DNS: Domain Name System

People: many identifiers:
- SSN, name, Passport #

Internet hosts, routers:
- IP address (32 bit) - used for addressing datagrams
- "name", e.g., gaia.cs.umass.edu - used by humans

Q: map between IP addresses and name?

Domain Name System:
- distributed database implemented in hierarchy of many name servers
- application-layer protocol host, routers, name servers to communicate to resolve names (address/name translation)
- note: core Internet function implemented as application-layer protocol
- complexity at network’s “edge”

DNS name servers

Why not centralize DNS?
- single point of failure
- traffic volume
- distant centralized database
- maintenance

doesn’t scale!

- no server has all name-to-IP address mappings

local name servers:
- each ISP, company has local (default) name server
- host DNS query first goes to local name server

authoritative name server:
- for a host: stores that host’s IP address, name
- can perform name/address translation for that host’s name
DNS Query Example:

Bayou.UH.EDU> nslookup
Default Server: Masala.CC.UH.EDU
Address: 129.7.1.1

> www.yahoo.com
Server: Masala.CC.UH.EDU
Address: 129.7.1.1

Non-authoritative answer:
Name: www.yahoo.akadns.net
Addresses: 216.32.74.53, 216.32.74.55, 216.32.74.50, 216.32.74.51
216.32.74.52
Aliases: www.yahoo.com

set querytype=ANY

> www.yahoo.com
Server: Masala.CC.UH.EDU
Address: 129.7.1.1

Non-authoritative answer:
www.yahoo.com canonical name = www.yahoo.akadns.net

Authoritative answers can be found from:
YAHOO.com nameserver = ns1.YAHOO.com
YAHOO.com nameserver = ns3.europe.YAHOO.com
YAHOO.com nameserver = ns5.dcx.YAHOO.com
ns1.YAHOO.com internet address = 204.71.200.33
ns3.europe.YAHOO.com internet address = 194.237.108.51
ns5.dcx.YAHOO.com internet address = 216.32.74.10

DNS: Root name servers

- contacted by local name server that cannot resolve name
- root name server:
  - contacts authoritative name server if name mapping not known
  - gets mapping
  - returns mapping to local name server
- ~ dozen root name servers worldwide
Simple DNS example

host surf.eurecom.fr wants IP address of gaia.cs.umass.edu
1. Contacts its local DNS server, dns.eurecom.fr
2. dns.eurecom.fr contacts root name server, if necessary
3. root name server contacts authoritative name server, dns.umass.edu, if necessary

DNS example

Root name server:
- may not know authoritative name server
- may know intermediate name server: who to contact to find authoritative name server
**DNS: iterated queries**

**recursive query:**
- puts burden of name resolution on contacted name server
- heavy load?

**iterated query:**
- contacted server replies with name of server to contact
- "I don’t know this name, but ask this server"

**DNS: caching and updating records**

- once (any) name server learns mapping, it caches mapping
  - cache entries timeout (disappear) after some time
- update/notify mechanisms under design by IETF
  - RFC 2136
DNS records

**DNS**: distributed db storing resource records (RR)

**RR format**: (name, value, type, ttl)

- **Type=A**
  - name is hostname
  - value is IP address

- **Type=NS**
  - name is domain (e.g. foo.com)
  - value is IP address of authoritative name server for this domain

- **Type=CNAME**
  - name is an alias name for some "cannonical" (the real) name
  - value is cannonical name

- **Type=MX**
  - value is hostname of mailserver associated with name

DNS protocol, messages

**DNS protocol**: 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>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>question (variable number of questions)</td>
<td>answers (variable number of resource records)</td>
</tr>
<tr>
<td>authority (variable number of resource records)</td>
<td>additional information (variable number of resource records)</td>
</tr>
</tbody>
</table>
2: Application Layer

DNS protocol, messages

- Name, type fields for a query
- RRs in response to query
- Records for authoritative servers
- Additional “helpful” info that may be used

<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>
</tbody>
</table>

- Questions (variable number of questions)
- Answers (variable number of resource records)
- Authority (variable number of resource records)
- Additional information (variable number of resource records)

Socke programming

**Goal:** learn how to build client/server application that communicate using sockets

**Socket API**
- Introduced in BSD.1 UNIX, 1981
- Explicitly created, used, released by apps
- Client/server paradigm
- Two types of transport service via socket API:
  - Unreliable datagram
  - Reliable, byte stream-oriented

**Socket**
- A host-local, application-created/owned, OS-controlled interface (a "door") into which application process can both send and receive messages to/from another (remote or local) application process
### Socket types

- **Application program**
  - **Stream socket interface** (SOCK_STREAM)
  - **Datagram socket interface** (SOCK_DGRAM)
  - **Raw socket interface** (SOCK_RAW)

