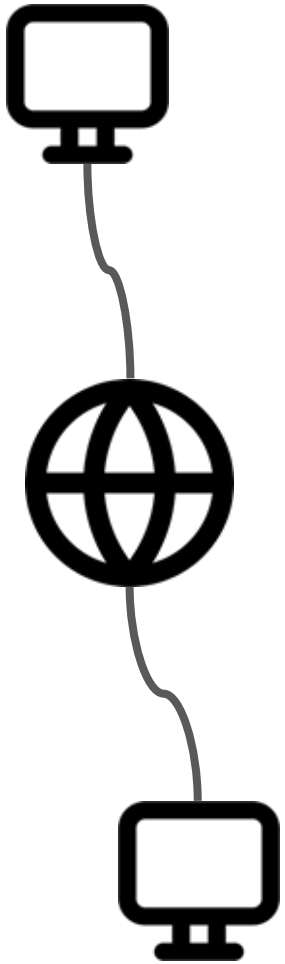
Socket Programming

COS 316 Precept 3

Abstractions



- How can two different computers exchange data?
  - Complex process, involves many different components, links, etc.
  - Computers may have different hardware, operating systems, …
- **Abstractions** avoid us having to worry about this
  - A way of reducing implementation complexity into simpler concepts
  - Focus on their *abstraction paradigm*
- Many examples for abstractions in modern systems
  - Files, Terminals (TTYs), …
- Today: *sockets*!

What are sockets? And connections?

- **Connection**
  - Many different definitions!
  - In this context: an *established* method to communicate between
    a process on one host (A) and a process on another host (B)
  - A *communication channel*
  - An abstraction; in this case spanning multiple (physical) systems
- **Socket**
  - An *endpoint* of a given connection
    - Connections are established between two sockets
  - Just another abstraction! The *system-local* abstraction of a connection

Client – Server Communication

- A *paradigm* describing how a connection is initiated between two sockets

Client

- *Actively* **initiates** the connection
- Typically “sometimes on” (e.g., web browser on your phone / laptop)
- Needs to *dial* the server
  → thus requires its address!

