Go Track

PowerUp! SG Tech Traineeship – Software Engineering

Learning Objectives – Mod 14

At the end of the course, participants should be able to:

- Define a standard Network Model.
- Define standard network protocols.
- Examine the different network protocols available.
- Define the components in a HTTP setup.
- Create a simple application using Go packages for HTTP.
- Examine the different components of a HTTP server.
- Demonstrate a working HTTP server.

The Web in Go

- Interesting read for Go as ranked by IEEE
  - https://spectrum.ieee.org/top-programming-languages-2022
- Solves slow compilation and execution in large distributed software.
- Own native support of frameworks and libraries.
- Community resources for HTTP.
  - Follows Opensource Policy.

ISO OSI Protocol

- A network
  - A communications system that connects devices together.
  - In the form of copper wire, ethernet, fibre or wireless.
  - Information is shared via the network
    - Uses same language to communicate.
    - Also known as protocol.
  - Influenced by the ISO OSI Model published in 1984.
  - Exist in two general forms
    - Local Area Network (LAN).
    - Wide Area Network (WAN).

* ISO - International Organization for Standardization
* OSI - Open System Interconnection

Inter versus Intra

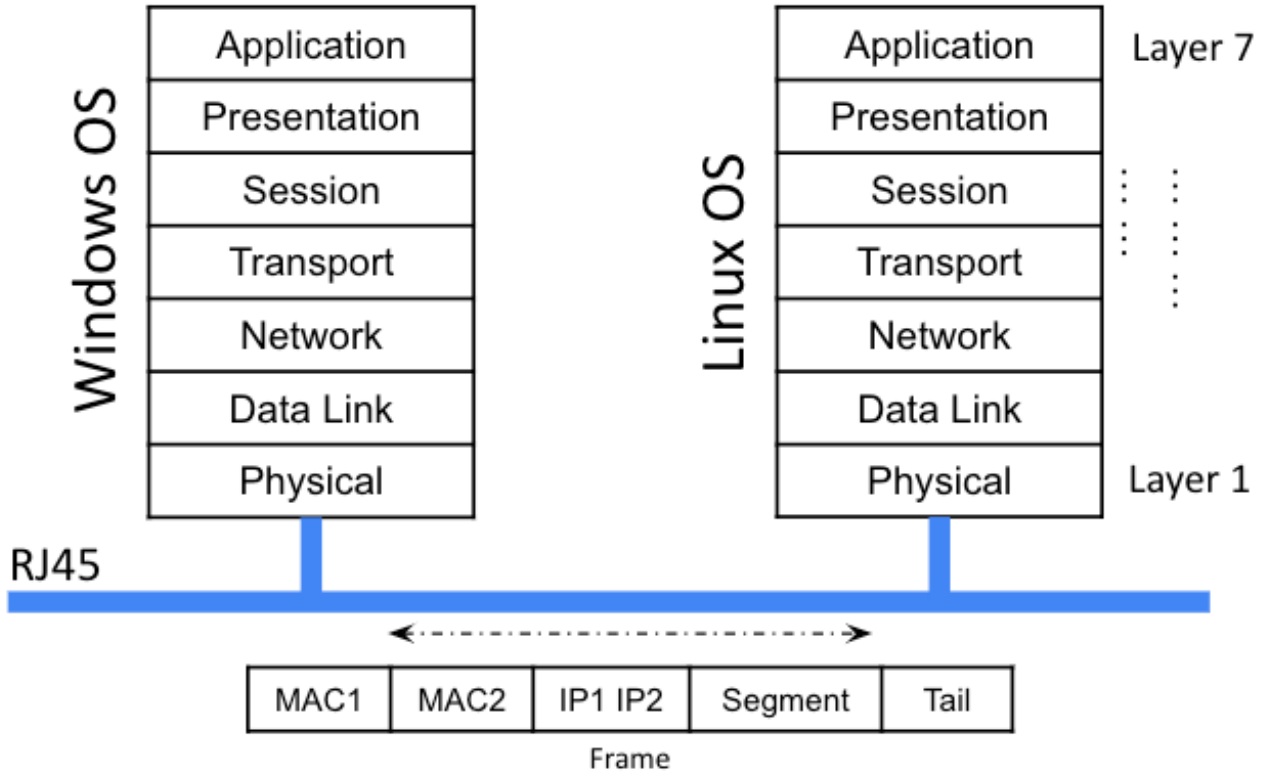
- **Inter**net
  - interconnection of two or more distinct networks in the form of LANs or WANs.
  - Not controlled by single body.
- **Intra**net
  - internet of networks limited to a single organization.
  - single administrative control, single set of policy.

ISO OSI Model

- Conceptual model to serve as a standard of communication for telecommunication or computing systems.
- Abstract model of several different layers.
- Each layer is served by the one below and serves the layer above.
- Reference Standardization began since 1977.
- Maintained by Internet Engineering Task Force (IETF).

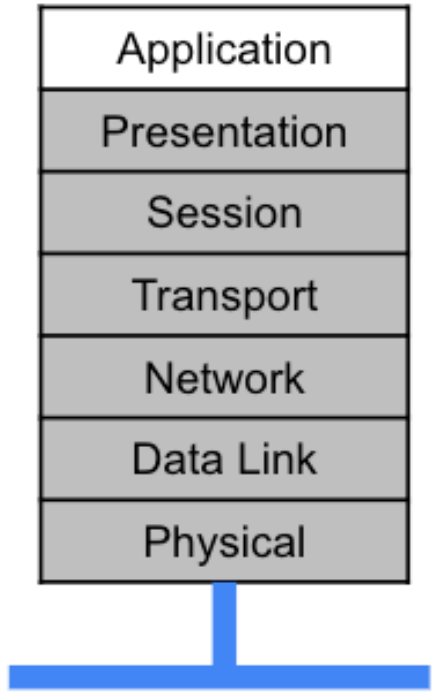
ISO OSI Protocol

- Consists of multiple layers, each layer consist of protocol unique to it.
  - Application Layer
  - Presentation Layer
  - Session Layer
  - Transport Layer
  - Network Layer
  - Data Link Layer
  - Physical Layer



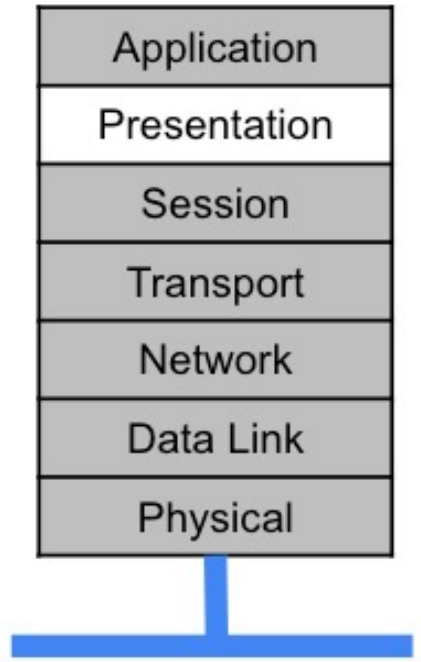
ISO OSI Protocol

- Application Layer
  - Provides services for applications.
  - Allows user activities using protocols.
  - Network Applications
    - E.g Chrome, Firefox, Outlook, Skype.
  - Application layer protocols support Network Applications
    - FTP allows for File transfer.
    - HTTP/S allows for Web Surfing.
    - SMTP allows for Emails.
    - Telnet allows for Virtual Terminals.



ISO OSI Protocol

- Presentation Layer
  - Receives information from Application layer.
  - Converts information from text and numbers to machine binary.
  - Translation to machine understandable data
    - E.g. ASCII → Binary
  - Data compression – reduce space for faster transmission
    - Lossy transmission.
    - Lossless transmission.
  - Encryption/ Decryption to enhance data integrity
    - Secure Sockets Layer.

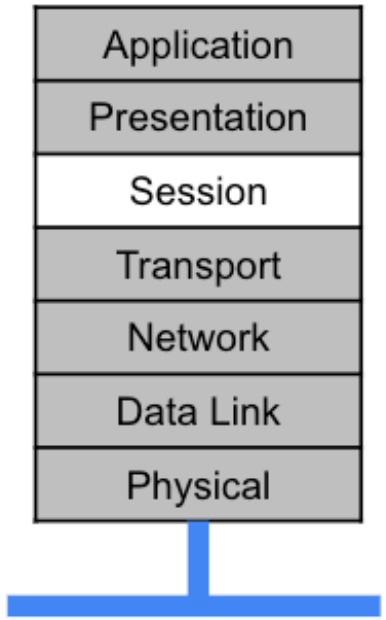


ISO OSI Protocol

- Lossy
  - Some original data is removed
  - Used when data is not important e.g. loss of quality of images and video
  - After decompression, file is not the same as original
  - Examples, JPEG and MP3
- Lossless
  - No removal, rearranging of data to become efficient
  - Compacting repeated data with frequency/ data pair
    - AAAAAAAeBBBCCCCCb --- 7Ae3B5Cb
    - 0000011110 --- 504110

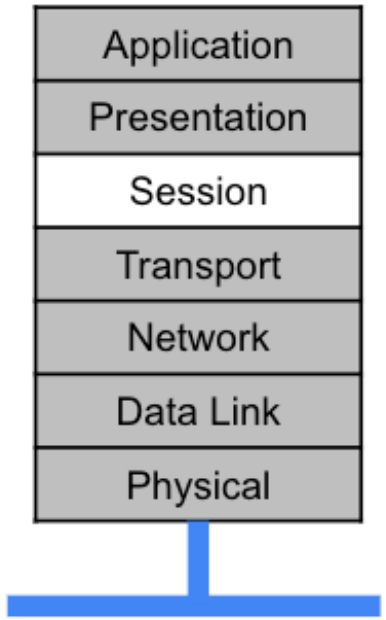
ISO OSI Protocol

- Session Layer
  - Setting up and managing connection
  - Enables sending and receiving of data
  - Termination of session at the end.
  - Application Programming Interfaces
    - NETBIOS
      - Applications between computers to communicate.



