Week.3.txt

- January 25th, 2021
    - Outline
        - Principles of network applications
        - Web and HTTP
            - This is where we are at
        - DNS
        - Socket Programming
    - Web and HTTP
        - HTTP is used every time you load a web page in your browser
            - It is an application layer protocol
        - Web pages consist of multiple objects
            - Some objects are dynamic, and some are static
                - Dynamic objects can be created by the time of day or be user specific content
            - Examples of objects:
                - HTML file
                - JPEG image
                - Java applet
                - Audio/Video file
                - Etc.
            - A web page consists of a base HTML-file which includes several referenced objects
                - Each object is addressable by a URI
                    - i.e. www.someschool.edu/someDept/pic.gif
                        - The host name is: www.someschool.edu
                        - The path name is: someDept/pic.gif
                        - This entire thing is a URI
                            - Host + Path name == URI
                            - URI stands for uniform resource identifier
                - When you combine the URI with the protocol, you get a URL
                    - URL stands for Uniform Resource Locator
                        - i.e. https://www.website.edu/page/index.html
    - HTTP Overview
        - HTTP stands for HyperText Transfer Protocol
            - It is the web's application layer protocol
            - Every browser uses it
        - Web services are based off the client-server model
            - Client
                - Corresponds to the browser application that runs on your device
                - Sends requests and recieves content from web servers
                    - The recieved information is rendered by/on the browser/screen
            - Servers
                - Typically run in data centers
                - Recieve requests from clients, and send objects in

response to those requests
    - Protocols determine:
        - Format, semantic, and syntax of the message
        - Specify the order of message that's being exchanged
        - Corresponding action of what client/server will do upon receiving the message
- HTTP Overview Continued
    - The lower layer transport protocols need to be specified
        - HTTP 1.1 and 1.0 utilize TCP for the transport layer
            - TCP is used for reliability
        - Newer versions of HTTP (3.0) use a modified version of UDP
            - HTTP 3.0 was invented by Google
        - There are advantages and disadvantages in using each transport layer protocol
            - TCP is reliable but can be slow
            - TCP is susceptible to ahead of line blocking
                - i.e. Some content needs to be transferred before you can request the next one
    - Steps In Using TCP:
        1. Client initiates TCP connection to server
            - Essentially this creates a socket from client to server
            - Port used is 80
                - This port is associated with HTTP
        2. Server accepts TCP connection from client
        3. Requests and responses are exchanged between client and server
        4. TCP connection is closed
    - HTTP is stateless
        - The server only reponds to the message that the client sends in the request
            - It maintains no information about past client requests
        - However, cookies are used to save states and maintain session specific information
            - The implementation of states are complex
                - TCP is a stateful protocol
                    - Maintains information about:
                        - Was connection established
                        - Has connection been teared down
                        - Did the message get delivered
                    - TCP is a lot more complex than HTTP
            - If the server/client crashes, then their views of "state" may be inconsistent and must be reconciled
- HTTP Connections
    - There are different types of HTTP protocol versions and connections
        - This changes the way we can send multiple objects within or across multiple TCP connections

    - After loading the base HTML file, the browser searches for other URIs, and requests those pages accordingly
        - This begs the question:
          Should you wait for each object to transfer before requesting the next one, and each object has its own TCP connection?
            OR
          Should you send everything over a single TCP connection
    - Non Persistent HTTP
        - At most one object will be sent over one TCP connection
            - After an object is sent, the TCP connection is closed
                - This is repeated for each object
        - HTTP 1.0 uses non-persistent HTTP
            - HTTP 1.0 is less utilized nowadays
        - Requires multiple TCP connections to load content
    - Persistent HTTP
        - Allows multiple objects to be sent over a single TCP connection between client and server
        - HTTP 1.1 uses persistent HTTP by default
            - HTTP 1.1 is the most used
- Non-Persistent HTTP Example
    - The following URL is entered into a browser:
      www.someSchool.edu/someDepartment/home.index
      (Contains text, references to 10 JPEG images)
    - Steps
        - 1.a
            - HTTP client initiates TCP connection to HTTP server at: www.someSchool.edu on port 80
        - 1.b
            - HTTP server at host is waiting for TCP connections at port 80
            - It accepts incoming connections and notifies the client
        - 2.
            - HTTP client sends HTTP request message (containing URL) into TCP connection socket
                - The message indicates that client wants the object: someDepartment/home.index
        - 3.
            - Upon receiving the request, the server sends a response message containing requested object, and sends message into its socket
                - The server encapsulates the base HTML page of the Department's web page
        - 4.
            - Server closes HTTP connection
        - 5.
            - The client recieves the HTML file, displays it, and finds 10 referenced JPEG objects

            - The time between the request message sent, and the response message is received, is referred to as the Round Trip Time (RTT)
        - 6.
            - Steps 1-5 are repeated for each object in the base HTML file
                - This is done in parallel
        - Note: The TCP connection is closed after receiving the base HTML file, and each referenced object requires its own TCP connection to be retrieved
    - Non-persistent HTTP incurs more overhead because a TCP connection is established for every object/file
        - Establishing a TCP connection is time consuming
    - Referenced objects can be retrieved in parallel
        - You don't have to retrieve each object one by one, they can be retrieved at the same time
    - Typically, servers limit how many TCP connections they accept from each host
        - This is because TCP consumes server resources
        - If too many TCP connections are sent, then they must wait for current TCP connections to terminate
- Persistent HTTP Example
    - The following URL is entered into a browser: www.someSchool.edu/someDepartment/home.index
    - Steps
        - 1.a
            - HTTP client initiates TCP connection to HTTP server at: www.someSchool.edu on port 80
        - 1.b
            - HTTP server at host is waiting for TCP connections at port 80
                - It accepts incoming connections and notifies the client
            - Steps 1a and 1b are identical for both persistent and non-persistent HTTP
                - This handshake cannot be skipped, thus this latency exists for both persistent and non-persistent HTTP
        - 2
            - HTTP client sends HTTP request message into TCP connection socket
                - The message indicates that the client wants an object at: someDepartment/home.index
        - 3
            - HTTP server receives request message, and sends a response message containing requested object into its socket
        - 4
            - HTTP client recieves response message containing HTML file, displays file, and finds 10 referenced

