#In the name of Allah

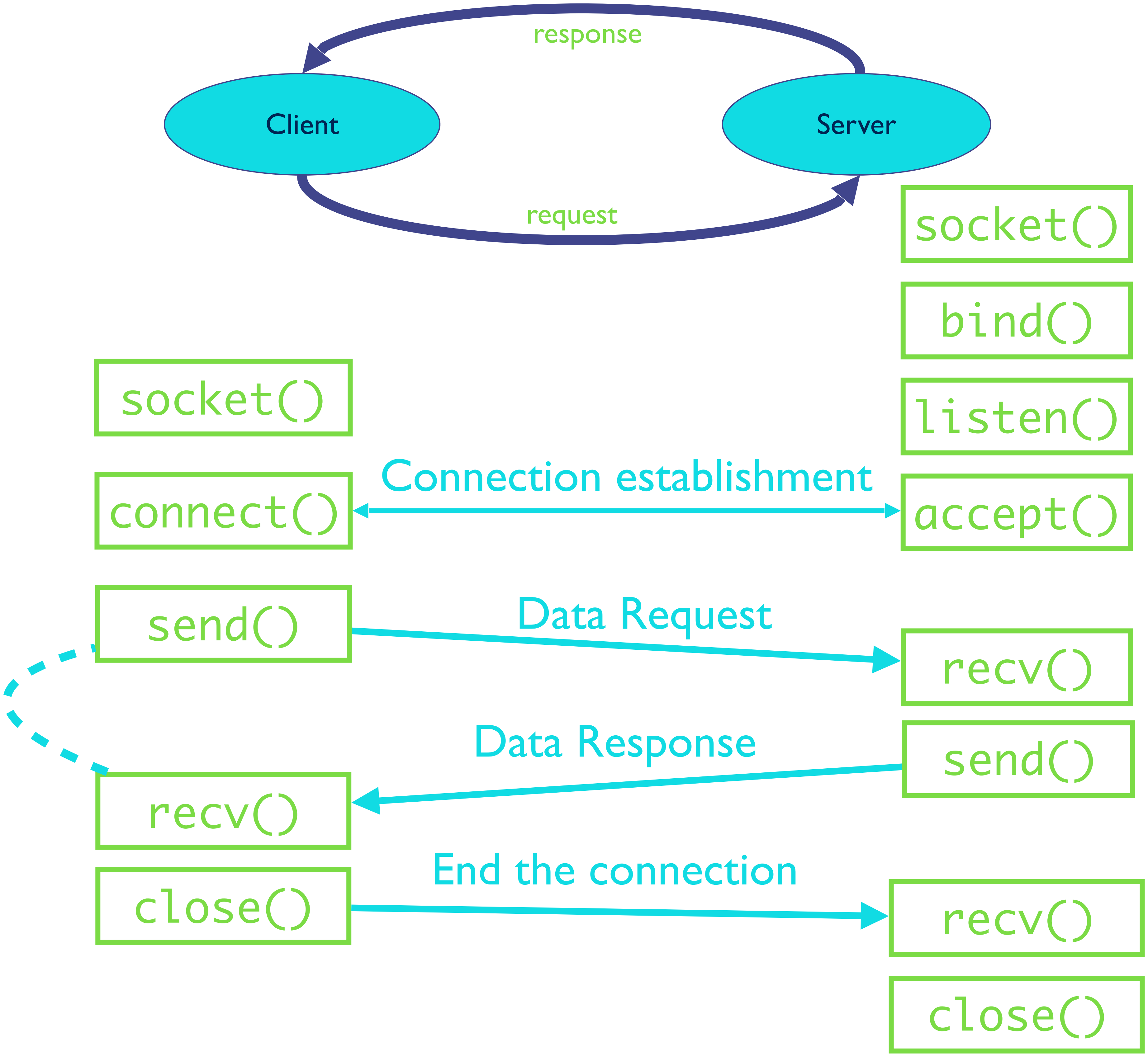
Computer Engineering Department
Sharif University of Technology

CE443- Computer Networks

Socket Programming

Acknowledgments: Lecture slides are from Computer networks course thought by Jennifer Rexford at Princeton University. When slides are obtained from other sources, a reference will be noted on the bottom of that slide.