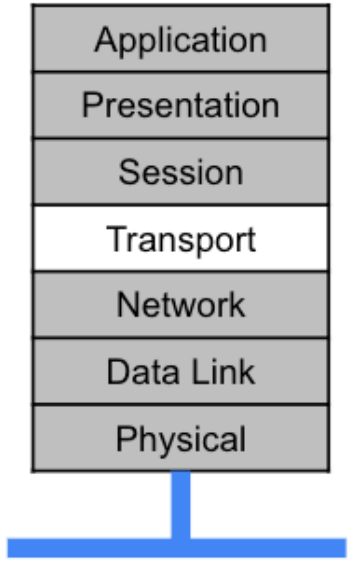
ISO OSI Protocol

- Session Layer
  - Authentication & Authorization
    - Username / Password to establish session
    - User Privileges to confirm access to data
  - Session management
    - Tracks the data package of files like type (text, image) and recipient of the data package.



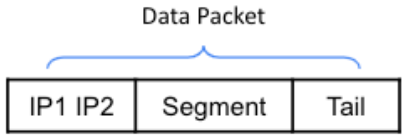
ISO OSI Protocol

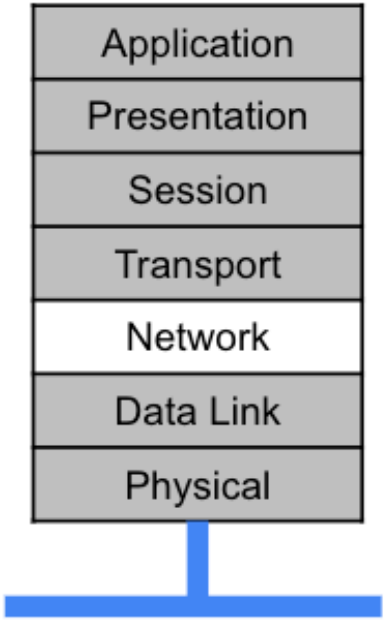
- Transport Layer
  - Controls the reliability of communications.
  - Segmentation
    - Breaks data package to small segments
      - Contains sequence number to reassemble segments in correct order.
      - Contains port number to direct data to correct application
  - Flow Control
    - Controls the amount of data transmitted
    - Channel to send regulating signal to throttle transmission rate.
  - Error Control
    - Automatic Repeat Request to request repeat delivery of lost or corrupted data using checksum for example.



ISO OSI Protocol

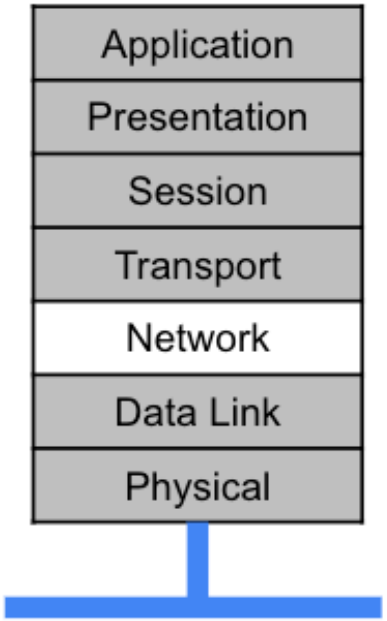
- Network Layer
  - Routes data packets across different networks to different computers
  - Logical Addressing
    - Values of IPv4 or IPv6 and Mask
  - Routing
    - Based on logical addressing
    - Unique IP address with Mask
      - Determine network address and Host address.
      - E.g. 192.168.1.31 with mask 255.255.255.0
        - Network is 192.168.1.x and host is 31





ISO OSI Protocol

- Network Layer
  - Path determination
    - Best route of sending the data from source to destination
      - Open Shortest Path First
      - Border Gateway Protocol
      - Intermediate System to Intermediate system



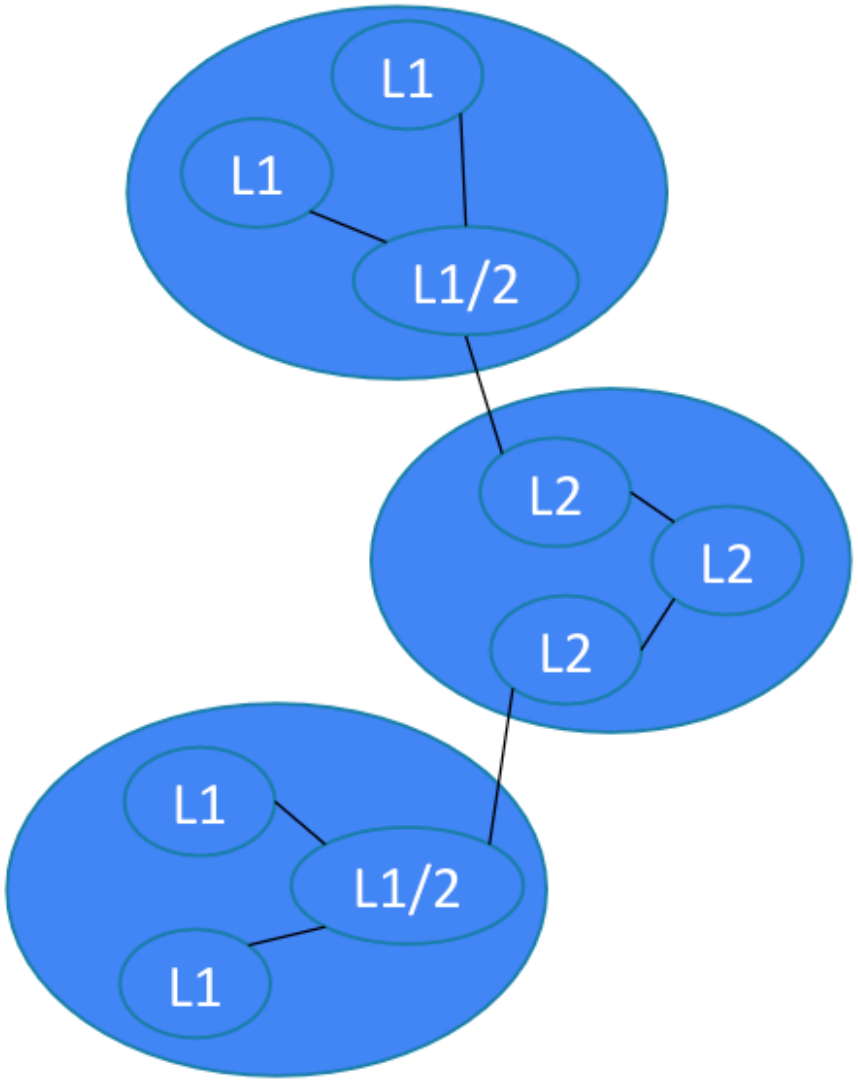
ISO OSI Protocol

- Network Layer
  - Open Shortest Path First
    - Routing Standard – interior Gateway Protocol
    - Consider router links as nodes
    - Uses linked state database
      - Data of network exchanged between routers using messaging
    - Calculates the cost of traversing the related routers from source to destination.
      - Cost = 1 0000 0000 / bandwidth in bps

Interface	Default Bandwidth	Cost
Serial	1544Kbps	64
Ethernet	10 000 Kbps	10
Fast Ethernet	100 000 Kbps	1

ISO OSI Protocol

- Network Layer
  - Border Gateway Protocol
    - Standard adopted for exterior gateway protocol
      - Routing and reachability information between routers in the internet
    - Does routing decisions based on paths, network policies or rules set by network administrator.
    - Communicates on port 179 to maintain *link alive* message
    - Continuous update to network changes like links break, resolved and routers offline or online.
  - Internal Border Gateway Protocol in a local network.
  - External Border Gateway Protocol in internet.

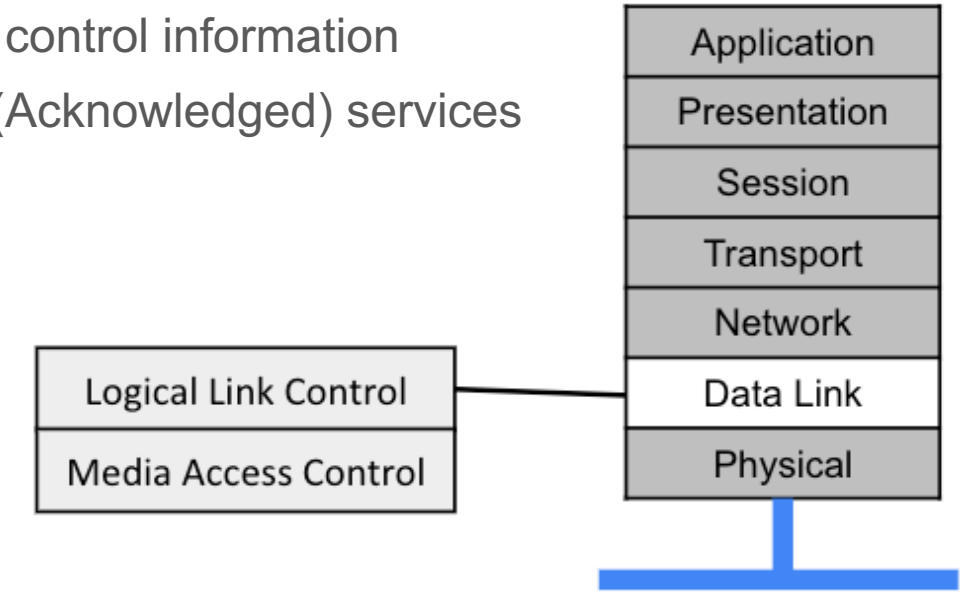


ISO OSI Protocol

- Network Layer
  - Intermediate System to Intermediate system
    - Consider router as intermediate system
    - Linked state routing protocol similar to OSPF
    - Large scale networks
    - L1 router maintains routing information in L1 area
    - L2 router maintains only routing information in L2 backbone area
    - L1/2 router maintains routing between L1 area and L2 area

