COS 316
Precept:
Socket Programming

Naming

- Why is naming important? What would happen if we didn't name things?

- Naming is important because it gives us a way to find and access things
  - how Amazon knows how to deliver packages to you
  - how to access stored objects in memory

- Socket
  - their names enable the system (and others) to know how to find/contact it

Abstraction Clarification

- “A way of modeling things”
- Don‘t worry about the exact implementation
- Focus on the paradigm
- Socket abstraction

What are Sockets/Connections?

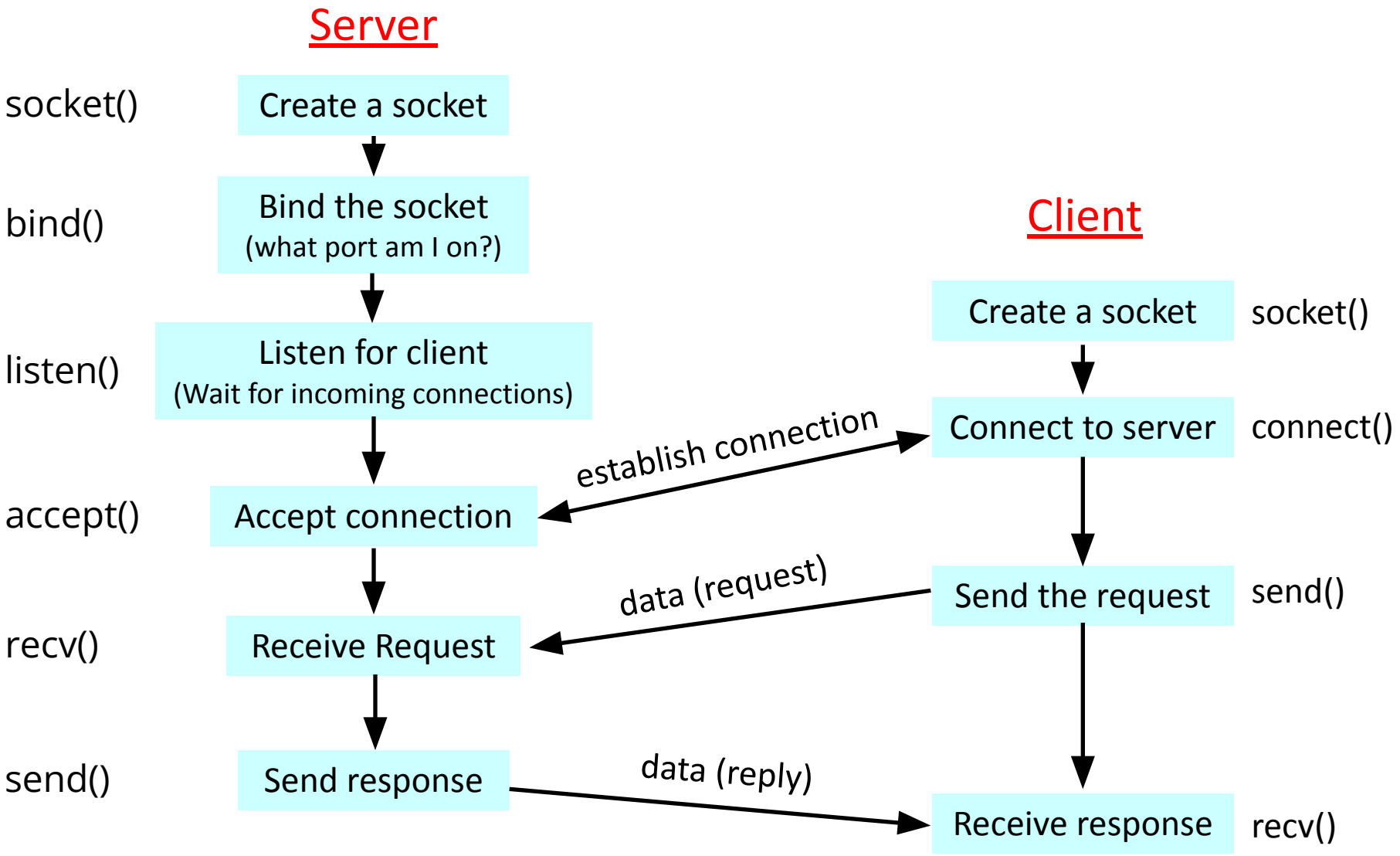
- **Connection**
  - A process on one host (host A) communicates with a process on another host (host B) via a connection
  - A communication channel
  - Another abstraction

- **Socket**
  - In order for host A to start a connection with host B, host A needs to know where and how to contact host B
  - This endpoint on host B is what we call a socket

