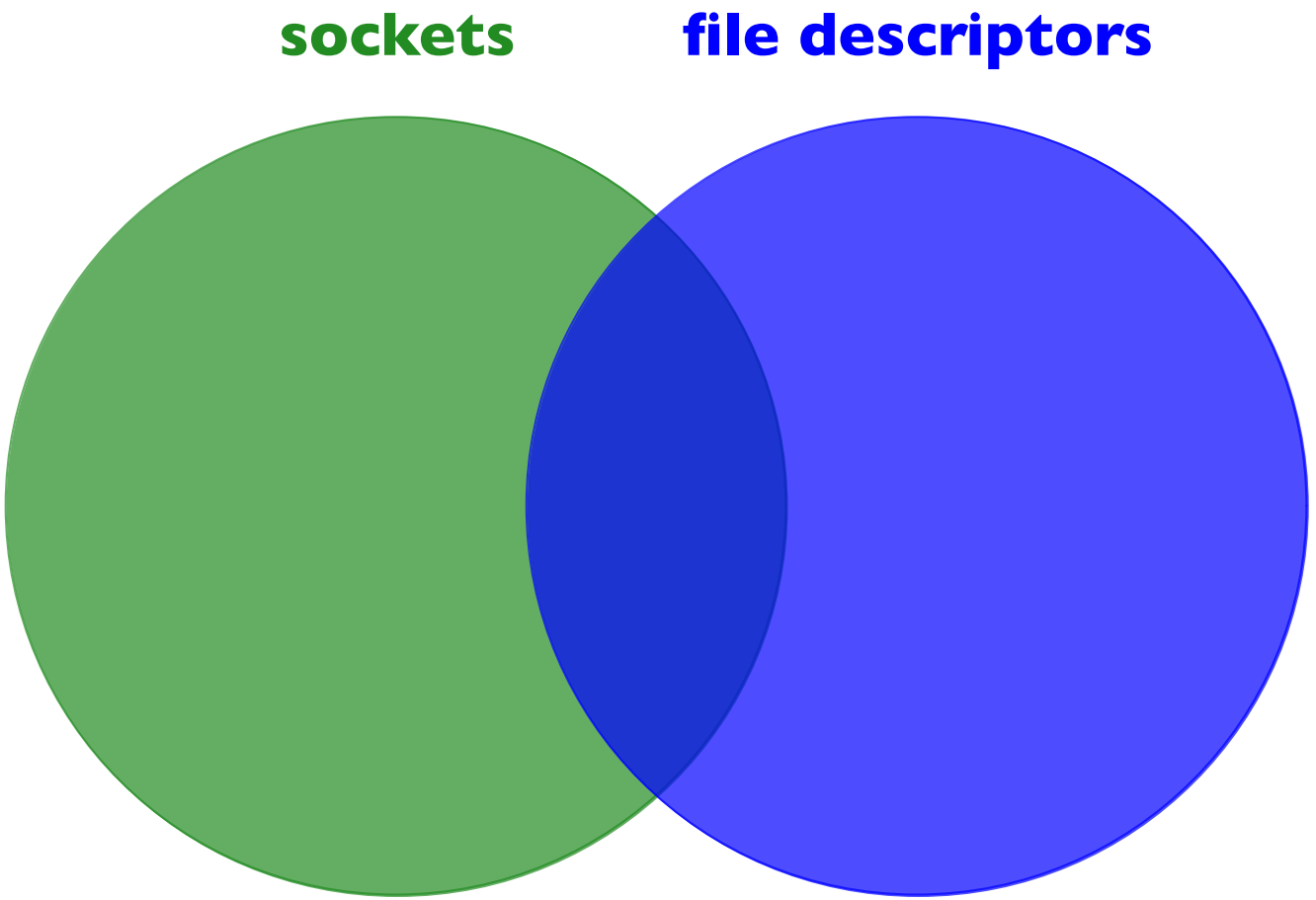
