Computer Systems and Networks

ECPE 170 – Jeff Shafer – University of the Pacific

Networking: UDP & DNS

User Datagram Protocol (UDP)

UDP versus TCP

	TCP	UDP
Reliable?	**Yes** *(Via acknowledgements and retransmitting)*	**No**
Connection-oriented?	**Yes** *(Server has one socket per client)*	**No** *(Server has one socket and all messages from all clients are received on it)*
Programming model?	**Stream** *(continuous flow of data – may get a little bit at a time)*	**Datagram** *(data is sent in its entirety or not at all. Size of each datagram is small)*
Applications	**HTTP (Lab 8)** *Web, email, file transfer*	**DNS (Lab 9)** *Streaming Audio/Video, Gaming*

User Datagram Protocol (UDP)

- UDP: no "connection" between client and server
  - No handshaking
  - Sender explicitly attaches IP address and port of destination to each message
  - Receiver can extract IP address, port of sender from received datagram

application viewpoint

UDP provides unreliable transfer of groups of bytes ("datagrams") between client and server

User Datagram Protocol (UDP)

- Each UDP message is self-contained and complete
- Each time you read from a UDP socket, you get a complete message as sent by the sender
  - *That is, assuming it wasn't lost in transit!*
- Think of UDP sockets as putting a stamp on a letter and sticking it in the mail
  - *No need to establish a connection first*
  - *Receiver has no idea "letter" is arriving until they look in the mailbox*

Python UDP Programming

- Two new functions: `sendto()` and `recvfrom()`

```
server_ip = 1.2.3.4
port = 5678
dest_addr = (server_ip, port)
s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
...
...
bytes_sent = s.sendto(raw_bytes, dest_addr)
...
...
max_bytes = 4096
(raw_bytes, src_addr) = s.recvfrom(max_bytes)
```

Domain Name System (DNS)

IP Addresses

- IP version 4 addresses are 32 bits long
- IP version 6 address are 128 bits long
- Every network interface has at least one IP address
  - A computer might have 2 or more IP addresses
- IPv4 addresses are usually displayed in dotted decimal notation
  - Each byte represented by decimal value
  - Bytes are separated by a period
  - IP address `0x8002C2F2 = 128.2.194.242`

Motivation

- IP addresses are hard to remember
  - `198.16.253.143`? Or was it `.146`?
- Human-friendly names are much better
  - `engineering.pacific.edu`
- How can we translate between the two?

Early Days (prior to 1983)

- Each computer on the ARPAnet (early Internet) had a single file
  - `hosts.txt` maps all known host names to IP address
- Master list maintained by SRI Network Information Center
  - Email them if your mapping changes
  - New list produced 1-2 times a week
  - All hosts download the new list
- **Problems with this approach?**

Domain Name System (DNS)

- **Distributed database** implemented in hierarchy of many **name servers**
- **Application-layer protocol**
  - Hosts, routers, and name servers communicate to resolve names (address/name translation)
  - Core Internet function implemented as application-layer protocol

DNS is Decentralized

- No single point of failure
- No distant centralized database
- Easier maintenance
  - Take one or a dozen servers offline without issue
- Support high traffic volume
- *** **Scalability** ***

How many DNS requests/second globally?