#Typical Client-Server



response

Client

Server

request

socket()

bind()

socket()

listen()

Connection establishment

connect()

accept()

send()

Data Request

recv()

Data Response

send()

recv()

End the connection

close()

recv()

close()

#Client Programming

- Create stream socket (socket())
- Connect to server (connect())
- While still connected:
  - send message to server (send())
  - receive (recv()) data from server and process it
- Close TCP connection and Socket (close())

#Client: Learning Server Address/ Port

- **Server typically known by name and service**
  - "www.google.com" and "http"
- **Which must be translated into IP address and port #**
- **Translating the server's name to an address**
  - ✦ int **getaddrinfo**(const char *node, const char *service,
  - ✦ const struct addrinfo *hints, struct addrinfo **res);
  - ✦ void **freeaddrinfo**(struct addrinfo *res);
  - ✦ int **getnameinfo**(const struct sockaddr *sa, socklen_t
  - ✦ salen,char *host, size_t hostlen, char *serv, size_t
  - ✦ servlen, int flags);
- Check Linux Man pages for details

#Client: Learning Server Address/ Port

```
struct addrinfo {
    int              ai_flags;     // AI_PASSIVE, AI_CANONNAME, etc.
    int              ai_family;      // AF_INET, AF_INET6, AF_UNSPEC
    int              ai_socktype;  // SOCK_STREAM, SOCK_DGRAM
    int              ai_protocol;        // use 0 for "any"
    size_t            ai_addrlen;        // size of ai_addr in bytes
    struct sockaddr *ai_addr;       // struct sockaddr_in or _in6
    char             *ai_canonname; // full canonical hostname
    struct addrinfo *ai_next;       // linked list, next node
};
struct sockaddr {
    unsigned short    sa_family;    // address family, AF_xxx
    char              sa_data[14];  // 14 bytes of protocol address
};
```

#Client Creating a Socket: socket()

```
int socket(int domain, int type, int protocol)
```

Operation to create a socket

- ✓ Returns a descriptor (or handle) for the socket
- ✓ Originally designed to support any protocol suite

Domain: protocol family

- ✓ PF_INET for the Internet

Type: semantics of the communication

- ✓ SOCK_STREAM: reliable byte stream
- ✓ SOCK_DGRAM: message-oriented service

Protocol: specific protocol

- ✓ UNSPEC: unspecified
- ✓ (PF_INET and SOCK_STREAM already implies TCP)

#Client: Send/Rcv Data and Close

```
int connect(int sockfd, struct sockaddr
*server_address,socketlen_t addrlen)
```