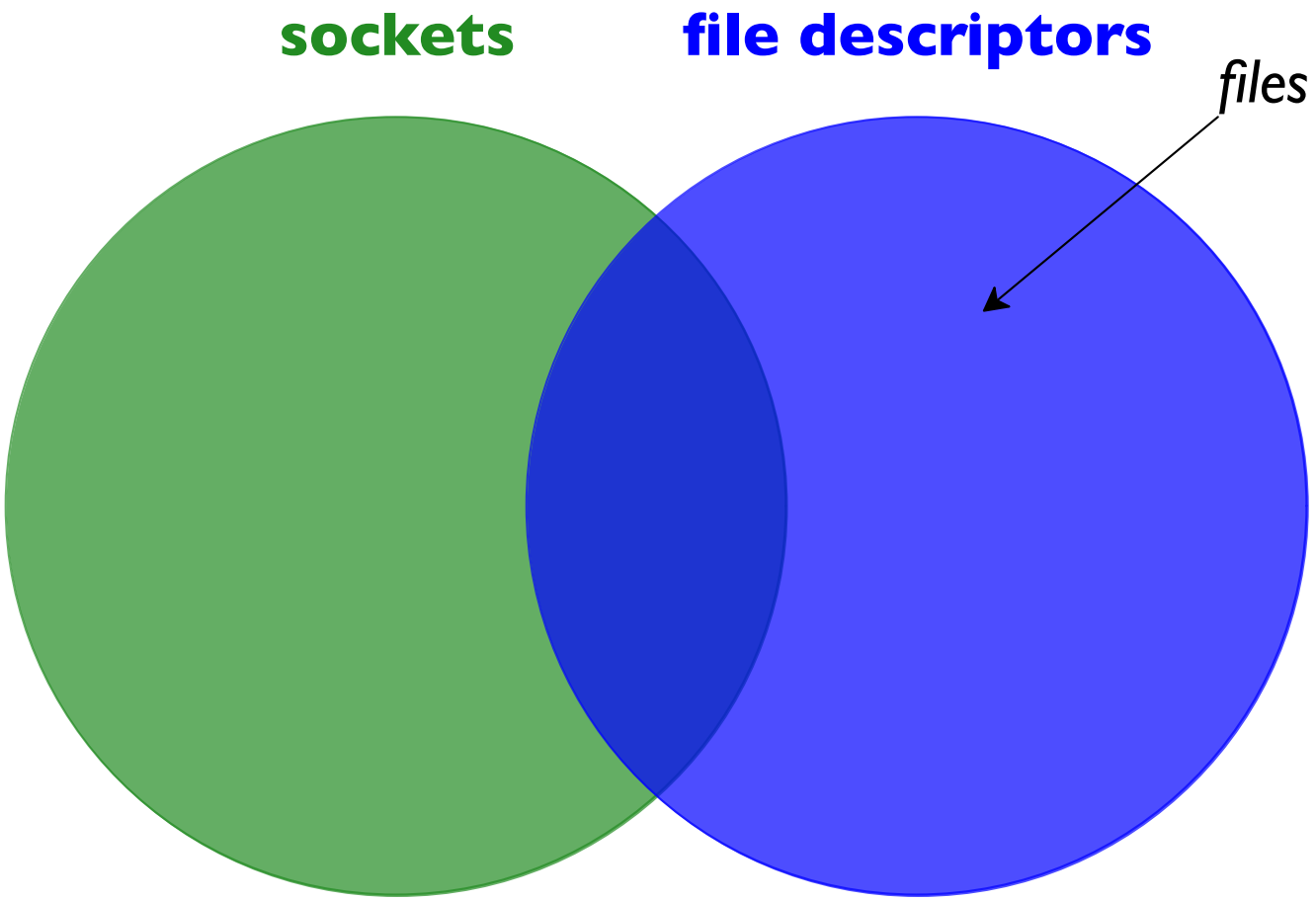
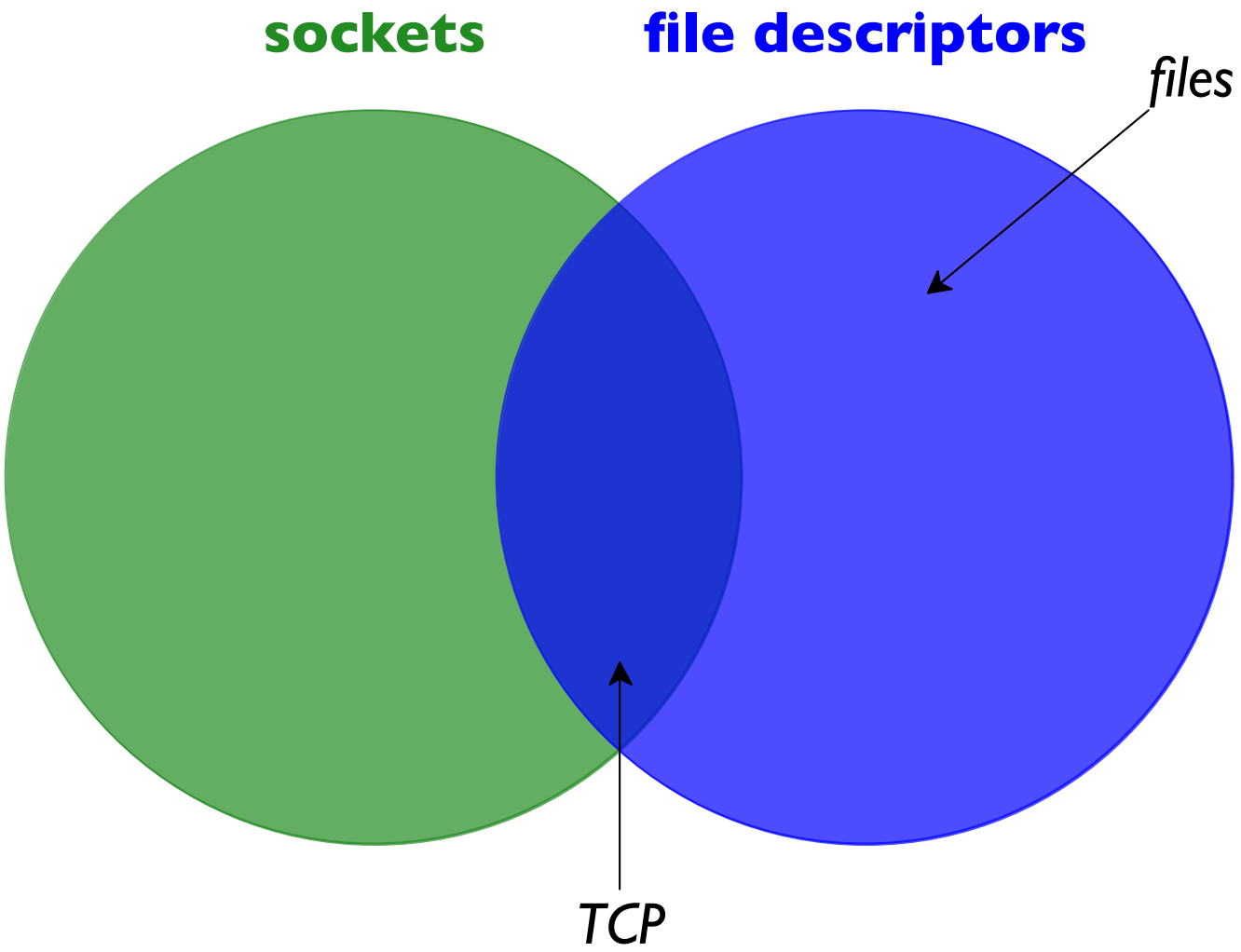
