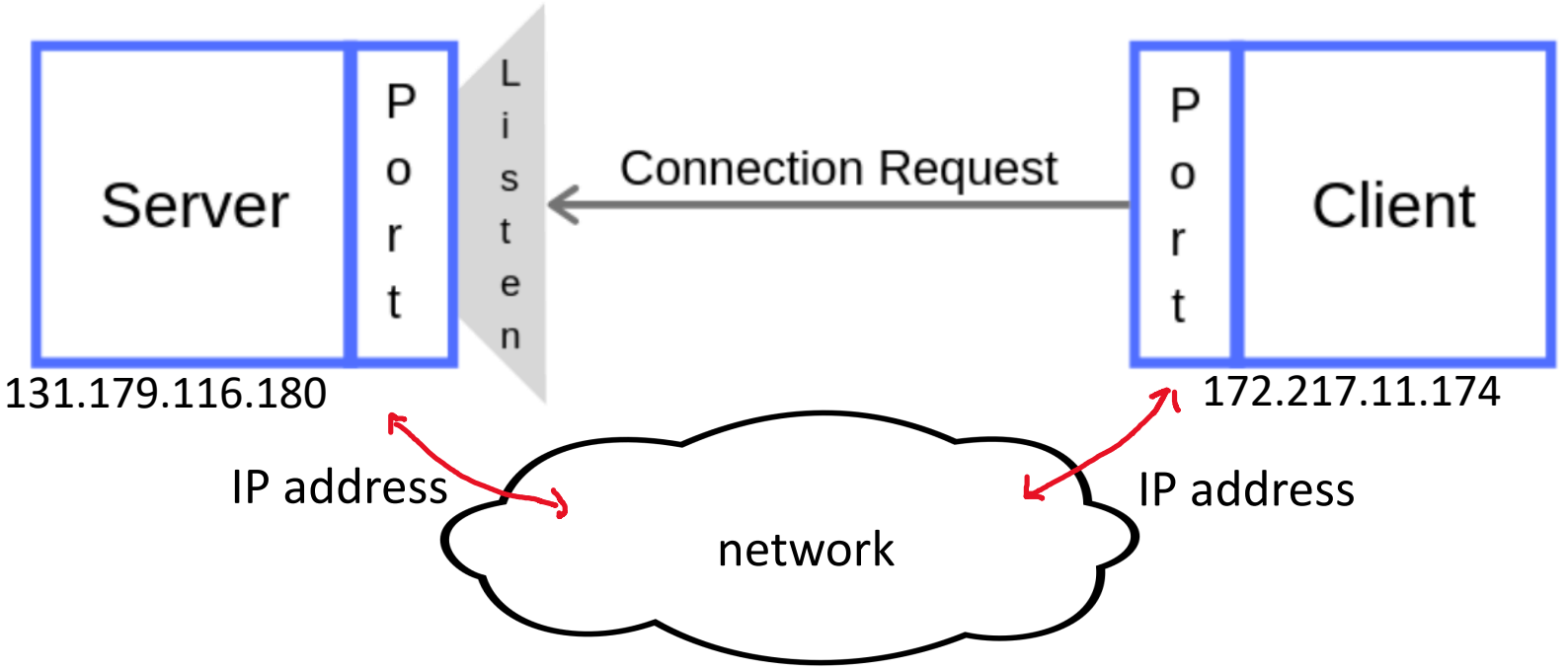


Background: C/S Model, IP/Port



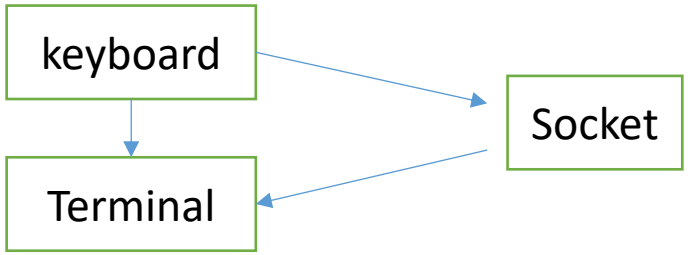
- IP address identifies which host the data is destined to
- Port number identifies which services on the host the data is destined to
  - e.g. HTTP 80, SSH 22, FTP DATA 20, MySQL 3306, … (0-1024 reserved)

