University of Glasgow

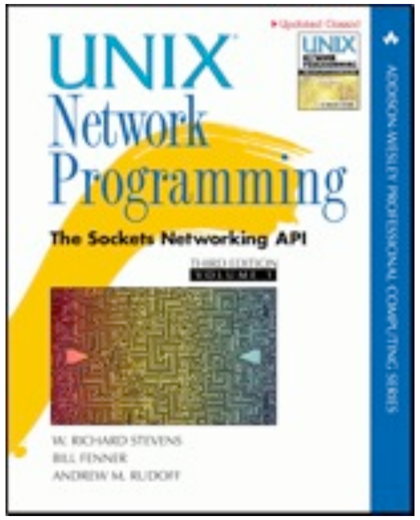
Network Programming in C: The Berkeley Sockets API

Networked Systems 3
Laboratory Sessions

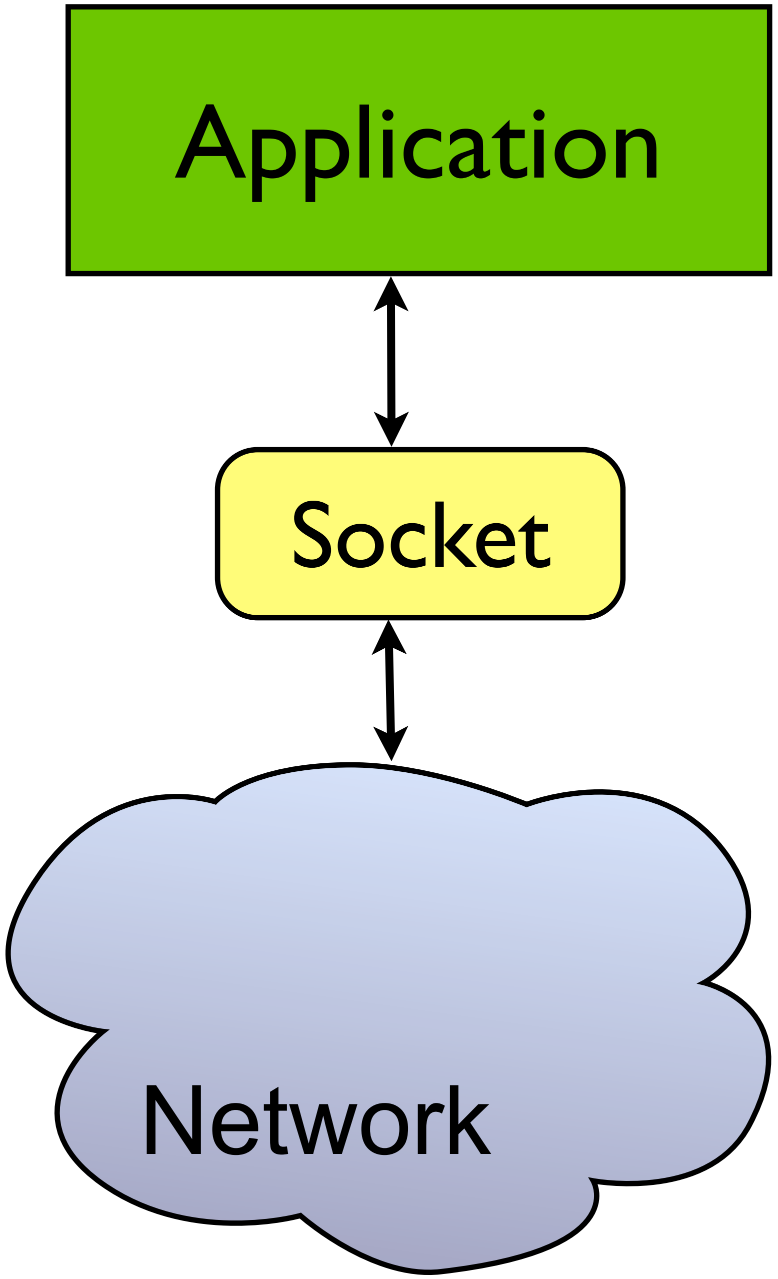
The Berkeley Sockets API

- Widely used low-level C networking API
- First introduced in 4.BSD Unix
  - Now available on most platforms: Linux, MacOS X, Windows, FreeBSD, Solaris, etc.
  - Largely compatible cross-platform

- Recommended reading:
  - Stevens, Fenner, and Rudoff, “Unix Network Programming volume 1: The Sockets Networking API”, 3rd Edition, Addison-Wesley, 2003.



