Sockets

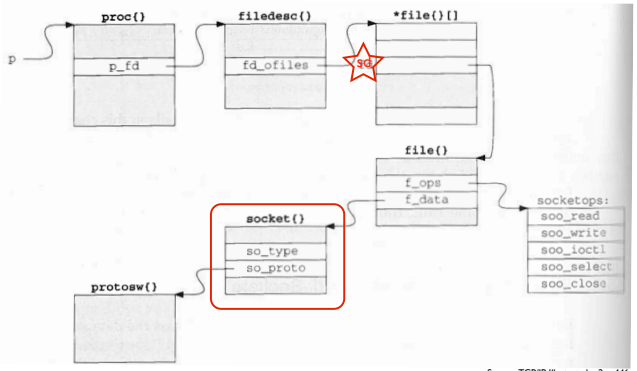
What exactly are sockets?
- an endpoint of a connection
- similar to UNIX file I/O API (provides a file descriptor)
- associated with each end-point (end-host) of a connection
  - identified by the IP address and port number of *both* the sender and receiver

Berkeley sockets is the most popular network API
- runs on Linux, FreeBSD, OS X, Windows
- fed off the popularity of TCP/IP
- can build higher-level interfaces on top of sockets
  - e.g., Remote Procedure Call (RPC)

Based on C, single threaded model
- does not require multiple threads

Useful sample code available at
- http://www.kohala.com/start/unpv12e.html

Process File Table and Socket Descriptor



Stevens TCP/IP Illustrated v. 2 p. 446

Types of Sockets

Different types of sockets implement different service models
- Stream v.s. datagram

Stream socket (aka TCP)
- connection-oriented
- reliable, in order delivery
- at-most-once delivery, no duplicates
- used by e.g., ssh, http

Datagram socket (aka UDP)
- connectionless (just data-transfer)
- "best-effort" delivery, possibly lower variance in delay
- used by e.g., IP telephony, streaming audio, streaming video, Internet gaming, etc.

Simplified E-mail Delivery

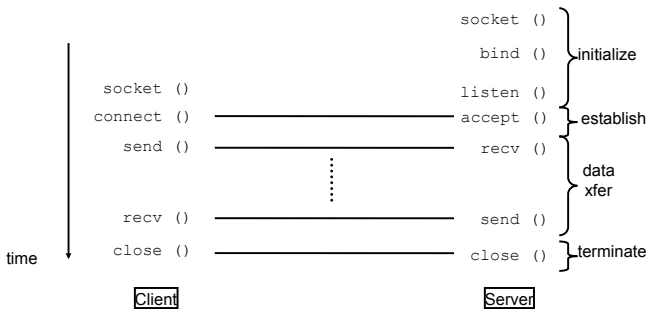
You want to send email to friend@cs.usc.edu

At your end, your mailer
- translates cs.usc.edu to its IP address (128.125.1.45)
- decides to use TCP as the transport protocol (Why?)
- creates a socket
- `connect`s to 128.125.1.45 at the well-known SMTP port # (25)
- parcels out your email into packets
- `send`s the packets out

On the Internet, your packets got:
- transmitted
- routed
- buffered
- forwarded, or
- dropped

At the receiver, smtpd
- must make a "receiver" ahead of time:
- creates a socket
- decides on TCP
- `bind`s the socket to smtp's well-known port #
- `listen`s on the socket
- `accept`s your smtp connection requests
- `recv`es your email packets

Stream/TCP Sockets



Stream/TCP Socket

Server:

- server process must first be running
- server must have created socket (door) that welcomes client's contact

Client:

- creates client-local TCP socket
- specifies IP address, port number of server process
- When client contacts server: 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
- source port numbers used to distinguish clients

Initialize (Client)

```
int sd;
if ((sd = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP)) < 0) {
  perror("socket");
  printf("Failed to create socket\n");
  abort();
}
```

`socket()` creates a socket data structure and attaches it to the process's file descriptor table

Handling errors that occur rarely usually consumes most of systems code

Establish (Client)

```
struct sockaddr_in sin;

struct hostent *host = gethostbyname(argv[1]);
unsigned int server_addr = *(unsigned long *) host->h_addr_list[0];
unsigned short server_port = atoi(argv[2]);

memset(&sin, 0, sizeof(sin));

sin.sin_family = AF_INET;
sin.sin_addr.s_addr = server_addr;
sin.sin_port = htons(server_port);

if (connect(sd, (struct sockaddr *) &sin, sizeof (sin)) < 0) {
  perror("connect");
  printf("Cannot connect to server\n");
  abort();
}
```