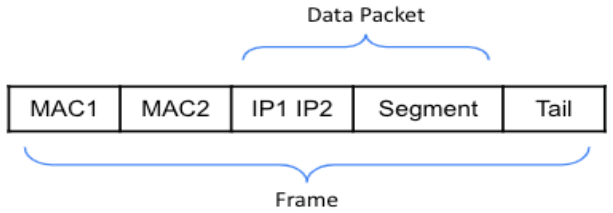
ISO OSI Protocol

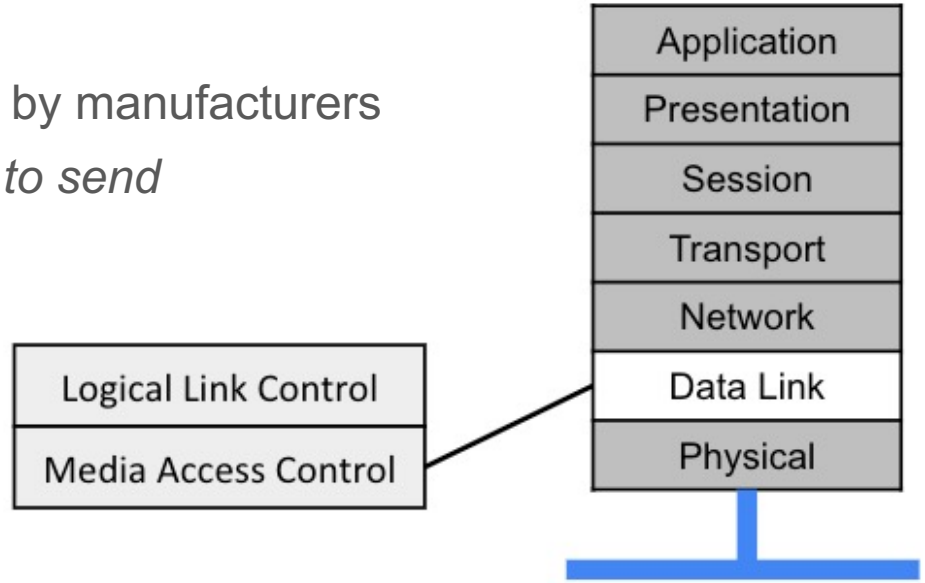
- Data Link Layer
  - Logical Link Control (Upper sublayer)
    - Identifies the network layer protocol and adds control information
    - Provides LLC1 (Unacknowledged) and LLC2 (Acknowledged) services
      - Flow Control



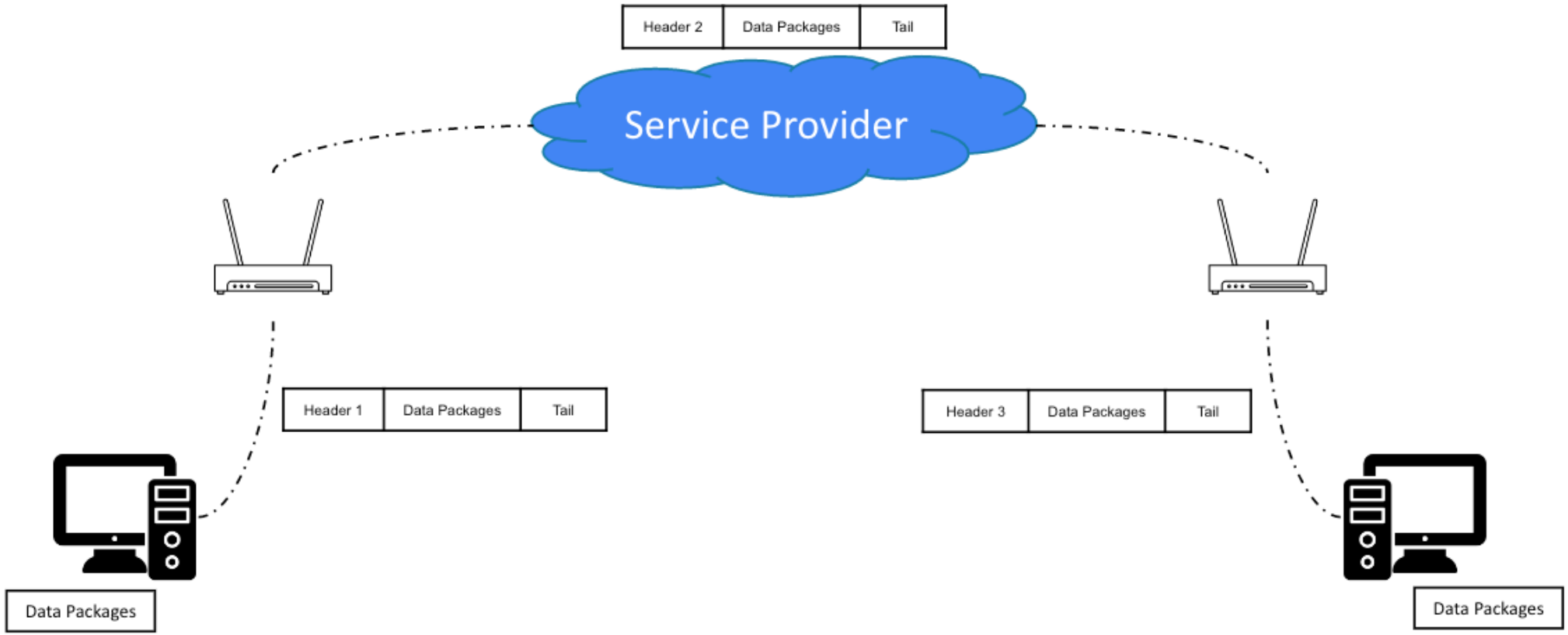
ISO OSI Protocol

- Data Link Layer
  - Media Access Control (Lower sublayer)
    - Uses physical MAC addressing, a sequence of 12 alphanumeric numbers in hexadecimal
      - Embedded into Network Interface Card (NIC) by manufacturers
    - Framing by adding header to form the *data frame to send*
    - Removes header on receiving
    - Error control
    - Access Control to prevent collision



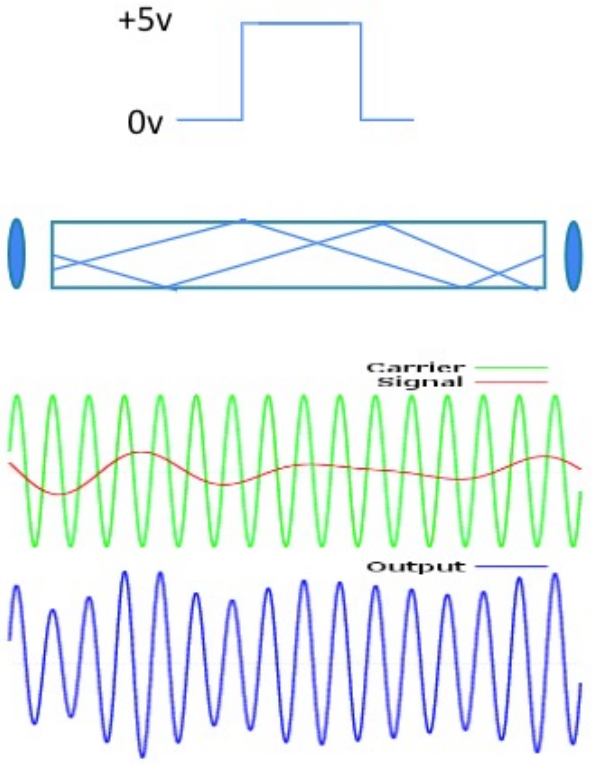


ISO OSI Protocol

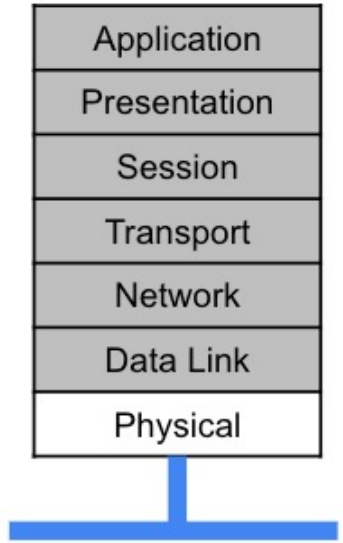


ISO OSI Protocol

- Physical Link Layer
  - Converts Bits to signals
  - Copper wire
    - Signal of High and Low
  - Optical fibre wire
    - Light signal
  - Air
    - Radio signal



Source: https://commons.wikimedia.org/wiki/File:Amplitude-modulation.png



System Details

- Makes use of package “net”
  - Interface
    - Name
    - HardwareAddr
  - InterfaceByName
    - Addrs

```
interfaces, err := net.Interfaces()
if err != nil {
    fmt.Println(err)
    return
}

for _, i := range interfaces {
    fmt.Printf("Interfaces: %v\n", i.Name)
    fmt.Printf("MAC address: %v\n", i.HardwareAddr)

    byName, err := net.InterfaceByName(i.Name)
    if err != nil {
        fmt.Println(err)

    }

    addresses, err := byName.Addrs()
    for k, v := range addresses {
        fmt.Printf("Interface Address # %v: %v\n", k, v.String())

    }
    fmt.Println()
}
```

Activity1
Exploring System Details

Transport Type

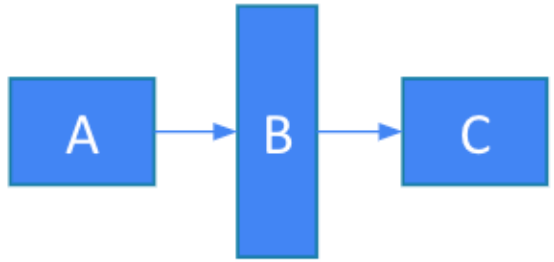
- Recall on Transport layer of the OSI ISO Model
- Connection Oriented
  - Single connection for the session.
  - 2-way communications flow along the connection.
  - Connection is broken after the session is over. Example: a Phone Call
- Connectionless
  - Messages sent independent of each other. Example: A mailing system.
  - Data may arrive out of order.
- Connection oriented types may be used on top of connectionless ones and vice versa.

Internet, Transmission Control and User Datagram Protocol

- Internet Protocol
  - Not reliable
  - Responsible for delivering packets from source to destination according to IP addresses.
- Transmission Control Protocol
  - Reliable protocol, resends if there is no delivery to destination
  - Used for establishing connections, transferring data, sending acknowledgements and closing connections.
- User Datagram Protocol
  - Not reliable, can be lost, duplicated or arrive out of sequence order
  - Send at a higher speed than TCP

Distributed Computing

- Client-Server system
  - asymmetric system
  - client send requests to server and server responds
- Peer-to-peer system
  - Both able to start or to respond to messages
  - Both act with the same functionalities
- Filter
  - Pass information to a middle component that modifies the information before passing it to a third.