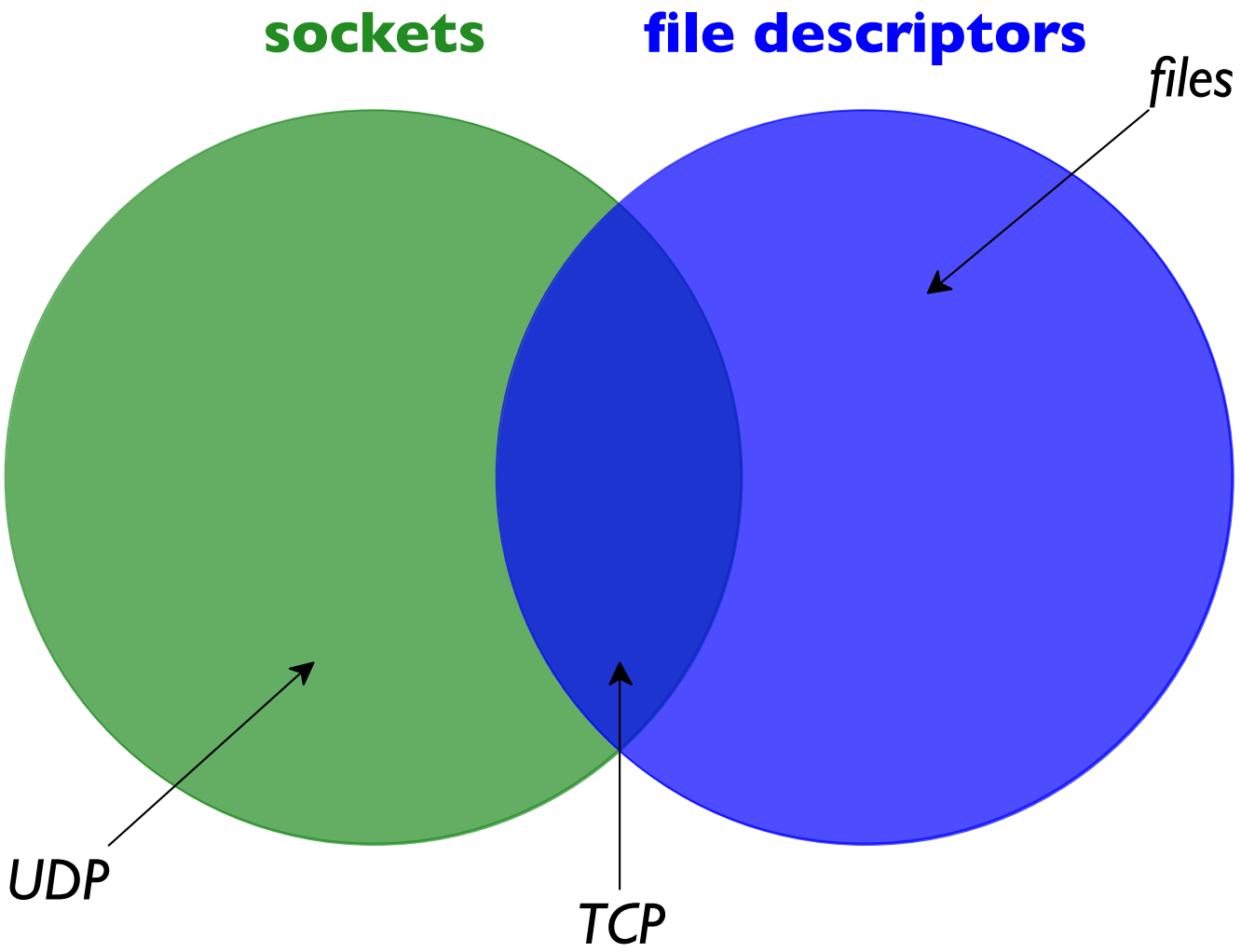