Background: Socket



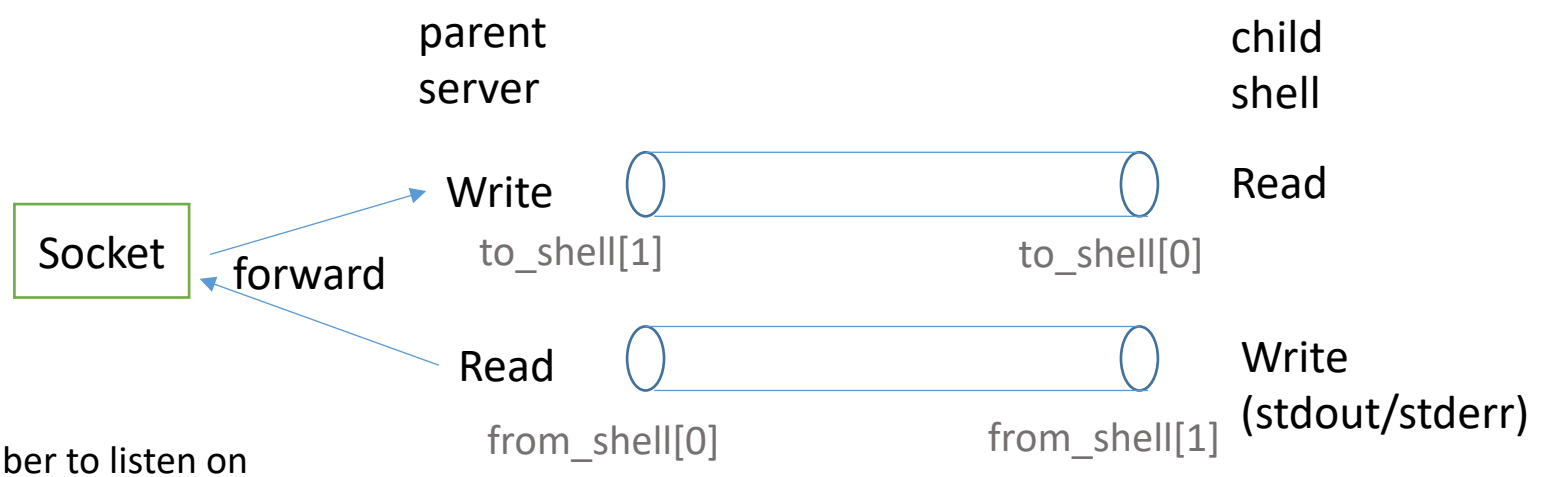
- **A socket is “one endpoint of a two-way communication link between two programs running on the network."**
  - IP address identifies which host the data is destined to
  - Port number identifies which services on the host the data is destined to
    - e.g. HTTP 80, SSH 22, FTP DATA 20, MySQL 3306, … (0-1024 reserved)

Client



- Options
  - **--port=***portnum* (mandatory): port number used to set up connection
  - **--log=***filename:* maintains a record of data sent over the socket
  - **--compress** (second part of project)
- I/O
  - Use **non-canonical** (character at a time, no echo) terminal behavior you used in Project 1A.
  - send input from the **keyboard** to the **socket** (while echoing to the **display**)
  - send input from the **socket** to the **display**
- Special character handling
  - If a ***^D*** or ***^C*** is entered on the terminal, simply pass it through to the server like any other character.
- Error handling
  - **unrecognized argument:** print an informative **usage message** (on **stderr**) and exit with a **return code of 1**.
  - **system call fails**: print an informative **error message** (on **stderr**) and exit with a **return code of 1**.
- Exit
  - Before exiting, **restore** normal terminal modes.

Server



- Options
  - **--port=***portnum* (mandatory): port number to listen on
  - **--shell** (can be omitted) fork a child process, which will exec a shell to process commands received
  - **--compress** (second part of project)
- I/O
  - Redirect the shell process's stdin/stdout/stderr to the appropriate pipe ends (similar to what you did in Project 1A).
  - Input received through the network **socket** should be forwarded through the **pipe to the shell**.
  - Input received from the **shell pipes (**stdout/stderr from the shell) should be forwarded out to the network **socket**.
- Special character handling
  - Because the server forks the shell (and knows its process ID), the processing of ***^C*** (turning it into a **SIGINT** to the **shell**) must be done in the server process.
  - when the server encounters a ***^D*** it should **close** the **write side of the pipe** to the shell (as was done in project 1A).
- Error handling
  - Receive **EOF or SIGPIPE** from the **shell pipes** (e.g. the shell exits): harvest the shell's completion **status**, log it to **stderr** (as you did in project 1A), and exit with a **return code of 0**.
  - Receive **EOF** or **read error** from the **network connection**, (e.g., the client exits): **close** the **write pipe** to the shell, **await** an **EOF** from the pipe from the shell, and harvest and report its termination **status** (as above).
  - **unrecognized argument:** print an informative **usage message** (on **stderr**) and exit with a **return code of 1**.
  - **system call fails**: print an informative **error message** (on **stderr**) and exit with a **return code of 1**.
- Exit
  - Before exiting, **restore** normal terminal modes.

Testing

- Start Server
  - Listen on port greater than 1024
  - Do not display anything (other than error messages) on the server side
- Start Client
  - Connect to same port as server
  - You will see your commands echoed back to the terminal as well as the output from the shell

Shutdown

- Server (Server initiates shutdown)
  - an *exit(1)* command is sent to the shell, or the server closes the write pipe to the shell.
  - the **shell exits**, causing the server to receive an **EOF** on the read pipe from the shell.
  - the server collects and reports the shell's **termination status**.
  - the server **closes** the network **socket** to the client, and **exits**.
- Client
  - the client continues to process output from the server until it receives an error on the network socket from the server.
  - the client **restores** terminal modes and **exits**.

TCP socket: service setup

TCP Client

TCP Server

TCP socket: service setup

TCP Client

TCP Server

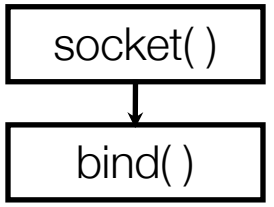
socket( )