Client Server System

- Two logical parts – two-tiered
  - Server - Provide services
  - Client - Request for services

- Typically can run on separate machines on a network
  - Allows for different users to access powerful resources on the server from lower end computers
  - Server holds data while the client is responsible for the user interface
  - Application logic could be distributed between the client and the server
- Can be three-tiered with middle tiered holding most of the application logic.

User Datagram Protocol

- Used as a data information transmission protocol.
- Can be coupled with IP to transmit messages to other destinations.
- Designed by David P Reed (1980)
- Request for Comment (RFC) 768
  - Reference: https://www.rfc-editor.org/rfc/rfc768
- Consist of UDP datagram header and the payload needed.
  - UDP datagram header catered as defined in RFC768.
  - Payload is user defined.

Simple UDP Server

- Uses checksums for data integrity.
- No handshake needed, no overheads. Ideal to design time-sensitive or real-time systems.
- For reliability to be increased, applications will have to implement certain “add-on” features.
  - Keep alive messages
  - Timeout restart messages
  - Error recovery messages

Simple UDP Server

- Support multicast using IP address and port numbers assigned.
  - One IP Address sending/ receiving via multiple ”port channels”
  - Port number 0 to 65535. Port 0 reserved.
- Internet Assigned Numbers Authority (IANA)
  - 0 ~ 1023 : Well Known Ports
    - https://en.wikipedia.org/wiki/List_of_TCP_and_UDP_port_numbers
  - 1024 ~ 49151 : User Defined Ports
  - 49152 ~ 65535 : Ephemeral Ports
- Different standards, different ranges
  - RFC 6056 : Ephemeral Ports 1024 ~ 65535

Simple UDP Server & Client

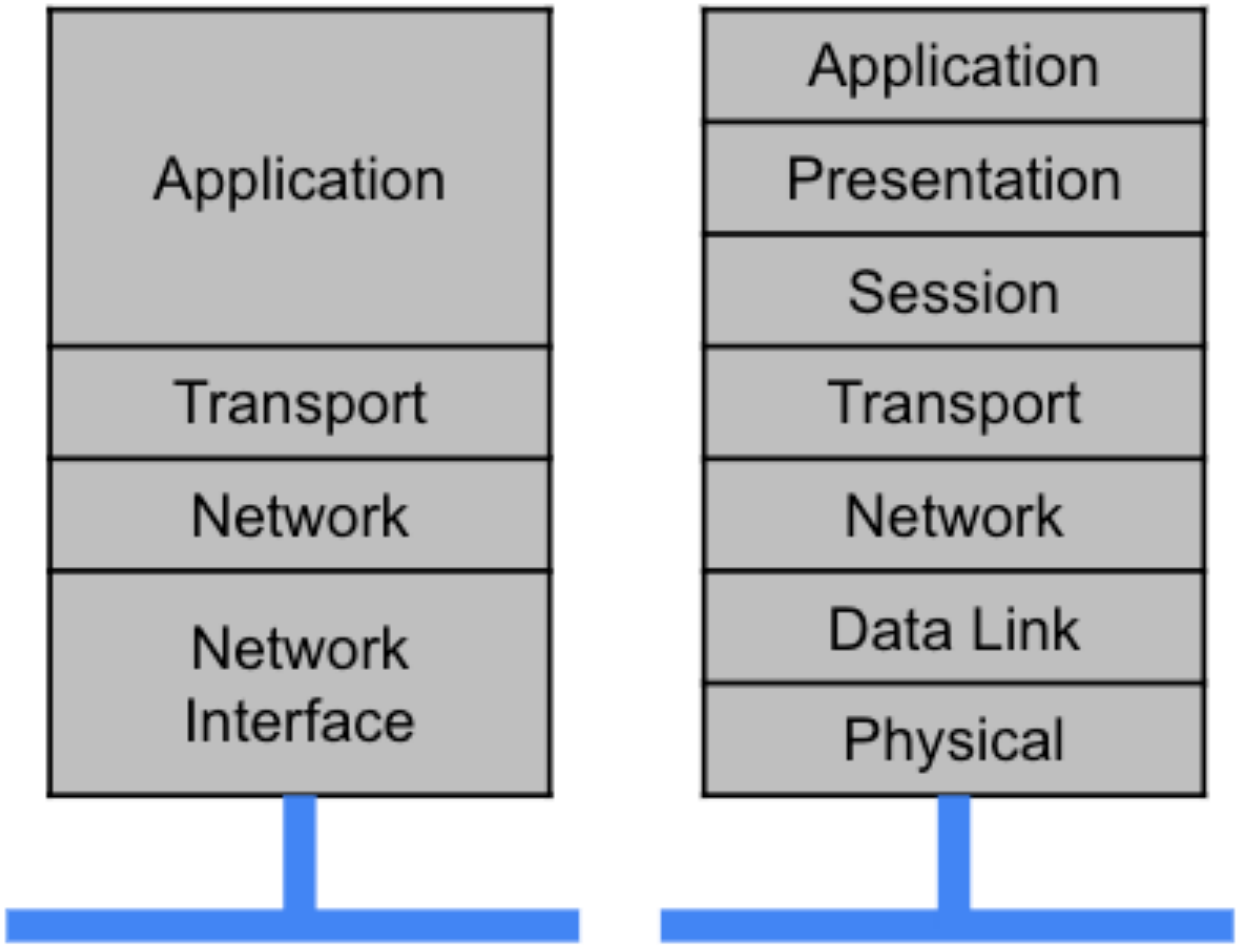
- Implementation using
  - Go package “net”
    - Provides interface for network I/O
  - Go package “fmt”
    - For formatting
  - Go package “bufio”
    - Implements buffered I/O
- Uses methods of net
  - ResolveUDPAddress
  - ListenUDP
- Uses buffer as a temporary storage
- Use of go routine to cater for buffered output.

```
func startUDPServer() {
        addr, err := net.ResolveUDPAddr("udp", ":5331")
        myListener, err := net.ListenUDP("udp", addr)
        if err != nil {
                log.Fatalln(err)
                return
        }
        defer myListener.Close()

        for {
                buffer := make([]byte, 1024)
                len, addr, err := myListener.ReadFrom(buffer)

                if err != nil {
                        fmt.Println("Continue")
                        continue
                }

                go handle(myListener, addr, buffer[:len])
        }
}
```

Activity2
Simple UDP Server-Client

TCP/IP Model

- Created by Defense Advanced Research Projects Agency (DARPA) in 1970s for ARPANET.
- Originally for Unix
- Maintained by Internet Engineering Task Force (IETF)



TCP/IP Protocol

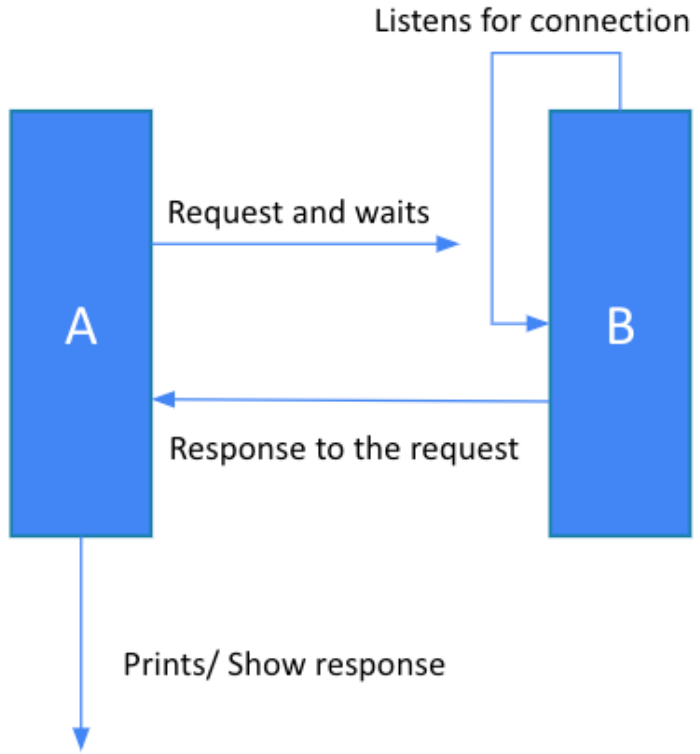
- Used as a data information transmission
- Layouts standards for development of the Internet
- Comes from two protocols – TCP and IP
- IP by its own is not reliable and is only responsible for encapsulation of data packages.
- TCP provides the reliability between the two points of transfer.

Simple TCP Server

- Consist of request and response messages.
  - Client make requests (get / post)
  - Server fulfil responses base on route
- Use of TCP to transport data with HTTP on top.
  - Defined by Internet Engineering Task Force (IETF)
    - https://www.ietf.org/standards/
  - Recommendations on how internet is built
- Request for Comment (RFC) 7230
  - reference: https://tools.ietf.org/html/rfc7230

Simple TCP Server

- Simple setup of Client-Server setup
  - Server
    - Listen for a connection from the client.
    - Accepts the connection and prints out the message
    - Respond to the client
  - Client
    - Sends a request to Server and waits for a response.
    - Receives message from server.
    - Prints out the message.