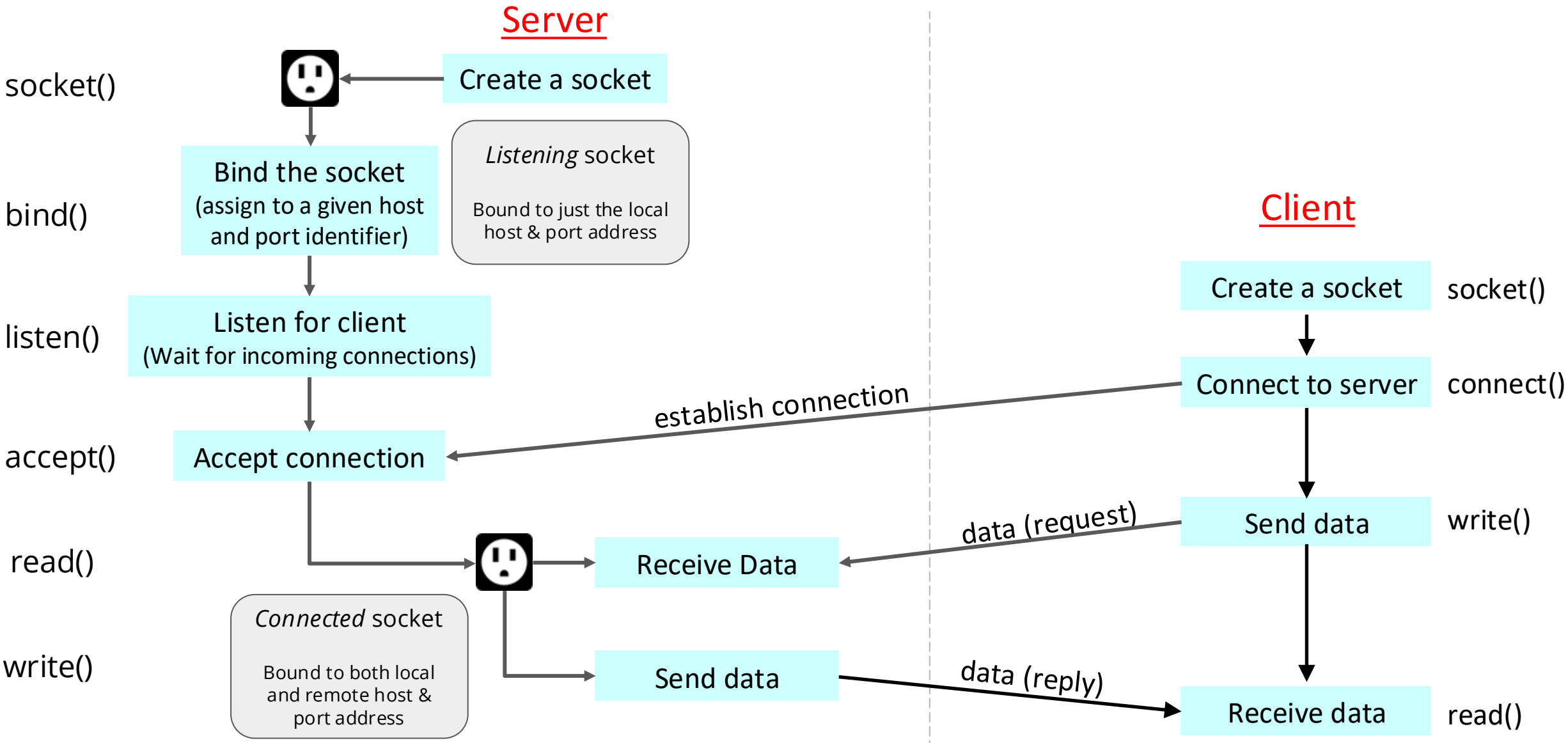
Server

- *Passively* **listens** for and **accepts** connections
- Typically “always on” (e.g., web server for google.com in some data center)
- Must be reachable under some address

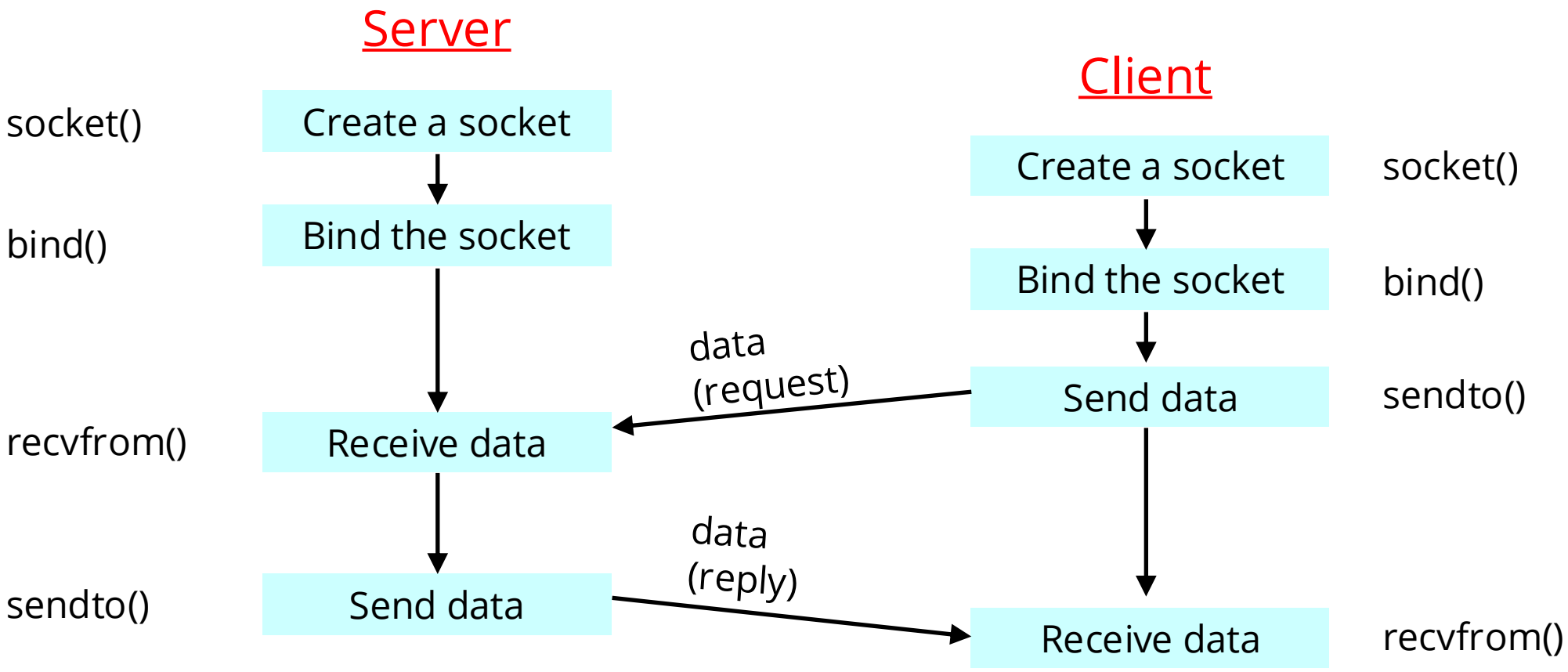
Recall: a connection is established between two processes on some hosts

