Network protocols

- IP content: version, source IP address; destination IP address
  - Each IP packet contains both a header (20 or 24 bytes long) and data (variable length). **The header** includes the IP addresses of the source and destination. **The data** is the **actual content**, such as a string of letters or part of a webpage.
  - When the router receives a packet, it looks at its **IP heade**r. The most important field is **the destination IP address, which tells the router where the packet wants to end up.**
- TCP/UDP: source port; destination port
  - **The source port number, and the destination port number** are contained in the first header word of each TCP segment and UDP packet.
- Notion of well-known ports **(you should know the well-known port for HTTP)**
  - Port 80: HTTP use TCP/UDP - **the port that the server "listens to" or expects to receive from a Web client**, assuming that the default was taken when the server was configured or set up.
  - Port 53: **zone transfers, for maintaining coherence between the DNS database and the server**.
- TCP vs. UDP characteristics, TCP three-way handshake
  - **TCP:** - connection, REALIABLE, byte streams, 3 ways handshake**. TCP sends streams that are combined into a buffer**.
    - 3 ways hand shake : **1/ syn, 2/syn + ack, 3/ack**
  - **UDP**: - user datagrams, no reliability, FAST. UDP send "packets" that are never combined and **only receive 1 package.**

- **IPv4** : 32 Bit decimal number. Example: 187.89.31.226:80, can be convert to decimal or heximal

Socket:

- **UDP/SOCK_DGRAM is a datagram-based protocol, that involves NO connection**. You send any number of datagrams and receive any number of datagrams. It's an "unreliable service".
- **TCP/SOCK_STREAM** is a "reliable" or "confirmed" service.

- **underlying transport protocol? –** datagrams: udp and sockets for tcp
- **socket functions**:
  - socket(ipv4,sock_dgram/sock_stream):initialize, bind(): assign a local IP address and port with a socket, connect()- optional for udp can also use sendto(). Listen(), accept() – optional for tcp, and udp don't use. Close() : both udp and tcp
  - **getaddrinfo() usage: list of IPV4/6 > 0**
  - bind() : associate local port and IP address to that socket, the server needs bind where client knows where/which port it can go, because client can choose the port. If client doesn't set bind, server can set for you

b/ **how each function differs or is applicable (or not) to client or server**

- sock_stream : one socket per client/ sock_degram: one socket handles everything
- server+client: socket() + close()
- only client: connect()- client gets a port. Different from bind() that we don't care which port the client is using.
- only by server: bind() needs to specify which port it receives data on/listen()/accept()

- read()/write() operations on (blocking) sockets of both types (i.e., SOCK_DGRAM and SOCK_STREAM)

- sock_stream: read the rest on the next call / sock_degram: **trunk case(cut off)** if its less than what in the buffer, if its more than the buffer, it will just read the buffer amount and wait no more

- what happens if remote/serve side closes connection: local is the client
  - Read/write return 0/ Client get closed if its making call to the server and server is closed
  - Both UDP and TCP, it blocks and waits if there is no data in the buffer.It does not return zero

URL: parts breakdown: protocol, hostname, port,

<Protocol> :// <Hostname> : <Port> / <URI(where client pass argument to sever)>. Ex. https://www.example.com:8080/index.html

DNS: Input(www.hedafa.com) -> DNS STUB resolver (Queries) + DNS Recursion resolver (Queries + resources)(both way) -> output (192.0.2.1)

HTTP

Request line: <Method> <URL> <Version>/ Ex. GET https://www.example.com:8080/home/index.htm HTTP/1.0 /r/n/r/n

Response line: <version> <return code> <English description of return code>\r\n

Headers / end-of-headers signal: <Name> : <Data>

Content-length header:

Tells the recipient how long the request/response BODY is – needed to determine when the server is done sending after reading in the double return (\r\n\r\n)

GET vs. POST

Get request does NOT include a body/ **Get response** DOES include a body. Get: asking for a resources

Post request DOES include a body/ **Post response** DOES include a body. Post: Asking to create a new resources