socket ( ): Create a socket

TCP socket: service setup

TCP Client

TCP Server

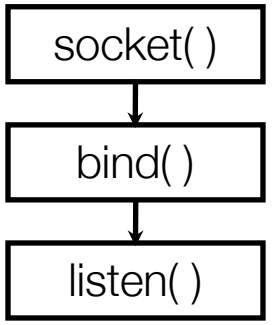


bind( ): bind a socket to a local IP address and port #

TCP socket: service setup

TCP Client

TCP Server

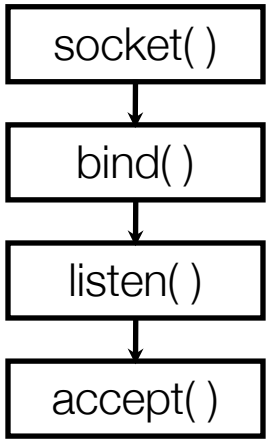


listen( ): passively waiting for connections

TCP socket: service setup

TCP Client

TCP Server

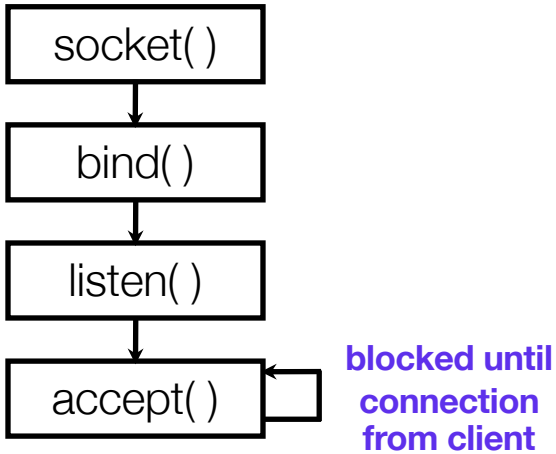


accept( ): accept a new connection

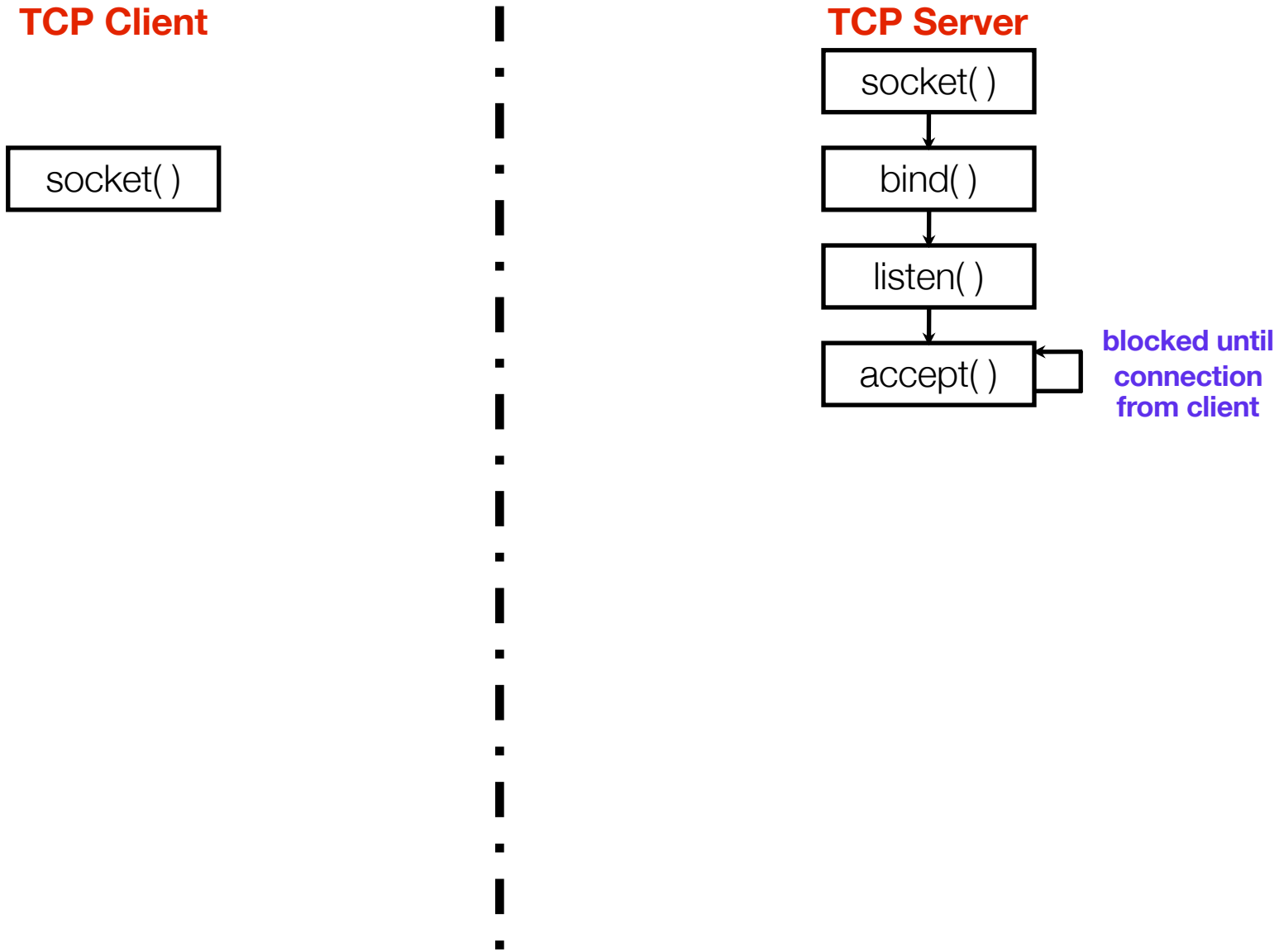
TCP socket: service setup

TCP Client

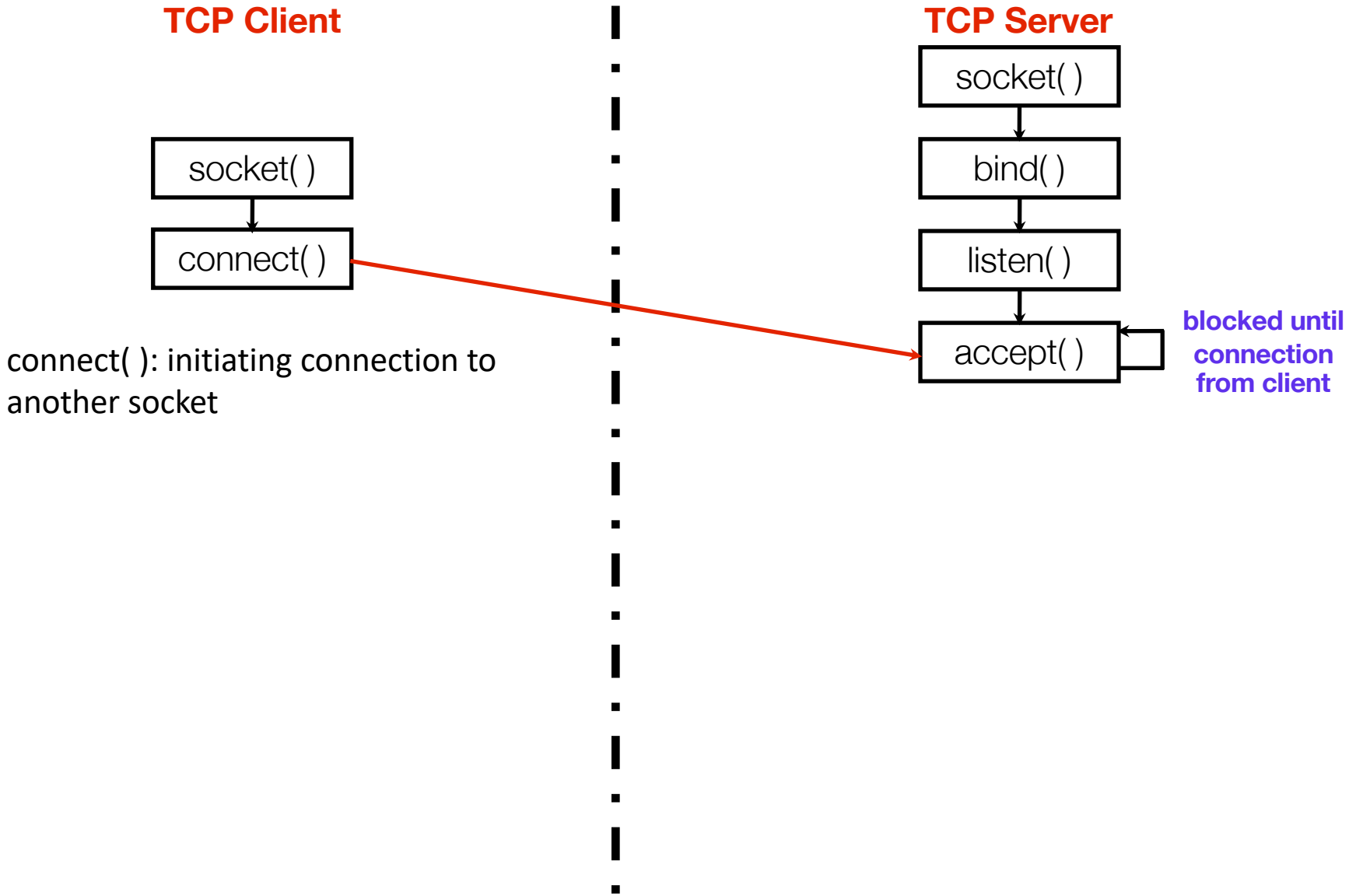
TCP Server



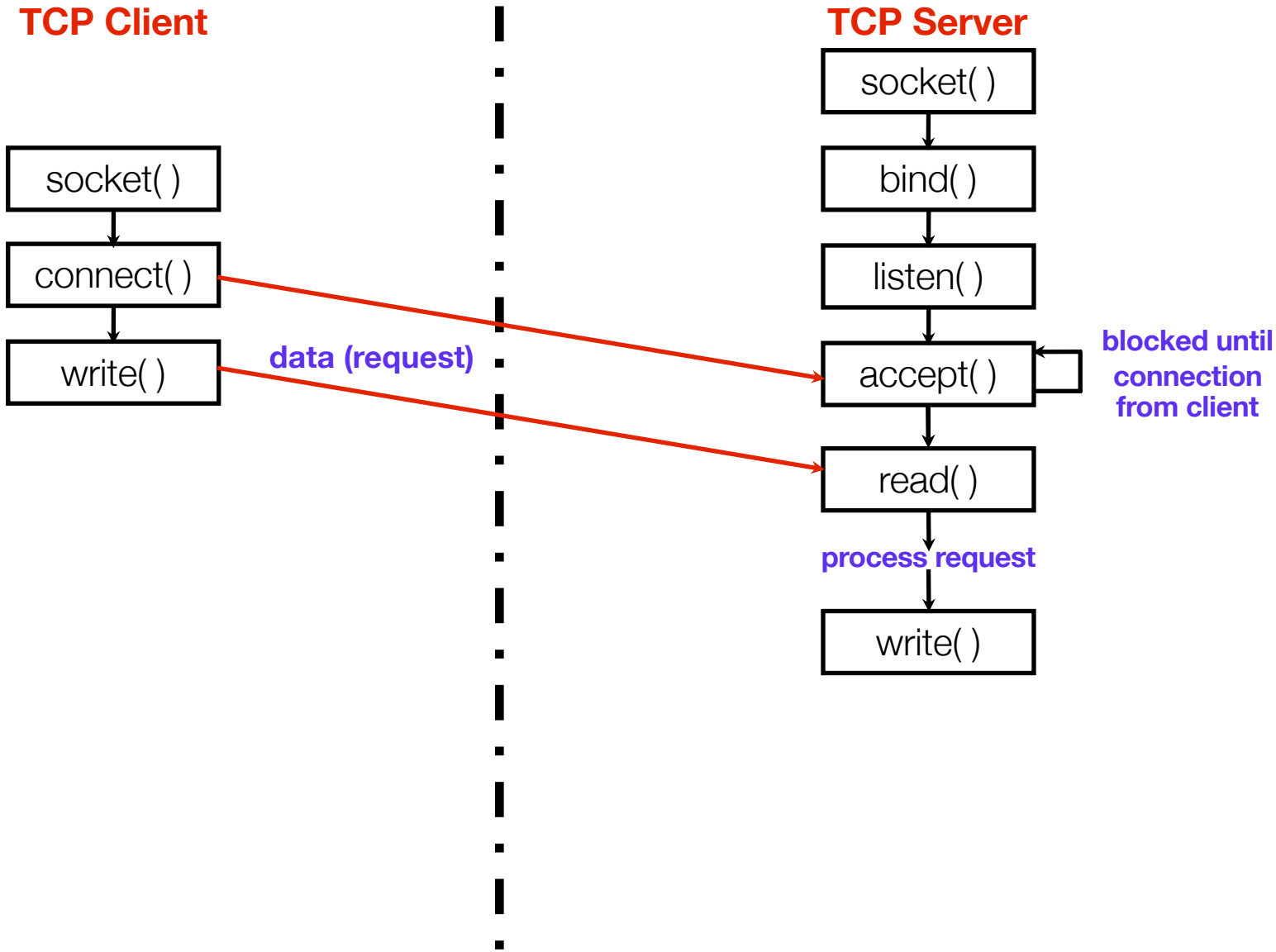