Concepts

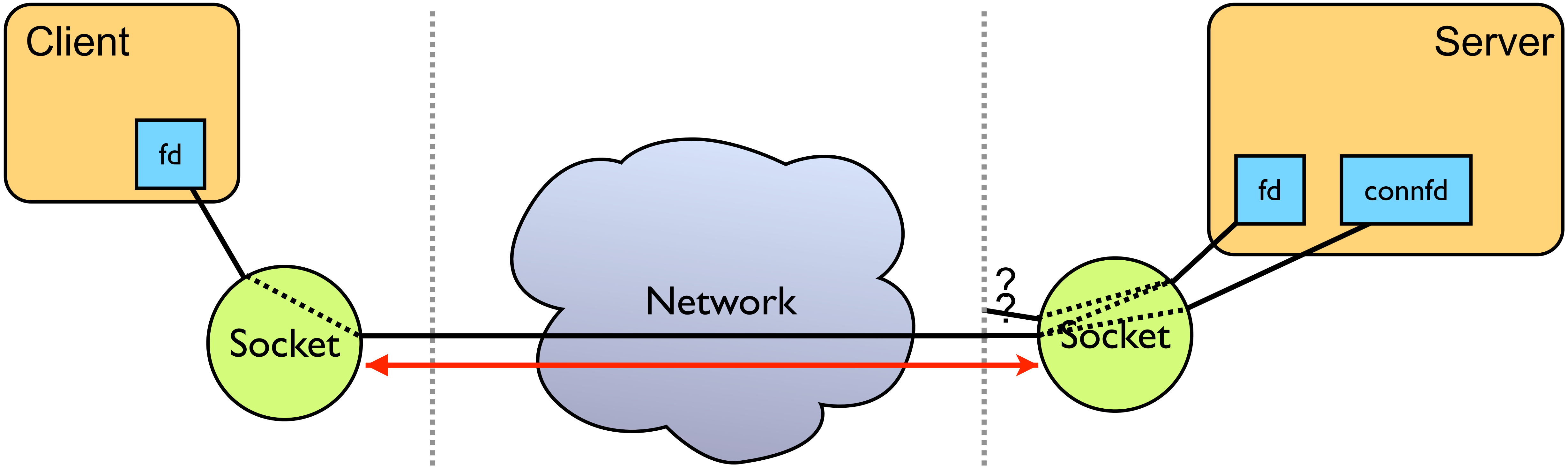


- Sockets provide a standard interface between network and application
- Two types of socket:
  - Stream – provides a virtual circuit service
  - Datagram – delivers individual packets
- Independent of network type:
  - Commonly used with TCP/IP and UDP/IP, but not specific to the Internet protocols
  - Only discuss TCP/IP sockets today

What is a TCP/IP Connection?

- A reliable byte-stream connection between two computers
  - Most commonly used in a client-server fashion:
    - The server listens on a well-known *port*
      - The *port* is a 16-bit number used to distinguish servers
      - E.g. web server listens on port 80, email server on port 25
    - The client connects to that port
  - Once connection is established, either side can write data into the connection, where it becomes available for the other side to read
- The Sockets API represents the connection using a *file descriptor*

TCP/IP Connection



```
int fd = socket(...)


connect(fd, ..., ...)

write(fd, data, datalen)

read(fd, buffer, buflen)

close(fd)
```

