Week 10 2

CSC209 Fall 2023

Dr. Demetres (dee) Kostas

November 23, 2023

Contents

1 Announcements

- Lab 10 this week
  - Lab 11 will be the last!
- After the lecture
  - you should be able to
  - understand what
    - *needs to be done*
      - for A4

2 Sockets

- recalling pipes
  - when we said `pipe()`
    - the system created
  - some sort of buffer
    - and two `fd`'s
      - one to put data
      - one to get data

2.1 What does `socket` do?

- create something similar...
- but gives us 1 `fd`!

- there are various implementations
  - but the socket
  - is essentially an endpoint
    - (the `fd`)
  - that we can read/write data
    - **both** on same `fd`

2.2 (Continued)

- this endpoint automatically
  - formats data going in
  - and converts data before reading
    - according to the args:
      - **domain**
      - IPv4 == `AF_INET`
      - **type**
      - TCP == `SOCK_STREAM`
      - **protocol**
      - `0` for CSC209

2.3 So we have a first system call

- one for creating an end point
  - for both read and write
- represented by a single `fd`
- such that *the system*
  - will basically *package*
    - the data we write
  - or *unwrap* the data
    - that comes in for read
- according to the format

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

- return value is the fd
  - or `-1` on error
- We mentioned two "packets" (type)
  - TCP and UDP
    - these are the types
      - `SOCK_STREAM` (TCP)
      - guaranteed error detection
      - `SOCK_DGRAM` (UDP)
      - no guarantees

2.5 So... Internet now?

- not quite
- we've made a socket
  - like making a pipe
- let's connect it to our network

3 The "server" end

- this is the end
- that does not *initiate*
  - the "conversation"
  - "passive" end
- it works by waiting
  - for connections

3.1 `bind()`

- once we have a socket...
  - "structure"
- we can manually assign it
  - to a specific port (and address)
    - to wait for connection
- basically, the socket's `fd`
  - needs to be connected
  - to a specific port
    - on the network device
    - so *others* can remote access

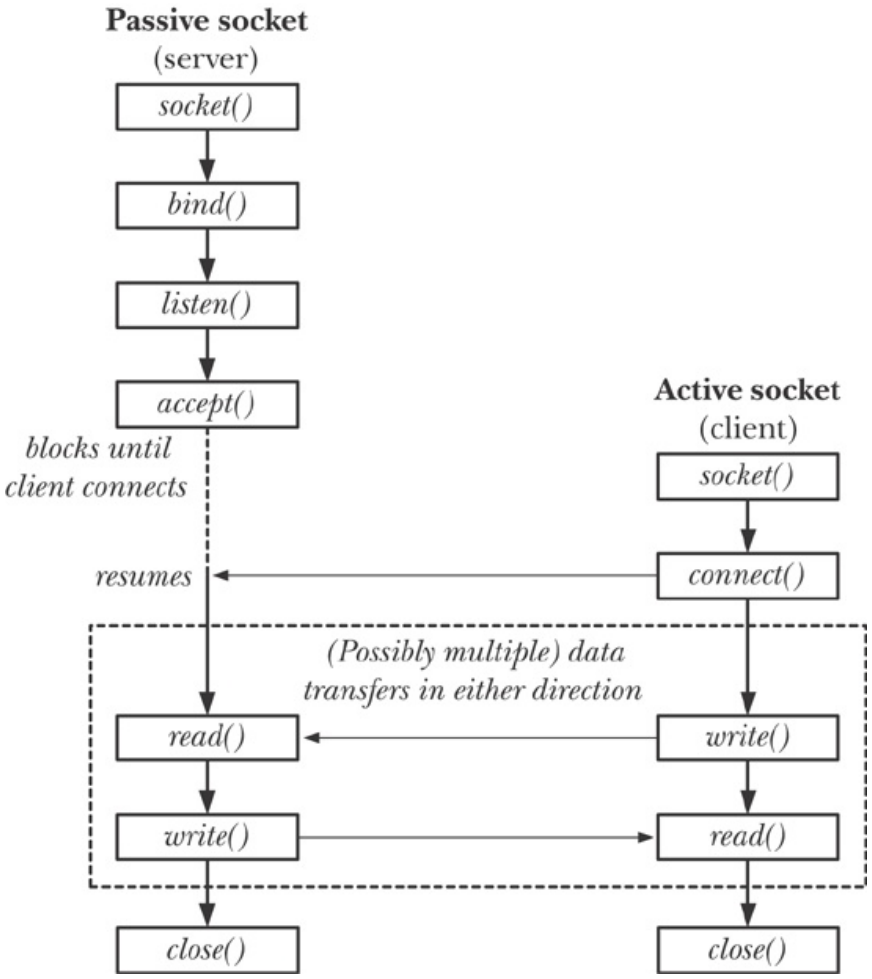
3.2 `listen()`

- alternatively just say this
- `bind` forces a *specific* port
  - but we could just tell the system
  - that this socket
    - should *listen* for data
- similar to `bind`
  - but the system just assigns
    - the socket to a port on the
    - network interface randomly

4 The "client" end

- this end *requires*
  - a "server" to be available
  - and **connects** to it
- it typically
  - just uses whatever free port
    - is available
  - it doesn't affect the user

5 Architecture lifecycle



6 System calls in detail

- let's look at the *additional*
  - system calls that one socket
    - needs to connect to another
- we understand the architecture

- we just need to use
    - the available calls correctly
- again the focus is TCP/IPv4

6.1 We saw `socket`

- 9 times out of 10
    - it will look like this

```
int socket_fd = socket(AF_INET, SOCK_STREAM, 0);
```