TCP socket: service setup



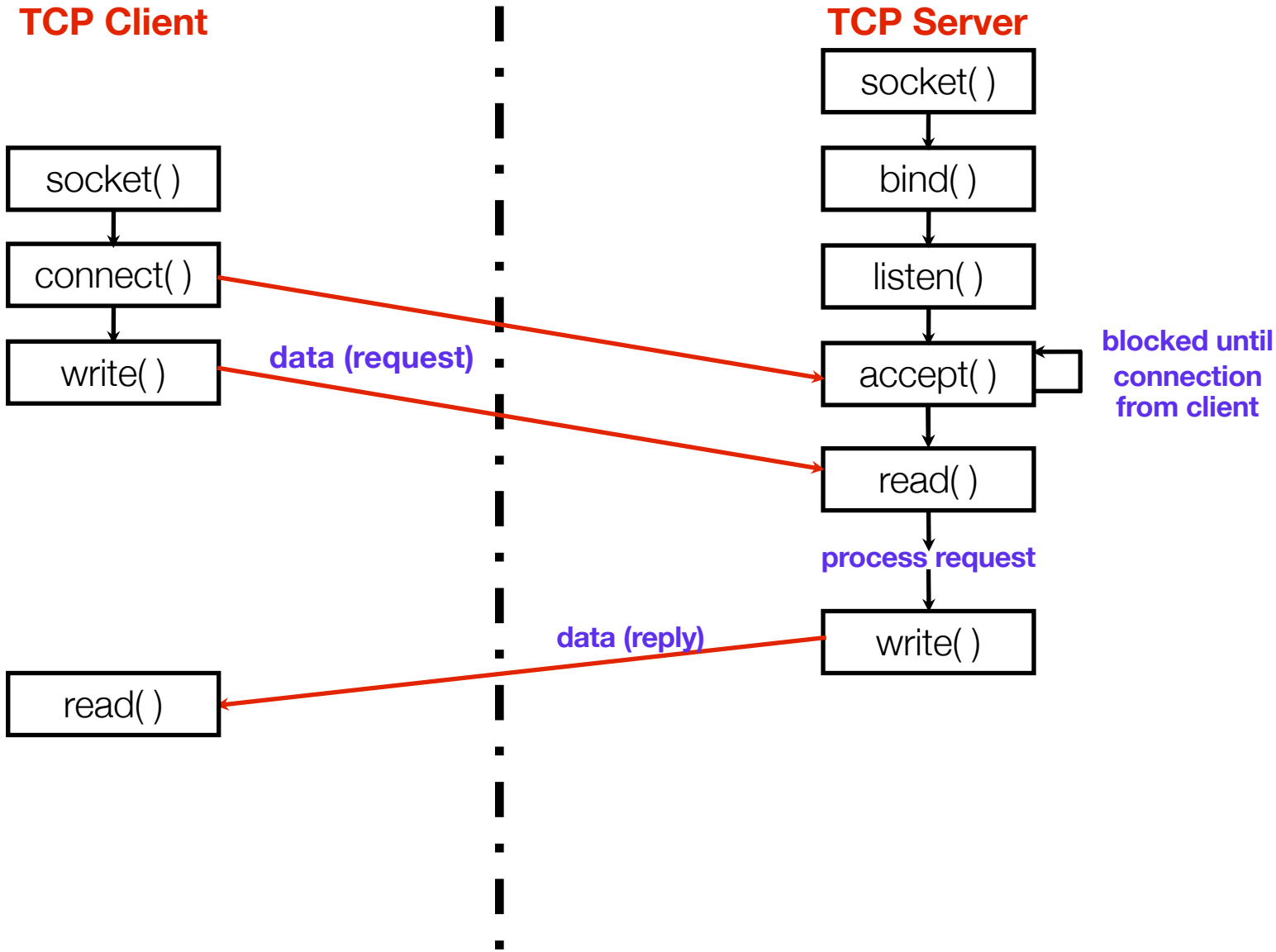
TCP socket: establish connection



TCP socket: send and receive data



TCP socket: send and receive data



Headers

```
/* headers */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/wait.h>
#include <netinet/in.h>
…
```