Simple TCP Server

- Implementation using
  - Go package “net”
    - Provides interface for network I/O
  - Go package “fmt”
    - For formatting
  - Go package “bufio”
    - Implements buffered I/O

```
import (
    "bufio"
    "fmt"
    "net"
)
```

Simple TCP Server

- Implementation steps
  - Preset function for setting up
  - Separate Go routine to handle incoming and outgoing information

```
func StartTCPServer() {
    myListener, err := net.Listen("tcp", ":5331")
    if err != nil {
        log.Fatalln(err)
        return
    }
    defer myListener.Close()

    for {
        conn, err := myListener.Accept()
        if err != nil {
            log.Println(err)
            continue
        }
        go handle(conn)
    }
}
```

Simple TCP Server

- Implementation steps
  - Go routine to handle incoming and outgoing information

```
func handle(conn net.Conn) {

   for {
      data, err := bufio.NewReader(conn).ReadString('\n')
      if err != nil {
         fmt.Println(err)
         return
      }
      fmt.Println("Received :", string(data))
      retMsg := string(data) + "\n"
      conn.Write([]byte(retMsg))
   }
}
```

Simple TCP Client

- Implementation using
  - Go package “net”
    - Provides interface for network I/O
  - Go package “fmt”
    - For formatting
  - Go package “bufio”
    - Implements buffered I/O

```
import (
    "bufio"
    "fmt"
    "net"
)
```

Simple TCP Client

- Implementation steps

```
func StartTCPClient() {
    conn, err := net.Dial("tcp", "localhost:5331")
    if err != nil {
        log.Fatalln("Connection fails")
        return
    }

    for {
        reader := bufio.NewReader(os.Stdin)
        fmt.Println("Key in your message: ")
        message, _ := reader.ReadString('\n')
        fmt.Fprintf(conn, message+"\n")

        recMessage, _ := bufio.NewReader(conn).ReadString('\n')
        fmt.Println("Received : ", recMessage)
    }
}
```

Activity2
Simple TCP Server-Client

Hypertext Transfer Protocol (HTTP)

- Protocol for distributed, collaborative and hypermedia information systems
- Foundation of data communication for the world wide web
- Development of HTTP initiated in 1989
- Dictates the format of communication using RFC 7230 as defined by IETF

Hypertext Transfer Protocol (HTTP)

- *Taking reference to RFC 7230*
- Start line
  - A HTTP message of either a request from client or a response from server.
- HTTP request-line (request)
  - Consist of request line, header and optional message body
  - Request-line: Method SP Request-URI SP HTTP-Version CRLF
  - Example : GET /path/to/file/home.html HTTP/1.0
- HTTP status-line (response)
  - Consist of status line, header and optional message body
  - Status-line: HTTP-Version SP Status-Code SP Reason-Phrase CRLF
  - Example: HTTP/1.0 200 OK or HTTP/1.0 404 Error

Hypertext Transfer Protocol (HTTP)

- Header Fields
  - reference: https://en.wikipedia.org/wiki/List_of_HTTP_header_fields
- Simple example
  - curl -v www.google.com

```
* Rebuilt URL to: www.google.com/
*   Trying 74.125.24.104...
* TCP_NODELAY set
* Connected to www.google.com (74.125.24.104) port 80 (#0)
> GET / HTTP/1.1
> Host: www.google.com
> User-Agent: curl/7.55.1
> Accept: */*
>
< HTTP/1.1 200 OK
< Date: Mon, 21 Sep 2020 10:43:06 GMT
< Expires: -1
< Cache-Control: private, max-age=0
< Content-Type: text/html; charset=ISO-8859-1
< P3P: CP="This is not a P3P policy! See g.co/p3phelp for more info."
< Server: gws
< X-XSS-Protection: 0
< X-Frame-Options: SAMEORIGIN
< Set-Cookie: 1P_JAR=2020-09-21-10; expires=Wed, 21-Oct-2020 10:43:06 GMT
< Set-Cookie: NID=204=QVRZ_TS0tNygarOtJAPorzbg90H888aop6JnJbp4aV0UBeuZtQI
oF3X6nAntSbWGN1E-nZiQB_aC8; expires=Tue, 23-Mar-2021 10:43:06 GMT; path=/
< Accept-Ranges: none
< Vary: Accept-Encoding
< Transfer-Encoding: chunked
<
```

Hypertext Transfer Protocol (HTTP)

- HTTP Standard Codes
- Response from server to client requests
- 5 standard classes
  - 1xx - Information
  - 2xx – Success
  - 3xx – Redirection
  - 4xx – Client Error
  - 5xx – Server Error
- Unofficial codes specified by different services
  - https://en.wikipedia.org/wiki/List_of_HTTP_status_codes#Unofficial_codes

Hypertext Transfer Protocol (HTTP)

- Common response success codes
  - 2xx series

2xx	200	**Ok**. Standard HTTP response for success.
	202	**Accepted.** Requests accepted but yet to process them.
	206	**Partial Content.** Only Partial content is delivered. I.e. wget command of data split to range header

Hypertext Transfer Protocol (HTTP)

- Common response client error codes
  - 4xx series

4xx	400	**Bad Request**. Server confused by request. Bad Syntax or characters in URL error.
	401	**Not Authorised**. Requested page is protected and require authentication.
	403	**Forbidden.** No permission to access requested page.
	404	**Not Found**. Requested page does not exist on server.
	408	**Request Timeout**. Server timed out waiting for request.

Hypertext Transfer Protocol (HTTP)

- Common response server error codes
  - 5xx series

5xx	500	**Internal Server Error.** Generic error in server, would need to examine logs to see why server responded with internal error.
	502	**Bad Gateway.** An invalid response from the server while it is working as a gateway.
	503	**Service Unavailable.** The server is temporarily overloaded or down and cannot complete the request.

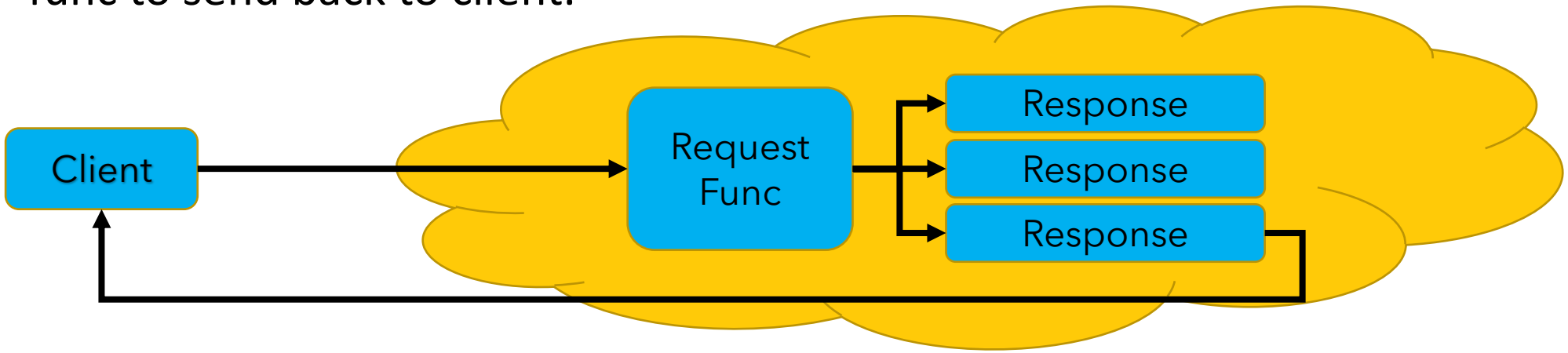
Hypertext Transfer Protocol (HTTP)

- Redirection
  - Stored in browser cache and cookies.
  - Affects the behavior of browser at next request to same URL.

3xx	300	**Moved Permanently.** Requested page has been moved permanently to a new location.
	302	**Moved Temporarily.** Request is served from a different location and is a temporary arrangement.
	304	**Not Modified.** Served from cached page as resource has not been modified.
	305	**Use Proxy**. Requested resource is only available through proxy. Need relevant proxy to get requested page.

Simple TCP Server with HTTP Request/ Response

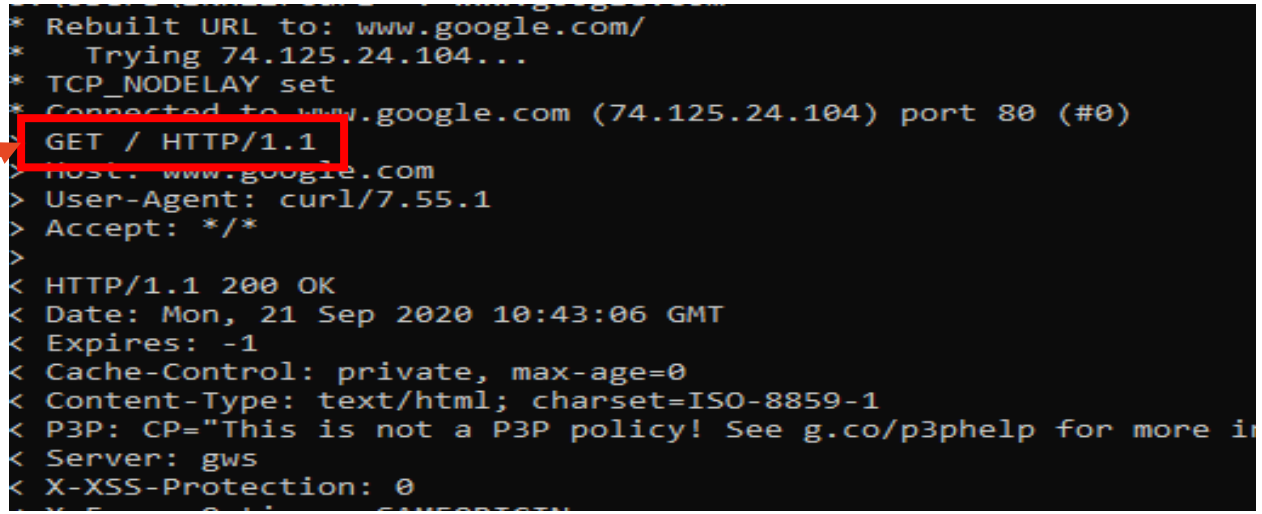
- TCP Server uses the HTTP format for request to determine what is needed as a response.
  - Request in compliance to RFC 7230
  - Response to the client also in compliance to RFC 7230
- Makes use of go routine to handle request and use corresponding response func to send back to client.