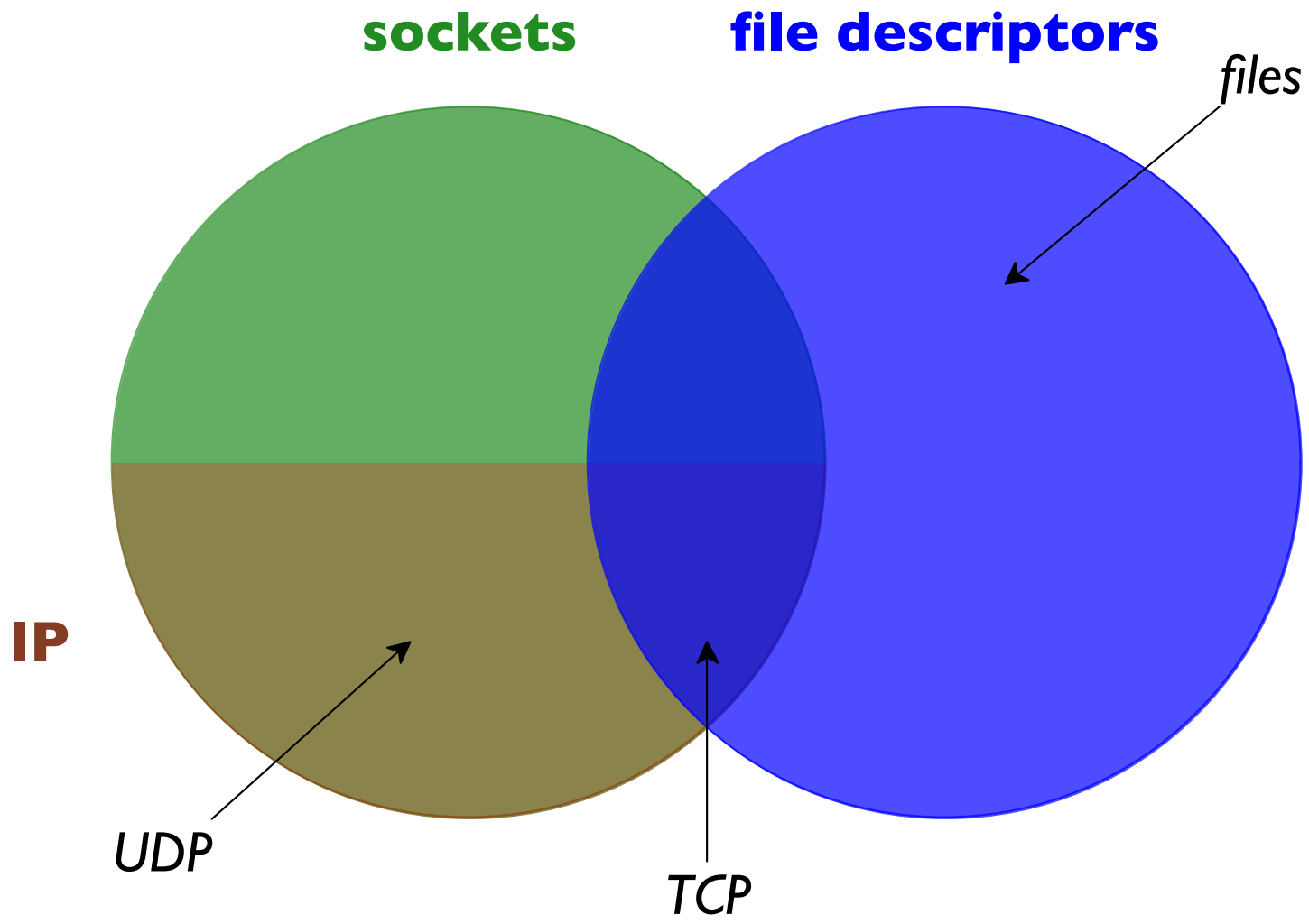