- **TCP**
- **UDP**
- **IP**

- **Physical and data link layers**

### Socket Functions

**Server:**
- create endpoint: `socket()`
- bind address: `bind()`
- specify queue: `listen()`
- wait for connection: `accept()`

**Client:**
- bind address: `bind()`
- connect to server: `connect()`
- transfer data:
  - read():
  - write():
  - recv():
  - send():
- datagrams: `recvfrom()`
- terminate:
  - close():
  - shutdown()
socket() System Call

```c
int socket (int family, int type, int protocol);
```

<table>
<thead>
<tr>
<th>family</th>
<th>type</th>
<th>protocol</th>
<th>Actual protocol</th>
</tr>
</thead>
<tbody>
<tr>
<td>AF_INET</td>
<td>SOCK_DGRAM</td>
<td>IPPROTO_UDP</td>
<td>UDP</td>
</tr>
<tr>
<td>AF_INET</td>
<td>SOCK_STREAM</td>
<td>IPPROTO_TCP</td>
<td>TCP</td>
</tr>
<tr>
<td>AF_INET</td>
<td>SOCK_RAW</td>
<td>IPPROTO_ICMP</td>
<td>ICMP</td>
</tr>
<tr>
<td>AF_INET</td>
<td>SOCK_RAW</td>
<td>IPPROTO_RAW</td>
<td>(raw)</td>
</tr>
</tbody>
</table>

Socket-programming using TCP

**Socket:** a door between application process and end-end-transport protocol (UCP or TCP)

**TCP service:** reliable transfer of bytes from one process to another
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 client-local 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 client
  - allows server to talk with multiple clients

TCP provides reliable, in-order transfer of bytes ("pipe") between client and server

Example client-server app:
- client reads line from standard input (inFromUser stream), sends to server via socket (outToServer stream)
- server reads line from socket
- server converts line to uppercase, sends back to client
- client reads, prints modified line from socket (inFromServer stream)

Input stream: sequence of bytes into process
Output stream: sequence of bytes out of process
Socket interface for connection-oriented concurrent server

A parent server creates many children; each child server serves only one client.

Client
- socket (...)
- connect (...)
- Repeat as needed
  - write (...)
  - read (...)
  - close (...)

Server
- socket (...)
- bind (...)
- listen (...)
- Repeat (infinitely)
  - accept (...)
  - fork (...)
  - close (listening)

Parent server
- close (accepting)

Child server
- Repeat as needed
  - read (...)
  - Process
  - write (...)

- close (accepting)

Relationship between the client and the server

Client
- a. After connect, before accept
- b. After accept
- c. After fork
- d. After parent closes ephemeral port
- e. After child closes well-known port

Server
- Parent
  - b. After accept
  - c. After fork
  - d. After parent closes ephemeral port
  - e. After child closes well-known port
TCP Concurrent Server Program

```c
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
#include <netinet/in.h>
#include <stdio.h>
#include <string.h>
#define MAXBUF 256

void main(void) {
    char buf[MAXBUF];
    int listenSocket;
    int acceptSocket;
    int clientAddrLen;
    struct sockaddr_in  serverAddr;
    struct sockaddr_in  clientAddr;
    listenSocket = socket(AF_INET, SOCK_STREAM, 0);
    memset(&serverAddr, 0, sizeof(serverAddr));
    serverAddr.sin_family = AF_INET;
    serverAddr.sin_port = htons(a-well-know-port);
    serverAddr.sin_addr.s_addr = htonl(INADDR_ANY);
    bind(listenSocket, &serverAddr, sizeof(serverAddr));
    listen(listenSocket, 1);
    clientAddrLen = sizeof(clientAddr);
    for (;;) {
        acceptSocket = accept(listenSocket, &clientAddr, &clientAddrLen);
        pid = fork();
        if (pid != 0) { /* parent */
            close(acceptSocket);
            continue;
        } /* if */
        else { /* child */
            close(listenSocket);
            memset(buf, 0, MAXBUF);
            while (read(acceptSocket, buf, MAXBUF) > 0) {
                PROCESS (.........);
                memset(buf, 0, MAXBUF);
                write(acceptSocket, buf, MAXBUF);
                memset(buf, 0, MAXBUF);
            } /* while */
            close(acceptSocket);
        } /* else */
    } /* for */
} /* main */
```