Simple TCP Server with HTTP Addon

- Similar implementation except changes to handle func
- Redesign handle to accept incoming requests from a browser
- Search for first request line based on HTTP protocol and activate relevant response func with HTML – Multiplexer of response

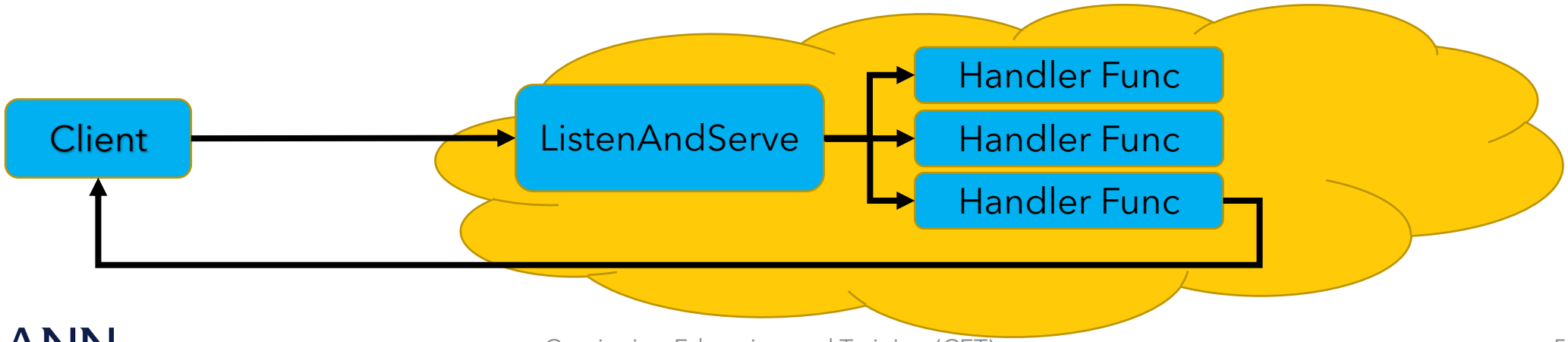
Request



Activity3
Simple TCP Server with HTTP add-on

Simple HTTP Server using Go packages

- Go HTTP Package
- Prewritten code for creating HTTP Server.
- ListenAndServe method
  - Starts server
  - Requires two information
    - IP address to listen to
    - Handler to be used to serve content, nil for default handler



Simple HTTP Server using Go packages

- Basics of HTTP Server component
  - Handler Function
  - Request Receiver
  - ResponseWriter

```
type Handler interface{
        ServeHTTP(ResponseWriter, *Request)
}
```

```
type requestMethod int

func (m requestMethod) ServeHTTP(w http.ResponseWriter, r *http.Request)
{
   fmt.Fprintln(w, “This code will run if you call it.")
}

func main() {
   var request requestMethod
   http.ListenAndServe(":8080", request)
}
```

```
Func ListenAndServe(addr string, handler Handler) error
```

Simple HTTP Server using Go packages

- Request Receiver
  - Struct of methods to use to cater to HTTP properties
  - For example the “Method” property allows for setting the method of HTTP
    - GET, POST, PUT, etc
- ResponseWriter
  - Interface to allow setting of properties of response in a HTTP
    - Set Header, Write Header, Write response etc.

Simple HTTP Server using Go packages

- Handle function
  - Simple method of HTTP to register a pattern string to a handler to trigger a response.
- Makes use of the receiver and method of functions to map the pattern to the function.
- ListenAndServe handler is set to nil typically and the default serve multiplexer is used.

```
type requestMethod int

func (m  requestMethod) ServeHTTP(res http.ResponseWriter, r *
http.Request) {
        /*code block*/
}

func main() {
        Var a requestMethod
        http.Handle(<string pattern>, a)
        http.ListenAndServe(":8080", nil)
}
```

Activity 4
Simple Serve HTTP for Single and Multiple

HTTP with routing

- A HTTP using *req.URL.Path*
  - Determines the different cases in the additional path.
  - Returns the respective pages or response needed.

```
type myURL int

func( receiver) ServeHTTP)(res, req)
{
        switch req.URL.Path{
            case 1:
                /*code block*/
            case 2:
                /*codeblock*/
        }
}

var a myURL
http.ListenAndServe(port, a)
```

Activity 4
HTTP with Routing

HTTP with ServeMux

- A HTTP request multiplexer
  - Routing function
  - Matches incoming request against a list of assigned patterns and calls the relevant handlers.
- Steps to implement
  - Create object for ServeMux
  - Assign handle to ServeMux object
  - Assign ServeMux to ListenAndServe

```
func main() {
    var f1 feature1
    var f2 feature2

    mux := http.NewServeMux()
    mux.Handle("/feature1/", f1)
    mux.Handle("/feature2", f2)

    http.ListenAndServe(":8080", mux)
}
```

Activity 5
HTTP with Serve Mux

HTTP with Handle and HandleFunc

- Handle function
  - Maps the string pattern to a handler to trigger a response

```
http.Handle(<string pattern>, handler)
http.ListenAndServe(port, nil)
```

- Handlerfunc function
  - An adaptor to allow for functions to be treated as a handler.

```
http.HandlerFunc(<function name>)
```

Activity 6
Handle to HandlerFunc

Handle, HandlerFunc and HandleFunc

- Handle maps the string pattern to a handler to trigger a response.
- HandlerFunc is an adaptor to allow functions to be used as HTTP handlers.
- HandleFunc maps the given handler function into a default serveMux to a corresponding pattern string.

HTTP with HandleFunc

- Adaptor to allow ordinary functions to be used as HTTP handlers
- Function names are assigned directly as the handlers

```
func feature1(res http.ResponseWriter, req *http.Request) {
    io.WriteString(res, "Feature1")
}
func feature2(res http.ResponseWriter, req *http.Request) {
    io.WriteString(res, "Feature2")
}
func main() {
    http.HandleFunc("/feature1", feature1)
    http.HandleFunc("/feature2", feature2)

    http.ListenAndServe(":8080", nil)
}
```

Activity 6
HandleFunc

HTTP in GO

- In ascending levels of elegance of development,
- Direct handler defined interface and passed to ListenAndServe
- Use of ServeMux to handle multiple routing.
  - Creation of a multiplexer - http.NewServeMux()
  - tags response route into mux.Handle(route, handler)
  - DefaultServeMux by passing nil for handler using http.Handle.
- Use of HandleFunc to handle multiple routing.
  - Creation of multiple route by registering route to function defined.
  - http.HandleFunc(route, functionName)

Handler Not Found, Chrome

- For some browsers, there is always a request for favicon.ico during refreshes
- Go has an inbuilt that can be used directly to response with “404 page not found”
- Use of http.Handle to respond with http.NotFoundHandler()

```
func main() {
    http.Handle(“/favicon.ico”, http.NotFoundHandler())
    http.HandleFunc("/feature1", feature1)
    http.HandleFunc("/feature2", feature2)

    http.ListenAndServe(":8080", nil)
}
```

Activity 6
HTTP Not Found

File Serving using HTTP

- FileServer
  - returns handler serving HTTP requests with the contents of the file system
  - Can be used for static file server

- Uses http.FileServer with http.Dir(<directory>)
  - *Http.Handle("/", http.FileServer(http.Dir(".")))*
  - FileServer uses "index.html" if there is one

- Addition of usual HandleFunc would allow for control of response.

Simple HTTP Client

- Similar in nature to a server
- Uses HTTP package
  - Get method to trigger retrieval of desired page.

```
package main

import (
    "fmt"
    "net/http"
)

func main() {

    data, err := http.Get(<URL>)
    if err != nil {
        fmt.Println(err)
        return
    }
    fmt.Println(data)
}
```

Setting Properties for Server/ Client

- Moves away from conventional Go built-in Http server.
  - Assign a pointer of the server to a variable and set the relevant properties
    - For example, “timeout” property to prevent locks

```
func main() {
   c := &http.Client{
      Timeout: 15 * time.Second,
   }
   request, err := http.NewRequest("GET", "http://127.0.0.1:8080", nil)
   if err != nil {
      fmt.Println("Get:", err)
      return
   }
   httpData, err := c.Do(request)
   if err != nil {
      fmt.Println("Error in DO")
      return
   }
   fmt.Println("Status code:", httpData.Status)
   header, _ := httputil.DumpResponse(httpData, false)
   fmt.Print(string(header))
}
```

Activity 7
Client and Server Comms

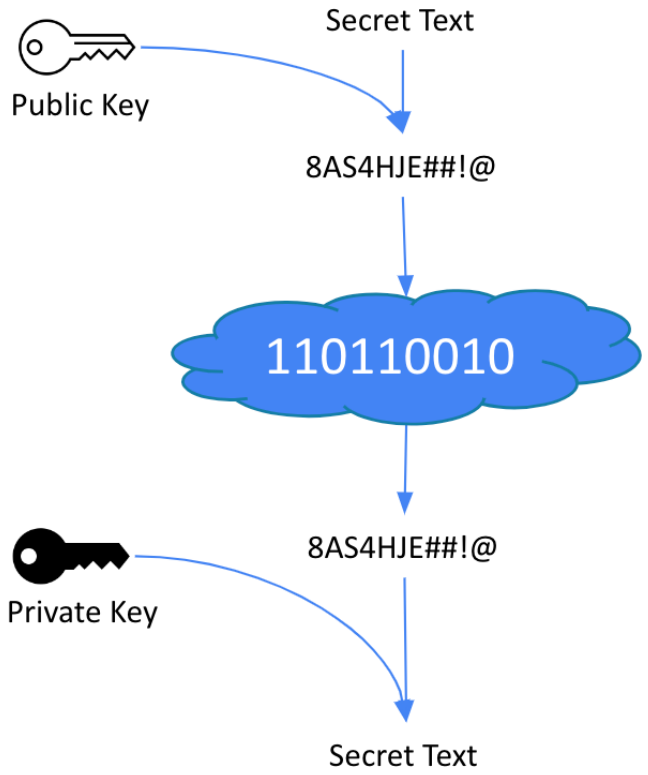
Recall Concurrency

- In last session on concurrency, we handled counter with concurrency with Mutex.
- HTTP can be added onto the concurrency and run concurrency as a backend service.
- Multiple Clients may have access and make changes to the same element at any given one time.