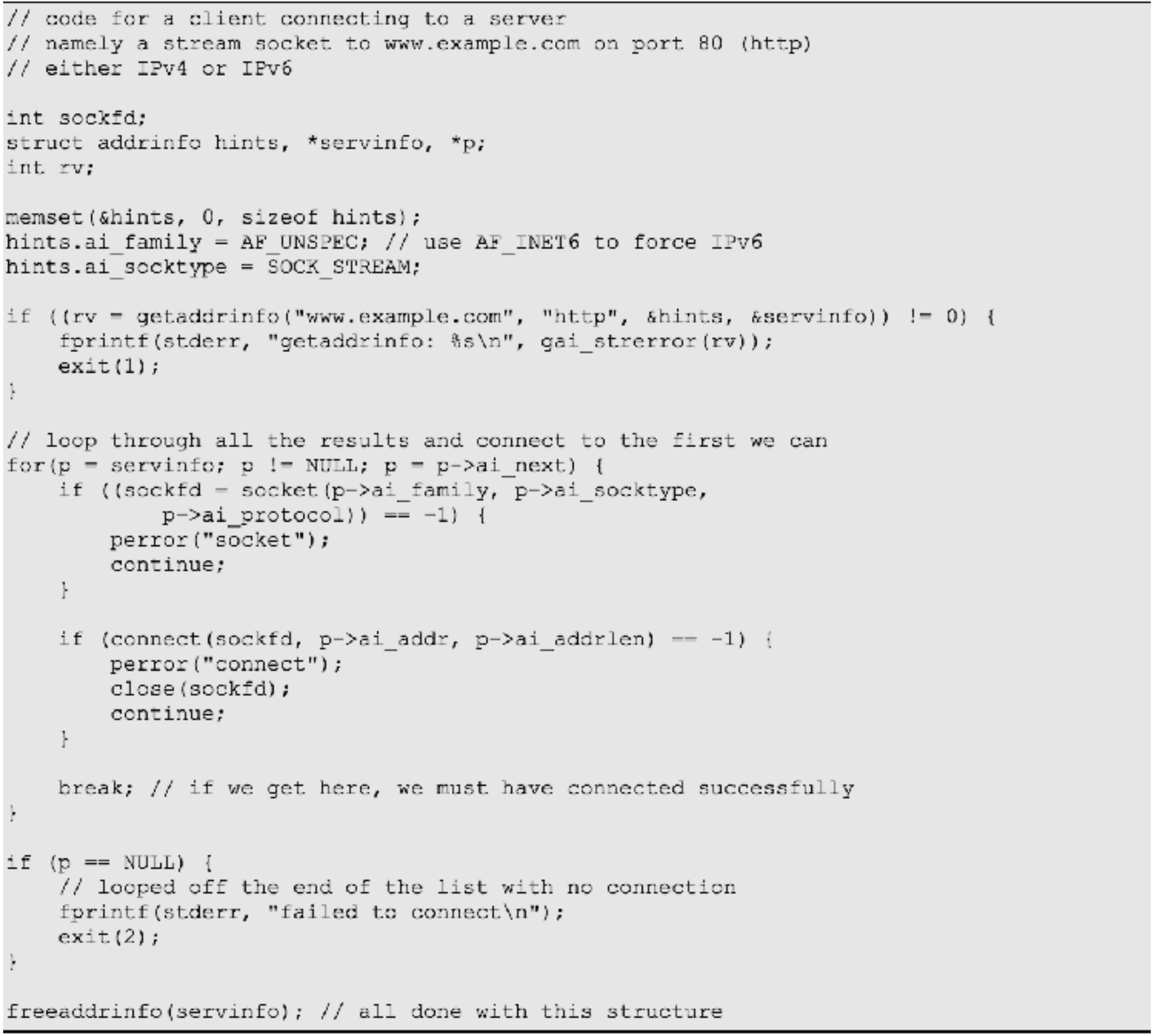
Client contacts the server to establish connection

- ✓Associate the socket with the server address/port
- ✓Acquire a local port number (assigned by the OS)
- ✓Request connection to server, who will hopefully accept

Establishing the connection

- ✓Arguments: socket descriptor, server address, and address
- ✓size
- ✓Returns 0 on success, and -1 if an error occurs

