Principles of Network Applications Overview

Web and HTTP -- implementing web server
Email, SMTP, IMAP -- implementing web server
The Domain Name System (DNS)
P2P Applications -- final project
Video Streaming and Content Distribution
Socket Programming -- already covered

Application Layer and HTTP

IMAP is a type of email protocol.

The final project will be P2P: Torrent client that is modified such that we can connect to others

Video Streaming can have strictly information based sharing but can also function with delta. Essentially the change in the frames is sent instead of the entire image.

Network Applications include:

- Zoom
- Social Media
- Web servers
- Texting
- Streaming video
- P2P
- Skype → different protocol all on its own
- Real Time (RT) video
- Internet search
    - Http requests
        - These are recursive
- Remote login
- And more…

Networking applications are implemented on end devices and are connected by autonomous systems. The lower layers are abstracted away at this layer with something called “overlay networks”. These are built on top of the already existing networks. These are located in specific regions, and will use TCP or UDP to implement them depending on the scenario. In terms of sockets, it is sticking a bunch of sockets together to create the web.

The code written for these only needs to be written in the core. This will allow for most of the concerns to be “deleted”, allowing the developer to focus on the application layer.

Client-Server Model

An example of the client server model would be having a phone communicate to a website. This website would have a dedicated machine for the domain to ensure access.

Properties:

- Always on
- Has a permanent IP address

While the server will have a permanent IP address the client will have dynamic one (it can change).

Examples:

- HTTP
- IMAP
- FTP

P2P Architecture

In this model there is no server, only arbitrary communication between two machines. Something like a torrent network client would fall under this category. In this scenario, neither of the machines involved are always on, they can enter and leave as necessary. This makes the architecture more challenging and complex as there is no permanent IP involved on either side. The two machines will only request and provide information in return. In this sense they will need to upload something when a download occurs. The availability of space is also dynamic, as the number involved grows the capacity will as well. It will change to meet the demands.

Process Communication

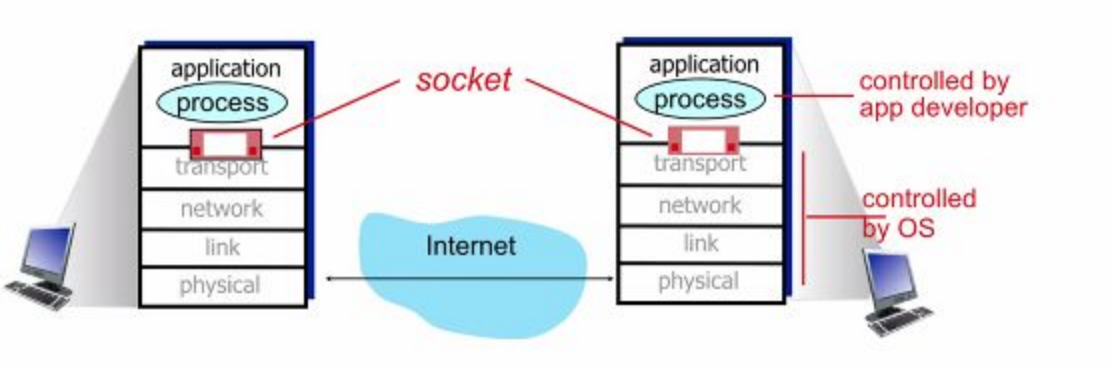
Process → program running within a host

- Within the same host, two processes communicate using inter-process communication
- Processes within different hosts communicate using messages (socket)

In client-server model, the client will initiate the communication and the server waits to be communicated with.

In the P2P model, both applications support client and server processes

All the lower levels (transport, data link) are dealt with by the OS using sockets. In this case the sockets act like a "door" to the other host. The OS will send the message through the door without worrying about all of the details.



Socket Addressing

Host IP (normally IPv4, but expanded with IPv6)
Port # → assigned to a process essentially "binding" a process to a port

Examples:

- HTTP → port 80
- Mail → port 25
- HTTPS → port 443

Protocol at the Application Layer

Define how the processes communicate, need four things

- Type: i.e HTTP is a Request and Response type
- Message Syntax: the layout of the packet
- Format of the data: ascii, binary, etc.?
- Rules for the request: when and how will messages be sent?

Many protocols are open protocols, so that everyone knows how to communicate. However, not all protocols are open, for example, Skype has a proprietary protocol.

Most of these protocols feature a handshake and some form of state to implement reliability (for example TCP). Some protocols need 100% reliability, but not all do. Audio for example can be lossy, (and as such implement UDP) while file download and transfer cannot be lossy.