        JPEG objects
    - 5
      - Steps 2-4 are repeated for each referenced object, sequentially
        - This is done using a single TCP connection from step 1
          - In non-persistent HTTP, step 1 is repeated for every object
    - 6
      - Once the web page has been loaded, the HTTP server closes the TCP connection after some time, or when the client closes the connection
  - A single TCP connection is reused while retrieving different objects in the same page
- Persistent HTTP
  - There are two variants of persistent HTTP
    1. Persistent without pipelining
       - Objects must be retrieved sequentially
       - Client issues new request only when previous response has been receieved
         - After getting the base HTML, each additional object is retrieved one-by-one. Once the first object is receieved, the second object is requested, and so on so forth
       - Each object takes one round trip time (RTT)
    2. Persistent with pipelining
       - Client sends request as soon as it encounters a referenced object
         - After parsing the base HTML file, the client immediately sends a request to fetch all referenced objects
           - Client does not have to wait for the completion of previous objects to send requests for subsequent objects
         - Can take as little as one round trip time (RTT) for all referenced objects
       - Default in HTTP 1.1
  - Pipelining VS. Non-pipelining is application behavior
  - In practice, pipelining may not be faster than non-pipelining due to traffic congestion, network delays, large files, etc.
    - In pipelining, the objects share the same TCP connection and pipe
- HTML Request Message
  - HTTP messages, request and response, are in a human readable format
    - This format is called ASCII; American Standard Code For Information Interchange
  - HTTP messages start with a request line
    - It contains the method, followed by a space, the URL,

        another space, the version of the protocol, and ends with a carriage return
          - Below the request message are the header lines
            - Below the header lines is the (entity) body
      - Examples of methods in the request line:
        - GET
        - POST
        - HEAD
      - The carriage return line feed at the start of each line indicates end of header line
      - The keep alive time indicates how long to keep the connection before discarding it
        - Once the time is reached, a new connection needs to be established
- Method Types
  - HTTP 1.0 has the following methods:
    - GET
      - Request an object specified by the URL
    - POST
      - Request that the server accept the entity closed in the request
    - HEAD
      - Asks the server to leave the requested object out of the response message
        - Retains meta data/info
  - HTTP 1.1 has the following methods::
    - GET, POST, HEAD
      - This is from HTTP 1.0
    - PUT
      - Uploads file/resource in entity body to path specified in URL field
    - DELETE
      - Deletes resource specified in the URL field
        - Requires authentication of some kind
- Uploading Form Input
  - Often times we need to input information into a form, send it to the server, and get some response back
    - There are two ways to do this
      1. Utilize the POST method
        - The information you want to send in the form is included in the body of the HTTP request message
          - i.e.

            ```
            POST /test/demo_form.asp HTTP/1.1
            Host: w3schools.com
            name1=value1&name2=value2
            ```
          - Reloading the page or pressing the back button gives you a warning, asking if you want to resubmit the data
          - This information is protected in the body, and is not visible, unless they check the

body
                - The data is not cached because it is part of the body of the HTTP request message
            2. Use the GET method
                - The information is uploaded in the form of name-value pairs in the URL field of the request line
                    - i.e. /test/demo_form.asp?name1=value1&name2=value2
                - All the information is included in the URL
                    - Reloading or pressing the back button does not do anything
                    - This information is visible to everyone who looks
                    - It can be cached because the data is part of the URL
    - Summary Table:

| | GET | POST |
|---|---|---|
| BACK Button / Reload | Harmless | Data will be resubmitted (the browser should alert the user that the data is about to be resubmitted) |
| Visibility | Data is visible to everyone in the URL | Data is not displayed in the URL |
| Cached | Can be cached | Not cached |

- Security Vulnerabilities
    - There are potential problems of using GET
        - If the URL is very long, an attacker can cause a buffer overflow in a server with buggy software by getting/posting a very long URI
            - i.e. A URI with 50,000 characters at the end
                - This happens if the server doesn't check the size of the input
        - Since the information is transferred in clear-text, an attacker can retrieve important details
            - i.e. Customer name, customer ID, product in cart, price, etc.
- HTTP Response Message
    - A response message starts with a status line
        - It contains protocol version, status code, status phrase
            - i.e. HTTP/1.1 200 OK\r\n
                - Lines are separated with '\r\n'
        - Other information includes: Date, server, last modified,

etc.
    - HTTP has a conditional GET, that tells the server to only respond to the message if the contents of page are modified after a certain time
        - This helps in avoiding reloading unchanged pages and reduces traffic
            - The web server handles this check
    - A regular HTTP GET responds with the URI in the body of the HTTP message that is sent back to the client
- HTTP Response Status Codes
    - Status codes appear in the 1st line in server-to-client response message
    - Some response status codes are:
        - 200 OK
            - Request succeeded; request object later in message
        - 301 Moved Permanently
            - Requested object moved, new location specified later in this message
                - i.e. (Location: ...)
        - 400 Bad Request
            - Request message not understood by server
        - 404 Not Found
            - Requested document not found on this server
        - 505 HTTP Version Not Supported
            - i.e. Client running HTTP 1.1 trying to communicate with server running HTTP 1.0
    - There are a variety of status codes associated with response messages
- Trying Out HTTP
    - It's possible to load a web page using your console
        - You don't have to use the browser
    - i.e.
        1. Telnet to your favorite web server
            - i.e. telnet www.google.com 80
        2. Type in a GET HTTP request
            - i.e. GET /index.html HTTP/1.1
              Host:www.google.com
        3. Look at the response message sent by HTTP server (OR use Wireshark to look at captured HTTP request/response)