CGI

- How CGI is implemented by a Web server (Web server perspective)
  - how to run CGI: fork(), dup2(), close() right after dup2(), setenv(queries string), execve(setenv) – start running the code
  - fork()-> pthread_create(), wait()->pthread_join(), exit()-pthread_exit()
  - YOU CANT IMPLEMENT CGI WITH THREADS, you use daemons
- How a CGI program operates inside a Web server (CGI program perspective)
  - How to get input – queries string that set by the server
  - QUERY_STRING (stuff after "?" in URL) is an environment variable youcan get with getenv()
  - CGI program uses the passed in cgiargs from the URI before setting QUERY_STRING to cgiargs, dup2 its stdout to the connectionfdand then execve'd
  - If (Fork() == 0) { setenv("QUERY_STRING", cgiargs, 1)/ Dup2(fd, STDOUT_FILENO)/ Execve(filename, emptylist, environ); }
  - The URI minus the query string is typically the path to the CGI program (This iswhat the server gives to execve()) - stuff before the "?"
- How a client passes data (both GET and POST data) to a Web server: As Params in the query string (both GET & POST) or in request body (POST)

Threads / Semaphores

- **shared between threads: heap, file descriptors, global variable, Static variable**(store in "permgem" of heap, use to exchange information between threads). Stack(un-sharable, hold only the local variable and not the variable on the heap
  - **thread**(a unit of execution) will have its own stack, but all the **threads** in a process will **share** the **heap**. And FASTER THAN process
  - threads share data while processes do not, the stack is not shared for both processors and threads. **Thread context : thread ID, stack, program counter**
  - **Processes** don't share memory with other processors. **To share info, they must use IPC**
- basic properties of a semaphore: mutex lock. It can have only two values – 0 and 1. Its value is initialized to 1. It is used to implement the solution of critical section problems with multiple processes. Sepharmore - used to provide synchronization of tasks/ low-level synchronization mechanism. Semaphore will always hold a non-negative integer value.
- semaphore initialization (sem_init), wait (sem_wait), post (sem_post)
  - sem_init: initialize
  - sem_wait: check and decrement, make sure its never below 0
    - if sem>0**: decrement** and continue; if sem = 0 : wait until sem>0
  - sem_post**: increment** and wake 1 thread, just only 1 because it has a bunch of threads waiting on it.
- data sharing / protection / thread safety
- **creating and joining threads, detached vs. joinable threads** – review lab and code
  - pthread_detach(thread) – thread is no longer able to synchronize , release its resources
  - pthread_join(thread) – wont release any resources even after the thread runs its cources
- Using a binary semaphore as a mutex
- **Shared data paradigms - principles and example code**

Producer-consumer: everyone is the writers cause they will change the queue, only one person have access at one (3 sephamores)

-Mutex On data - tells us if we can access the shared data. Binarysemaphore, data is available to change 1, or data is not 0

-Items - Represents items available for the consumers. Starts at 0, increments when the producer calls post on items and puts an item on the queue

-Slots - Represents the **number of slots** we have in our Queue,**initialized to 4 if our queue size is 4. (can be #n)**

Producer

- calls wait on slots - checks to see if there is a sloton the queue, this decrements slots.
- calls wait on Mutex - checks to see if data issavailable
- puts the item on the Queue
- Calls post on Mutex to say data can be changed again.
- Finally calls Post on Items to let the consumer know there is an item

Consumer

-Calls wait on items - If yes decrements items

-Calls wait on Mutex

-If both return true, it pulls the item off the queue

-Calls post on Mutex

Calls post on slots - It pulled the item off the queue and so there are most spots in the queue and excuse what on the

Readers-writers:

- If there is a lot of readers, then the writers will go to starvation
- If a writer has a resource checked out then no readers are allowed toread to avoid corrupt data.
- The most common solution - Writer Mutex and a reader semaphore. File checkout the writer mutex or try to get that mutex.
- readers : call semwait() – checking the Rlock = 1 and Wlock = 1. Semwait() on # of locks -> everything now at "0" -> POST
- writers(still locked) – if we are last reader, put the writer lock back