sockets
file descriptors


sockets
file descriptors
files

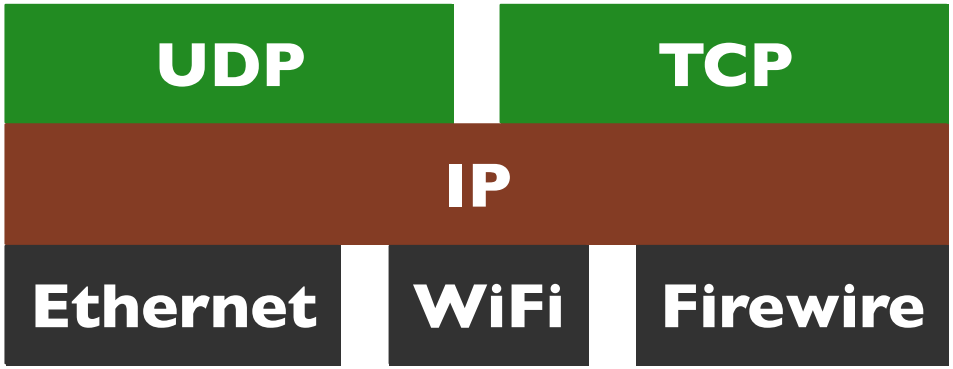

sockets
file descriptors
files
TCP


sockets
file descriptors
files
UDP
TCP


sockets
file descriptors
files
IP
UDP
TCP

IP

IP is an addressing scheme and packet format



IP

- Each node has a 32-bit address written in four parts, e.g.

  `192.168.1.100`

- "Directly" connected to other addresses that match within the ***netmask***

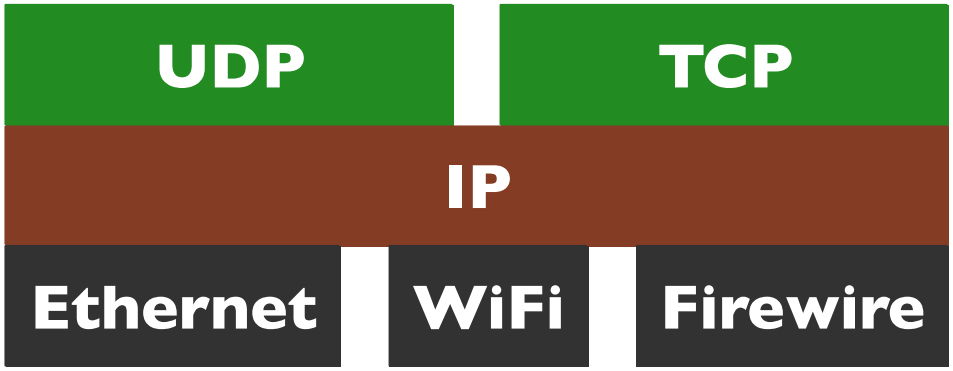
  netmask **`255.255.255.0`**:

  ⇒ **`192.168.1.100`** on subnet of **`192.168.1.28`**

  ⇒ **`192.168.1.100`** not on subnet of **`192.168.2.100`**

Interfaces

A machine may have multiple IP ***interfaces***



Try running `ifconfig` or `ipconfig`

Getting an Address

- Static addressing: user/administrator tells the OS to use a particular address and netmask
- ***DHCP***: machine gets address from a server to which it is "directly" connected
  - Exploits IP netmask-constrained broadcast without knowing the subnet address
- ***NAT*** makes multiple nodes look like one

UDP and TCP

Applications practically never create raw IP packets

- An exception: **`ping`**

Primarily two choices for layers over IP:

- ***UDP*** packet-based, not reliable
- ***TCP*** stream-based, reliable

IP Message

The destination of an IP message is

- a host address
- a protocol (e.g., TCP or UDP)
- a port number

Port numbers range from 1 to 65535

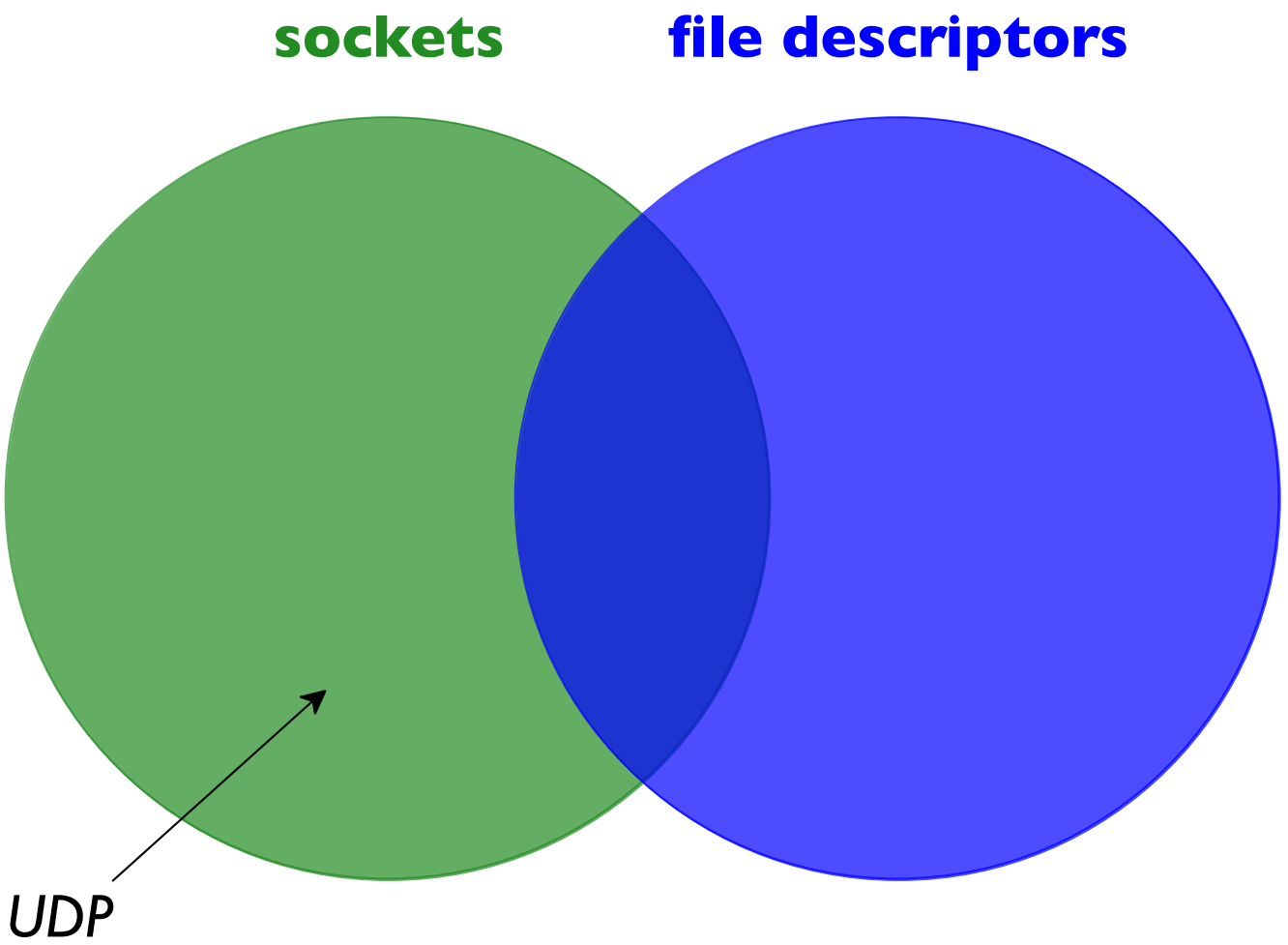
Port numbers below 1024 require special privilege