- January 27th, 2021
    - User Server State: Cookies
        - HTTP is a stateless protocol
            - The response messages in HTTP entirely depend on what has been sent in the request message
                - HTTP does not take past history into account when determining the response
                - Persistent states are useful for:
                    - Maintaining previous browsing session

        - i.e. Shopping cart at an online website
      - Customizing content
        - i.e. Change theme to dark mode instead of default
      - Restriction
        - i.e. Paywalls for news articles
  - EU requires sites to follow GDPR regulations to increase user privacy and control over their data
    - Sites need permission to store cookies and track users
  - Cookies utilize the information content that is carried in an HTTP message, and combine the implementation on the browser and server side
    - This allows a user's information to be tracked throughout their session
    - Cookies do not modify the HTTP protocol in any way
      - The headers and information remain the same
  - There are four parts to a cookie
    1. Cookie header line of HTTP response message
    2. Cookie header line in next HTTP request message
    3. Cookie file kept on user's host, managed by user's browser
    4. Back-end database at website
  - Example of cookies in action:
    - Susan uses her laptop to access the Internet from the PC
    - Susan visits Amazon for the first time
    - Amazon creates a unique ID and adds an entry in the backend database for the cookie ID
      - If Susan visits Amazon later in the day, the cookie ID and other state information is sent to Amazon in the HTTP message body
- Cookies: Keeping State
  - Cookies are stored locally on your browser
  - Each cookie has a unique ID associated to it
  - The browser may warn the user if a cookie is created and stored locally
  - Cookies allow websites to generate a custom response/ information for the user
  - Cookies may be associated with an expiration timer
  - If the user deletes cookies for a website, then cookies are generated again when they visit the website
  - Cookies add state to HTTP; a stateless protocol
- Cookies: Usage
  - Cookies can be used for:
    - Authorization
      - Allows for one time authentication, so the user does not have to re-enter information over and over again
    - Shopping carts
      - Your shopping cart stays intact after closing the website or browser session
    - Recommendations

            - The web server can customize the content it shows you based on your viewing preferences
                - i.e. YouTube showing you dozens of cat videos, because you clicked on one cat video
        - User session state
            - The website can maintain state, customize content, and allows easier access
                - i.e. News website allowing you to read 3 free papers every month, and anything more requires subscribing. Deleting your cookies for the website may get around this paywall
    - Cookies and privacy:
        - Allow sites to learn a lot about you
            - i.e. Google serving ads "most relevant to you"
        - Third party cookies can track users aross multiple sites
            - If multiple sites use the same advertising network, then they can easily track your browsing history
        - Cookies are a privacy nightmare
            - Banners/ads across multiple websites can track a user's web history
            - It's a tradeoff between privacy and convenience
- Outline
    - Principles of network applications
    - Web and HTTP
    - DNS (Domain Name System)
        - This is the next topic
    - Socket programming
- Domain Name System (DNS)
    - Is essential to internet service
    - Network hosts are addressed using IP address(es), not domain names
        - The IP address is the dot decimal representation
            - i.e. 130.113.64.30
        - This is evident when using `traceroute`
            - It will return the IP address next to the domain name
    - The primary purpose of the domain name system is to map between IP addresses and hostnames
        - The DNS makes it much easier for both humans and computers
            - Humans are more comfortable using names for addressing
            - Computers are better/faster at using numbers for addressing
        - The DNS provides an intermediate layer between domain name and IP address
            - This layer follows good software engineering practices
                - i.e. The domain name "www.google.com" will remain constant, but next year it may be

                    hosted on a different machine using a different IP address. This drastic change does not affect the availability of the website
        - The DNS decouples the domain name from the IP address of the host
            - Even if the IP address changes, the domain name will remain the same, allowing users to continue using the website as if nothing changed
- DNS Service
    - The job of the DNS service is to translate hostname to IP address
    - Servers can have an alias; a different name
        - The canonical name is what the general public uses to access the site
            - i.e. www.apple.com
        - The alias is the alternative name
            - i.e. www.cdn-123.apple.yyz.com
                www.cdn-002.apple.nyc.com
    - If the website is hosted on several web servers, then the canonical domain name can map to multiple different IP addresses
        - Can be used for load balancing
            - `traceroute` will warn you if a domain name has multiple addresses
- DNS: Domain Name System
    - There are two parts to the domain name services
        1. Physical part
            - Refers to the information the DNS contains
            - This is a distributed database that is implemented in a hierarchical manner, and consists of many name servers
                - The key information that needs to be stored is the mapping between host name and (multiple) IP addresses
                - Since the database is gigantic, it needs to be done in a distributed and hierarchical manner
        2. Protocol part
            - This is the application layer protocol
                - Focuses on interacting with the DNS, querying a host name, and getting a response
                    - i.e. How does address/name translation work?
                - Looks at how the message is being encoded, and how can it be interpreted?
    - The DNS is not implemented on a giant centralized server because:
        - Single point of failure
            - Information needs to be distributed across different servers and continents
        - Traffic volume
            - Huge amount of traffic being routed to a single