Thus, an *address* is composed of a host identifier (IP address) and a process identifier (port number)

Stream Sockets (TCP): Connection-oriented



Datagram Sockets (UDP): Connectionless

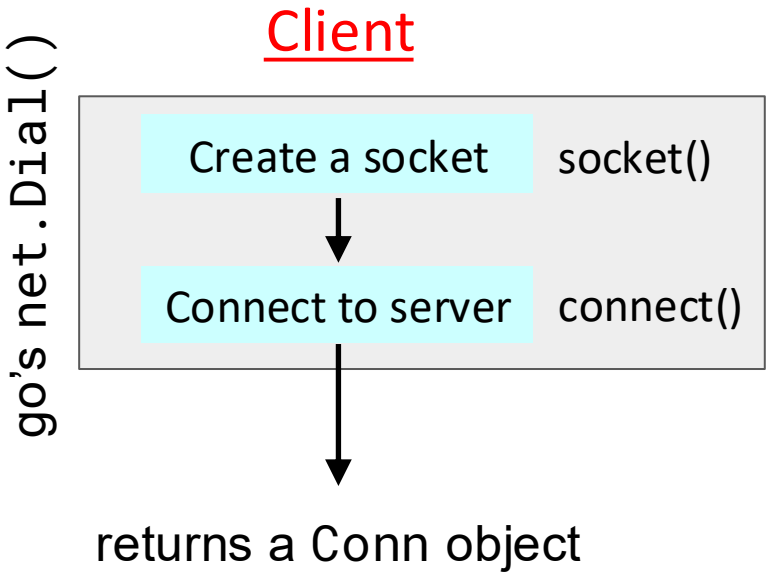


Assignment 1

- Write a pair of programs implementing the server – client connection-oriented socket paradigm
  - Using “stream sockets” (TCP)
- Two files you’ll modify: **client.go** and **server.go**
- Having a client send data to a server
  - And let the server print this data
- **This precept does not address all requirements of the assignment! Purpose is to give you an idea of how to get started.**

Client – Milestone 1: Connect to a Server

- We'll need to retrieve the server address from the command line
  … and connect to it
- go's `net.Dial` function looks promising!
  - Read its documentation to figure out the expected server address format
- Read the server address from the command line arguments
  - You can find those in `os.Args` in go!
  - The first argument (`os.Args[0]`) is always the executable name



