Network Applications: DNS Details;
Network App Programming--UDP

Y. Richard Yang

http://zoo.cs.yale.edu/classes/cs433/

9/19/2017
Outline

- Admin and recap
- DNS
  - High-level design
  - Details
  - Extensions/alternatives
- Network app programming
  - UDP
Admin

- Assignment Two linked on the schedule page
Recap: Domain Name System (DNS)

- **Function**
  - map between (domain name, service) to value, e.g.,
    - (www.cs.yale.edu, Addr) -> 128.36.229.30
    - (cs.yale.edu, Email) -> netra.cs.yale.edu
Recap: DNS

- Key design features of DNS
  - Hierarchical domain name space allowing delegation
  - Recursive or iterative queries
Outline

- Admin and recap
- DNS
  - High-level design
  - Details
  - Extensions/alternatives
- Network app programming
DNS Message Format?

Basic encoding decisions: UDP/TCP, how to encode domain name, how to encode answers...
Observing DNS Messages

- **Capture the messages**
  - DNS server is at port 53
    - Display and clear DNS cache
      - [https://support.apple.com/en-us/HT202516](https://support.apple.com/en-us/HT202516) (e.g., MAC)
  - Try to load the dns-capture file from class Schedule page, if you do not want live capture
**DNS Protocol, Messages**

**DNS protocol**: typically over UDP (can use TCP); *query* and *reply* messages, both with the *same message format*

<table>
<thead>
<tr>
<th>Identification</th>
<th>Flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of questions</td>
<td>Number of answer RRs</td>
</tr>
<tr>
<td>Number of authority RRs</td>
<td>Number of additional RRs</td>
</tr>
<tr>
<td><strong>Questions</strong></td>
<td><strong>Name, type fields for a query</strong></td>
</tr>
<tr>
<td>(variable number of questions)</td>
<td>(RRs in response to query)</td>
</tr>
<tr>
<td><strong>Answers</strong></td>
<td><strong>Records for authoritative servers</strong></td>
</tr>
<tr>
<td>(variable number of resource records)</td>
<td>(Authority)</td>
</tr>
<tr>
<td><strong>Authority</strong></td>
<td><strong>Additional “helpful” info that may be used</strong></td>
</tr>
<tr>
<td>(variable number of resource records)</td>
<td>(Additional information)</td>
</tr>
<tr>
<td><strong>Additional information</strong></td>
<td>(variable number of resource records)</td>
</tr>
</tbody>
</table>
DNS Details

- Header (Sec. 4.1.1 of https://www.ietf.org/rfc/rfc1035.txt)
- Encoding of questions (Sec. 4.1.2):
  - [Label-length label-chars]
- Encoding of answers (Sec. 4.1.3)
  - Pointer format
    (http://www.iana.org/assignments/dns-parameters/dns-parameters.xhtml)

- See example DNS packets
Name Encoding

Queries
- gmail.com: type A, class IN

Name: gmail.com

[Name Length: 9]
[Label Count: 2]
Type: A (Host Address) (1)
Class: IN (0x0001)
Message Compression
(Label Pointer)

```
Transaction ID: 0x6332
Flags: 0x8180 Standard query response, No error
Questions: 1
Answer RRs: 1
Authority RRs: 4
Additional RRs: 4

Queries
  > gmail.com: type A, class IN

Answers
  > gmail.com: type A, class IN, addr 216.58.219.223

Answer: offset 12
```
Recap: DNS Protocol, Messages

Many features: typically over **UDP** (can use **TCP**); query and reply messages with the same message format; *length/content encoding of names; simple compression; additional info as server push*

<table>
<thead>
<tr>
<th>Identification</th>
<th>Flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of questions</td>
<td>Number of answer RRs</td>
</tr>
<tr>
<td>Number of authority RRs</td>
<td>Number of additional RRs</td>
</tr>
<tr>
<td></td>
<td>12 bytes</td>
</tr>
<tr>
<td></td>
<td>Name, type fields for a query</td>
</tr>
<tr>
<td>Questions (variable number of questions)</td>
<td>RRs in response to query</td>
</tr>
<tr>
<td>Answers (variable number of resource records)</td>
<td>Records for authoritative servers</td>
</tr>
<tr>
<td>Authority (variable number of resource records)</td>
<td>Additional “helpful” info that may be used</td>
</tr>
<tr>
<td>Additional information (variable number of resource records)</td>
<td></td>
</tr>
</tbody>
</table>
What DNS did Right?