        point is bad architecture, and makes load balancing very hard
    - Latency to access distant centralized database
      - DNS lookups should be very quick
    - Maintenance
      - Updating information in a centralized database is problematic
        - The volume of information is way too much
    * This does not scale *
- Distributed Hierarchical Database
  - At the top of the hierarchy is Root DNS Servers
    - There are only 13 logical root DNS servers across the globe
      - But there can be more than 13 physical ones
  - After the root DNS servers are the top level domain (TLD) servers
    - i.e. ".com", ".org", ".edu", ".info", etc.
  - Below the top level domains (TLDs) are the authoritative DNS servers
    - These are typically associated with an organization
    - They are called authoritative because they resolve the mapping between the host name, and the IP address of hosting servers in their organizations
      - i.e. McMaster has its own authoritative DNS server for any of the services on campus
      - Authoritative DNS servers are the last resort in DNS querying
        - i.e. If it can't find the resulting IP address at other DNS servers, it will check the authoritative DNS servers as a last resort
  - Local DNS servers are not part of the hierarchy
    - They are often times provided by your ISP (Internet Service Provider)
    - Stores and caches DNS resolutions, and then uses it to answer future DNS queries
- DNS: Root Name Servers
  - There are 13 logical servers in the world
    - They're physically located in different continents; quite distributed
  - Root name servers will contact authoritative name servers for the appropriate domain if the mapping is not known
    - This happens if the top level domains (TLD) do not know a mapping
    - It uses the suffix to determine which DNS server to contact to resolve the mapping
      - i.e. The "edu" TLD is contacted for a site ending in ".edu"
- TLD & Authoritative Servers
  - The TLD servers are responsible for different domains
    - i.e.

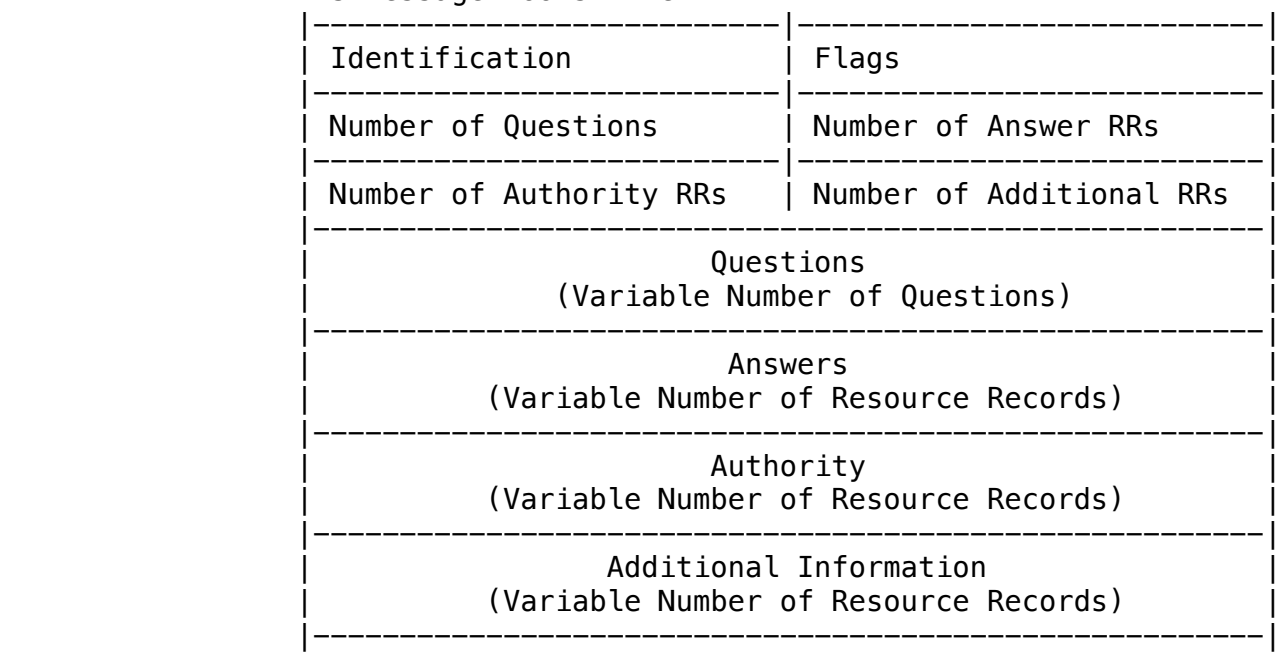
      ".com", ".edu", ".org", ".net"
    - i.e.
      ".uk", ".fr", ".jp", ".ca"
        - These are top level country domains
  - The bottom of the hierarchy has authoritative DNS servers
    - These are associated with different organizations that are in the position to provide a source of host name to IP mappings for organizational servers
      - i.e. Webmail service
    - Maintained by organization or its service provider
- Local Name Server
  - Does not strictly belong to the hierarchy
  - Typically, ISPs may provide the local name server
    - Also called "default name server"
    - Each ISP (residential ISP, company, university, etc.) has one
  - When a host makes a DNS query, it is sent to its local DNS server
    - If the mapping cannot be resolved, then the local DNS server will forward the query along the hierarchy
      - i.e. Local name server acts as a proxy
- DNS: Caching & Updating Records
  - Servers can learn the mapping between domain name and IP address
    - This is done by caching the mapping
      - The query response that is returned is cached
    - Cache entries disappear (timeout) after some time
      - Each cache entry is associated to a timeout value
  - TLD servers are typically cached in local name servers
    - Thus, root name servers are often times not visited
  - The specification for this is designed and maintained by the IETF
    - i.e. RFC 2136
- Queries
  - There are two types of queries to resolve the host name
    1. Recursive Queries
      - The server is asked to get the answer for you
        - If it fails, then it will send the query to the next server along the hierarchy
          - This may send the query further down until you reach the authoritative DNS servers, which will eventually return the query, and send it back the requesting hosts
          - If the mapping is cached, then it does not need to send the query to other DNS servers
      - i.e. Assume the host is mills.cas.mcmaster.ca and it wants to communicate with gaia.cs.umass.edu. It will send the query to the local DNS server. If the local DNS server cannot resolve the query, then it will send the query to the root DNS

