The Application Layer

Chapter 2 from the Kurose and Ross textbook
PART 2 – Sections 2.4 to 2.7
The Domain Name System
The 411 of the Internet

Host names

- Internet hosts may be identified by a unique hostname.

- Similar to telephone network, the hostname consists of a name (resource name) plus an address (domain name).

- Hostnames are good for people, but the Internet uses 32 bit IP addresses for sending datagrams.

Internet resource (Web server name)

www.wellesley.edu

Domain name

149.130.12.213

IP address in dotted decimal format
IP addresses*

Consists of four bytes expressed in decimal notation from 0 to 255

149.130.12.213

Hierarchical structure: scan from left to right yields more specific information about host's location

*You guessed it, more later when we get to the network layer.

Domain Name System (DNS)

• The Domain Name System is a distributed database implemented in a hierarchy of name servers.

• It is also an application-layer protocol that provides 411 service between hosts and name servers.

• All our old friends use DNS including HTTP, SMTP, and FTP.
HTTP uses Internet 411


2. DNS server does its thing and IP address is sent to DNS client.

3. DNS client receives reply and passes IP address to HTTP client so that browser can open a TCP connection to HTTP server.

4. HTTP server receives request and the rest is history.

Both HTTP and DNS clients live on the same host

HTTP and DNS servers live on different hosts
HTTP uses Internet 411


2. DNS server does its thing and IP address is sent to DNS client.

3. DNS client receives reply and passes IP address to HTTP client so that browser can open a TCP connection to HTTP server.

4. HTTP server receives request and the rest is history.

A number of hosts may get involved here, introducing additional (sometimes substantial) delay.

Wouldn’t a centralized design be simpler?

- Well, it would be a single point of failure.

- Traffic to and from a single server would be worse than Boston at 8:00 am on a Monday morning.

- A single name server would not be close to everyone. Someone in either New York City or Sidney, Australia would not be happy.

- Maintenance of a centralized database would be a nightmare.
Distributed, hierarchical database

Each ISP has a local name server where the client DNS request is sent first.

List is generated dynamically at startup and stored in /etc/resolv.conf

The cheese stands alone

From inside our domain; the outward facing DNS servers have be outsourced.
Root name servers

- If Cheers doesn’t know the IP address, she behaves as a DNS client and queries one of the root name servers.

- Each server is actually a cluster of replicated servers, for both security and reliability purposes.

Passing the buck

- Top-level domain servers are responsible for top-level domains (com, org, net, edu, and gov).

- The root server will probably refer us to one of these.*

*You see where this is going.
Authoritative name servers

- All within a domain hosts are registered with an authoritative name server* which is where we will probably be referred to next.

- The authoritative name server for a host is generally in the host's local ISP.

---

cis.poly.edu looks up gaia.cs.umass.edu

After the dance, the IP address would be cached locally for future reference.
But what if …

• … the TLD DNS server doesn’t know the authoritative DNS server for gaia.cs.umans.edu.

• Perhaps U Mass has a name server for the entire university, called dns.umass.edu, …

• … and each department has its own name server that is authoritative for all hosts in the department.

Busy servers

• **Recursive query**: Server A makes a request on behalf of server B, then forwards the address.

• This puts the burden of name resolution on the contacted server.

• Root servers have places to go, people to meet.
Iterated queries

- Iterated query:
  - Server A queries server B.
  - If Server B doesn’t know, she responds with the name of the next server in the chain.

- In effect, server B says, “I don’t know this name. Go ask so-and-so.”

DNS caching

- In a query chain, when a DNS server receives a DNS reply it caches the mapping in its local memory.

- When a second query arrives for the same hostname, it can provide the IP address, even if it is not authoritative for the hostname.

- DNS servers typically discard cached information after a couple of days.
DNS resource records

- **TTL** is the time to live.
- **Name** and **Value** depend on type.
- **Type=A**
  - **Name** is hostname and **Value** is IP address
- **Type=NS**
  - **Name** is domain and **Value** is the hostname of an authoritative server that knows how to obtain IP addresses for hosts in the domain
- **Type=CNAME**
  - **Name** is alias for some canonical name and **Value** is the canonical name.
- **Type=MX**
  - **Value** is the name of mailserver associated with **Name**.