`connect()` initiates connection (for TCP)

Sending Data Stream (Client)

```
int
send_packets(char *buffer, int buffer_len)
{
   sent_bytes = send(sd, buffer, buffer_len, 0);

   if (send_bytes < 0)
        perror("send");

   return 0;
}
```

- returns how many bytes are actually sent
- must loop to make sure that all is sent (except for blocking I/O, see UNP Section 6.2)

What is blocking and non-blocking I/O?
Why do you want to use non-blocking I/O?

Initialize (Server)

```
int sd;
int optval = 1;
if ((sd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
  perror("opening TCP socket");
  abort();
}
if (setsockopt sd, SOL_SOCKET, SO_REUSEADDR, &optval,
  sizeof(optval)) <0) {
  perror("reuse address");
  abort();
}
```

`SO_REUSEADDR` allows server to restart or multiple servers to bind to the same port # with different IP addresses

Initialize (Server `bind` addr)

```
struct sockaddr_in sin;

memset(&sin, 0, sizeof (sin));

sin.sin_family = AF_INET;
sin.sin_addr.s_addr = INADDR_ANY;
sin.sin_port = htons(server_port);

if (bind(sd, (struct sockaddr *) &sin, sizeof (sin)) < 0) {
  perror("bind");
  printf("Cannot bind socket to address\n");
  abort();
}
```

`bind()` used only by server, to "label" a socket with an IP address and/or port#

- Why do we need to label a socket with a port#?
- Must each service have a well-known port?
- Why do we need to label a socket with IP address?
- What if we want to receive packets from all network interfaces of the server machine?
- Why not always receive from all interfaces?
- What defines a connection?

Initialize (Server `listen`)

```
if (listen(sd, qlen) < 0) {
 perror("error listening");
 abort();
}
```

- specifies max number of pending TCP connections waiting to be accepted (using `accept()`)
- only useful for connection oriented services, but may be used by UDP also
- TCP SYN denial of service attack

API design question: why not merge `bind()` and `listen()`?

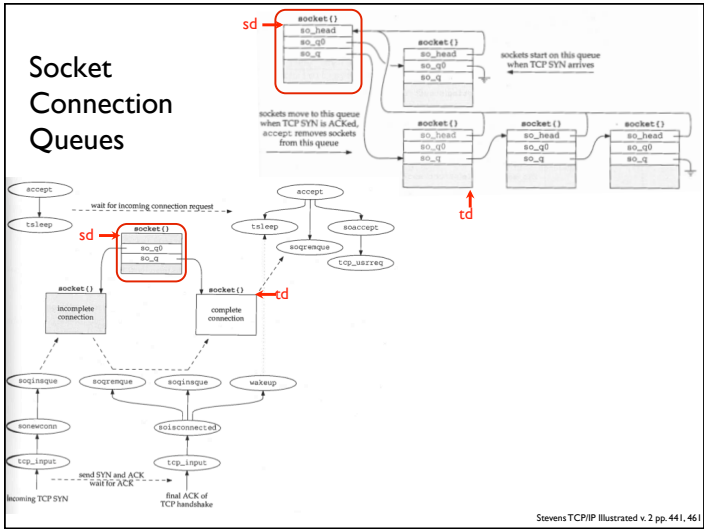
Establish (Server `accept`)

```
int addr_len = sizeof(addr);
int td;

td = accept(sd, (struct sockaddr *) &addr, &addr_len);

if (td < 0) {
  perror("error accepting connection");
  abort();
}
```

- waits for incoming client connection
- returns a connected socket (different from the `listen`ed to socket)

API design question: why not merge `listen()` and `accept()`?

Socket Connection Queues



Stevens TCP/IP Illustrated v. 2 pp. 441, 461

Receiving Data Stream (Server)

```
int
receive_packets(char *buffer, int buffer_len, int
  *bytes_read)
{
    int left = buffer_len - *bytes_read;
    received = recv(td, buffer + *bytes_read, left, 0);
    . . . .
    return 0;
}
```

- returns the number of bytes actually received
- 0 if connection is closed, -1 on error
- if non-blocking: -1 if no data, with `errno` set to `EWOULDBLOCK`
- must loop to ensure all data is received
- Why doesn't recv return all of the data at once?
- How do you know you have received everything sent?