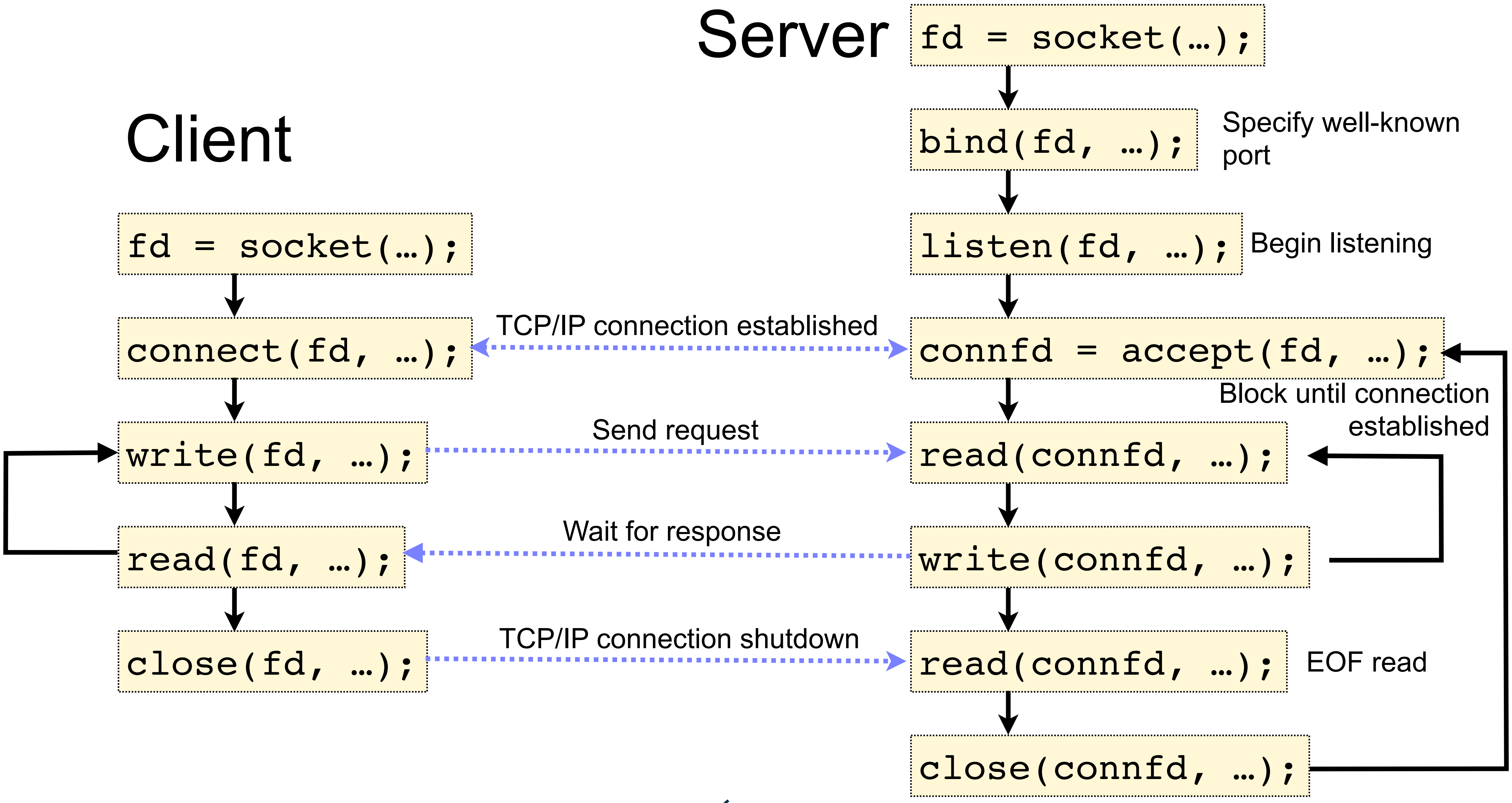
```
int fd = socket(...)
bind(fd, ..., ...)
listen(fd, ...)
connfd = accept(fd, ...)

read(connfd, buffer, buflen)

write(connfd, data, datalen)

close(connfd)
```

TCP/IP Connection



Server

```
fd = socket(…);
bind(fd, …);
listen(fd, …);
connfd = accept(fd, …);
read(connfd, …);
write(connfd, …);
read(connfd, …);
close(connfd, …);
```