How to write a client?

```
/* include all the headers */
int client_connect(char* hostname, unsigned int port) {

    //1.  create socket

     //2.  fill in socket address information

    //3.  connect socket with corresponding address


}
```

Functions

- **int socket(int socket_family, int type, int protocol);**
  - Create a socket
  - returns the socket descriptor or -1(failure). Also sets errno on failure
  - **socket_family**: protocol family
    - **AF_INET** for IPv4, **AF_INET6** for IPv6, **…**
  - **type**: communication style
    - **SOCK_STREAM** for TCP
    - **SOCK_DGRAM** for UDP
  - **protocol**: protocol within family, which is typically set to 0

How to write a client?

```
/* include all the headers */
int client_connect(char* hostname, unsigned int port) {
/* e.g. host_name:"google.com", port:80, return the socket for subsequent communication */

    int sockfd;
    struct sockaddr_in serv_addr; /* server addr and port info */
    struct hostent* server;
     //1.  create socket
    sockfd = socket(AF_INET, SOCK_STREAM, 0);

     //2.  fill in socket address information


    //3.  connect socket with corresponding address
return sockfd;
}
```

Socket programming API: essential structs

- sockfd — socket descriptor. Just a regular **int**.
- sockaddr_in — struct for socket address info

```
struct sockaddr_in { // used for IPv4 only
    short          sin_family;  // addr family, AF_INET
    unsigned short sin_port;    // port number
    struct in_addr sin_addr;    // internet address
    unsigned char  sin_zero[8]; // padding zeros
};
```

How to write a client?

```
struct sockaddr_in { // used for IPv4 only
    short          sin_family;  // addr family, AF_INET
    unsigned short sin_port;    // port number
    struct in_addr sin_addr;    // internet address
    unsigned char  sin_zero[8]; // padding zeros
};
```

```
/* include all the headers */
int client_connect(char* hostname, unsigned int port) {
/* e.g. host_name:"google.com", port:80, return the socket for subsequent communication */

    int sockfd;
    struct sockaddr_in serv_addr; /* server addr and port info */
    struct hostent* server;
     //1.  create socket
    sockfd = socket(AF_INET, SOCK_STREAM, 0);

     //2.  fill in socket address information
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_port = htons (port);
    server = gethostbyname(hostname); /* convert host_name to IP addr */
    memcpy(&serv_addr.sin_addr.s_addr, server->h_addr, server->h_length); /* copy ip address from server to serv_addr */
    memset(serv_addr.sin_zero, '\0', sizeof serv_addr.sin_zero); /* padding zeros*/

    //3.  connect socket with corresponding address

return sockfd;
}
```

How to write a client?

```
struct sockaddr_in { // used for IPv4 only
    short          sin_family;  // addr family, AF_INET
    unsigned short sin_port;    // port number
    struct in_addr sin_addr;    // internet address
    unsigned char  sin_zero[8]; // padding zeros
};
```

```
/* include all the headers */
int client_connect(char* hostname, unsigned int port) {
/* e.g. host_name:"google.com", port:80, return the socket for subsequent communication */

    int sockfd;
    struct sockaddr_in serv_addr; /* server addr and port info */
    struct hostent* server;
     //1.  create socket
    sockfd = socket(AF_INET, SOCK_STREAM, 0);

     //2.  fill in socket address information
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_port = htons (port);
    server = gethostbyname(hostname); /* convert host_name to IP addr */
    memcpy(&serv_addr.sin_addr.s_addr, server->h_addr, server->h_length); /* copy ip address from server to serv_addr */
    memset(serv_addr.sin_zero, '\0', sizeof serv_addr.sin_zero); /* padding zeros*/

    //3.  connect socket with corresponding address
    connect(sockfd, (struct sockaddr*) &serv_addr, sizeof(serv_addr));

return sockfd;
}
```

How to write a server: body (I)

```
int server_connect(unsigned int port_num)
/* port_num: which port to listen, return socket for subsequent communication*/
{
    int sockfd, new_fd;         /* listen on sock_fd, new connection on new_fd */
    struct sockaddr_in my_addr;         /* my address */
    struct sockaddr_in their_addr; /* connector addr */
    int sin_size;

    /* create a socket */
    sockfd = socket(AF_INET, SOCK_STREAM, 0);
```

How to write a server: body (II)

```
struct sockaddr_in { // used for IPv4 only
    short          sin_family;  // addr family, AF_INET
    unsigned short sin_port;    // port number
    struct in_addr sin_addr;    // internet address
    unsigned char  sin_zero[8]; // padding zeros
};
```

```
// ...
/* set the address info */
my_addr.sin_family = AF_INET;
my_addr.sin_port = htons(port_num); /* short, network byte order */
my_addr.sin_addr.s_addr = INADDR_ANY
/* INADDR_ANY allows clients to connect to any one of the host's IP address.

memset(my_addr.sin_zero, '\0', sizeof(my_addr.sin_zero)); //padding zeros

/* bind the socket to the IP address and port number */
bind(sockfd, (struct sockaddr *) &my_addr, sizeof(struct sockaddr));
```

The htons() function
converts from host byte order to network byte order

The inet_addr() function
converts the Internet host address from IPv4 numbers-and-dots notation to network byte order.