Activity 8
Concurrency HTTP

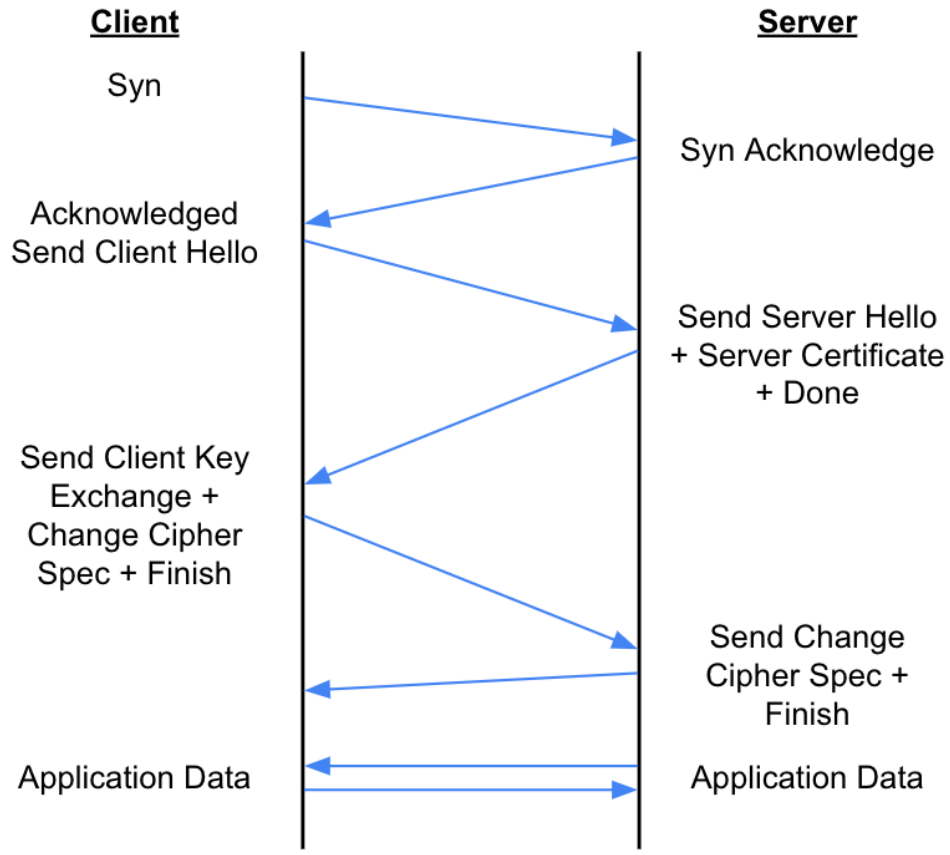
HTTP versus HTTPS

- HTTP + Security = HTTPS
  - Uses encryption protocol to encrypt communications.
  - Security system that uses
    - Private Key
    - Public key
  - Private Key used to decrypt information encrypted by public key
  - Public Key available to everyone to encrypt information. This information can be decrypted by the Private Key.



Secured Sockets Layer (SSL) or Transport Layer Security (TLS)

- SSL is the former name for TLS
- TLS is an encryption protocol
- Operates on a handshake protocol where messages are exchanged to
  - Acknowledge each other
  - Verify each other
  - Establish encryption algorithm
  - Agree on session keys.
- Both HTTPS and TLS settings are also available in Go for deployment of secured network server client.



Learning Objectives – Mod 15

At the end of the course, participants should be able to:

- Define the use of templates.
- Examine the different types of template attributes.
- Demonstrate the use of templates.

Templates

- Uses predefined pieces of information
- To create and merge data to a single document. i.e. Create HTMLs
- Create single document about a web page and allows data to be inserted and merged.
- Allows for personalized results or pages to be served to users.

Templates

- Dynamic, customised content generator found in Go.
- Not to be mistaken with templates for coding or frameworks for other languages etc.
- Can be combined with some frameworks/ templates.
- Consist of few areas
  - Actions
  - Conditions
  - Loops
  - Functions
  - Pipelines

Templates

- Two packages in Go
  - text template
    - Foundation for templates
    - Generates textual output using data
  - html template
    - Additional functionalities for HTMLs to be safe against code injection.

Templates - Creating

Template creation

- Writing in code to create HTML web pages using text or string concatenation
- Uses “go run ***<filename>*** > ***<output file>***”
- Coupled with simple ”html generator” code to loop through all files to auto generate and populate with information.

```
func main() {
	name := "Welcome to Go"
	templateBasic := `
	<!DOCTYPE html>
	<html lang = "en'>
	<head>
	<meta charset = "UTF-8">
	<title>Hello!</title>
	</head>
	<body>
	<h1>` + name + `</h1>
	</body>
	</html>
	`
	fmt.Println(templateBasic)
}
```

Activity 9
Creating Templates

Templates - Parsing

- Package “Template” is used as a container to read in datafiles.
  - Datafiles can be of **any** extension.

- To read in data files
  - Use of template methods
    - template.ParseFiles
      - read in multiple files as multiple strings.
    - template.ParseGlob
      - read in a directory of files.
    - template.Must
      - read template and error to do error check.

Templates - Parsing

- To output the parsed files
  - Uses os.Stdout
    - display content of container holding template.
  - Uses os.Create
    - write to a new html file.
  - template.Execute
    - os.Stdout to execute.
  - template.ExecuteTemplate
    - os.Stdout to execute template specified from container of templates.
- func init() to execute only once on init.
  - Unique property of init( ) over main( ) to preload templates

Activity 9
Output HTML from Go & Reading in Templates

Templates – Passing Data

- Uses “ {{.}} ” as placement location.
- The use of “.” (dot) is where the current data that is passed in from template.ExecuteTemplate function.
  - ***template.ExecuteTemplate(os.Stdout, templateName, data)***
- Can be used to pass in aggregate
  - piece of data
  - a struct of data slice or map

Templates – Passing Variables

- Uses " ***{{$variableName : = . }}*** " as variable declaration and assigning data.

- The use of ***"{{$variableName}}***" denotes the use of variables in the html and passed using template.ExecuteTemplate function similar to passing data method.

  - ***{{$variableName}}*** can be used as a placeholder.

Templates – Passing Composites

- uses " ***{{range . }}***" and "***{{end}}***" as range of given composite.
- "***{{.}}***" as placement for composites.
- Respective return output variables assigned are adapted accordingly in the HTML.

Templates – Passing Composites

- For Slice,
  - Assign variable “{{range $index, $element := .}} and “{{end}}”
  - “{{$index}} {{$element}}” as placeholders in HTML.

- For Map,
  - Assign variable “{{range $key, $val := .}} and “{{end}}”
  - “{{$key}} {{$val}}” as placeholders in HTML.

- For Struct,
  - Uses “{{.fieldName}}” directly
  - Aassign variable using “{{$variableName := .fieldName}}”
  - “{{$variableName}}” as placeholders in HTML

Activity 10
Passing Variables

Templates – Passing Functions

- Use of type FuncMap to define mapping from names to functions

- In HTML, the key trigger the value that is mapped to a function in Go file.

```
Var functionMap = template.FuncMap{
        <string1> : function1,
        <string2>: funciton2,
}
```

```
<html>
        <p> This is a function being called .<string1></p>
</html>
```

Templates – Passing Data to Functions

- Function is added using
  - *template.Must(template.New(“”).Funcs(functionMap).ParseFiles(datafile))*
  - New template is to be created first before loading all functions in to the template ”object”
  - Sequence of creation is important.

- Function is created prior to reading ParseFiles.

- Use of *“{{FunctionKeyName .}}”* to pass data to the function in HTML.

Templates - Methods

- Any methods declared as a method of the object can be passed into template.
- Template access as “{{.MethodName}}” (<dot> method)
  - Creation of method is similar to normal method creation using receiver syntax
  - Assess via object using dot notation similar to normal object creation.
- Object passed into template using template.Execute as data.

Activity 10
Functions

Templates – Global Functions

- Predefined global functions in template.
  - reference : https://godoc.org/text/template for available predefined functions
    - eq (arg1 == arg2)
    - ne (!= arg2)
    - lt (arg1 < arg2)
    - le (arg1 <= arg2)
    - gt (arg1 > arg2)
    - ge (arg1 >= arg2)
- Pipelines
  - Use of "|" in existing funcMap to "{{ . | function1 | function2 | function3}}"
    - Move data from function 1 to function 2 and eventually function 3.

Activity 10
Global Functions and Pipelines

Templates - Nested

- Uses Keyword "***{{define "templateName"}}***" to declare and "***{{end}}***"
- Placeholder uses keyword "***{{template "templateName" .}}*** to link to the template from another datafile and pass data using "." (dot).
- Can declare multiple nested template in datafile and invoke with ***templateName***.
- Generally used to modularise the template datafiles.

```
{{define "name"}}
    /* code block */
{{end}}


{{template "name"}}
```

Activity 10
Nested Templates

Learning Objectives – Mod 16

At the end of the course, participants should be able to:

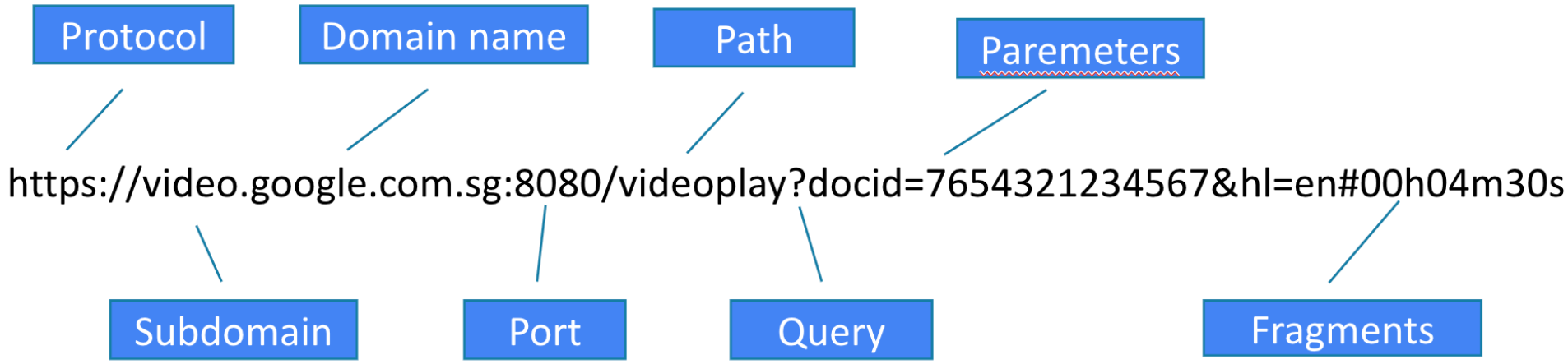
- Define the components that make up a URL.
- State the basic use of tokens in web development.
- Examine the use of XML and JSON in Go.
- Demonstrate the use of Go in web development.