- Hierarchical delegation avoids central control, improving manageability and scalability

- Redundant servers improve robustness
  - see [http://www.internetnews.com/dev-news/article.php/1486981](http://www.internetnews.com/dev-news/article.php/1486981) for DDoS attack on root servers in Oct. 2002 (9 of the 13 root servers were crippled, but only slowed the network)

- Caching reduces workload and improves robustness

- Proactive answers reduce # queries on server and latency on client
Problems of DNS

- Simple query model, relatively static resource values and types make it harder to implement generic service discovery
  - Service discovery of all printers, lw2.printer.cs.yale.edu, lw3.printers.cs.yale.edu, ...
  - Although theoretically you can update the values of the records, it is rarely enabled

- Early binding (separation of DNS query from application query) does not work well in mobile, dynamic environments
  - e.g., load balancing, locate the nearest printer

- Each local domain needs servers, but an ad hoc domain may not have a DNS server
Outline

- Admin and recap
- DNS
  - High-level design
  - Details
  - Extensions/alternatives
Discussions

- What extension(s) to standard DNS operations do we need to allow service discovery, say to implement Bonjour (discover all local printers)?
  - each printer needs to provide the following info: host, port, printer info (e.g., support postscript)
Leverage DNS message format, but each node can announce its own services
Realizing DNS-SD without Central DNS Server: mDNS

- **Multicast in a small world**
  - no central address server
    - each node is a responder
  - link-local addressing
    - send to multicast address: 224.0.0.251

![Diagram showing network connections with IP addresses 169.254.1.219, 169.254.4.51, 169.254.10.29, and 169.254.1.219 connected to Network, and Printer with IP 169.254.10.29]
Example

- Use the `dns-sd` command on Mac as example
  - Advertise (register) an LPR printer on port 515

```bash
dns-sd -R "My Test" _printer._tcp . 515
pdl=application/postscript
```
<table>
<thead>
<tr>
<th>MDNS</th>
<th>Standard query response 0x0000 PTR, cache flush Ys-MacBook-Pro.local PTR, cache flush Ys-MacBook-Pro.local NSEC, ca...</th>
</tr>
</thead>
<tbody>
<tr>
<td>24 15:749230</td>
<td>224.0.0.251</td>
</tr>
<tr>
<td>25 15:750136</td>
<td>fe80::1c99:22de:a91f:ff02::fb</td>
</tr>
<tr>
<td>26 15:946487</td>
<td>172.27.27.21 224.0.0.251</td>
</tr>
<tr>
<td>27 15:946465</td>
<td>fe80::1c99:22de:a91f:ff02::fb</td>
</tr>
<tr>
<td>28 16:197838</td>
<td>172.27.27.21 224.0.0.251</td>
</tr>
<tr>
<td>29 16:197896</td>
<td>fe80::1c99:22de:a91f:ff02::fb</td>
</tr>
<tr>
<td>30 16:450462</td>
<td>172.27.27.21 224.0.0.251</td>
</tr>
<tr>
<td>31 16:450580</td>
<td>fe80::1c99:22de:a91f:ff02::fb</td>
</tr>
<tr>
<td>32 16:780958</td>
<td>172.27.27.21 224.0.0.251</td>
</tr>
<tr>
<td>33 16:780998</td>
<td>fe80::1c99:22de:a91f:ff02::fb</td>
</tr>
<tr>
<td>34 16:805259</td>
<td>172.27.27.21 224.0.0.251</td>
</tr>
<tr>
<td>35 16:805318</td>
<td>fe80::1c99:22de:a91f:ff02::fb</td>
</tr>
<tr>
<td>36 17:780216</td>
<td>172.27.27.21 224.0.0.251</td>
</tr>
<tr>
<td>37 17:780415</td>
<td>fe80::1c99:22de:a91f:ff02::fb</td>
</tr>
<tr>
<td>38 18:088077</td>
<td>172.27.27.21 224.0.0.251</td>
</tr>
<tr>
<td>39 18:089057</td>
<td>fe80::1c99:22de:a91f:ff02::fb</td>
</tr>
</tbody>
</table>
Offline Exercise