DNS protocol, messages

**query** and **reply** messages, both with same message format

- **msg header**
  - **identification**: 16 bit # for query, reply to query uses same #
  - **flags**:
    - query or reply
    - recursion desired
    - recursion available
    - reply is authoritative

<table>
<thead>
<tr>
<th>Identification</th>
<th>Flags</th>
</tr>
</thead>
<tbody>
<tr>
<td># questions</td>
<td># answer RRs</td>
</tr>
<tr>
<td># authority RRs</td>
<td># additional RRs</td>
</tr>
<tr>
<td>questions (variable # of questions)</td>
<td></td>
</tr>
<tr>
<td>answers (variable # of RRs)</td>
<td></td>
</tr>
<tr>
<td>authority (variable # of RRs)</td>
<td></td>
</tr>
<tr>
<td>additional info (variable # of RRs)</td>
<td></td>
</tr>
</tbody>
</table>
To create a domain, we first
- **register the domain name**, say mynewdomain.edu
  - with a commercial entity known as a registrar
  - including the IP addresses of our primary and second authoritative servers.

The registrar would then enter a type NS and type A records into the TLD servers for each name server.

For example, suppose your primary name servers was named dns1.mynewdomain.edu:
- (mynewdomain.edu, dns1.mynewdomain.edu, NS)
- (dns1.mynewdomain.edy, 212.212.212.1, A)
**Scary times**

- To say that DNS is an important component of the Internet infrastructure is putting it mildly.
- Without it many important services (Web, email, ...) would cease to function.
- How vulnerable is DNS to attack? And who would want to attack it?

**DDoS bandwidth-flooding attack**

- A large-scale DDoS attack against DNS root servers was launched on October 21, 2002.
- The attackers leveraged a botnet to send truck loads of ICMP ping messages to each of the 13 DNS root servers.
- Fortunately, the root servers were protected by packet filters, configured to block all ICMP ping messages.
- A more effective DDoS attack would send a deluge of DNS queries to top-level-domain servers.

* Distributed Denial of Service (DDoS) attacks
Man-in-the-middle attack

- In a man-in-the-middle attack, the attacker intercepts queries from hosts and returns bogus replies.

- In the DNS poisoning attack, the attacker sends bogus replies to a DNS server, tricking the server into accepting bogus records into its cache.

- Either attack could be used to redirect an unsuspecting Web user to the attacker's website.

Peer-to-peer applications

There are more than you think!
Pure P2P architecture

- No always-on server
- Arbitrary end systems directly communicate
- Peers are intermittently connected and change IP addresses
- Examples:
  - file distribution (BitTorrent)
  - Streaming (KanKan)
  - VoIP (Skype)

File distribution: client-server vs P2P

- Question: how much time to distribute file (size F) from one server to N peers?
  - peer upload/download capacity is limited resource

\[ u_s: \text{server upload capacity} \]
\[ d_i, d_j: \text{peer i download capacity} \]
\[ u_i, u_j: \text{peer i upload capacity} \]
File distribution time: client-server

- Server transmission: must sequentially send (upload) N file copies:
  - time to send one copy: $F/\text{us}$
  - time to send N copies: $NF/\text{us}$

- client: each client must download file copy
  - $d_{\min} = \text{min client download rate}$
  - min client download time: $F/d_{\min}$

\[
D_{C\text{-}S} \geq \max\{NF/\text{us}, F/d_{\min}\}
\]

increases linearly in N

File distribution time: P2P

- Server transmission: must upload at least one copy
  - time to send one copy: $F/\text{us}$

- client: each client must download file copy
  - min client download time: $F/d_{\min}$

- clients: as aggregate must download $NF$ bits
  - max upload rate (limiting max download rate) is $u_s + \sum u_i$

\[
D_{P2P} \geq \max\{F/\text{us}, F/d_{\min}, NF/(u_s + \sum u_i)\}
\]

increases linearly in N …
… but so does this, as each peer brings service capacity
Client-server vs. P2P: example

Client upload rate = \( u \), \( F/u = 1 \) hour, \( u_s = 10u \), \( d_{\text{min}} \geq u_s \)

P2P file distribution: BitTorrent

- file divided into 256Kb chunks
- peers in torrent send/receive file chunks

**tracker** tracks peers participating in torrent

**torrent** group of peers exchanging chunks of a file

Alice arrives …
… obtains list of peers from tracker
… and begins exchanging file chunks with peers in torrent
P2P file distribution: BitTorrent

- **Peer joining torrent:**
  - has no chunks, but will accumulate them over time from other peers
  - registers with tracker to get list of peers, connects to subset of peers ("neighbors")

- While downloading, peer uploads chunks to other peers
- Peer may change peers with whom it exchanges chunks
- **Churn:** peers may come and go
- Once peer has entire file, it may (selfishly) leave or (altruistically) remain in torrent