server. If the root DNS server cannot resolve the query, then it will send it to the TLD DNS server. If TLD cannot resolve the query, then it will check the authoritative DNS servers, and they will resolve it. The resolution is sent back to the DNS server, the result is cached, and this process repeats until it reaches the requesting host
        2. Iterative Queries
            - The local DNS server resolves the host name for you
                - It asks each DNS server to resolve the mapping
                    - First it will ask root DNS, then TLD, and so on
                    - The intermediate DNS servers cannot cache the response because they do not communicate with each other
                - The merit in using iterative queries is to not overload the DNS servers with queries
            - i.e. If the local DNS server does not have the information, then it will ask the root DNS server. If the root DNS server cannot resolve the host name, it will tell the local DNS server to ask the TLD DNS server. If TLD DNS server can't resolve the host name, then it will tell the local DNS server to ask an authoritative DNS server. The authoritative DNS server will resolve the host name, and return the mapping to the local DNS server, which will send the resolution to the requesting host
    - DNS Message Example
        - The `dig` command line tool returns information pertaining to DNS queries
            - i.e. dig www.mcmaster.ca
        - The answer section includes information about the server
            - i.e. www.mcmaster.ca 3600 IN CNAME pinwps02.uts.mcmaster.ca
              pinwps02.uts.mcmaster.ca 3600 IN A 130.113.64.30
        - The authority section contains information about what authoritative DNS servers are present
            - This information can be cached for subsequent queries
        - DNS messages utilize user datagram protocol (UDP)
            - UDP is used for sending and receiving query messages
                - UDP is preferred because of its speed
                    - TCP would be too slow, and is not needed because reliability is not a priority
                - If the UDP message is lost, then it is resent

- January 29th, 2021
    - DNS Message Example
        - DNS protocol is an application layer protocol that is

  utilized by network hosts to query the IP address of a particular domain name
    - It is also used among DNS servers to maintain consistency across a distributed database
  - DNS servers are organized in a hierarchical manner
    - At the very top are root DNS servers
      - Next is top level domain (TLD) DNS servers
        - TLDs are responsible for different domains
          - i.e. ".net", ".edu", ".com", etc.
      - The last level is authoritative DNS servers
        - Authoritative DNS servers are ultimately responsible for resolving the query, and return a mapped IP address
    - Caching allows DNS servers to resolve queries for host names they are not responsible for
  - DNS protocol is implemented via UDP as the transport layer
    - Servers will listen on port 53 for incoming queries
  - There are two ways to resolve a DNS query
    1. Recursive
    2. Iterative
  - DNS messages consist of several sections
    - The answer section is the most relevant
- DNS Records
  - DNS information is stored in resource records (RRs)
    - The format of a resource record is: (name, TTL, type, value)
      - TTL is Time To Live
        - It is used by the authoritative DNS for indicating how long the RR should remain valid
      - The value of the RR depends on the type
  - There are many types of DNS records:
    - Type=A (Address)
      - name = hostname
      - value = IP address
        - This is what the DNS needs to resolve and return
      - i.e. pinwps02.uts.mcmaster.ca. 3600 IN A 130.113.64.30
        - Hostname = pinwps02.uts.mcmaster
        - IP Address = 130.113.64.30
    - Type=NS (Name Server)
      - name = domain
      - value = name of DNS server for domain
      - i.e. uts.mcmaster.ca. 86400 IN NS pindns03.mcmaster.ca
        - Name = uts.mcmaster.ca
        - Value = pindns03.mcmaster.ca
    - Type=CNAME (Canonical Name)
      - name = hostname
      - value = canonical name
        - This is the real name of the server

        - i.e. www.mcmaster.ca 3600 IN CNAME
                                    pinwps02.uts.mcmaster.ca
            - Hostname  = www.mcmaster.ca
            - Canonical = pinwps02.uts.mcmaster.ca
      - Type=MX (Mail eXchanger)
        - name = domain in email address
        - value = canonical name of mail server
- DNS Protocol, Messages
  - DNS messages contain multiple resource records
  - There are two types of DNS messages; query and reply
    - Both have the same message format, but have different flags in the message header that indicate if it is a query or reply message
  - A DNS message looks like:




  - The message header contains the identifier
    - The identifier is a 16 bit number that is used to indicate the identifier of the query message
      - The response message uses the same identifier
  - The flag field specifies:
    - If the message is a query or reply
      - This is how the two are distinguished
    - Is recursion desired?
      - The alternative is iterative query
    - Is recursion DNS query available?
      - Were the previous DNS queries recursive?
    - Is the response from an authoritative DNS server?
      - Indicates if the response is a cached resolution
  - After the flag field, there is information about:
    - Number of questions included in the message
      - In a query message you may have 1 to N questions
        - i.e. Name, type fields, etc. for a query