Network - State

- The technical term for stored information that a program has access to at a given instant of time
  - For example, a logged user has access to certain information on the server.
- Applied in cookies or via data passing during POST or GET
- POST requires FORM and GET requires URL
  - POST has four letters and so does FORM
  - GET has three letters and so does URL

Network - URL

- URL allows for data to be “appended” after the “?”
- Values are stored in identifier=value
- Multiple values are separated with “&”



Network - URL

- Unique ID can be appended in URL to allow identification
- Makes use of http.Request FormValue
- Returns empty string if there is no value

```
func function(w http.ResponseWriter, req * http.Request){
        v := req.FormValue(<identifier>)
        /* Code block */
}

// Page URL – https://localhost:8080/?<identifier>= value
```

Activity 11
Pass values in URLs

Network - Forms

- A form allows data to be passed through either with a request body payload or URL.
  - Form method of post would require the sending of data through request body's payload
  - Form method of get would require the value sent via URL.

```
func function(w http.ResponseWriter, req * http.Request){

        v := req.FormValue("myText")
        w.Header().Set("Content-Type", "text/html; charset = utf-8")
        io.WriteString(w, `
        <form method = "post">
        <input type="text" name = "myText">
        <input type="submit">
        </form>
        <br>` +v)
}
```

Activity 11
Forms / Single and Multiple Entries

Network – Uploading, Reading & Storing

- Simple text file upload to server via client
- Make use of HTML post method of enctype multipart/form-data
- request and ioutil to read
  - *req.Method*
    - determines the method used from requester
  - *req.FormFile*
    - returns the file, fileheader and error
  - *ioutil.ReadAll(<file>)*
    - reads all content of file
- os to create and write
  - *os.Create(filepath.Join(<directory>, <filename>))* returns writer
  - *writer.Write(<filename>)*

Activity 11
Upload Read & Store

Network - Redirects

- Uses the 3xx series of the error code of HTTP
- Makes use of series 3xx predefined “http.Status”
  - http.StatusSeeOther
  - http.StatusMovedPermanently
- Syntax is *http.Redirect(w, req, <redirect url”, http.Status)*
- General use of *return* to prevent further running of codes after redirects. Failure of doing so might cause redirect functions to be stuck.

Activity 12
Redirects

Network - Cookie

- Data stored on user computer by web browser while website is browsed.
  - Domain specific
  - Records user activity or to contain state data of user
    - E.g. User ID, User Cart, User clicks, etc
  - Struct of fields to manipulate during creation and read
  - Encrypted during transport from server to client

Network - Cookie

- Stuct of Cookie
  - https://go.dev/src/net/http/cookie.go

```
type Cookie struct {
        Name string
        Value string

        Path string // optional
        Domain string // optional
        Expires time.Time // optional
        RawExpires string // for reading cookies only

        // MaxAge=0 means no 'Max-Age' attribute specified.
        // MaxAge<0 means delete cookie now, equivalently 'Max-Age: 0'
        // MaxAge>0 means Max-Age attribute present and given in
        seconds
        MaxAge int
        Secure bool
        HttpOnly bool
        SameSite SameSite
        Raw string
        Unparsed []string // Raw text of unparsed attribute-value pairs
}
```

Network - Cookie

- Creation a Cookie object
  - *&http.Cookie{ /* Struct Creation */}*
- Setting a Cookie object
  - *http.SetCookie(responseWriter, Cookie Object)*
- Reading a Cookie object
  - *Req.Cookie(<Cookie Object Name>)*

Network - Cookie

- Deleting a Cookie object
  - Makes use of the Cookie MaxAge field
  - Set value to -1 to delete Cookie object
  - Set Cookie object
- Multiple cookie objects can be set using unique cookie names

Activity 13
Cookies

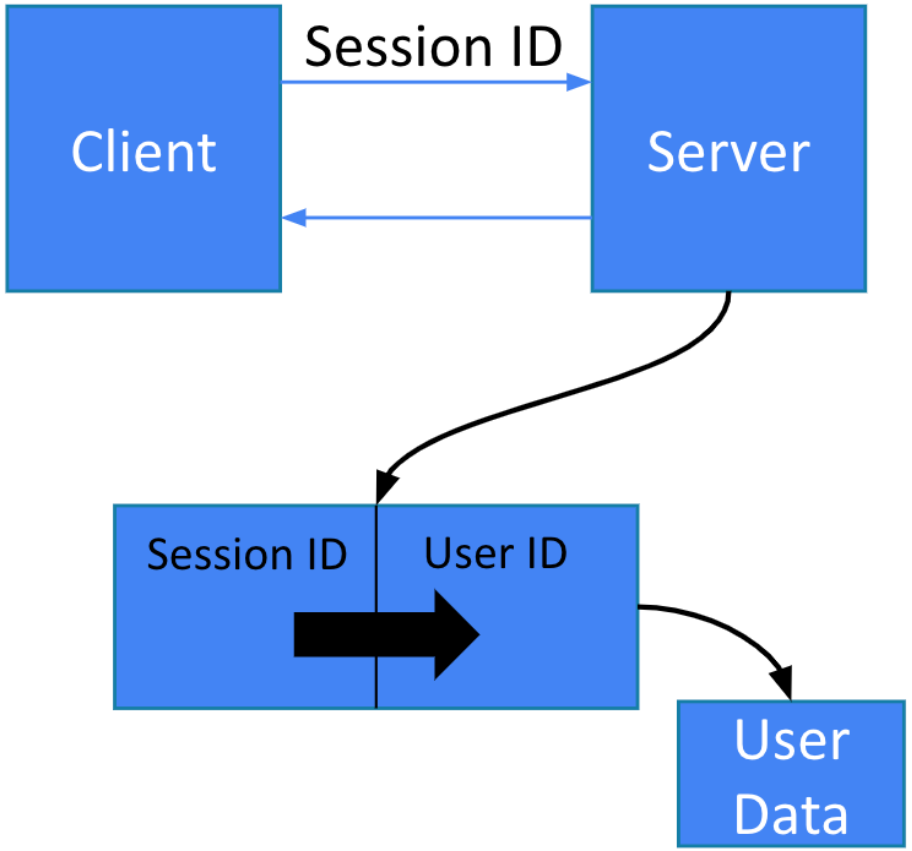
Network - Session

- Unique ID with the use of Universally Unique Identifier (UUID) packages
  - import "github.com/satori/go.uuid“
  - 128-bit number
- Makes use of
  - Cookie object to read User unique ID for example with a UUID
  - Append unique ID to URL

> “… only after generating 1 billion UUIDs every second for approximately 100 years would the probability of creating a single duplicate reach 50%.” -- Wikipedia

Network - Session

- Session ID sent to Server
- Server compares Session ID and retrieves User ID which will then extract user information.
- Process User Data and adapt data to response to request.



Network – Encryption in Login & Logout

- Use of package “golang.org/x/crypto/bcrypt”
  - Provides functions to do hash encryption
  - *func GenerateFromPassword(password [ ] byte, cost int) ([ ] byte, error)*
    - Property “cost” of encryption algo to determine difficulty of encrypt and decrypt
    - Used to generate signup of user
  - *func CompareHashAndPassword(hashedPassword, password [ ] byte) error*
    - Used to compare password and return result at login

Network – Encryption in Login & Logout

- Login
  - Check form entry and compare hash values
  - Create session
  - Set Server session to current User
- Logout
  - Delete session from Server session list
  - Set Cookie on User end to -1 MaxAge
  - Redirect to page
  - Can also update the remaining sessions to remove outdated session

Activity 14
Sessions and Login Logout

Network - JSON

- JSON
  - JavaScript Object Notation
  - Open standard file format
  - Human-readable format & Language independent
  - Commonly used by most programming languages

Network - JSON

- Go built in package for JSON.
  - Maps between JSON and Go values.
  - follows the definition of RFC 7159
- Sending Go data structure across HTTP as JSON
  - Encode with Marshal
- Receive JSON into Go data structure from URL
  - Decode with Unmarshal

Network - JSON

- Uses Go package “encoding/JSON”
- Online convertors for JSON to Go struct directly
  - Provide structure for receiving JSON
  - https://mholt.github.io/json-to-go
  - https://transform.tools/json-to-go
- JSON data given when request is made to URL
- Decode and unmarshal to Go variables upon receiving from request.
- Display the data on HTML response to a client request.

Network - JSON

- JSON example
  - Calls API with GET
  - Receives and unmarshal

```
func temperaturepage(ress http.ResponseWriter, req *http.Request)
{
   result, err := http.Get("https://api.data.gov.sg/v1/environment/24-
hour-weather-forecast")
   if err != nil {
      log.Println(err.Error())
   }
   JSONData, _ := ioutil.ReadAll(result.Body)
   var weather WeatherJSON
   err = json.Unmarshal(JSONData, &weather)
   if err != nil {
      fmt.Println(err)
   }
   // Remaining code to access weather variable.
   // WeatherJSON is the struct for JSON.
}
```

Network - XML

- XML
  - eXtensible Markup Language.
  - Readability is dependent on developer who created it.
  - Similar to HTML
    - Use of ***<tag>Element</tag>*** format
  - Generally used for data storing and transmitting.
  - Used as a form of configuration file.
  - Machine and human readable.

Network - XML

- Go built in package for XML.
  - Simple XML 1.0 Parser
  - follows the definition of RFC 7159
- Read in XML file and map data to a struct
  - Creation of struct using `xml : “<element tag name>”`
    - Struct field declaration, ***VariableName VariableType `xml: “<element tag name>”`***
    - Variable name must be Capitalized to allow for exporting of variable for accessibility.
  - Decode with Unmarshal
  - Access data in struct using init( ) function.

Network - XML

- Uses Go package “encoding/XML”
- Online convertors for XML to Go struct directly
  - Provide structure for receiving XML
  - https://jsonformatter.org/xml-formatter
  - https://www.onlinetool.io/xmltogo/
- Simple syntax
  - xml.unmarshal(***<readFile object>***, &***<struct object>***)

Activity 15
JSON and XML

Good Reads

- https://blog.gopheracademy.com/advent-2017/using-go-templates/