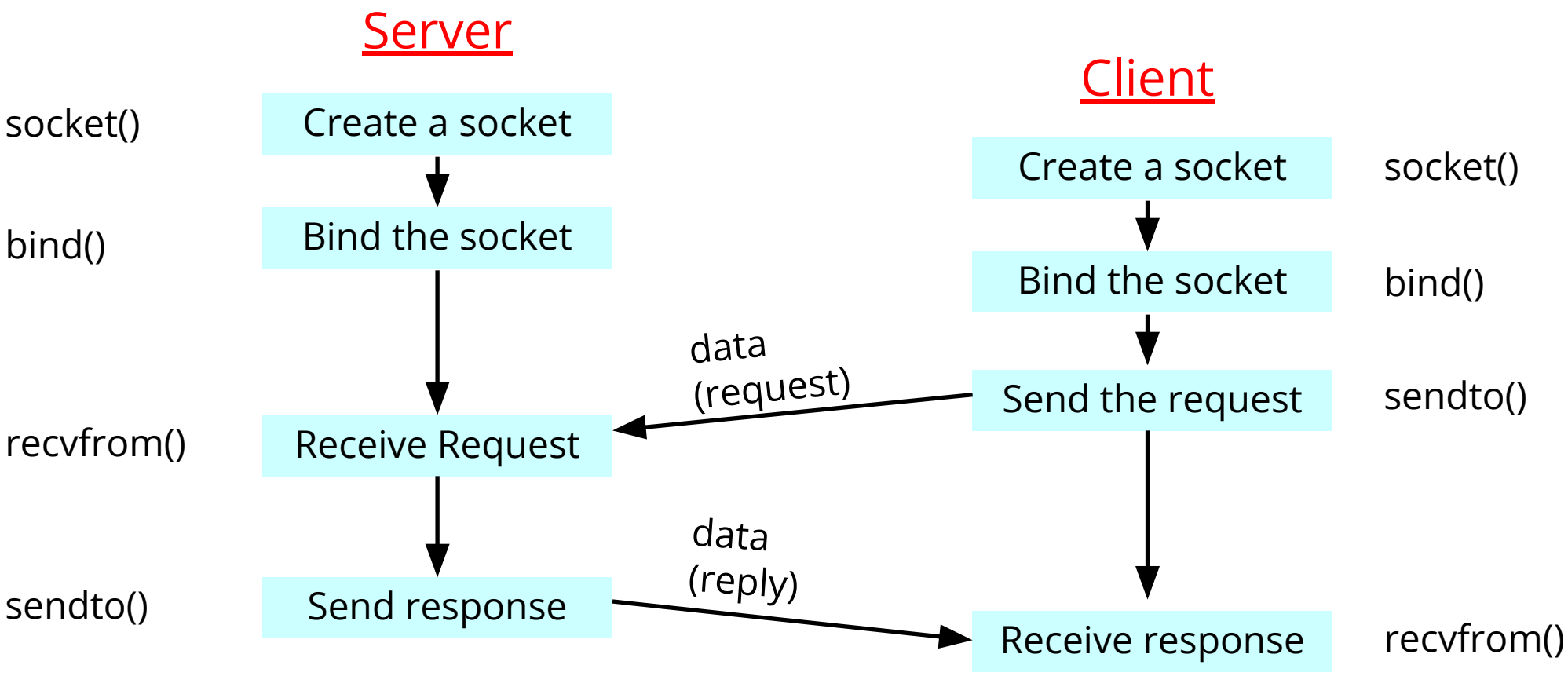
Client - Server Communication

- Client “sometimes on”
  - Initiates a request to the server when interested
  - E.g., Web browser on your laptop or cell phone
  - Doesn’t communicate directly with other clients
  - Needs to know server’s address
- Server is “always on”
  - Handles services requests from many client hosts
  - E.g., Web server for the www.cnn.com Web site
  - Doesn’t initiate contact with the clients
  - Needs fixed, known address

Stream Sockets (TCP): Connection-oriented



Datagram Sockets (UDP): Connectionless



Assignment 1

- Employ the client - server architecture
- Two files you'll modify: **client.go** and **server.go**
- Having a client send bytes to a server
- Implement the Stream Sockets (TCP): Connection-oriented

The `net` package

- `net.Listen` receives the ip, port, and protocol, and returns a `net.Listener`

- `net.Listener#Accept` waits for connections from clients
  - Once a client connects, `net.Accept` returns a `net.Conn` to be used for communication

- `net.Dial` connects to the given ip and port, with the specified protocol.
  - Once it is connected, `net.Dial` returns a `net.Conn` to be used for communication

Socket Server/Client: Go

SERVER

- `socket, err := net.Listen("tcp4", "127.0.0.1:8080")`
  - `net.Listen` performs the C `socket`, `bind` and `listen` system calls
  - socket is of type `net.Listener`
- `connection, err := server.Accept()`
  - `net.Accept` accepts an incoming client request
  - `connection` is of type `net.Conn`

CLIENT

- `connection, err := net.Dial("tcp4", "127.0.0.1:8080")`
  - Creates a TCP socket, establish connection
  - `connection` is of type `net.Conn`

`net.Conn`

- **`net.Conn.Read`** reads from the connection
  - Wrap the connection in **`bufio.Reader`**
- **`net.Conn.Write`** writes to the connection
- **`net.Conn.Close`** closes the connection

`net/http` (Useful in Future)

- A collection of useful functions for handling and processing http requests

Tips and Common gotcha

- fmt.Sprintf could be handy
- Don’t print the entire buffer
- Convert bytes to string when print
- Client needs to close() at end of connection
- EOF is not a character, it’s a type of error

Resources

- https://beej.us/guide/bgipc/html/multi/unixsock.html

Echo Demo Code

- The one shown in Precept

```go
package main

import (
    "fmt"
    "log"
    "net"
)

func main() {
    ln, err := net.Listen("tcp", "localhost:8080")
    if err != nil {
        log.Fatalf("Failed to setup a listener - %v\n", err)
    }
    defer ln.Close()
    conn, err := ln.Accept()
    if err != nil {
        log.Fatalf("Failed to accept connection - %v\n", err)
    }
    defer conn.Close()
    buf := make([]byte, 1024)
    _, err = conn.Read(buf)
    if err != nil {
        log.Fatalf("Failed to read from connection %v\n", err)
    }
    fmt.Println(string(buf))
}
```

server.go

```go
package main

import (
    "fmt"
    "log"
    "net"
)

func main() {
    conn, err := net.Dial("tcp", "localhost:8080")
    if err != nil {
        log.Fatalf("Failed to connect to server - %v\n", err)
    }
    fmt.Fprintf(conn, "Hello world!")
}
```

client.go