        - Number of answer resource records (RRs) in the message
            - A reply message may have multiple answers, and no questions
                - i.e. RRs in response to query
        - Number of authority resource records (RRs)
            - i.e. Records for other authoritative servers
        - Number of additional resource records (RRs)
            - i.e. Additional "helpful" info that may be used
    - Complete information about questions, answers, authority, etc. are listed at the end of the DNS message
        - This is the message body and maps to the number included in the header field
    - All information in a DNS message can be viewed in Wireshark
        - This includes: header fields, questions, answers, and RRs
- More DNS Messages Example
    - The `dig` command can be used to make iterative queries to resolve the host name being looked up
        - To do this, `+trace` needs to be specified as an argument
            - i.e. dig +trace www.example.com
        - It will follow referrals from the root servers, showing the answers from each server that was used to resolve the lookup
        - In an iterative query, the local DNS server will work on your behalf to resolve the IP address of the requested host name
            - If the local DNS server does not have the resolution, it will query the root DNS server, then the TLD DNS server, and finally the authoritative DNS server
- DNS Messages Example
    - The `dig +trace` command will combine all the DNS reply messages, and print it out in one long trace
        - This is an iterative query because of the `+trace` argument
            - The local DNS server will query all (13) root DNS servers
                - The root DNS server will reply with TLD DNS servers
            - The local DNS server will query the TLD DNS servers provided by the root DNS
                - The TLD DNS server will reply with authoritative servers
            - The local DNS server will query the authoritative servers and get a resolution for the host name mapping
                - Any one of the authoritative servers will have the correct resolution to the requested host name

- Attacking DNS
    - DNS servers provide a very important service for the internet
        - DNS servers may domain names to IP addresses
            - This helps humans remember websites with ease, and makes it easier for computers to process requests
        - DNS lookups must be done every time you want to connect to a remote host
    - Due to their importance, DNS servers are targeted by hackers
        - One type of attack is a denial of service (DOS) attack
            - This is done by overwhelming the DNS server with a lot of traffic, and making it inoperable; it can't respond to DNS queries
        - i.e.
            - On October 21, 2002, the root DNS servers were bombarded with lots of traffic/requests for one hour
                - The attackers used an ICMP Ping attack on 13 root DNS servers
            - On February 6, 2007, the DNS servers were attacked for 24 hours
            - From November 30 to December 1st, in 2015, over 5 million queries were sent to root DNS servers, every second
    - Remedies/Solutions
        - Traffic filtering
            - Suspicious queries, or multiple queries of the same host are blocked or dropped, respectively
        - Distributing requests to other root servers
            - Even though there are only 13 logical root servers, many of them have multiple physical servers that help with load balancing
        - Local DNS servers cache IPs of TLD servers
            - This bypasses the need for querying root DNS servers
            - Queries answered locally remove unnescessary load from root servers
    - TLD servers are subject to attackers
        - On August 27, 2013, attackers went after China's ".cn" top level domain (TLD)
            - All the network hosts under the ".cn" domain became inaccessible because the TLD DNS server could not resolve the domain names
        - This is potentially more dangerous
    - Redirect Attacks
        - Man in the middle (MITM)
            - Intercept queries
                - i.e. User gets a fake DNS response crafted by an attacker, and ends up visiting a phishing website or a honeypot
        - DNS poisoning
            - Send bogus replies to DNS servers, causing them to

            cache them, and then sending them to other users
              - This changes the mapping between the host name and the IP address
                - Users can be sent to a phishing website or a honeypot
              - This poisons the local DNS server
        - Exploit DNS for DDOS
          - Send queries with spoofed source address to a target IP
            - i.e. Resolve different domain names to the same IP address, so multiple people flood the same site with requests
    - Solution for DNS attacks
      - There's not way to tell if DNS messages come from a legitimate DNS server or an attack
        - In theory, anyone can generate a DNS reply message because it is in clear text
          - An attacker can exploit this and target IP addresses
      - In order to combat DNS attacks, communication utilizes a crytographically signed response
        - An attacker cannot successfully spoof DNS messages, because cryptographic signature cannot be faked
          - i.e. DNSSEC, DNSCurve, Etc.
- Outline
    - Principles of network applications
    - Web and HTTP
    - DNS (Domain Name System)
    - Socket Programming
      - This is the next few slides
- Socket Programming
    - This is the interface between the application and transport layer
    - A socket is a door between application processes that allows them to send information from the sending host to the receiving host
      - An application on the sending host can send data to an application on the receiving host
      - The socket is the metaphorical door
        - It is the interface between the application and transport layer
    - There are two socket types for two transport services:
      1. UDP
          - Unreliable datagram
          - Connection-less
      2. TCP
          - Reliable
          - Byte stream oriented
          - Connected oriented
    - Application Example:

        1. Client reads a line of characters (data) from its keyboard and sends data to the server
        2. The server receives the data and converts characters to uppercase
        3. The server sends the modified data to the client
        4. The client receives the modified data and displays the line on its screen
- Socket Programming With UDP
    - In a UDP socket, you do not need to establish a connection before sending data
        - There's no handshake between server and client
        - The sending program explicitly attaches the destination IP address, and port number to EACH packet
            - This is because a connection is not established
        - The receiving program extracts the IP address and port number from the received packet
    - In UDP, transmitted data may be lost or received out-of-order
    - From an application viewpoint, UDP provides unreliable transfer of groups of bytes (aka datagrams) between client and server
- Client/Server Socket Interaction Via UDP
    - Client point of view:
        1. Create a socket
            - i.e. socket()
            - i.e. A UDP socket
        2. Send the data to the server
            - i.e. sendto()
            - Must provide the IP address and port number of the UDP server along with the message/data
                - In UDP, this applies to every single packet
        3. Wait for response from server
            - i.e. recvfrom()
            - Client waits until server responds to the message
        4. Close the socket
            - i.e. close()
            - The socket is closed when everything is done, and client/server do not need to exchange any more information
        - In UDP, every single packet sent/recieved to/from the client/server must have an IP address and port number attached
            - This is because in UDP, there is no handshake between client and server that sets up a connection
    - Server point of view:
        1. Create a socket
            - i.e. socket()
        2. Bind to a specific port
            - i.e. bind()
            - This is analogous to waiting at the door and

expecting a message to come through the door
- The port number is the application's address
- When the transport layer receives the packet, it will look at the port number to determine which application process the message needs to be delievered to
- i.e. DNS servers bind to port 53