TCP Concurrent Server Program (cont'd)

```c
for (;;) {
    acceptSocket = accept(listenSocket, &clientAddr, &clientAddrLen);
    pid = fork();
    if (pid != 0) { /* parent */
        close(acceptSocket);
        continue;
    } /* if */
    else { /* child */
        close(listenSocket);
        memset(buf, 0, MAXBUF);
        while (read(acceptSocket, buf, MAXBUF) > 0) {
            PROCESS (.........);
            memset(buf, 0, MAXBUF);
            write(acceptSocket, buf, MAXBUF);
            memset(buf, 0, MAXBUF);
        } /* while */
        close(acceptSocket);
    } /* else */
} /* for */
} /* main */
```
TCP Concurrent Client Program

```c
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/socket.h>
#include <netdb.h>
#include <netinet/in.h>
#include <stdio.h>
#include <string.h>
#define MAXBUF 256

void main(void) {
    char buf[MAXBUF];
    int activeSocket;
    struct sockaddr_in remoteAddr;
    struct sockaddr_in localAddr;
    struct hostent *hptr;
    activeSocket = socket(AF_INET, SOCK_STREAM, 0);
    memset(&remoteAddr, 0, sizeof(remoteAddr));
    remoteAddr.sin_family = AF_INET;
    remoteAddr.sin_port = htons(a-well-know-port);
    hptr = gethostbyname("a-domain-name");
    memcpy((char*)&remoteAddr.sin_addr.s_addr,
           hptr->h_addr_list[0], hptr->h_length);
    memset(&buf, 0, MAXBUF);
    while (gets(buf)) {
        write(activeSocket, buf, MAXBUF);
        memset(&buf, 0, MAXBUF);
        read(sockds, buf, MAXBUF);
        printf("%s\n", buf);
        memset(&buf, 0, MAXBUF);
    } /* while */
    close(activeSocket);
} /* main */
```

Client/server socket interaction: TCP

<table>
<thead>
<tr>
<th>Server (running on hostid)</th>
<th>Client</th>
</tr>
</thead>
<tbody>
<tr>
<td>create socket, port=x, for incoming request: welcomeSocket = ServerSocket()</td>
<td>create socket, connect to hostid.port=x clientSocket = Socket()</td>
</tr>
<tr>
<td>wait for incoming connection request connectionSocket = welcomeSocket.accept()</td>
<td>send request using clientSocket</td>
</tr>
<tr>
<td>read request from connectionSocket</td>
<td>read reply from clientSocket</td>
</tr>
<tr>
<td>write reply to connectionSocket</td>
<td>close clientSocket</td>
</tr>
<tr>
<td>close connectionSocket</td>
<td>close connectionSocket</td>
</tr>
</tbody>
</table>

TCP connection setup
Example: Java client (TCP)

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

class TCPClient {
    public static void main(String argv[]) throws Exception {
        String sentence;
        String modifiedSentence;

        BufferedReader inFromUser = new BufferedReader(new InputStreamReader(System.in));
        Socket clientSocket = new Socket("hostname", 6789);
        DataOutputStream outToServer = new DataOutputStream(clientSocket.getOutputStream());

        BufferedReader inFromServer = new BufferedReader(new InputStreamReader(clientSocket.getInputStream()));
        sentence = inFromUser.readLine();
        outToServer.writeBytes(sentence + '
');
        modifiedSentence = inFromServer.readLine();
        System.out.println("FROM SERVER: " + modifiedSentence);
        clientSocket.close();
    }
}
```

Example: Java client (TCP), cont.

Create 
input stream

Create 
client socket,
connect to server

Create 
output stream
attached to socket

BufferedReader inFromUser = new BufferedReader(new InputStreamReader(System.in));
Socket clientSocket = new Socket("hostname", 6789);
DataOutputStream outToServer = new DataOutputStream(clientSocket.getOutputStream());

sentence = inFromUser.readLine();
outToServer.writeBytes(sentence + \n);
modifiedSentence = inFromServer.readLine();
System.out.println("FROM SERVER: " + modifiedSentence);
clientSocket.close();
```
Example: Java server (TCP)

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

class TCPServer {
    public static void main(String argv[]) throws Exception {
        String clientSentence;
        String capitalizedSentence;

        ServerSocket welcomeSocket = new ServerSocket(6789);
        while(true) {
            Socket connectionSocket = welcomeSocket.accept();
            BufferedReader inFromClient =
                new BufferedReader(new InputStreamReader(connectionSocket.getInputStream()));
            clientSentence = inFromClient.readLine();
            capitalizedSentence = clientSentence.toUpperCase() + '
';
            outToClient.writeBytes(capitalizedSentence);
        }
    }
}
```

Example: Java server (TCP), cont

```java
DataOutputStream outToClient =
    new DataOutputStream(connectionSocket.getOutputStream());

clientSentence = inFromClient.readLine();

capitalizedSentence = clientSentence.toUpperCase() + 'n';

outToClient.writeBytes(capitalizedSentence);
```
Socket programming with UDP

UDP: no "connection" between client and server
- no handshaking
- sender explicitly attaches IP address and port of destination
- server must extract IP address, port of sender from received datagram

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

Socket interface for connectionless iterative server

Each server serves many clients but handles one request at a time. Requests from different clients can be mingled together.
UDP Iterative Server Program