Creating a Socket

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

- domains: **`PF_INET`**, **`PF_UNIX`**, ...
- types: **`SOCK_STREAM`**, **`SOCK_DGRAM`**, ...
- protocols: **`"tcp"`**, **`"udp"`**, ...
  convert string to a number with **`getprotoent()`**

UDP



Sending a UDP Message

```
ssize_t sendto(int socket,
               void *buffer, size_t length,
               int flags,
               struct sockaddr *dest_addr,
               socklen_t dest_len);
```

Need to build an address...

Binding a Socket

```
int bind(int socket,
         struct sockaddr *address,
         socklen_t address_len);
```

Need to build an address...

Receiving a UDP Message

```
ssize_t recv(int socket,
             void *buffer, size_t length,
             int flags);
```

Computing an IP Address

```
struct sockaddr_in addr;
```

- set **`serv_addr.sin_family`** to **`AF_INET`**
- set **`serv_addr.sin_port`** to a port number
- set **`serv_addr.sin_addr.s_addr`** to a numerical IP address

Getting a numerical address:

- Convert a hostname string with **`gethostbyname()`**
- Use **`INADDR_ANY`** with **`bind()`**

See **udp_recv.c**, **udp_send.c**, **udp_recvfrom.c**, **udp_lh_recv.c**, **udp_sendfrom.c**

Binding to a Destination

```
int connect(int socket,
            struct sockaddr *address,
            socklen_t address_len);

ssize_t send(int socket,
             const void *buffer, size_t length,
             int flags);
```