3. Wait for incoming messages
- i.e. recvfrom()
- The server will wait until it receives a message/ data from a client
- It is blocked from doing anything else until it receives a datagram from the client

4. Process the data
- The server will process the data
- i.e. Perform complex mathematical simulations
- i.e. Spell and grammar check an essay

5. Reply to the client
- i.e. sendto()
- The server will construct another message, and send it to the client
- Server must include the client IP address and port number when sending the message
- In UDP, this applies to every single packet

- The server needs to be always on, and always anticipating a request

- Example App: UDP Client
  - i.e. UDPClient.py

```
# Include Python's socket library
from socket import *

# This is the IP address of localhost
host = '127.0.0.1'

# The port number
serverPort = 12000

# Create UDP socket
clientSocket = socket(AF_INET, SOCK_DGRAM)

# Get user keyboard input
message = input('Input lowercase sentence: ')

# Attach server name, port to message; send into socket
clientSocket.sendto(bytes(message, 'utf-8'), (host,
                                              serverPort))

# Read reply (buffersize) chars from socket into string
moddedMessage, serverAddress = clientSocket.recvfrom(2048)
```

```
        # Print out received string
        print(moddedMessage.decode('utf-8'))

        # Close socket
        clientSocket.close()
```

- 'localhost' is mainly used for testing
  - This is a private address
- 'SOCK_DGRAM' is a UDP socket
  - Uses UDP to send packets
- In UDP, every packet needs to specify destination IP address and port number
  - This is because every datagram could be sent to a different host or application
    - This is not the case for TCP, because it sets up a connection ahead of time
- The 'recvfrom()' function returns a message from the server and the server's address
  - 2048 is the buffer size
    - This is the size of the message

- Example App: UDP Server
  - i.e. UDPServer.py

```
        # Include Python's socket library
        from socket import *

        # The socket will run on this port
        serverPort = 12000

        # Create UDP socket for server
        serverSocket = socket(AF_INET, SOCK_DGRAM)

        # Bind socket to local port number 12000
        serverSocket.bind(("", serverPort))
        print('The server is ready to receive')

        # Loop forever
        while 1:
            # Read from UDP socket into message, getting client's
            #   address (client IP and port)
            message, clientAddress = serverSocket.recvfrom(2048)
            modifiedMessage = (message.decode('utf-8')).upper()
            # Send upper case string back to the client
            serverSocket.sendto(bytes(modifiedMessage, 'utf-8'),
                                                clientAddress)
```

  - 'SOCK_DGRAM' is a UDP socket
  - The difference between a client and server is that the server needs to bind to a port
    - The client does not explicitly bind to a port
      - This is taken care of by the operating system, and

            is included in the header that is received by the server
              - The number is arbitrarily selected by the operating system
    - The 'recvfrom()' function returns the message and client address
      - The client address is used when the server returns a message to the client
      - 2048 is the buffer size
          - This is the size of the message
- Socket Programming With TCP
    - TCP maintains a connection between the server and the client
      - The client must connect to the server
          - A server process must first be running
          - The server must have a socket (door) that accepts a client's requests/messages
    - Before any application data can be exchanged, the server needs to be put in a 'ready' state where it can accept requests from the client
      - This is in addition to binding the server to a port
          - Then, the client can connect to the server by creating a TCP socket, specifying the IP address, and port number of the process on the server
            - This is how the client TCP establishes a connection to the server TCP
    - TCP servers are passively waiting for connection requests
      - TCP clients are actively opening connections and sending requests to TCP servers
      - TCP servers and clients can only exchange data after a connection has been established
    - The IP address and port number needs to be specified during the connection request stage
      - However, there is no need to include the IP address and port number in subsequent (send or receive) messages
    - When contacted by a client, the server TCP creates a new socket for the server process to communicate with the client
      - This allows the server to talk with multiple clients
      - Source port numbers are used to distinguish clients
- Client/Server Socket Interaction Via TCP
    - Client point of view:
      1. Create a socket
          - Similar to UDP
              - i.e. socket()
      2. Send connection request to TCP server
          - The connection request specifies the IP address and port number of the TCP server
          - i.e. connect()
      3. Write or read data from the established TCP pipe
          - The TCP client can send data to the TCP server, and receive a response, with new data

        - i.e. write(), send()
    4. TCP socket is closed
        - Once the socket is closed, an EOF notification is sent to the TCP server, so it can close its connection and terminate any process/threads pertaining to the pipe
        - i.e. close()
- Server point of view:
    1. Create a socket
        - The process is similar to UDP
            - i.e. socket()
    2. Bind the server to a particular port
        - This tells the socket that the process is going to work with a specific port
            - This changes information in the socket data structure
        - i.e. bind()
    3. The server starts listening in order to accept incoming connections
        - It listens on the specific port that it is binded to
            - This is unique to TCP
        - The TCP socket is blocked until there's an incoming connection request
        - i.e. listen()
    4. Waiting for a connection request from a TCP client
        - After calling 'listen()' the TCP socket immediately enters the 'accept()' stage
            - The TCP socket is blocked until it gets a request from a TCP client
        - i.e. accept()
    5. Once a connection is received from a TCP client, a TCP connection is established, and this socket can be used for exchanging data
    6. TCP server accepts data from TCP client, modifies it, and sends it back to the TCP client
        - i.e. write(), read()
    7. The TCP pipe is closed when it is no longer needed
        - i.e. close()
    - In TCP, one socket is used for setting up a connection, and another socket is used for exchanging application data
- Example App: TCP Client
    - i.e. TCPClient.py

```
# Include Python's socket library
from socket import *

# This is the server name
serverName = '127.0.0.1'
```

```
        # This is the port that the server is binded to
        serverPort = 12000

        # Create a client socket
        # SOCK_STREAM corresponds to a TCP socket
        clientSocket = socket(AF_INET, SOCK_STREAM)

        # Create TCP socket for server at remote port of 12000
        # Client actively connects to the server (at name & port)
        clientSocket.connect((serverName, serverPort))

        # Read input from user
        message = input('Input lowercase sentence: ')

        # Send message to TCP server
        # No need to attach server name or port
        clientSocket.send(bytes(message, 'utf-8'))

        # Get response from server and store it
        modifiedMessage = clientSocket.recv(2048)

        # Print response from above
        print('From server: ' + modifiedMessage.decode('utf-8'))

        # TCP client socket is closed
        clientSocket.close()
```