- Use the `dns-sd` command on Mac as example
  - Advertise (register) a web page on local machine

  ```
  dns-sd -R "My Test" _http._tcp . 80
  path=/path-to-page.html
  ```
Issue: How to Query

- Query needs a back pointer, PTR records
- Exercise: Use the dns-sd command on Mac as example
  - Browse web pages on local machines
    - `dns-sd -B _http._tcp`
  - See the page in Safari Bonjour
Network Service Discovery in Android

- Based on DNS-SD/mDNS
- Foundation for peer-to-peer/Wi-Fi Direct in Android

- See https://developer.android.com/training/connect-devices-wirelessly/nsd.html for programming using nsd
General Service/Naming Discovery Paradigm: Linda

“Distributed workspace” by David Gelernter in the 80’s at Yale

Very influential in naming and resource discovery

Key issues
- How to name services/resources
- How to resolve names
The Linda Paradigm

- Naming scheme:
  - arbitrary tuples (heterogeneous-type vectors)

- Name resolution:
  - Nodes write into shared memory
  - Nodes read matching tuples from shared memory
    - exact matching is required for extraction
Linda: Core API

- **out()**: writes tuples to shared space
  - example: `out("abc", 1.5, 12).
  - result: insert ("abc", 1.5, 12) into space

- **read()**: retrieves tuple copy matching arg list (blocking)
  - example: `read("abc", ?, A, ? B)
  - result: finds ("abc", 1.5, 12) and sets local variables
    A = 1.5, B = 12. Tuple ("abc", 1.5, 12) is still resident in space.

- **in()**: retrieves and deletes matching tuple from space (blocking)
  - example: same as above except ("abc", 1.5, 12) is deleted

- **eval(expression)**: similar to out except that the tuple argument to eval is evaluated
  - example: `eval("ab", -6, abs(-6))` creates tuple ("ab", -6, 6)
Linda Extension: JavaSpaces

- Industry took Linda principles and made modifications
  - add transactions, leases, events
  - store Java objects instead of tuples
  - a very comprehensive service discovery system

- Definitive book, “JavaSpaces Principles, Patterns, and Practice”
  - 2 of 3 authors got Ph.D.’s from Yale
Outline

- Admin and recap
- DNS
  - High-level design
  - Details
  - Extensions/alternatives
- Network app programming
  - UDP
Socket Programming

Socket API

- introduced in BSD4.1 UNIX, 1981

- Two types of sockets
  - connectionless (UDP)
  - connection-oriented (TCP)
Services Provided by Transport

- **User data protocol (UDP)**
  - multiplexing/demultiplexing

- **Transmission control protocol (TCP)**
  - multiplexing/demultiplexing
  - reliable data transfer
  - rate control: flow control and congestion control
Big Picture: Socket

**Diagram: Host or server**

- **Process**
  - Controlled by application developer
  - Buffers, states
- **Socket**
  - Controlled by operating system
- **Internet**

**Diagram: Host or server**

- **Process**
  - Controlled by application developer
  - Buffers, states
Outline

- Recap
- DNS
- Basic network application programming
  - Overview
  - UDP (Datagram Socket)
DatagramSocket (Java) (Basic)

- **DatagramSocket()**
  constructs a datagram socket and binds it to any available port on the local host.

- **DatagramSocket(int lport)**
  constructs a datagram socket and binds it to the specified port on the local host machine.

- **DatagramPacket(byte[] buf, int length)**
  constructs a DatagramPacket for receiving packets of length length.

