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
- 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: 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

---

**Diagram:**
- Requesting host: `surf.eurecom.fr`
- Local name server: `dns.eurecom.fr`
- Root name server
- Authoritative name server: `dns.umass.edu`
- `gaia.cs.umass.edu`
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**: \((\text{name}, \text{value}, \text{type}, \text{ttl})\)

- **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></th>
<th>identification</th>
<th>flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>number of questions</td>
<td></td>
<td></td>
</tr>
<tr>
<td>number of answer RRs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>number of authority RRs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>number of additional RRs</td>
<td></td>
<td></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)
DNS protocol, messages

<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>

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

12 bytes
Socket programming

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

**Socket API**
- introduced in BSD4.1 UNIX, 1981
- explicitly created, used, released by apps
- client/server paradigm
- two types of transport service via socket API:
  - unreliable datagram (UDP)
  - reliable, byte stream-oriented (TCP)

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
Sockets: Conceptual View

**Diagram:**

- **USER APPL.**
- **SOCKET LAYER**
  - `buffered data yet to be sent`
  - `bind()`
  - `sendto()`
  - `create()`
  - `recvfrom()`
- **TRANSPORT LAYER**
  - **Port Number**
  - **Socket Parameters**
Referencing Book

**TITLE:** UNIX Network Programming: Networking APIs Sockets & XT1
by **Author:** W. Richard Stevens
**ISBN:** 013490012X
**Publisher:** Prentice Hall PTR
**Publish Date:** 10/01/1997
**Binding:** Hardcover , 1240 pages , 2 edition
**List Price:** USD 71.00
**Connection-Oriented Application**

1. **Server gets ready to service clients**
   - **Creates** a socket
   - **Binds** an address to the socket
     - Server’s address should be made known to clients

2. **Client contacts the server**
   - **Creates** a socket
   - **Connects** to the server
     - Client has to supply the address of the server

3. **Accepts** connection requests from clients

4. **Further communication is specific to application**
Creating a socket

```c
int  socket(int family, int service, int protocol)
```

- **family**: symbolic name for protocol family
  - `AF_INET, AF_UNIX`
- **type**: symbolic name for type of service
  - `SOCK_STREAM, SOCK_DGRAM, SOCK_RAW`
- **protocol**: further info in case of raw sockets
  - typically set to 0

Returns **socket descriptor**
More on Socket Descriptor

- A 5-tuple associated with a socket
  - \{protocol, local IP address, local port, remote IP address, remote port\}
    - socket() fills the protocol component
    - local IP address/port filled by bind()
    - remote IP address/port by accept() in case of server
    - in case of client both local and remote by connect()

- Complete socket is like a file descriptor
  - Both send and recv through same socket

- Accept returns a new complete socket
  - Original one can be used to accept more connections
Streams and Datagrams

- Connection-oriented reliable byte stream
  - SOCK_STREAM based on TCP
  - No message boundaries
  - Multiple writes may be consumed by one read

- Connectionless unreliable datagram
  - SOCK_DGRAM based on UDP
  - Message boundaries are preserved
  - Each sendto corresponds to one recvfrom
Binding Socket with an Address

int bind(int sd, struct sockaddr *addr, int len)

- **sd**: socket descriptor returned by socket()
- **addr**: pointer to sockaddr structure containing address to be bound to socket
- **len**: length of address structure

Returns 0 if success, -1 otherwise
Specifying Socket Address

```c
struct sockaddr_in {
    short         sin_family;   /* set to AF_INET */
    u_short       sin_port;    /* 16 bit port number */
    struct in_addr sin_addr;   /* 32 bit host address */
    char          sin_zero[8]; /* not used */
};

struct in_addr {
    u_long        s_addr;      /* 32 bit host address */
};
```
Bind Example

```c
int sd;
struct sockaddr_in ma;
sd = socket(AF_INET, SOCK_STREAM, 0);
ma.sin_family = AF_INET;
ma.sin_port = htons(5100);
ma.sin_addr.s_addr = htonl(INADDR_ANY);
if (bind(sd, (struct sockaddr *) &ma, sizeof(ma)) != -1)
...
```
Connecting to Server

int connect(int sd, struct sockaddr *addr, int len)

- `sd`: socket descriptor returned by `socket()`.
- `addr`: pointer to sockaddr structure containing server’s address (IP address and port).
- `len`: length of address structure.