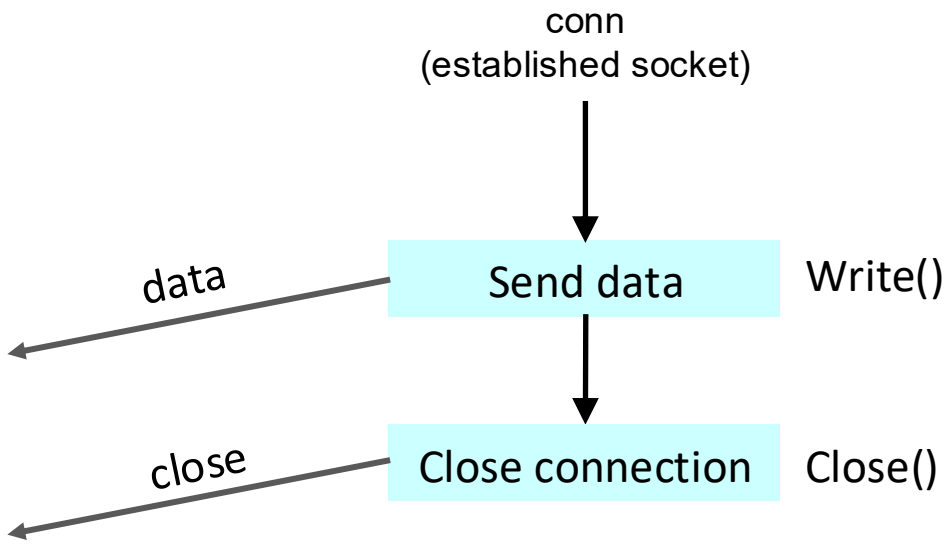
Client – Milestone 1: Connect to a Server

```
func main() {
    // get the server's IP and port from the command line
    if len(os.Args) < 2 {
            panic("Provide the server IP and port as the first command line argument")
    }
    address := os.Args[1]

    // use net.Dial to connect ip and port, with the specified protocol.
    conn, err := net.Dial("tcp", address)
    if err != nil {
            panic("An error occurred while connecting to the server!")
    }
```

Client – Milestone 2: Write Data & Close Connection

- The client will need to read a message from standard input
  - Place the message in a buffer
  - Write the contents of that buffer to the socket

- Use `conn.Write` to write some bytes to an established connection
- Use `conn.Close` to close a connection
  - This informs the opposite end socket that the connection is no longer established
  - Both sides can close a connection!