#Programming in C: Client

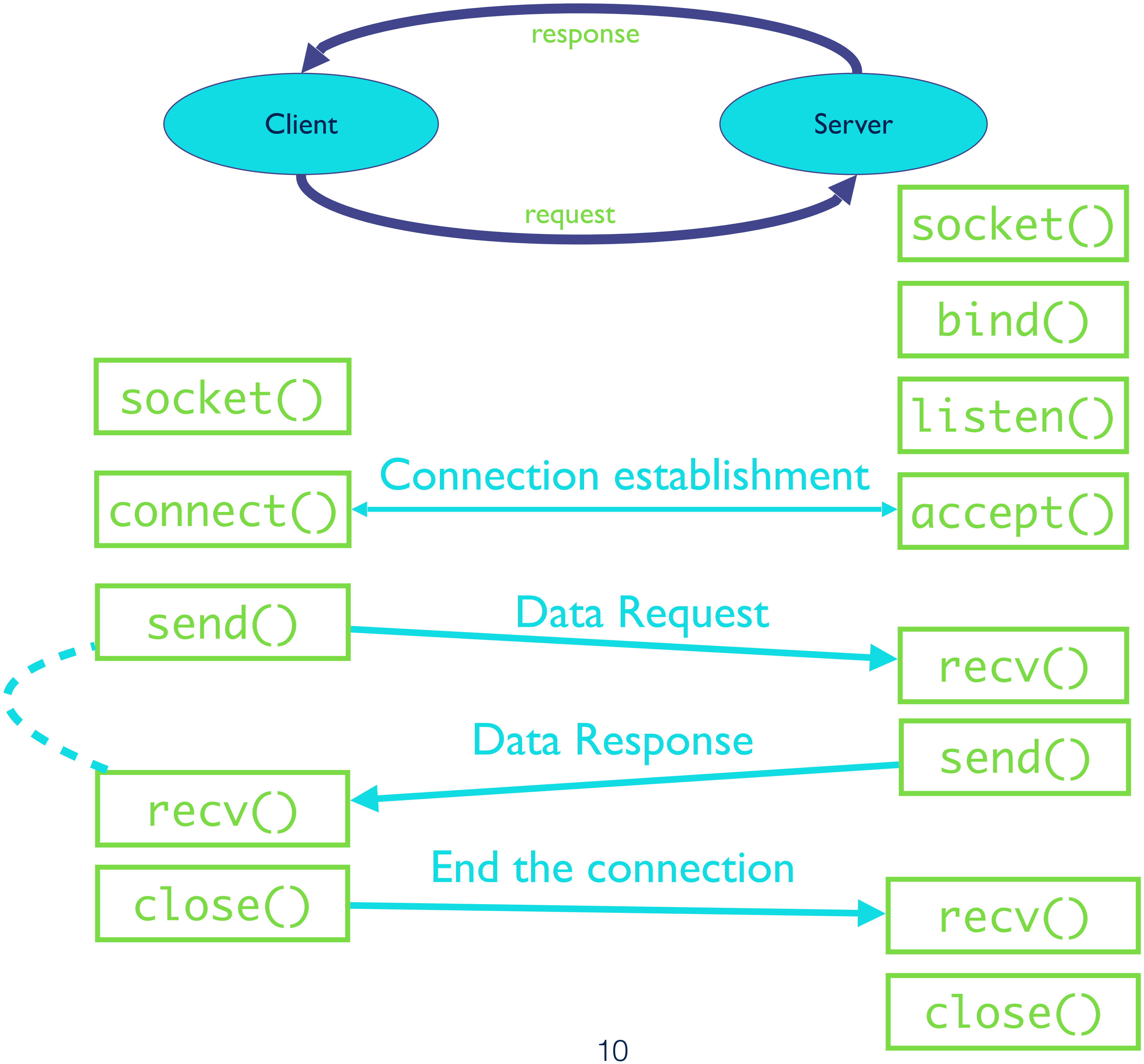


```c
// code for a client connecting to a server
// namely a stream socket to www.example.com on port 80 (http)
// either IPv4 or IPv6

int sockfd;
struct addrinfo hints, *servinfo, *p;
int rv;

memset(&hints, 0, sizeof hints);
hints.ai_family = AF_UNSPEC; // use AF_INET6 to force IPv6
hints.ai_socktype = SOCK_STREAM;

if ((rv = getaddrinfo("www.example.com", "http", &hints, &servinfo)) != 0) {
    fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(rv));
    exit(1);
}

// loop through all the results and connect to the first we can
for(p = servinfo; p != NULL; p = p->ai_next) {
    if ((sockfd = socket(p->ai_family, p->ai_socktype,
            p->ai_protocol)) == -1) {
        perror("socket");
        continue;
    }

    if (connect(sockfd, p->ai_addr, p->ai_addrlen) == -1) {
        perror("connect");
        close(sockfd);
        continue;
    }

    break; // if we get here, we must have connected successfully
}

if (p == NULL) {
    // looped off the end of the list with no connection
    fprintf(stderr, "failed to connect\n");
    exit(2);
}

freeaddrinfo(servinfo); // all done with this structure
```