Returns 0 if success, -1 otherwise.
Connect Example

```c
int sd;
struct sockaddr_in sa;
sd = socket(AF_INET, SOCK_STREAM, 0);

sa.sin_family = AF_INET;
sa.sin_port = htons(5100);
sa.sin_addr.s_addr = inet_addr("128.101.34.78");
if (connect(sd, (struct sockaddr *) &sa, sizeof(sa)) != -1)
    ...
```
Connection Acceptance by Server

int accept(int sd, struct sockaddr *from, int *len)

- sd: socket descriptor returned by socket()
- from: pointer to sockaddr structure which gets filled with client’s address
- len: length of address structure

Blocks until connection requested or error
- returns a new socket descriptor on success
Connection-oriented Server

```c
int    sd, cd, calen;
struct sockaddr_in   ma, ca;

sd = socket(AF_INET, SOCK_STREAM, 0);
ma.sin_family = AF_INET;
ma.sin_port = htons(5100);
ma.sin_addr.s_addr = htonl(INADDR_ANY);
bind(sd, (struct sockaddr *) &ma, sizeof(ma));

listen(sd, 5);
calen = sizeof(ca);

cd = accept(sd, (struct sockaddr *) &ca, &calen);
...read and write to client treating cd as file descriptor...
```
sockid = socket() 

bind() 

sockid is used for waiting for client connection request 

listen() 

newsockid = accept() 

create a child process, fork(), to handle communication (provide service) to client 

child process 

parent process 

child communications read() / write() with client and provides service via newsockid 

close(newsockid) and exit() 

Typical Server Structure
Input/Output Multiplexing

- **Polling**
  - Nonblocking option using `fcntl()`/`ioctl()`
  - Waste of computer resources

- **Asynchronous I/O**
  - Generates a signal on an input/output event
  - Expensive to catch signals

- **Wait for multiple events simultaneously**
  - Using `select()` system call
  - Process sleeps till an event happens
Select System Call

int select(int maxfdp1, fd_set *readfds,
           fd_set *writefds, fd_set *exceptfds,
           struct timeval *timeout)

- maxfdp1: largest numbered file descriptor + 1
- readfds: check if ready for reading
- writefds: check if ready for writing
- exceptfds: check for exceptional conditions
- timeout: specifies how long to wait for events
Timeout in Select

- Wait indefinitely till there is an event
  - Pass NULL to the `timeout` argument
- Don’t wait beyond a fixed amount of time
  - Pass pointer to a `timeval` structure specifying the number of seconds and microseconds.
- Just poll without blocking
  - Pass pointer to a `timeval` structure specifying the number of seconds and microseconds as 0
Working with File Descriptor Set

- **Set is represented by a bit mask**
  - Keep a descriptor in/out the set, turn on/off corresponding bit
    - Using FD_ZERO, FD_SET and FD_CLR
    - Use FD_ISSET to check for membership

- **Example:**
  - Make descriptors 1 and 4 members of the readset
    ```c
    fd_set   readset;
    FD_ZERO(&readset);
    FD_SET(1, &readset);
    FD_SET(4, &readset);
    ```
  - Check if 4 is a member of readset
    ```c
    FD_ISSET(4, &readset);
    ```
Return Values from Select

- Arguments `readfds` etc are value-result
- Pass set of descriptors you are interested in
- Select modifies the descriptor set
  - Keeps the bit on if an event on the descriptor
  - Turns the bit off if no event on the descriptor
- On return, test the descriptor set
  - Using `FD_ISSET`
Select Example

```c
fd_set   readset;
FD_ZERO(&readset);
FD_SET(0, &readset);
FD_SET(4, &readset);
select(5, &readset, NULL, NULL, NULL);
if  (FD_ISSET(0, &readset) {
        /* something to be read from  0 */
}
if  (FD_ISSET(4, &readset) {
        /* something to be read from  4 */
}
```
Servers and Services

- **Mapping between names and addresses (DNS)**
  - Host name to address: `gethostbyname()`
  - Host address to name: `gethostbyaddr()`
  - Try using command `host`
    - Example: “`host mail.cs.umn.edu`” or “`host 128.101.35.200`”

- **Mapping from a service to a port number**
  - Use `getservbyname()`
  - Look at `/etc/services` or try `ypcat services`
Operations on Socket

- `getpeername()`
  - returns remote address part of socket tuple

- `getsockname()`
  - returns local address part of socket tuple
Utility Functions

- `inet_addr()`, `inet_ntoa()`
  - dotted decimal string to/from 32-bit address
- `htonl()`, `htons()`, `ntohl()`, `ntohs()`
  - byte ordering functions
- `bcopy()`, `bzero()`, `bcmp()`
  - byte operations
Project 1: Conference App.

- Template will be provided
- SunOS 5.8
- Read project description before starting
Chapter 2: Summary

Our study of network apps now complete!

- application service requirements:
  - reliability, bandwidth, delay
- client-server paradigm
- Internet transport service model
  - connection-oriented, reliable: TCP
  - unreliable, datagrams: UDP

- specific protocols:
  - http
  - ftp
  - smtp, pop3
  - dns

- socket programming
  - client/server implementation
  - using tcp, udp sockets
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