BitTorrent: requesting, sending file chunks

**requesting chunks:**
- At any given time, different peers have different subsets of file chunks
- Periodically, Alice asks each peer for list of chunks that they have
- Alice requests missing chunks from peers, rarest first

**sending chunks: tit-for-tat**
- Alice sends chunks to those four peers currently sending her chunks at highest rate
  - other peers are choked by Alice (do not receive chunks from her)
  - re-evaluate top 4 every 10 secs
- Every 30 secs: randomly select another peer, starts sending chunks
  - "optimistically unchoke" this peer
  - newly chosen peer may join top 4
BitTorrent: tit-for-tat

1. Alice “optimistically unchokes” Bob
2. Alice becomes one of Bob’s top-four providers; Bob reciprocates
3. Bob becomes one of Alice’s top-four providers

higher upload rate: find better trading partners, get file faster!

Video Streaming
A view into Content Distribution Networks
Internet video

- Videos are sequences of images, displayed at a constant rate, e.g., 24 or 30 images per second.

- Uncompressed images consist of arrays of pixels, each pixel encoded as a number representing the pixel’s luminance and color.

- Algorithms can compress video to most any bit rate. Of course, higher bit rate means better image quality.

HTTP streaming video

- The video is stored on a server as an ordinary file, with a specific URL.

- The client establishes an TCP connection and issues a GET request for that URL.

- File is sent, and collected in a client buffer. Once this buffer exceeds a predetermined threshold, the video begins to play.
Dash it all

- HTTP streaming is extensively deployed, but has a major shortcoming.
- All clients receive the same encoding despite their bandwidth.
- Dynamic Adaptive Stream over HTTP (DASH) encodes multiple versions.
- The client dynamically requests chunks of a few seconds: high quality when bandwidth is high, and lower quality when bandwidth is scarce.

Keeping score

- Each video version is stored in the HTTP server, with a different URL.
- The client first requests a manifest file, which provides a URL for each version along with its bit rate, and requests a chunk.
- While downloading chunks the client measures bandwidth to determine which version to select next.
Distributing on-demand

- YouTube distributes hundreds of millions of video streams to users around the world.

- They could build a single massive data center to store and stream all videos.

- But this approach is not without fault.

Content distribution networks (CDNs)

- A content provider (YouTube) distributes its content through a CDN company (Akamai).

- The CDN company purchases large numbers of servers from vendor (IBM), and ...

- ... installs the servers in hosting centers (Exodus).
Two placement philosophies

- The philosophy pioneered by Akamai is to enter deep into the access network of the ISPs by deploying server clusters in access ISPs all over the world.
- The goal is to get close to the end users.
- However, maintaining and managing the clusters is a bit of a challenge.

Bring home

- A second design philosophy (Limelight and others) is to bring the ISPs home by building large clusters at a smaller number (tens) of locations and ...

- ... connecting these using a private high-speed network.
- CNDs typically locate simultaneously near Internet Exchange Points.
Okay, so how . . .

- . . . when a browser requests an object, does the CDN intercept the request and
- how does is determine a suitable CDN servers cluster and redirect the request to a server in that cluster?

CDN content access scenario

2. Resolve http://netcinema.com/6Y7B23V via Bob’s local DNS.
3. netcinema’s authoritative DNS returns URL http://KingCDN.com/NetC6y&B23.
4. Resolve http://KingCDN.com/NetC6y&B23 via KingCDN’s authoritative DNS, which returns IP address of KingCDN server with video.
5. request video from KINGCDN server, streamed via HTTP.
Cluster selection strategies

• How does a CND determine the “best” server cluster to dynamically direct a client?

• CND’s typically employ proprietary cluster selection strategies to answer this question.

• There are a number of approaches.

Geographically closest

• This one is easy to understand.

• Use commercial geo-location databases (e.g., Quova, Max-Mind) to map each LDNS IP address to the closest geographic location.

• Works well for a large fraction of clients ...
Current traffic conditions

- CNDs can perform periodic real-time measurements of delay and loss performance between their clusters and clients.
- Clients are routed according to current traffic patterns.

Case study: Netflix

- Netflix generated 37% of the downstream US Internet traffic in 2015.
- There are two major components to its video distribution: the Amazon cloud and its own private CDN infrastructure.
Before it can distribute a movie

- Content ingestion. Netflix obtains a master version of the film and uploads it to host in the Amazon cloud.

- Content procession. The cloud creates many different formats for the movie at multiple bit rates.
- Uploading versions to its own CDN.