#Server Programming: Servers Differ From Clients

- Passive open
  - Prepare to accept connections
  - … but don’t actually establish
  - … until hearing from a client
- Hearing from multiple clients
  - Allowing a backlog of waiting clients
  - ... in case several try to communicate at once
- Create a socket for each client
  - Upon accepting a new client
  - … create a new socket for the communication

#Typical Client-Server



response

Client

Server

request

socket()

bind()

listen()

accept()

socket()

connect()

Connection establishment

send()

Data Request

recv()

Data Response

send()

recv()

End the connection

close()

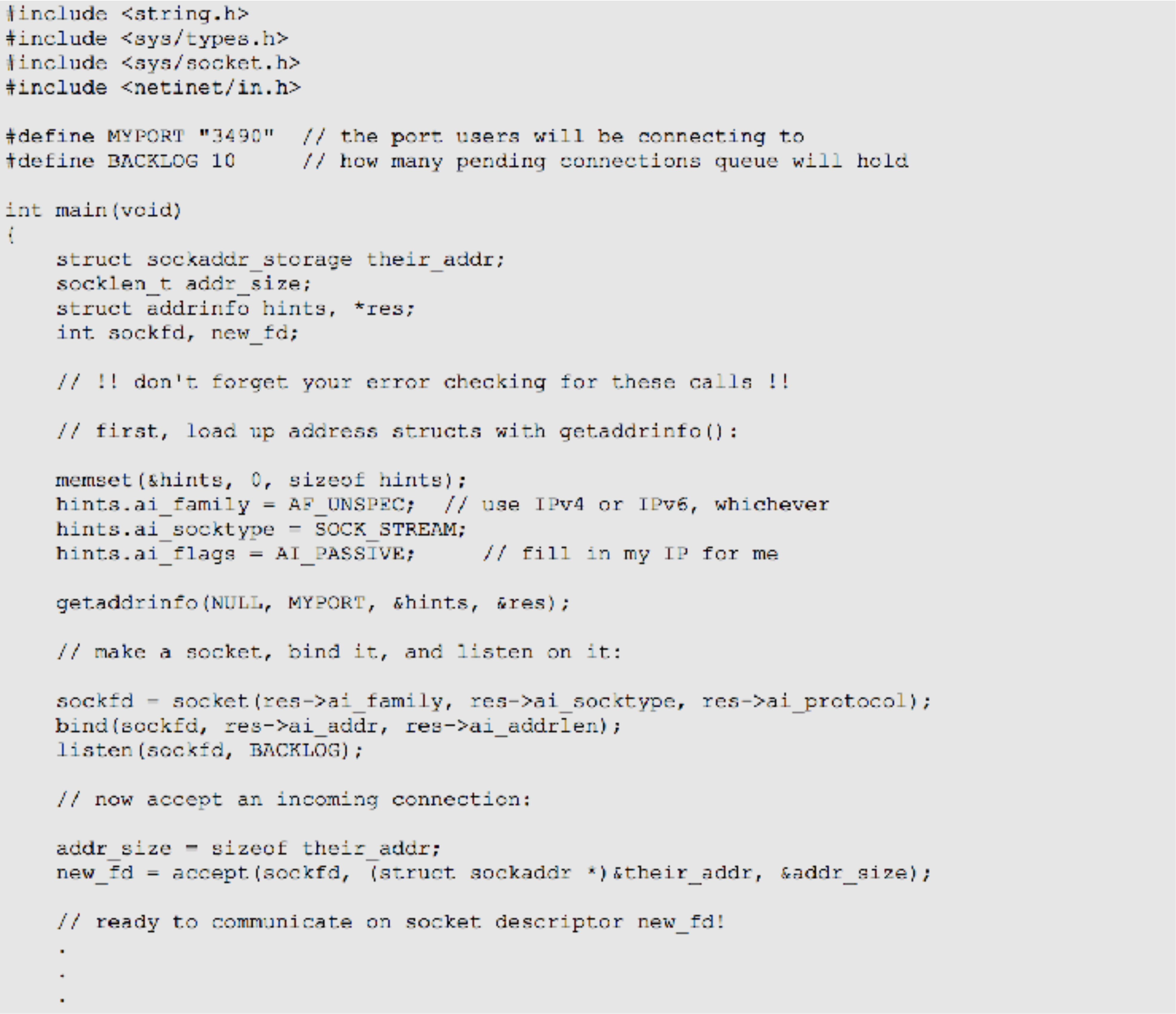
recv()