Another aspect is time sensitivity. These applications are normally going to be more ok with loss so that they don't have to implement as much overhead and they will run or respond faster.

A lot of the time these applications need some sort of minimum throughput in order to be effective. Things like audio and video need to be transmitted at a specific rate with no delay.

Transport Layer Security (TLS):

Applications use this to provide data integrity, end-point authentication, and provides encrypted TCP connections

Data Integrity:

- TCP -- reliable (web transactions)
- UDP -- not reliable (audio -- loss tolerant)

Timing;

- Some apps require low delay to be "Effective"

Throughput:

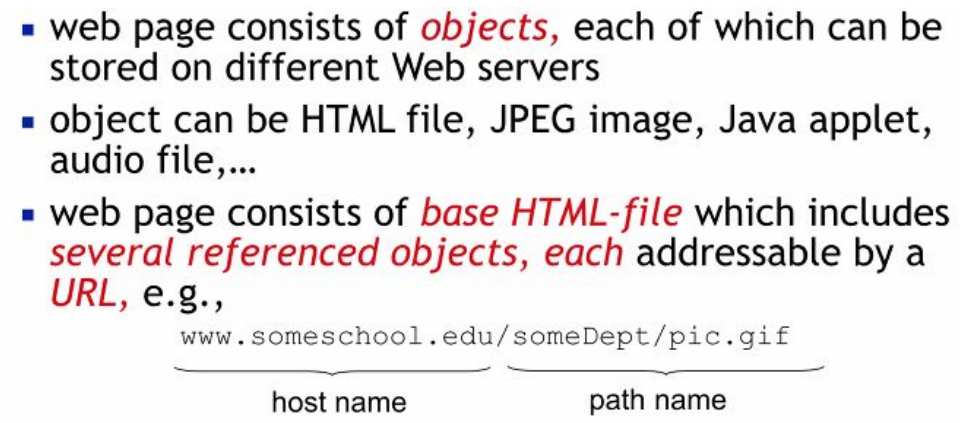
- Some apps require minimum amount of throughput to be "Effective"
- Other apps are elastic -- make use of whatever throughput they get (at beginning they can be lossy, but later they can be recovered and fixed)

Security:

- encryptions, data integrity, etc.

HTTP

→ Hyper-Text Transport Protocol



HTTP is a type of Client/Server architecture that will run on Port 80. It is a stateless protocol that will initialize a TCP connection, exchange the information from the server to the client, and then

close the TCP channel. Since it is stateless it maintains no information for past requests. Why is this a good thing? Why is this desirable? Maintaining state is complex. There is a way to have the protocol maintain state but that could lead to errors. On failure, there is a chance that the state could be inconsistent or flawed upon restarting. Without state there is no need to implement the solution to such a problem eliminating a lot of overhead. The way to "maintain state" with HTTP is to just keep the channel open longer. This is known as "Persistent HTTP".

Non-Persistent HTTP vs. Persistent HTTP

Non-Persistent HTTP	Persistent HTTP
One at a time One object for each connection Not very efficient More overhead Many in parallel	"Conga line" (one after another) Multiple objects for each connection Leave the connection open No extra RTT overhead Time in half

Note: RTT: time for a small packet to travel from client to server and back
Note: TCP connections are closed for every object that is opened, so there is a lot of overhead

So what if they all come in on port 80?

- They can all come in on the same port but can leave through any ort

The request and response for HTTP are all in ASCII, meaning that a dev can read them.

There are a few different methods for HTTP (GET, POST, HEAD, PUT)
GET will place the information in the URL, while POST will put it in the data section of the request, essentially hiding it.
HEAD will retrieve the header only, this can be used to make sure that a site is not vulnerable.
PUT will upload information to the destination.

Server codes are used to communicate response and state. Normally a code of 200 means that all is working as it is meant to, 404 indicates the page was not found, 500 for internal server error, and more… These codes are useful for debugging.

Other parts of the HTTP request are Date Last Modified (used for caching), tag (also for caching), content length, and others. Content length is an important field. (WHY???)

Cookies:

- Cookies can be used in the request and response and are a solution to the lack of state for HTTP. Essentially the state is stored in a cookie on the client's computer and on the server and that information is included in the request and the server will use the information in the cookie to fill out the response.
- There are privacy issues that arise with cookies.
    - End up with 3rd party tracking

*Does SSO work between processes? (*e.g. sign in using Chrome, then open Firefox and be automatically signed in.)

No because chrome and Firefox don't have the same plugins