- **DatagramPacket(byte[] buf, int length, InetAddress address, int port)**
  constructs a datagram packet for sending packets of length length to the specified port number on the specified host.

- **receive(DatagramPacket p)**
  receives a datagram packet from this socket.

- **send(DatagramPacket p)**
  sends a datagram packet from this socket.

- **close()**
  closes this datagram socket.
Connectionless UDP: Big Picture (Java version)

Server (running on serv)
- create socket, port=x, for incoming request:
  - serverSocket = DatagramSocket(x)

  read request from serverSocket

  generate reply, create datagram using client host address, port number

  write reply to serverSocket

Client
- create socket, clientSocket = DatagramSocket()

  Create datagram using (serv, x) as (dest addr. port), send request using clientSocket

  read reply from clientSocket

  close clientSocket
Example: UDPServer.java

- A simple UDP server which changes any received sentence to upper case.
Java Server (UDP): Create Socket

```java
import java.io.*;
import java.net.*;

class UDPServer {
    public static void main(String[] args) throws Exception {
        DatagramSocket serverSocket = new DatagramSocket(9876);
        // Create datagram socket and bind at port 9876
        // Check socket state:
        // %netstat -a -p udp -n
    }
}
```
System State after the Call

server
128.36.232.5
128.36.230.2

UDP socket space

address: {*}9876
snd/recv buf:

“*” indicates that the socket binds to all IP addresses of the machine:
% ifconfig -a

local address
why shown as “*”?

address: {128.36.232.5:53}
snd/recv buf:
Binding to Specific IP Addresses

**Server**
Public address: 128.36.59.2
Local address: 127.0.0.1
UDP socket space

- **address**: {127.0.0.1:9876}
snd/recv buf:

- **address**: {128.36.59.2:9876}
snd/recv buf:

- **address**: {*:6789}
snd/recv buf:

- **address**: {128.36.232.5:53}
snd/recv buf:

InetAddress sIP1 = InetAddress.getByName("localhost");
DatagramSocket ssock1 = new DatagramSocket(9876, sIP1);

InetAddress sIP2 = InetAddress.getByName("128.36.59.2");
DatagramSocket ssock2 = new DatagramSocket(9876, sIP2);

DatagramSocket serverSocket = new DatagramSocket(6789);
UDP Demultiplexing

**Server**

Public address: 128.36.59.2
Local address: 127.0.0.1

<table>
<thead>
<tr>
<th>Address</th>
<th>Protocol</th>
<th>Source Address</th>
<th>Source Port</th>
<th>Destination Address</th>
<th>Destination Port</th>
</tr>
</thead>
<tbody>
<tr>
<td>{127.0.0.1:9876}</td>
<td>UDP</td>
<td>127.0.0.1</td>
<td>9876</td>
<td>127.0.0.1</td>
<td>9876</td>
</tr>
<tr>
<td>{128.36.59.2:9876}</td>
<td>UDP</td>
<td>127.0.0.1</td>
<td>9876</td>
<td>128.36.59.2</td>
<td>9876</td>
</tr>
<tr>
<td>{128.36.232.5:53}</td>
<td>UDP</td>
<td>128.36.232.5</td>
<td>53</td>
<td>128.36.59.2</td>
<td></td>
</tr>
</tbody>
</table>

UDP demultiplexing is based on matching (dst address, dst port)
UDP Demultiplexing

Server
Public address: 128.36.59.2
Local address: 127.0.0.1

UDP socket space
address: {127.0.0.1:9876}
snd/recv buf:

address: {128.36.59.2:9876}
snd/recv buf:

address: {*:6789}
snd/recv buf:

address: {128.36.232.5:53}
snd/recv buf:

UDP demultiplexing is based on matching (dst address, dst port)

Client on server
P1

| SP: x   |
| DP: 9876 |
| S-IP: A |
| D-IP: 127.0.0.1 |

P3

| SP: y   |
| DP: 6789 |
| S-IP: C |
| D-IP: 128.36.59.2 |

client
IP: C
Per Socket State

- Each Datagram socket has a set of states:
  - local address
  - send buffer size
  - receive buffer size
  - timeout
  - traffic class

