COS 316 Precept: Socket Programming
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

socket() -> Create a socket

bind() -> Bind the socket (assign to a given host and port identifier)

listen() -> Listen for client (Wait for incoming connections)

accept() -> Accept connection

read() -> Receive Data

write() -> Send data

Listening socket
Bound to just the local host & port address

Connected socket
Bound to both local and remote host & port address

Server

Create a socket

Client

Create a socket

socket()

Connect to server

connect()

Send data

write()

data (request)

Receive data

read()

data (reply)
Datagram Sockets (UDP): Connectionless

**Server**
- `socket()`
  - Create a socket
- `bind()`
  - Bind the socket
- `recvfrom()`
  - Receive data
- `sendto()`
  - Send data

**Client**
- `socket()`
  - Create a socket
- `bind()`
  - Bind the socket
- `sendto()`
  - Send data
- `recvfrom()`
  - Receive data

Data (request) is sent from Client to Server, and data (reply) is sent back from Server to Client.
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: minimal client – server example
  - Available at https://github.com/cos316/precepts/tree/main/precept2
- 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

Create a socket  socket()

Connect to server  connect()

returns a Conn object
```
Client – Milestone 2: Write Data & Close Connection

- Client contains code for reading a message from the **standard input**
  - Message is placed in the message buffer
  - `bytes_read` indicates the number of bytes that have been read into the buffer
  - Go supports “sub-slicing an array” like so: `my_array[:number_of_elements]`

- **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!
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 := net.Listen("tcp", fmt.Sprintf("%s:%d", host, port))
```
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**

```plaintext
net.Listener

Accept() blocks until a client connects!

Accept a connection

go's net.Conn
(owns a connected socket underneath)

Read()

Receive Data

Client establishing connection

message
```
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 4: Receiving Data

- Now, let’s actually print the client’s message!
  - Similar to reading on the client side
  - `Read()` reads to a buffer, returns the number of bytes
  - Use `fmt.Println` to print a subset of the buffer’s contents
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