For UDP, **`connect()`** is just a convenience

See **`udp_many_send.c`**

UDP Summary

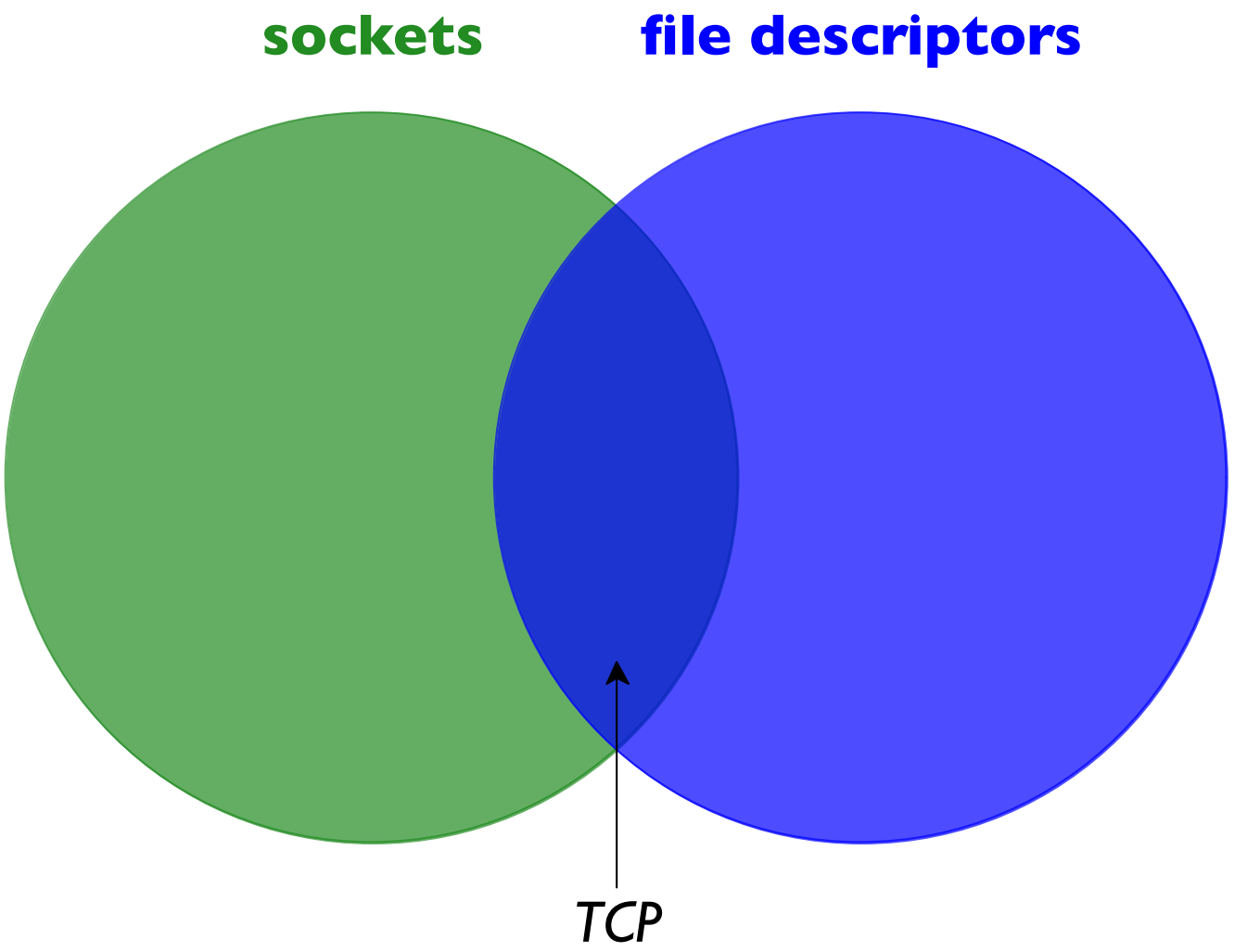
- About as simple as possible
- No guarantees about delivery
- No guarantees on order of messages

UDP in the OS

OS needs to maintain

- A mapping from port numbers to process+socket
  - Handle incoming messages
  - Disallow multiple uses of port numbers
- Little buffering for messages going our or coming in

TCP



Creating a TCP Connection

Client:

- **`socket()`** and **`connect()`** *N* times

  socket works with **`send()`**, **`recv()`**, **`read()`**, and **`write()`**

Server:

- **`socket()`**, **`bind()`**, and **`listen()`** once

  socket works only with **`accept()`**

- **`accept()`** [implicitly creates new socket] *N* times

  socket works with **`send()`**, etc.

Listening and Accepting TCP Connections

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

int accept(int socket,
           struct sockaddr *address,
           socklen_t *address_len);
```

See **tcp_server.c**, **tcp_client.c**

TCP Streams

A TCP connection allows both read and write

- **`close()`** ends both directions
- **`shutdown()`** ends one direction
  - shutdown output ⇒ other end recieves EOS
  - shutdown input ⇒ no message

See **`tcp_server2.c`**, **`tcp_client2.c`**

Reliable Data Delivery

When an IP packet is lost for a TCP connection, TCP re-sends the data

- Requires an ACK from other end
- Messages have IDs for ACKs and ordering

Resending uses ***exponential backoff***:

- Send message, wait *N* msecs for reply...
- Re-send message, wait 2*N* msecs for reply...
- Re-re-send message, wait 4*N* msecs for reply...

An ACK is needed even for a shutdown EOS

TCP in the OS

A program could make a TCP connection, send data, **`close()`** the connection, and exit

- OS typically allows the close and exit immediately
- Some TCP work work may survive the process, such as EOS ACKs

Absent an EOS, how does the OS know that no more data will arrive on a TCP connection?

- OS hedges with connection in **`TIME_WAIT`** state
- **`SO_REUSEADDR`** truncates **`TIME_WAIT`** state on listeners

See **`server.c`**