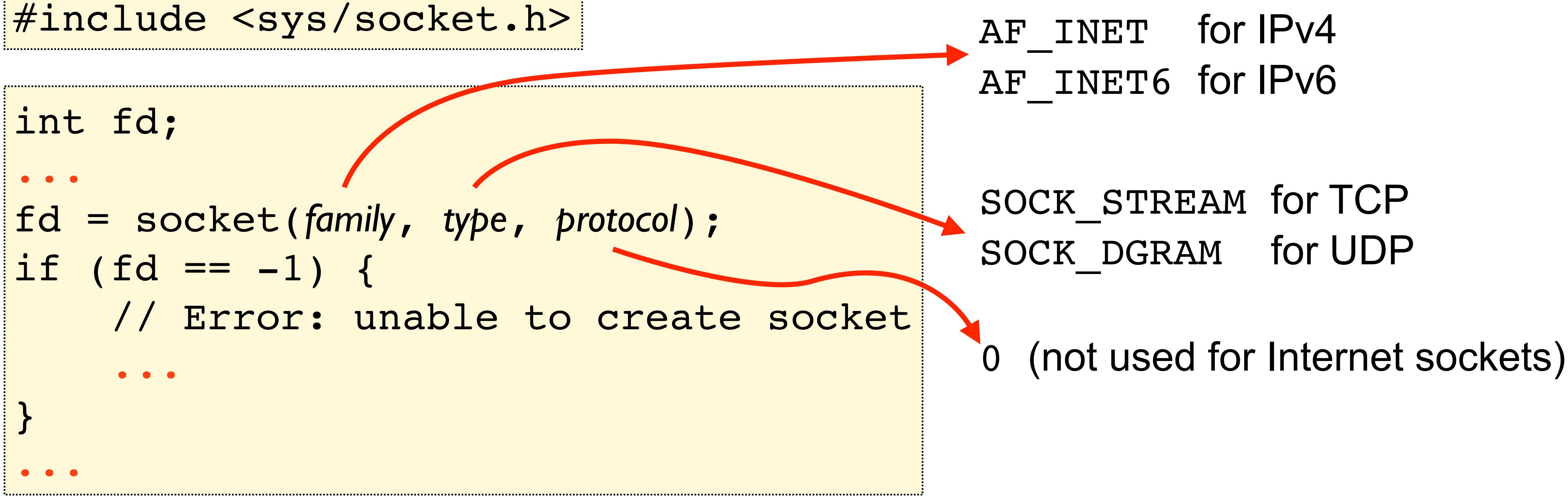
Client

```
fd = socket(…);
connect(fd, …);
write(fd, …);
read(fd, …);
close(fd, …);
```

Specify well-known port
Begin listening
TCP/IP connection established
Block until connection established
Send request
Wait for response
TCP/IP connection shutdown
EOF read

Creating a socket

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



```
int fd;
...
fd = socket(family, type, protocol);
if (fd == -1) {
    // Error: unable to create socket
    ...
}
...
```

`AF_INET` for IPv4
`AF_INET6` for IPv6

`SOCK_STREAM` for TCP
`SOCK_DGRAM` for UDP

`0` (not used for Internet sockets)

Create an unbound socket, not connected to network; can be used as either a client or a server

Handling Errors

Socket functions return `-1` and set the global integer variable `errno` on failure

```
fd = socket(family, type, protocol);
if (fd == -1) {
    // Error occurred; look at
    // errno to determine what
    // to do.
    ...
}
```

The Unix man pages list possible errors that can occur for each function

E.g. do "`man 2 socket`" in a terminal, and read the ERRORS section

Binding a Server Socket

- Bind a socket to a port on a network interface
  - Needed to run servers on a well-known port – with *addr* specified as INADDR_ANY
  - Not generally used on clients, since typically don’t care which port used

```
#include <sys/types.h>
#include <sys/socket.h>
...
if (bind(fd, addr, addrlen) == -1) {
    // Error: unable to bind
    ...
}
...
```

Listening for Connections

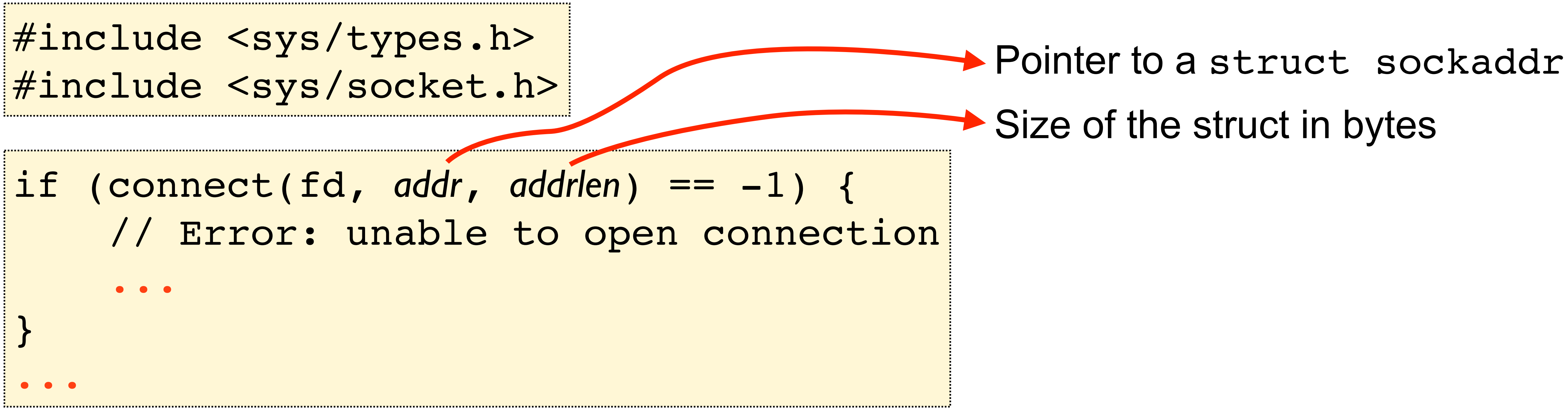
```
#include <sys/types.h>
#include <sys/socket.h>
```

```
if (listen(fd, backlog) == -1) {
    // Error
    ...
}
...
```