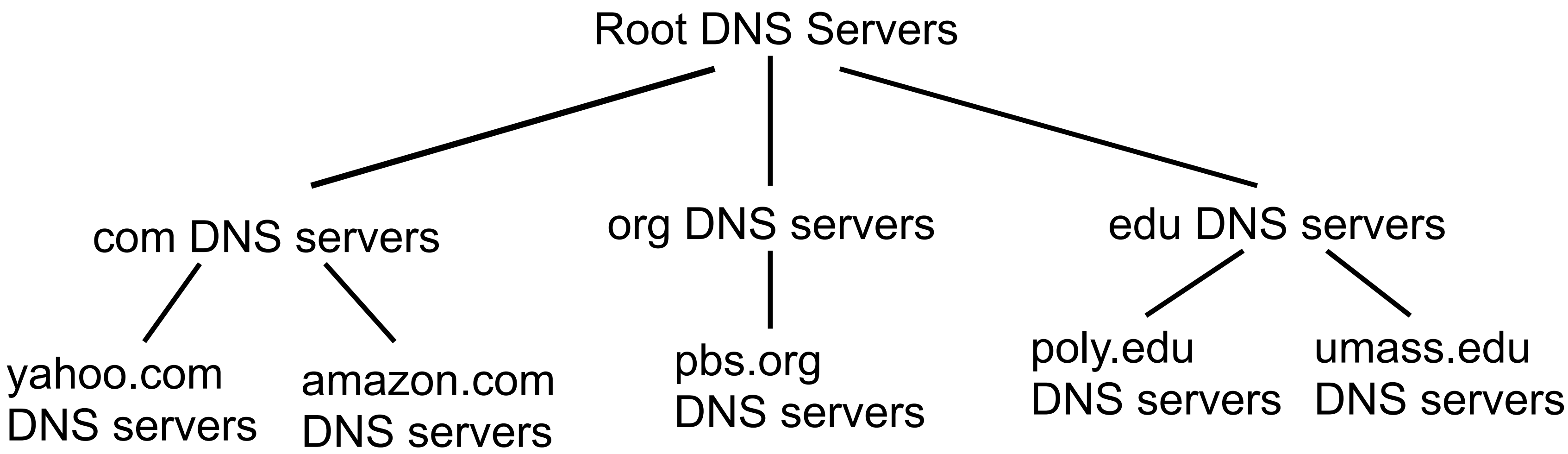
DNS: Scalability

- **Challenging to find data on <u>global</u> DNS requests/sec**
  - No global internet "dashboard"
  - Internet is a "network of networks"
- Would have to inquire with AT&T, Comcast, TimeWarner, Pacific, etc
  - They would have to check stats on all of their local servers
- **Google Public DNS**
  - 400 billion requests/day as of Dec 2014
  - 70% international
  - http://googlewebmastercentral.blogspot.com/2014/12/google-public-dns-and-location.html
- **OpenDNS**
  - 80 billion requests/day as of Sept 2015
  - http://system.opendns.com/

What's in a Name?

- `engineering.pacific.edu`
  - .edu is top-level domain
  - "pacific" belongs to .edu
  - "engineering" belongs to "pacific"
  - Hierarchical! Read from right to left

Distributed, Hierarchical Database



- Client wants IP for www.amazon.com
  1. Client queries a root server to find com DNS server
  2. Client queries com DNS server to get amazon.com DNS server
  3. Client queries amazon.com DNS server to get IP address for www.amazon.com