Connection `close` (Client and Server)

- `close()` marks socket unusable
- actual tear down depends on TCP
  `bind: Address already in use`
- socket option `SO_LINGER` can be used to specify whether `close()` should return immediately or abort connection or wait for termination
- The APIs `getsockopt()` and `setsockopt()` are used to query and set socket options (see UNP Ch. 7)
- Other useful options:
  - `SO_RCVBUF` and `SO_SNDBUF` used to set buffer sizes
  - `SO_KEEPALIVE` tells server to ping client periodically

How to Handle Multiple I/O Streams?

Where do we get incoming data?

- `stdin` (typically keyboard/mouse input)
- sockets

Asynchronous arrival, program doesn't know when data will arrive

Alternatives:

multithreading: each thread handles one I/O stream (482)

I/O multiplexing: a single thread handles multiple I/O streams

Flavors:

a. blocking I/O (default):
   - put process to sleep until I/O is ready
   - blocking for: device availability and I/O completion
   - by polling or use of `select()`

b. non-blocking I/O:
   - only checks for device availability
   - by polling or signal driven (not covered)

c. asynchronous I/O:
   - process is notified when I/O is completed (not covered)

Non-Blocking I/O: Polling

```
int opts = fcntl(sock, F_GETFL);
if (opts < 0) {
    perror("fcntl(F_GETFL)");
    abort();
}

if (fcntl(sock, F_SETFL, opts | O_NONBLOCK) < 0) {
    perror("fcntl(F_SETFL)");
    abort();
}
while (1) {
    if (receive_packets(buffer, buffer_len,
       &bytes_read) != 0) {
        break;
    }

    if (read_user(user_buffer, user_buffer_len,
                  &user_bytes_read) != 0) {
        break;
    }
}
```

get current socket option settings

set non-blocking I/O socket option

get data from socket

get user input

Blocking I/O: `select()`

`select(maxfd, readset, writeset, exceptset, timeout)`

- waits on multiple file descriptors/sockets and timeout
- application does not consume CPU cycles while waiting
- `maxfd` is the maximum file descriptor number + 1
  - if you have only one descriptor, number 5, `maxfd` is 6
- descriptor sets provided as bit mask
  - use `FD_ZERO`, `FD_SET`, `FD_ISSET`, and `FD_CLR` to work with the descriptor sets
- returns as soon as *one* of the specified sockets are ready to be read or written, or they have an error, or timeout exceeded
  - returns # of ready sockets, -1 on error, 0 if timed out and no device is ready (what for?)

Blocking I/O: `select()`

```
fd_set read_set;
struct timeval time_out;
while (1) {
    FD_ZERO(read_set);
    FD_SET(stdin, read_set); /* stdin is typically 0 */
    FD_SET(sd, read_set);
    time_out.tv_usec = 100000; time_out.tv_sec = 0;

    err = select(MAX(stdin, sd) + 1, &read_set, NULL, NULL, &time_out);

    if (err < 0) {
        perror ("select");
        abort ();
    }
    if (err > 0) {
        if (FD_ISSET(sd, read_set))
            if (receive_packets(buffer, buffer_len, &bytes_read) != 0)
                break;
        if (FD_ISSET(stdin, read_set))
            if (read_user(user_buffer, user_buffer_len, &user_bytes_read) != 0)
                break;
    }
    else { . . . /* timed out */ }
}
```

set up parameters for `select()`

run `select()`

interpret result

Blocking I/O: polling

Which of the following would you use? Why?

```
loop {
  select(. . . , timeout);

  recv();
} till done;
```

or:

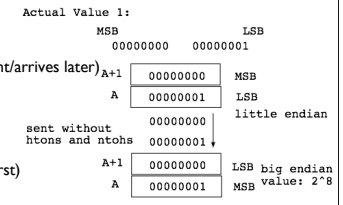
```
loop {
  sleep(seconds)

  recv();
} till done;
```

Byte Ordering

```
struct sockaddr_in sin;

memset(&sin, 0, sizeof (sin));

sin.sin_family = AF_INET;
sin.sin_addr.s_addr = IN_ADDR;
sin.sin_port = htons(server_port);

if (bind(sd, (struct sockaddr *) &sin, sizeof (sin)) < 0) {
  perror("bind");
  printf("Cannot bind socket to address\n");
  abort();
}
```