Tell the socket to listen for new connections

The *backlog* is the maximum number of connections the socket will queue up, each waiting to be `accept()`’ed

Connecting to a Server



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

```
if (connect(fd, addr, addrlen) == -1) {
    // Error: unable to open connection
    ...
}
...
```

Pointer to a `struct sockaddr`

Size of the struct in bytes

Tries to open a connection to the server

Times out after 75 seconds if no response

Specifying Addresses & Ports

- Must specify the address and port when calling `bind()` or `connect()`
  - The address can be either IPv4 or IPv6
  - Could be modelled in C as a union, but the designers of the sockets API chose to use a number of structs, and abuse casting instead

struct sockaddr

- Addresses specified via `struct sockaddr`
  - Has a data field big enough to hold the largest address of any family
  - Plus `sa_len` and `sa_family` to specify the length and type of the address
  - Treats the address as an opaque binary string

```
struct sockaddr {
    uint8_t        sa_len;
    sa_family_t    sa_family;
    char           sa_data[22];
};
```

struct sockaddr_in

- Two variations exist for IPv4 and IPv6 addresses
  - Use struct sockaddr_in to hold an IPv4 address
  - Has the same size and memory layout as struct sockaddr, but interprets the bits differently to give structure to the address

```
struct in_addr {
    in_addr_t      s_addr;
};

struct sockaddr_in {
    uint8_t        sin_len;
    sa_family_t    sin_family;
    in_port_t      sin_port;
    struct in_addr sin_addr;
    char           sin_pad[16];
};
```

struct sockaddr_in6

- Two variations exist for IPv4 and IPv6 addresses
  - Use struct sockaddr_in6 to hold an IPv6 address
  - Has the same size and memory layout as struct sockaddr, but interprets the bits differently to give structure to the address

```
struct in6_addr {
    uint8_t         s6_addr[16];
};

struct sockaddr_in6 {
    uint8_t         sin6_len;
    sa_family_t     sin6_family;
    in_port_t       sin6_port;
    uint32_t        sin6_flowinfo;
    struct in6_addr sin6_addr;
};
```

Working with Addresses

- Work with either `struct sockaddr_in` or `struct sockaddr_in6`
- Cast it to a `struct sockaddr` before calling the socket routines

```
struct sockaddr_in  addr;
...
// Fill in addr here
...
if (bind(fd, (struct sockaddr *) &addr, sizeof(addr)) == -1) {
    ...
```

Creating an Address: INADDR_ANY

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

- Servers often just want to listen on the default address – do this using INADDR_ANY for the address passed to `bind()`
- Convert port number using `htons(…)`

```
struct sockaddr_in  addr;
...
addr.sin_addr.s_addr = INADDR_ANY;
addr.sin_family      = AF_INET;
addr.sin_port        = htons(80);

if (bind(fd, (struct sockaddr *)&addr, sizeof(addr)) == -1) {
    ...
```

Creating an Address: Manually

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

- Clients want to connect to a specif[illegible] address – can use `inet_pton()` to cr[illegible]address, if you know the numeric I[illegible]s
- Convert port numb[illegible]g `htons(…)`

```
struct sockaddr_in  addr;
...
inet_pton(AF_INET, "1[illegible].240.1", &addr.sin_addr);
addr.sin_family [illegible]ET;
addr.sin_port[illegible]tons(80);

if ([illegible](fd, (struct sockaddr *)&addr, sizeof(addr)) == -1) {
```