```c
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
#include <netinet/in.h>
#include <stdio.h>
#include <string.h>
#define MAXBUF 256

void main(void) {
    char buf[MAXBUF];
    int passiveSocket;
    int clientAddrLen;
    struct sockaddr_in serverAddr;
    struct sockaddr_in clientAddr;
    passiveSocket = socket(AF_INET, SOCK_DGRAM, 0);
    memset(&serverAddr, 0, sizeof(serverAddr));
    serverAddr.sin_family = AF_INET;
    serverAddr.sin_port = htons(a-well-know-port);
    serverAddr.sin_addr.s_addr = htonl(INADDR_ANY);
    bind(passiveSocket, &serverAddr, sizeof(serverAddr));
    clientAddrLen = sizeof(clientAddr);
    while (true) {
        while (recvfrom(passiveSocket, buf, MAXBUF, 0, &clientAddr, &clientAddrLen) > 0) {
            PROCESS (.........);
            memset(buf, 0, MAXBUF);
            sendto(passiveSocket, buf, MAXBUF, 0, &clientAddr, clientAddrLen);
            memset(buf, 0, MAXBUF);
        }
    }
}
```

UDP Iterative Client Program

```c
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
#include <netinet/in.h>
#include <stdio.h>
#include <string.h>
#define MAXBUF 256

void main(void) {
    char buf[MAXBUF];
    int activeSocket;
    struct sockaddr_in remoteAddr;
    struct sockaddr_in localAddr;
    struct hostent *hptr;
    activeSocket = socket(AF_INET, SOCK_DGRAM, 0);
    memset(&remoteAddr, 0, sizeof(remoteAddr));
    memset(&buf, 0, sizeof(buf));
    while (gets(buf)) {
        sendto(activeSocket, buf, sizeof(buf), 0, &remoteAddr, sizeof(remoteAddr));
        printf("%s\n", buf);
        memset(&buf, 0, sizeof(buf));
    }
    close(activeSocket);
}
```
Client/server socket interaction: UDP

Server (running on hostid)

- create socket, port=x, for incoming request:
  - serverSocket = DatagramSocket()
- read request from serverSocket
- write reply to serverSocket specifying client host address, port number

Client

- create socket, clientSocket = DatagramSocket()
- Create, address (hostid, port=x), send datagram request using clientSocket
- read reply from clientSocket
- close clientSocket

Example: Java client (UDP)

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

class UDPClien{
    public static void main(String args[]) throws Exception {
        BufferedReader inFromUser =
            new BufferedReader(new InputStreamReader(System.in));
        DatagramSocket clientSocket = new DatagramSocket();
        InetAddress IPAddress = InetAddress.getByName("hostname");
        byte[] sendData = new byte[1024];
        byte[] receiveData = new byte[1024];
        String sentence = inFromUser.readLine();
        sendData = sentence.getBytes();
```
Example: Java client (UDP), cont.

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

Send datagram
to server

Read datagram
from server

DatagramPacket sendPacket = new DatagramPacket(sendData, sendData.length, IPAddress, 9876);
clientSocket.send(sendPacket);

DatagramPacket receivePacket = new DatagramPacket(receiveData, receiveData.length);
clientSocket.receive(receivePacket);
String modifiedSentence = new String(receivePacket.getData());
System.out.println("FROM SERVER:" + modifiedSentence);
clientSocket.close();

Example: Java server (UDP)

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 = new byte[1024];
        while(true) {
            DatagramPacket receivePacket = new DatagramPacket(receiveData, receiveData.length);
            serverSocket.receive(receivePacket);
        }
    }
}
Example: Java server (UDP), cont

```java
String sentence = new String(receivePacket.getData());
InetAddress IPAddress = receivePacket.getAddress();
int port = receivePacket.getPort();

String capitalizedSentence = sentence.toUpperCase();
sendData = capitalizedSentence.getBytes();
DatagramPacket sendPacket =
    new DatagramPacket(sendData, sendData.length, IPAddress, port);
serverSocket.send(sendPacket);
```
Chapter 2: Summary

Most importantly: learned about protocols

- typical request/reply message exchange:
  - client requests info or service
  - server responds with data, status code
- message formats:
  - headers: fields giving info about data
  - data: info being communicated
- control vs. data msgs
  - in-based, out-of-band
- centralized vs. decentralized
- stateless vs. stateful
- reliable vs. unreliable msg transfer
- "complexity at network edge"
- security: authentication