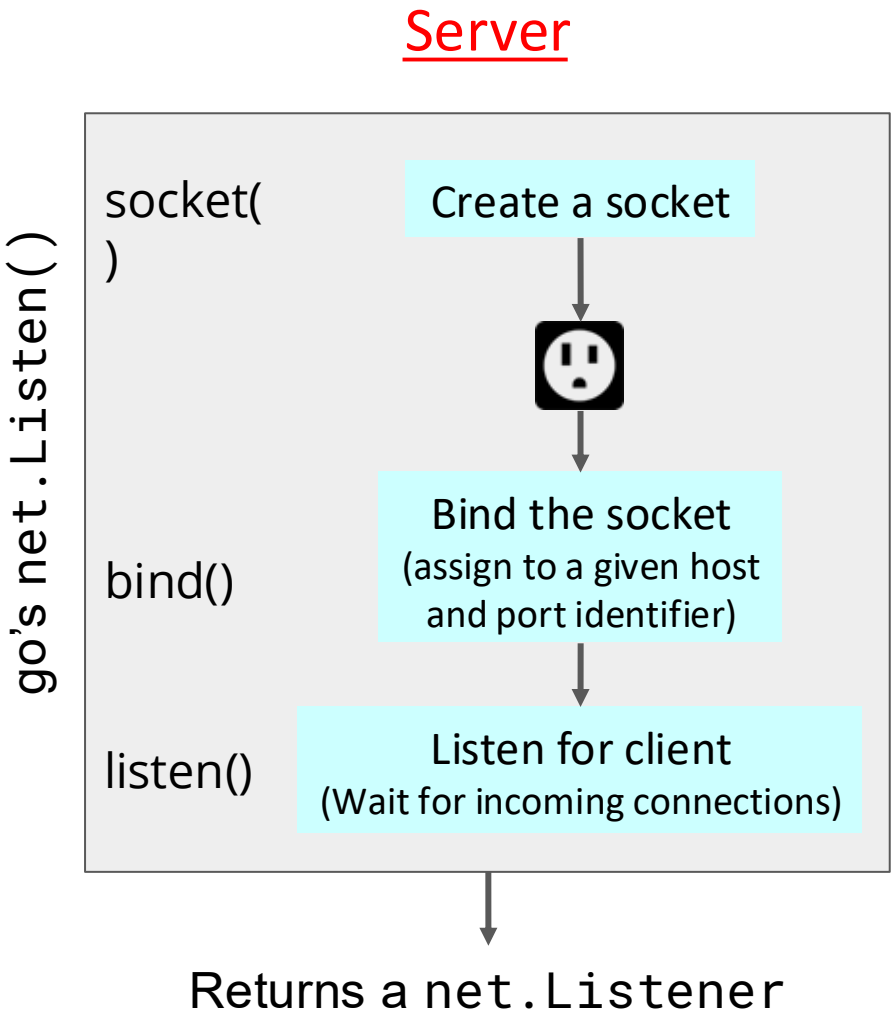


Client – Milestone 2: Write Data & Close Connection

```
	// net.Conn.Write writes to the connection
_, err = conn.Write(message[:bytes_read])
if err != nil {
	log.Fatalf("Failed to write message to connection -- %v!", err)
}

// net.Conn.Close closes to the connection
err = conn.Close()
if err != nil {
	log.Fatalf("Failed to close the connection -- %v!", err)
}
```

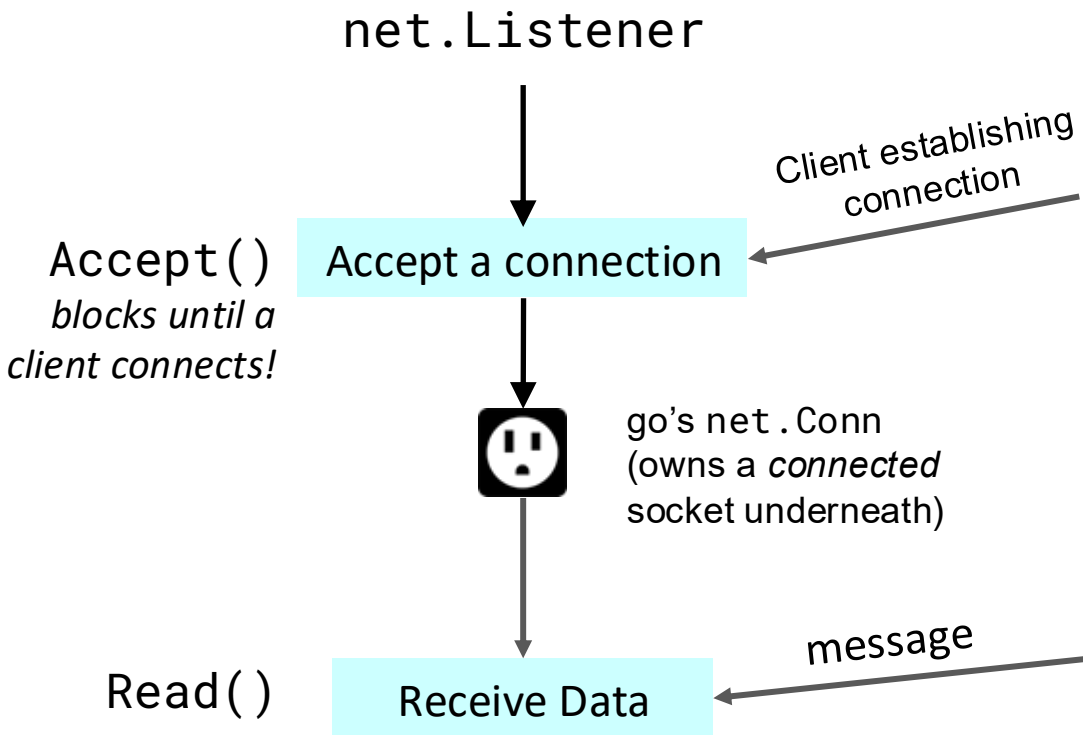
Server – Milestone 1: Create a Listening Socket

- To accept connections, our server must create a *listening* socket
  - The `net.Listen` function does that!
  - Returns a `Listener`, which owns a socket
- net.Listen takes a *listen* address
  - *Host*- and *process*-address of server (IP & port)
  - A server can have multiple host addresses! Listening on “localhost” or “127.0.0.1” only allows local connections.
- Use `fmt.Sprintf` to combine the host-address and port number