DON'T DO THIS – USE THE DNS INSTEAD

Creating an Address: DNS

- Prefer using DNS names to raw IP addresses
  - Use `getaddrinfo()` to look-up name in DNS
  - Returns a linked list of `struct addrinfo` values, representing addresses of the host

```
struct addrinfo {
    int              ai_flags;     // input flags
    int              ai_family;    // AF_INET, AF_INET6, ...
    int              ai_socktype;  // IPPROTO_TCP, IPPROTO_UDP
    int              ai_protocol;  // SOCK_STREAM, SOCK_DRAM, ...
    socklen_t        ai_addrlen;   // length of socket-address
    struct sockaddr *ai_addr;      // socket-address for socket
    char            *ai_canonname; // canonical name of host
    struct addrinfo *ai_next;      // pointer to next in list
};
```

Connecting via a DNS Query

```
struct addrinfo   hints, *ai, *ai0;
int i;

memset(&hints, 0, sizeof(hints));
hints.ai_family     = PF_UNSPEC;
hints.ai_socktype   = SOCK_STREAM;
if ((i = getaddrinfo(“www.google.com”, "80", &hints, &ai0)) != 0) {
    printf("Unable to look up IP address: %s", gai_strerror(i));
    ...
}

for (ai = ai0; ai != NULL; ai = ai->ai_next) {
    fd = socket(ai->ai_family, ai->ai_socktype, ai->ai_protocol);
    if (fd == -1) {
        perror("Unable to create socket");
        continue;
    }

    if (connect(fd, ai->ai_addr, ai->ai_addrlen) == -1) {
        perror("Unable to connect");
        close(fd);
        continue;
    }
    ...success, use the connection
    break;
}
if (ai == NULL) {
    // Connection failed, handle the failure...
}
```

Accepting Connections

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

```
int                connfd;
struct sockaddr_in cliaddr;
socklen_t          cliaddrlen = sizeof(cliaddr);
...
connfd = accept(fd, (struct sockaddr *) &cliaddr, &cliaddrlen);
if (connfd == -1) {
    // Error
    ...
}
...
```

Accepts a connection, returns *new* file descriptor for the connection (`connfd`) and client address (`cliaddr`)

Accepting Connections

- A TCP/IP server may have multiple connections outstanding
  - Can `accept()` connections one at a time, handling each request in series
  - Can `accept()` connections and start a new thread for each, allowing it to process several in parallel
- Each call to `accept()` returns a new file descriptor

Reading and Writing Data

```
#define BUFLEN 1500
...
ssize_t i;
ssize_t rcount;
char    buf[BUFLEN];
...
rcount = read(fd, buf, BUFLEN);
if (rcount == -1) {
    // Error has occurred
    ...
}
...
for (i = 0; i < rcount; i++) {
    printf(“%c”, buf[i]);
}
```

- The `read()` call reads *up to* BUFLEN bytes of data from connection – blocks until data available
- Returns actual number of bytes read, or –1 on error
- Data is *not* null terminated

Handling Multiple Sockets

The `select()` call tells you which of a group of sockets has data available to read

```
#include <sys/select.h>
...
int            fd1, fd2;
fd_set         rfds;
struct timeval timeout;
...
timeout.tv_sec  = 1;   // 1 second timeout
timeout.tv_usec = 0;

FD_ZERO(&rfds);
FD_SET(fd1, &rfds);
FD_SET(fd2, &rfds);

int rc = select(max(fd1, fd2) + 1, &rfds, NULL, NULL, &timeout);
if (rc == 0) ... // timeout
if (rc > 0) {
    if (FD_ISSET(fd1, &rfds)) {
        ... // Data available to read on fd1
    }
    if (FD_ISSET(fd2, &rfds)) {
        ... // Data available to read on fd2
    }
}
if (rc < 0) ... // error
```

Reading and Writing Data

```
char data[] = “Hello, world!”;
int  datalen = strlen(data);
...
if (write(fd, data, datalen) == -1) {
    // Error has occurred
    ...
}
...
```

The `write()` call sends data over a socket; blocks until all data can be written

Returns actual number of bytes written, or `-1` on error

Reading and Writing Data

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main()
{
        char     x[] = "Hello, world!";
        char    *y   = malloc(14);

        sprintf(y, "Hello, world!");

        printf("x = %s\n", x);
        printf("y = %s\n", y);

        printf("sizeof(x) = %d\n", sizeof(x));
        printf("sizeof(y) = %d\n", sizeof(y));

        printf("strlen(x) = %d\n", strlen(x));
        printf("strlen(y) = %d\n", strlen(y));

        return 0;
}
```

What gets printed?

Why?

Closing a Socket

```
#include <unistd.h>
```

```
close(fd);
```

Close and destroy a socket

Close the file descriptor for each connection, then the file descriptor for the underlying socket

Questions?