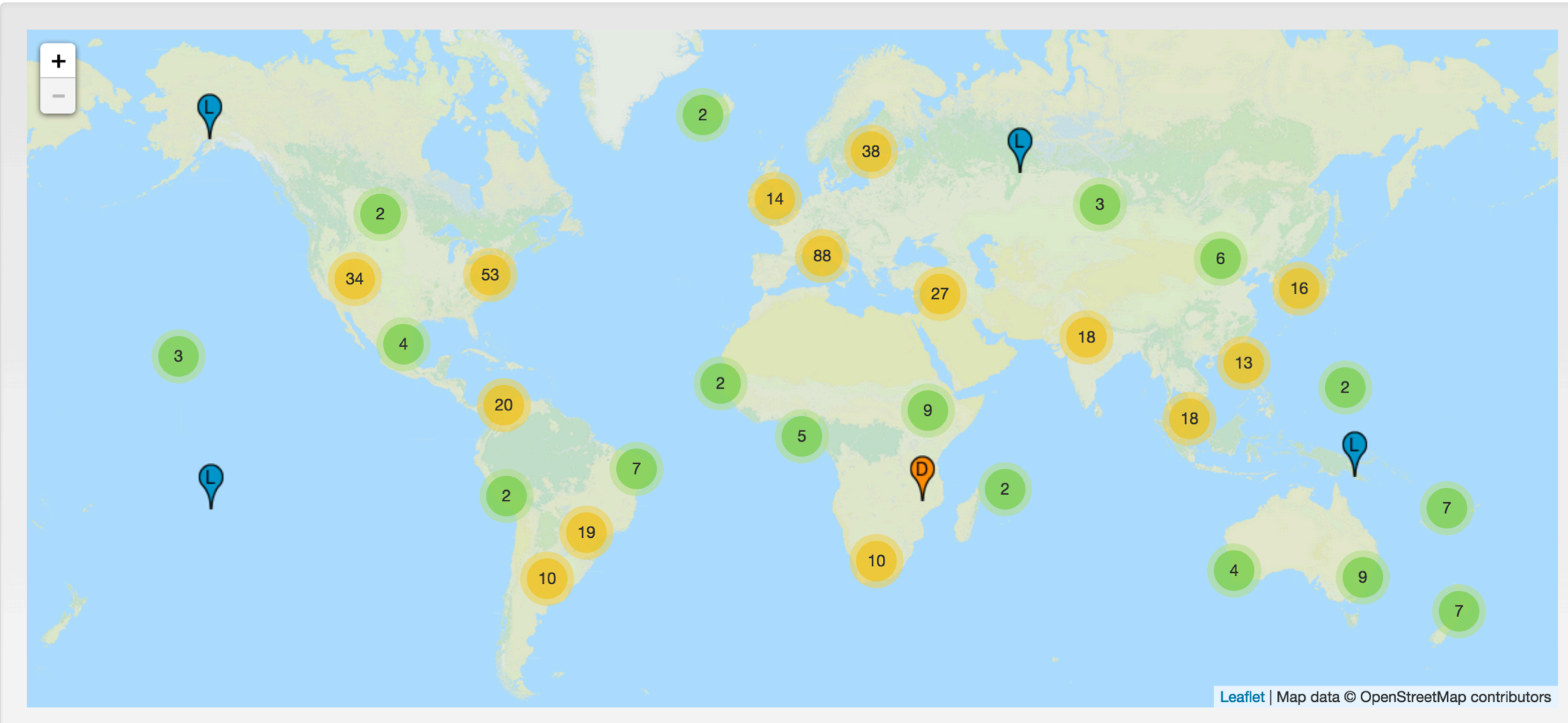
DNS: Root Name Servers

- Contacted by local name server that can not resolve top-level domain
- Root name server:
  - Contacts authoritative name server for TLD if name mapping not known
  - Gets mapping
  - Returns mapping to local name server

13 root name "servers" worldwide labeled `a` - `m`

- Each "server" is really a cluster
- Some clusters are geographically distributed
- 504 total in Fall 2014

DNS: Root Name Servers



http://www.root-servers.org/

DNS and UDP

- DNS uses UDP by default
  - It *can* use TCP, but it's rare
  - **Isn't this unreliable?**
- Why use UDP
  - Reliability not needed
    - DNS will just re-request if no response received (2-5 seconds)
  - Faster (in three ways!)
    - No need to establish a connection (RTT/latency overhead)
    - Lower per-packet byte overhead in UDP header
    - Less packet processing by hosts

Demonstrations

Demonstrations

1. DNS Client: `dns.py`
2. Wireshark packet capture

Programming Tips

The `struct` Module

- The details of variables are hidden in Python
  - For example, how many bytes is an integer?
- Need a method to deal with binary data for file I/O or network I/O: the `struct` module
  - Module performs conversions between basic Python datatypes and arrays of bytes

The `struct` Module

- Two main functions in the `struct` module
  - `pack`: convert a group of variables into an array of bytes
  - `unpack`: convert an array of bytes into a group of variables
- Similar to C's `printf` and `scanf`
- Each function requires a *format string* to describe how to pack or unpack the arguments

The `struct` Module

- Common format string options:
  - See https://docs.python.org/3/library/struct.html

Format	Python Type	Size (bytes)
B	Integer	1
H	Integer	2
L	Integer	4
Q	Integer	8

- `raw_bytes = struct.pack("BH", val1, val2)`
- `(val1, val2) = struct.unpack("BH", raw_bytes)`

The `struct` Module

- Endianness must be considered when doing file or network I/O with fields greater than one byte
- The first character of the format string determines the endianness

Character	Byte order	Size	Alignment
@	Native	Native	Native
=	Native	Standard	None
<	Little	Standard	None
>	Big	Standard	None
!	Network (Big)	standard	None

DNS Endianness

- **What endianness is your computer?**
  - Little endian (x86)
- **What endianness is the DNS protocol? (or most network protocols)**
  - Big endian
- **What fields in the DNS header does this matter for?**
  - Two-byte integer fields
    (question count, answer count, etc…)

Bit Fields

- **Warning!** `struct` only deals with bytes. It cannot handle fields with dimensions less than one byte
- Problem – Some of the DNS fields are only 1 bit, 3 bits, or 4 bits in size

2 bytes (16 bits)

```
QR | OPCODE | AA | TC | RD | RA | Resvd | RCODE
(1)    (4)    (1)  (1)  (1)  (1)   (3)     (4)
```

- **How can we handle this?**
  - Manual bit shifting (ala C) or **ctypes**

CTypes

```
import ctypes

# Define a 2-byte structure (equivalent to a 'uint16' variable in C)
class CustomStruct(ctypes.BigEndianStructure):
    _fields_ = [
        ("fieldA", ctypes.c_uint16, 1),  # 1-bit field - Most Sig BIT
        ("fieldB", ctypes.c_uint16, 6),  # 6-bit field
        ("fieldC", ctypes.c_uint16, 4),  # 4-bit field
        ("fieldD", ctypes.c_uint16, 5)   # 5-bit field - Least SIG BIT
    ]

# Create new instance of the 'CustomStruct' data type
special_variable = CustomStruct()

# Access the fields of the structure
special_variable.fieldA = 1
special_variable.fieldB = 18
special_variable.fieldC = 5
special_variable.fieldD = 17
```

CTypes

```
# Print out individual fields
print("Field A = %i" % special_variable.fieldA)
print("Field B = %i" % special_variable.fieldB)
print("Field C = %i" % special_variable.fieldC)
print("Field D = %i" % special_variable.fieldD)

# Convert the structure to a byte array and print it out
print(bytes(special_variable))

# Alternate printing method (won't decode bytes as ASCII)
hex_string = "".join("%02x " % b for b in bytes(special_variable))
print("0x%s" % hex_string)
```