See
http://download.java.net/jdk7/archive/b123/docs/api/java/net/DatagramSocket.html

Example: socket state after clients sent msgs to the server
import java.io.*;
import java.net.*;

class UDPServer {
    public static void main(String args[]) throws Exception {
        DatagramSocket serverSocket = new DatagramSocket(9876);
        byte[] receiveData = new byte[1024];
        byte[] sendData = null;

        while(true) {
            DatagramPacket receivePacket = new DatagramPacket(receiveData, receiveData.length);
            serverSocket.receive(receivePacket);
            Create space for received datagram
            Receive datagram
        }
    }
}
 DatagramPacket

- **Receiving**
  - `DatagramPacket(byte[] buf, int length)` constructs a `DatagramPacket` for receiving packets of length `length`.
  - `DatagramPacket(byte[] buf, int offset, int length)` constructs a `DatagramPacket` for receiving packets starting at `offset`, `length`.

- **Sending**
  - `DatagramPacket(byte[] buf, int length, InetAddress address, int port)` constructs a datagram packet for sending packets of length `length` to the specified port number on the specified host.
  - `DatagramPacket(byte[] buf, int offset, int length, InetAddress address, int port)`
Java Server (UDP): Processing

```java
import java.io.*;
import java.net.*;

class UDPServer {
  public static void main(String args[]) throws Exception {

    // process data
    String sentence = new String(receivePacket.getData(), 0, receivePacket.getLength());
    String capitalizedSentence = sentence.toUpperCase();
    sendData = capitalizedSentence.getBytes();

    // getData() returns a pointer to an underlying buffer array;
    // for efficiency, don't assume receive() will reset the rest of the array

    // getLength() returns how much data is valid.
  }
}
```
Java Server (UDP): Response

- **Java DatagramPacket:**
  - `getAddress()` / `getPort()` returns the source address/port
Java server (UDP): Reply

Get IP addr
port #, of
sender

\[
\text{InetAddress IPAddress} = \text{receivePacket.getAddress}();
\]

\[
\text{int port} = \text{receivePacket.getPort}();
\]

Create datagram
to send to client

\[
\text{DatagramPacket sendPacket} = \\
\text{new DatagramPacket(sendData, sendData.length,} \\
\text{IPAddress, port)};
\]

Write out
datagram
to socket

\[
\text{serverSocket.send(sendPacket)};
\]

End of while loop, loop back and wait for
another datagram

{...}
A simple UDP client which reads input from keyboard, sends the input to server, and reads the reply back from the server.
Example: Java client (UDP)

```java
import java.io.*;
import java.net.*;

class UDPClient {
    public static void main(String args[]) throws Exception {
        BufferedReader inFromUser =
            new BufferedReader(new InputStreamReader(System.in));
        String sentence = inFromUser.readLine();
        byte[] sendData = sentence.getBytes();

        DatagramSocket clientSocket = new DatagramSocket();
        InetAddress sIPAddress = InetAddress.getByName("servname");

        DatagramPacket dp = new DatagramPacket(sendData, sendData.length, sIPAddress, 1234);
        clientSocket.send(dp);
    }
}
```
Example: Java client (UDP), cont.

Create datagram with data-to-send, length, IP addr, port

```java
DatagramPacket sendPacket =
    new DatagramPacket(sendData, sendData.length, sIPAddress, 9876);
clientSocket.send(sendPacket);
```

Send datagram to server

```java
byte[] receiveData = new byte[1024];
DatagramPacket receivePacket =
    new DatagramPacket(receiveData, receiveData.length);
clientSocket.receive(receivePacket);
```

Read datagram from server

```java
String modifiedSentence =
    new String(receivePacket.getData());
System.out.println("FROM SERVER:" + modifiedSentence);
clientSocket.close();
```
Demo

%mac: java UDPServer
% netstat to see buffer

%cicada: java UDPClient <server>

% wireshark to capture traffic
Discussion on Example Code

- A simple upper-case UDP echo service is among the simplest network service.

- Are there any problems with the program?