Functions

- **int listen(int sockfd, int backlog);**
  - Put socket into passive state (wait for connections rather than initiating a connection)
  - returns 0 on success, -1 and sets errno on failure
  - **sockfd**: socket file descriptor returned by socket( )
  - **backlog**: the maximum number of connections this program can serve simultaneously

How to write a server: body (III)

```
    // ...
    listen(sockfd, 5); /* maximum 5 pending connections */

    sin_size = sizeof(struct sockaddr_in);
    /* wait for client's connection, their_addr stores client's address */
    new_fd = accept(sockfd, (struct sockaddr*)&their_addr, &sin_size));

return new_fd; /* new_fd is returned not sock_fd*/
}
```

store ip address of client

Functions

- **int accept(int sockfd, struct sockaddr* client_addr, int* addrlen);**
  - Accept a new connection
  - Return client's socket file descriptor or -1. Also sets errno on failure
  - **sockfd**: socket file descriptor for server, returned by socket( )
  - **client_addr**: IP address and port number of a client (returned from call)
  - **addrlen**: length of address structure = pointer to **int** set to **sizeof(struct sockaddr_in)**
  - **NOTE: client_addr and addrlen are result arguments**
    - i.e. The program passes empty client_addr and addrlen into the function, and the kernel will fill in these arguments with client's information

Functions

- **int close(int sockfd);**
  - close a socket
  - return 0 on success, or -1 on failure
  - After close, sockfd is no longer valid

After the connection

For brevity, error handling is omit

```
//server side
int main()
{
    char buffer[100];
    int sock_fd = server_connect(80);
    write(sock_fd, “hahaha”, 6);
    read(sock_fd, buffer, 100);
    //buffer will be “hihihi”
}
```

For brevity, error handling is omit

```
//client side
int main()
{
    char buffer[100];
    int sock_fd = client_connect(“microsoft.com”, 80);
    read(sock_fd, buffer, 100);
    //buffer will be “hahaha”
    write(sock_fd, “hihihi”, 6);
}
```

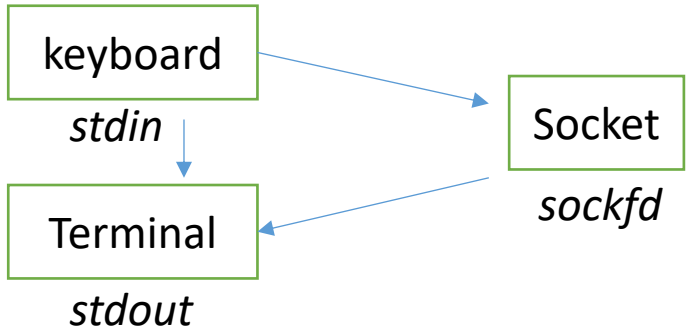
Everything is a file descriptor: files, pipe, stdin, stdout, stderr, socket

Benefits: Use the same set of APIs: read, write, poll for everything.

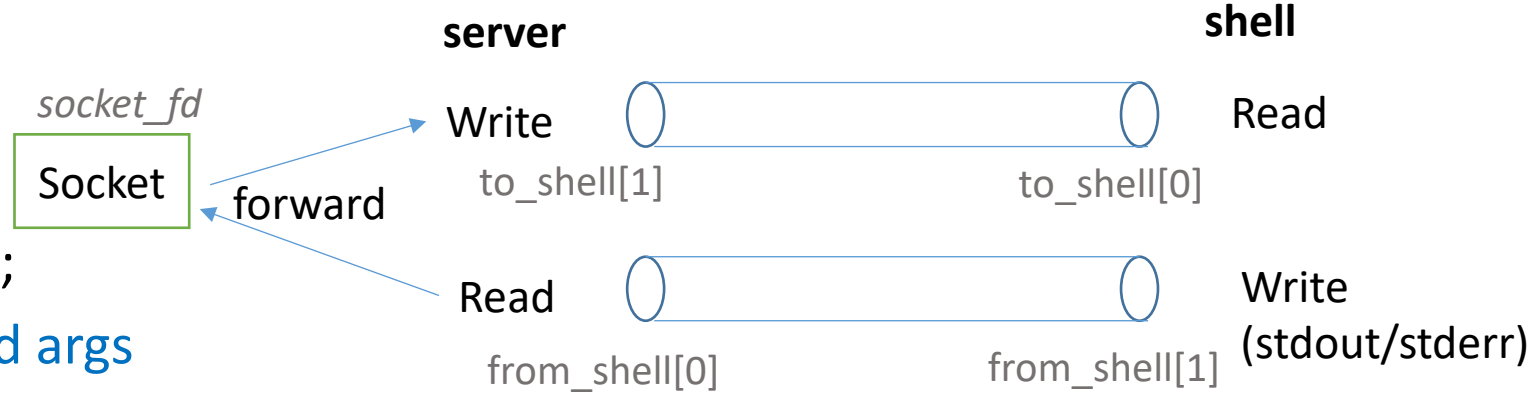
Drawback: Implementation is very complex, loss of efficiency.