- 'AF_INET' is an internet protocol
  - It corresponds to the IPV4 address family
- The port number of the TCP client is automatically allocated by the operating system
  - This is the case for UDP as well

- Example App: TCP Server
  - i.e. TCPServer.py

```
        #Include Python's socket library
        from socket import *

        # This is the port that the server will bind to
        serverPort = 12000

        # Create TCP welcoming socket
        serverSocket = socket(AF_INET, SOCK_STREAM)

        # Socket is bind to the specified port
        serverSocket.bind(("", serverPort))
        # "" means any network on the server

        # Server begins listening for incoming TCP requests on the
        #   binded port
        serverSocket.listen(1)
```

```
    print('The server is ready to receive')
    # The socket is now in listen mode and can receive
    #   connections from clients

    # Loop forever
    while 1:
        # Server waits on accept() for incoming requests
        connectionSocket, addr = serverSocket.accept()
        # New socket is created on return
        # The IP address & port number of client is returned

        # Read bytes (the message) from socket
        # Note: It does not read the address, like it does in
        #   UDP, and the 'UDPServer.py' example
        sentence = connectionSocket.recv(1024).decode('utf-8')

        # Convert message to uppercase
        capitalizedSentence = sentence.upper()

        # Sends the modified message to the client
        connectionSocket.send(bytes(capitalizedSentence,
            'utf-8'))

        # Close connection to this socket, but NOT the
        #   welcoming socket, which is the first socket
        #   that was created (in the beginning)
        connectionSocket.close()


    # This is the welcoming socket and needs to be closed
    #   after it is no longer needed
    # serverSocket.close()
```

- A server may have multiple interface cards, and thus multiple IPs
- 'SOCK_STREAM' corresponds to a TCP socket
- The 'accept()' method is a blocked call
  - It will not return until a connection arrives from the client side
  - Once a connection arrives, it returns the IP address and port number of the client
- 'connectionSocket' is different from 'serverSocket'
  - The 'serverSocket' sets up the connection between the client and the server
    - It only specifies the IP address and port number of the host
  - The 'connectionSocket' is used to exchange data
    - It contains the IP address and port number of both client and server
- The 'serverSocket' needs to be closed after the 'connectionSocket'

        - But if it is not closed, then the 'serverSocket' can be reused to create a new 'connectionSocket'
- Comparison of UDP & TCP Sockets

| | TCP | | UDP | |
|---|---|---|---|---|
| | Client | Server | Client | Server |
| Socket | SOCKET_STREAM | SOCKET_STREAM | SOCKET_DGRAM | SOCKET_DGRAM |
| Bind to a fixed port | | X | | X |
| Connection setup | X | X | | |
| Send | send() | send() | sendto(data, (IP, port)) | sendto(data, (IP, port)) |
| Recv | recv() | recv() | recvfrom() | recvfrom() |

        - TCP sockets use 'SOCKET_STREAM'
        - UDP sockets use 'SOCKET_DGRAM'
        - Both UDP and TCP servers need to bind to a fixed port
            - But the clients do not have to
                - The client's port is allocated by the operating system
        - TCP sockets need to go through a connection setup phase
            - i.e. 'accept()' from the TCP server
        - UDP does not establish a connection
        - In TCP, both client and server use the 'send()' method to deliver content
            - The destination IP address or port are not specified
        - In UDP, both client and server use to 'sendto()' method
            - The destination IP address and port number need to be specified
        - TCP sockets retrieve messages using 'recv()'
        - UDP sockets retrieve messages using 'recvfrom()'
            - This returns not only the message but also the source IP address and port number of the message
                - This is used to send subsequent messages to the source
- Port Number
    - Are used by client and server programs
    - Are application layer addresses to separate different processes
    - Is a field in the transport layer header

        - It is 16 bit
            - Range = 0 - 65535
    - Port numbers in the range from 0 to 1023 are well known ports or system ports
        - i.e.
            - SSH  = 22
            - SMTP = 25
            - HTTP = 80
    - Port numbers from 1023 to 49151 can be used without super-user (root) privileges, sometimes by a registered service
        - i.e. BitTorrent uses ports from 6888 to 6900
    - Ports greater than 49151 are private ports
        - This range differs from each operating system
            - i.e. Linux uses the port range 32768 to 60999
        - These ports are allocated by the operating system to the client program
            - These are short lived; they are released once the socket is closed, and the resources are removed
                - Other applications can now use the free'd up ports
- Chapter 2 Summary
    - Application architectures
        - Client/server
        - P2P
        - Hybrid
    - Application service requirements
        - Reliability, bandwidth, delay
            - Protocol is chosen by application developer
    - TCP, UDP, SSL
        - TCP is connected oriented
        - UDP is connection-less
    - HTTP
        - Non-persistent VS. Persistent
        - Messages
        - Method types
        - Cookies
    - Web caching
    - DNS
        - Services
        - 4 records
        - Hierarchical
        - Iterative, recursive query
        - Dig
        - Attacking DNS
            - DDOS attacks
            - Redirect attacks
    - Socket programming
        - TDP VS. UDP sockets to implement a client-server program