Server – Milestone 2: Accept a Connection & Read Data

- A `Listener` can *accept* an incoming client connection with the `Accept` method
  - returns a net.Conn, same as on Client!
- `net.Conn` can receive data through the `Read()` method
  - Takes a buffer as argument
- Accept a client connection



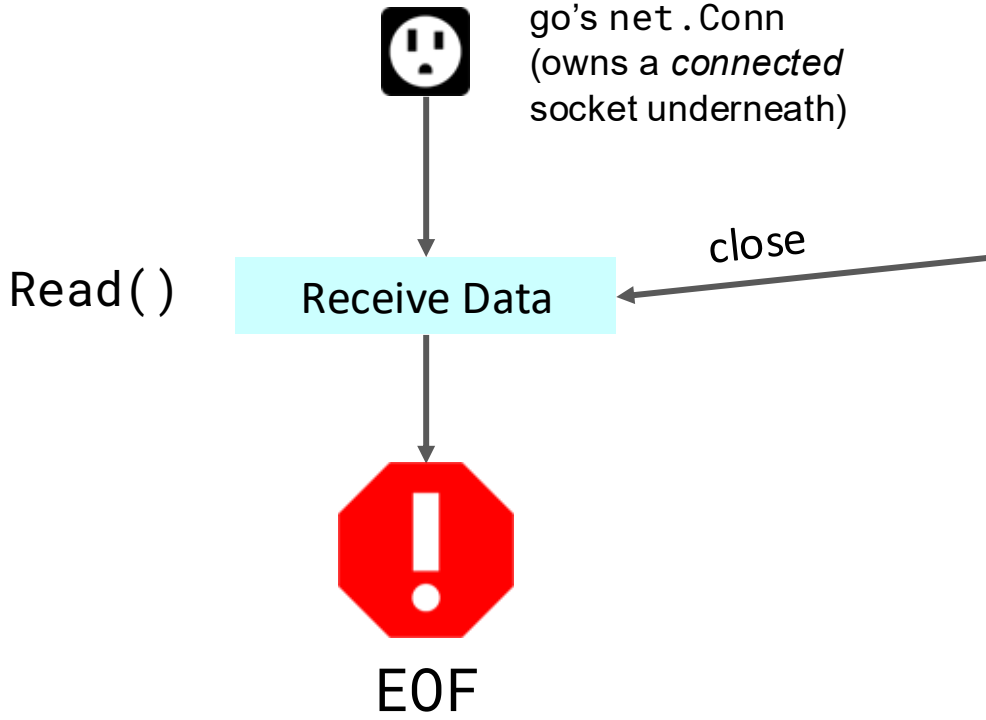
Server – Milestone 2: Accept a Connection & Read Data

```
// Loop waiting for connections
for {

    // look for a client to connect
    conn, err := listener.Accept()
    if err != nil {
              log.Fatalf("Failed to accept client connection -- %v!", err)
    }

    // Loop reading from the connection
    for {
              // create an input buffer
              message := make([]byte, RECV_BUFFER_SIZE)

              // read the data sent by the client
              bytes_read, err := conn.Read(message)
```

Server – Milestone 3: Handling a Client `Close()`

- Both sides can close a connection
  - What if that happens during a `Conn.Read()`?
- Conn.Read() returns an EOF error!
  - “End of file”
- Check for this error.

  If it occurs, close the connection.
  - `err` may be set to `nil` – check for this first!
  - `err` provides the `Error()` method, which returns error codes as strings



Server – Milestone 3: Handling a Client `Close()`

```
for {
    ...
        // read the data sent by the client
    bytes_read, err := conn.Read(message)
    if err != nil {
        if err == io.EOF {
                        // Client disconnected
                        break
        } else {
                        log.Fatalf("Failed to read from socket -- %v!", err)
        }
    }
        ...
}

// clean up/close the connection
err = conn.Close()
if err != nil {
    log.Fatalf("Error while closing the connection -- %v!", err)
}
```

Tips and Common gotcha

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