Netflix video streaming platform

1. Bob manages Netflix account
2. Bob browses Netflix video
3. Manifest file returned for requested video
4. DASH* streaming

*Dynamic Adaptive Streaming over HTTP
Socket Programming

Socket programming

- Goal:
  - learn how to build client/server applications that communicate using sockets
- Socket:
  - door between application process and end-end-transport protocol
Socket programming

- Two socket types for two transport services:
  - UDP: unreliable datagram
  - TCP: reliable, byte stream-oriented

**Application Example:**
1. Client reads a line of characters (data) from its keyboard and sends the 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

- UDP: no “connection” between client & server
  - no handshaking before sending data
  - sender explicitly attaches IP destination address and port # to each packet
  - rcvr extracts sender IP address and port# from received packet

- UDP:
  - transmitted data may be lost or received out-of-order

- Application viewpoint:
  - UDP provides unreliable transfer of groups of bytes (“datagrams”) between client and server
Client/server socket interaction: UDP

### server (running on serverIP)
- create socket, port= x:
  ```
  serverSocket = socket(AF_INET,SOCK_DGRAM)
  ```
- read datagram from serverSocket
- write reply to serverSocket specifying client address, port number

### client
- create socket:
  ```
  clientSocket = socket(AF_INET,SOCK_DGRAM)
  ```
- Create datagram with server IP and port=x; send datagram via clientSocket
- read datagram from serverSocket
- close clientSocket

Example app: UDP client

**Python UDPClient**

```python
from socket import *

serverName = 'hostname'
serverPort = 12000

clientSocket = socket(socket.AF_INET, socket.SOCK_DGRAM)
message = raw_input('Input lowercase sentence: ')  
clientSocket.sendto(message,(serverName, serverPort))

modifiedMessage, serverAddress = clientSocket.recvfrom(2048)
print modifiedMessage
clientSocket.close()
```
Example app: UDP server

**Python UDPServer**

```python
from socket import *
serverPort = 12000
serverSocket = socket(AF_INET, SOCK_DGRAM)
serverSocket.bind(('', serverPort))
print "The server is ready to receive"
while 1:
    message, clientAddress = serverSocket.recvfrom(2048)
    modifiedMessage = message.upper()
    serverSocket.sendto(modifiedMessage, clientAddress)
```

- **create UDP socket**
- **bind socket to local port number 12000**
- **loop forever**
- **Read from UDP socket into message, getting client’s address (client IP and port)**
- **send upper case string back to this client**

Socket programming with TCP

- **Client must contact server**
  - server process must first be running
  - server must have created socket (door) that welcomes client’s contact
- **Client contacts server by:**
  - Creating TCP socket, specifying IP address, port number of server process
- **When client creates socket:**
  - client TCP establishes connection to server TCP
- **When contacted by client, server TCP creates new socket for server process to communicate with that particular client**
  - allows server to talk with multiple clients
  - source port numbers used to distinguish clients (more in Chap 3)
- **Application viewpoint:**
  - TCP provides reliable, in-order byte-stream transfer (“pipe”) between client and server
Client/server socket interaction:

TCP

server (running on hostid)

create socket, port=x, for incoming request:
serverSocket = socket()

wait for incoming connection request
connectionSocket = serverSocket.accept()

write reply to connectionSocket
read request from connectionSocket
close connectionSocket

create socket, connect to hostid, port=x
clientSocket = socket()

send request using clientSocket
read reply from clientSocket
close clientSocket

Example app: TCP client

Python TCPClient

from socket import *
serverName = 'servername'
serverPort = 12000
clientSocket = socket(AF_INET, SOCK_STREAM)
clientSocket.connect((serverName, serverPort))
sentence = raw_input('Input lower case sentence: ') clientSocket.send(sentence)
modifiedSentence = clientSocket.recv(1024)
print 'From Server:', modifiedSentence
clientSocket.close()
```python
from socket import *

serverPort = 12000

serverSocket = socket(AF_INET,SOCK_STREAM)
serverSocket.bind(('',serverPort))
serverSocket.listen(1)

print('The server is ready to receive')

while 1:
    connectionSocket, addr = serverSocket.accept()
    sentence = connectionSocket.recv(1024)
    capitalizedSentence = sentence.upper()
    connectionSocket.send(capitalizedSentence)

connectionSocket.close()
```

*Python TCP Server*

- create TCP welcoming socket
- server begins listening for incoming TCP requests
- loop forever
- for incoming requests, new socket created on return
- read bytes from socket (but not address as in UDP)
- close connection to this client (but not welcoming socket)