6.2 `int bind(int sockfd, sockaddr *addr, socklen_t addrlen)`

- `-1` on error, and `sockfd` we know
- `socklen_t` is basically an `int`
    - its a `typedef` for holding integers
- `addrlen` is just the number
    - of bytes used for addr
        - to **not** overflow

```
struct sockaddr_in {
    sa_family_t     sin_family;     /* AF_INET */
    in_port_t       sin_port;       /* Port number */
    struct in_addr  sin_addr;       /* IPv4 address */
};
```

6.3 Code snippet from Lab 10

```
struct sockaddr_in *init_server_addr(int port) {
    struct sockaddr_in *addr = malloc(sizeof(struct sockaddr_in));

    // Allow sockets across machines.
    addr->sin_family = AF_INET;
```

```
    // The port the process will listen on.
    addr->sin_port = htons(port);

    // Clear this field; sin_zero is used for padding for the struct.
    memset(&(addr->sin_zero), 0, 8);

    // Listen on all network interfaces.
    addr->sin_addr.s_addr = INADDR_ANY;

    return addr;
}
```

6.4 `htons(port_num)`

- remember when we talked about
  - byte order
- `int` are 4 bytes
  - but left-right, or right-left?
- `htons`
  - ensures the 16-bit port number
    - isn't accidentally lost

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

- `0` on success, `-1` error
- listen's job is to enable
  - active/client connections
  - but it doesn't initiate
    - actual data transfer
      - this is for `accept()`
- As such, `backlog` is for
  - how many pending connections
  - the system must manage

6.6 See Lab 10 for helper functions

```
struct sockaddr_in *init_server_addr(int port);
int set_up_server_socket(struct sockaddr_in *self, int num_queue);
```

6.7 `int accept()`

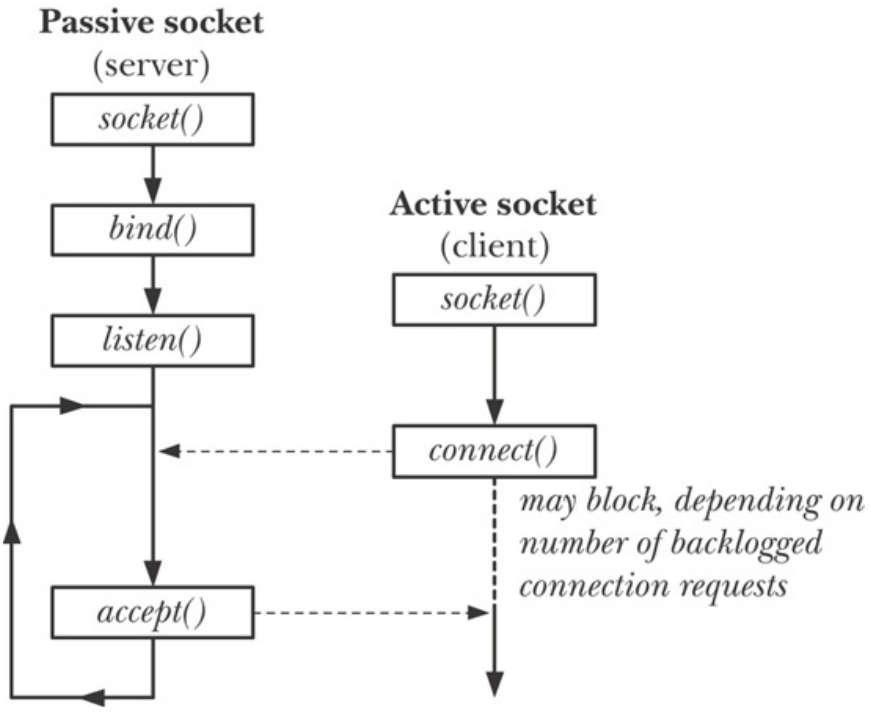
- blocks until there is connection
- creates **another** socket
  - the first one is to handle
    - connections
- *this one* is to read/write data
- will return a new fd for R/W
  - on success, or `-1` on error

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

6.8 `accept()` continued

- `addr`
  - is *filled* by `accept`!
- what about `*addrlen`?
  - `sizeof(addr)`
  - so the system doesn't overflow
- this holds the info
  - about the **active** peer
    - (client)
- `NULL` and 0 are valid
  - if we don't care

6.9 Connection lifecycle



6.10 `int connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen)`

- same as bind really
- but for the "active" side
  - client
- `0` on success, `-1` error

6.11 `close(sockfd)`

- if either side closes
  - system cleans up socket resources
- if *the other side* reads

    - as in, they haven't closed
        * they will receive EOF
- if *the other side* write
    - a `SIGPIPE` signal is raised

7 Final remarks

- a passive process on one machine
    - a.k.a *a server*
- plus any number of active
    - connectors
    - a.k.a. *clients*
- make up, what you might call
    - a client-server application
        * that uses TCP/IP

7.1 TCP/IP

- take the networking course
    - if you are finding this interesting
- some brief notes
    - every un-corrupted packet
        * is acknowledge upon recv
    - the stream of bytes are broken
        * into a sequence of packets
        * but users don't need to
            · manage this in any way

7.2 "Peers"

- rather than client/server
- I think the terminology
  - of passive vs active peers
  - makes more sense
    - peer applications
      - on either side of INET socket
- allowing a much more flexible
  - relationship
  - it *is* bidirectional...

7.3 `PF_INET` vs `AF_INET`

- assignment uses PF
- they are interchangeable on IPv4

7.4 sending/receiving data

- thus far we've seen
  - how to use `send()` and `recv()`
- you could also have used
  - more *raw* approaches
- `read()` and `write()`
  - could have worked
- and there is an additional pair
  - `sendto()` and `recvfrom()`