Client side code skeleton

```
int main(char * argc[], int argv) {
        process_cmd_args(argc, argv);
        socket_fd = client_connect(hostname, port);
        //hostname, port number obtained from cmd args
        set the terminal to the non-canonical, no echo mode
        while (1) {
                poll the socket_fd and stdin
                if (socket_fd ready to read)
                        read from socket_fd, process special characters, send to stdout
                if (stdin ready to read)
                        read from stdin, process special characters, send to stdout and socket_fd
                if (error)
                        Proceed to exit process: read every last byte from socket_fd, write to stdout,
                        restore terminal and exit.
        }
}
```



Server side code skeleton




```
int main(char * argc[], int argv) {
        process_cmd_args(argc, argv);
        socket_fd = server_connect(port);
        //port number obtained from cmd args
        Register SIGPIPE handler, Create PIPE, fork the new process, perform stdin/stdout redirection,
        while (1) {
                poll the socket_fd and from_shell[0]
                if (socket_fd ready to read)
                        read from socket_fd, process special characters, send to to_shell[1]
                if (from_shell[0] ready to read)
                        read from from_shell[0], process special characters, send to socket_fd
                if (error)
                        Proceed to exit process: read every last byte from from_shell[0], write
                        to socket_fd, get the exit status of the process and report to stderr.
        }
}
```

Compress

- Purpose
  - Reduce the amount of space data takes up (less bandwidth)
    - Require processing at sending/receiving end
    - Tradeoff between bandwidth and processing cost
- Use standard compression library
  - Depend on your system, install zilib package (library, documentation)
- Start simple
  - A simple program that reads from and writes to a pipe.
- Add **--compress** option (client & server)
  - If included, will enable compression (of all traffic in both directions).
    - compress traffic *before sending* it over the network
    - decompress traffic *after receiving* it

Compress

- Use **--log** option to verify that compression/decompression works
  - You should record
    - outgoing data post-compression
    - incoming data pre-decompression
  - You should expect to see:
    - no clear text going either to or from the shell
    - a reduction in the number of bytes sent in both directions

Compress the network connections Overview

Achieved with the zlib library.

```
int main(void)
{
        z_stream stream;
        init_compress_stream (&stream);  //init a compress stream.
        while (!done) {
            do_compress (&stream, orig_buf, orig_len, out_buf, out_len);
           //Use the zlib to compress the data from orig_buf and put it in out_buf.
            output the out_buf to the network.
        }
        end_compress_stream(&stream); //Close the compress stream
}
```

Original buffer

Output buffer

Note: purple text are user defined function names to encapsulate certain logic of the code, it's not standard functions from the zlib library

Basic functions

- From zlib.h (#include "zlib.h")
  - basic compression functions
    - deflateInit(), deflate(), and deflateEnd()
  - basic decompression functions
    - inflateInit(), inflate(), and inflateEnd()
  - stream data structure
    - The strm structure is used to pass information to and from the *zlib* routines

Compress

- Overall Logic

```
z_stream strm;  // strm structure is used to pass information to and from the zlib routines, and to maintain the deflate() state
unsigned char inbuf[CHUNK];  // input buffer for deflate().
unsigned char outbuf[CHUNK]; // output buffer for deflate().

//initialize compression stream
//while loop to compress data from inputbut to outputbuf
//end compression stream
```

initialize the zlib state for compression using **deflateInit()**

- The **zalloc**, **zfree**, and **opaque** fields in the strm structure must be initialized before calling deflateInit()

```
/* allocate deflate state */
strm.zalloc = Z_NULL; // set to Z_NULL, to request zlib use the default memory allocation routines
strm.zfree = Z_NULL;
strm.opaque = Z_NULL;
ret = deflateInit(&strm, Z_DEFAULT_COMPRESSION); /* pointer to the structure to be initialized, and the compression level */
if (ret != Z_OK) // make sure that it was able to allocate memory, and the provided arguments were valid.
{
    // code here: error handling
}
```

ret = **deflateInit**(&strm, level);

- **strm**: pointer to the structure to be initialized and
- **level**: the compression level (integer in the range of -1 to 9)
  - Lower compression levels result in faster execution, but less compression. Higher levels result in greater compression, but slower execution.
  - The zlib constant Z_DEFAULT_COMPRESSION, equal to -1, provides a good compromise between compression and speed and is equivalent to level 6.
- Return value **ret**:
  - Check against the *zlib* constant Z_OK

Deflate()

```
strm.avail_in =  /* number of bytes available at next_in */
strm.next_in =  inbuf; /* next input byte*/
strm.avail_out = sizeof outbuf; /* remaining free space at next_out */
strm.next_out = outbuf; /* next output byte */

while (strm.avail_in  > 0) {
        deflate(&strm, Z_SYNC_FLUSH); //use Z_SYNC_FLUSH  for independent msgs
/* deflate(). It takes as many of the avail_in bytes at next_in as it can process, and writes as many as avail_out bytes to next_out.
deflate will update next_in, avail_in, next_out, avail_out accordingly.*/
}
```

Deflate()

```
strm.avail_in =  /* number of bytes available at next_in */   minus 100bytes
strm.next_in =  inbuf; /* next input byte*/    minus 100bytes of data
strm.avail_out = sizeof outbuf; /* remaining free space at next_out */  decrease
strm.next_out = outbuf; /* next output byte */ increase

while (strm.avail_in  > 0) {
        deflate(&strm, Z_SYNC_FLUSH); //use Z_SYNC_FLUSH  for independent msgs
    Compress 100bytes of data
deflate will update next_in, avail_in, next_out, avail_out accordingly.*/
compressed_bytes = sizeof outbuf - strm.avail_out; //calculate size of compressed data
}
```

Close the module

```
deflateEnd(&stream); //frees up data structure
```

Ref:

https://www.zlib.net/manual.html

https://www.zlib.net/zlib_how.html