close()

#Server Programming: Preparing its Socket

- Create stream socket (socket())
- Bind port to socket (bind()) # local host and port
- Listen for new client (listen()) # How many clients?

#Programming in C: Server



```
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

#define MYPORT "3490"  // the port users will be connecting to
#define BACKLOG 10     // how many pending connections queue will hold

int main(void)
{
    struct sockaddr_storage their_addr;
    socklen_t addr_size;
    struct addrinfo hints, *res;
    int sockfd, new_fd;

    // !! don't forget your error checking for these calls !!

    // first, load up address structs with getaddrinfo():

    memset(&hints, 0, sizeof hints);
    hints.ai_family = AF_UNSPEC;  // use IPv4 or IPv6, whichever
    hints.ai_socktype = SOCK_STREAM;
    hints.ai_flags = AI_PASSIVE;     // fill in my IP for me

    getaddrinfo(NULL, MYPORT, &hints, &res);

    // make a socket, bind it, and listen on it:

    sockfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);
    bind(sockfd, res->ai_addr, res->ai_addrlen);
    listen(sockfd, BACKLOG);

    // now accept an incoming connection:

    addr_size = sizeof their_addr;
    new_fd = accept(sockfd, (struct sockaddr *)&their_addr, &addr_size);

    // ready to communicate on socket descriptor new_fd!
    .
    .
    .
```

#Server Programming: Handle No. of Clients

Many client requests may arrive

- Server cannot handle them all at the same time
- Server could reject the requests, or let them wait
- Define how many connections can be pending: backlog

Wait for clients

- int listen(int sockfd, int backlog)
- Arguments: socket descriptor and acceptable backlog
- Returns a 0 on success, and -1 on error

What if too many clients arrive?

- Some requests don’t get through
- The Internet makes no promises…
- And the client can always try again

#Server Programming: Accepting Client Connection

Now all the server can do is wait…

- Waits for connection request to arrive
- Blocking until the request arrives
- And then accepting the new request

Accept a new connection from a client

- int accept(int sockfd, struct sockaddr *addr, socketlen_t
- *addrlen)
- Arguments: socket descriptor, structure that will provide
- client address and port, and length of the structure
- Returns descriptor for a new socket for this connection

#Server Programming: Accepting Client Connection

Serializing requests is inefficient

- Server can process just one request at a time
- All other clients must wait until previous one is done
- May need to time share the server machine

Alternate between servicing different requests

- E.g. use multi-threading
- Or, start a new process to handle each request
- Allow the operating system to share the CPU across processes
- Or, some hybrid of these two approaches

#Client and Server: Cleaning House

Once the connection is open

- Both sides and read and write
- Two unidirectional streams of data
- In practice, client writes first, and server reads
- … then server writes, and client reads, and so on

Closing down the connection

- Either side can close the connection
- … using the close() system call

What about the data still "in flight"

- Data in flight still reaches the other end
- So, server can close() before client finishing reading

The End