Little-endian:
Most Significant Byte (MSB) in high address (sent/arrives later)
(Intel x86 and Alpha)

Big-endian: MSB in low address (sent/arrives first)
(PowerPC, Sun Sparc, HP-PA)



Bi-endian: switchable endians (ARM, PowerPC after G5, Alpha, SPARC V9)

Byte Ordering Solution

To ensure interoperability, **ALWAYS** translate `short`, `long`, `int`, `uint16_t`, `uint32_t`, to/from "network byte order" before/after transmission

Use these macros:
`htons()`: host to network short
`htonl()`: host to network long
`ntohs()`: network to host short
`ntohl()`: network to host long

Do we have to be concerned about byte ordering for `char` type?
How about `float` and `double`?

Establish (Client)

```
struct sockaddr_in sin;
struct hostent *host = gethostbyname(argv[1]); // argv[1] contains host name
unsigned int server_addr = *(unsigned long *) host->h_addr_list[0];
unsigned short server_port = atoi(argv[2]);

memset(&sin, 0, sizeof(sin));
sin.sin_family = AF_INET;
sin.sin_addr.s_addr = server_addr;
sin.sin_port = htons(server_port);

if (connect(sd, (struct sockaddr *) &sin, sizeof (sin)) < 0) {
  perror("connect");
  printf("Cannot connect to server\n");
  abort();
}
```

host name, e.g., www.eecs.umich.edu
- identifies a single host
- variable length string
- maps to one or more IP address
- `gethostbyname()` translates host name to IP address

Naming and Addressing

Example DNS name in ASCII string: www.eecs.umich.edu

Its IP address in dotted-decimal (dd) ASCII string: 141.212.113.110

Its IP address in 32-bit binary representation:
10001101 11010100 01110001 01101110

Why do we need names instead of using the addresses directly?

Why do we need addresses in addition to names?

Name and Address Manipulation

Syscalls to map name to/from address:
- dns to binary: `gethostbyname()`
- binary to dns: `gethostbyaddress()`

and to change representation:
- dd to binary: `inet_aton()`
- binary to dd: `inet_ntoa()`

dns to dd: `gethostbyname()` plus `inet_ntoa()`

`gethostbyname()` and `gethostbyaddr()` both return `struct hostent` that contains both binary & dd (See Fig. 11.2 of UNP)

Other useful syscalls:
- `gethostname()`: returns DNS name of current host
- `getsockname()`: returns IP address bound to socket (in binary)

Used when address and/or port is not specified (`INADDR_ANY`), to find out the actual address and/or port used
- `getpeername()`: returns IP address of peer (in binary)

Flat vs. Hierarchical Space

Example of flat name space:
- file system that doesn't support folders/sub-directories

Examples of hierarchical name space:
- Duncan McLeod, William Wallace

Examples of hierarchical address space:
- 5 Wilberforce Rd., Cambridge, Cambridgeshire, England, UK
- Japan, Tokyo-to, Minato-ku, Shirokanedai 4-chome 6-41
- +1 734 763 1583

Why form hierarchy?
- John Doe
- John Smith
- John Keynes
- John Woo

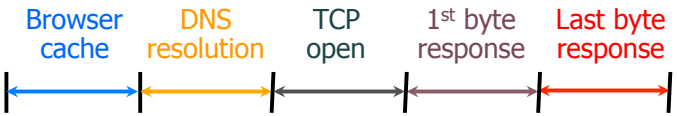
Advantage of hierarchical space: allows for decentralized management

Common Mistakes + Hints

Common mistakes:
- C programming
  - Use gdb
  - Use printf for debugging, remember to do `fflush(stdout);`
- Byte-ordering
- Use of `select()`
- Separating records in TCP stream
- Not knowing what exactly gets transmitted on the wire
  - Use tcpdump / Ethereal /wireshark

Hints:
- Use man pages (available on the web too)
- Check out WWW, programming books

Example: Many Steps in Web Download



Sources of variability of delay

- Browser cache hit/miss, need for cache revalidation
- DNS cache hit/miss, multiple DNS servers, errors
- Packet loss, high RTT, server accept queue
- RTT, busy server, CPU overhead (e.g., CGI script)
- Response size, receive buffer size, congestion
- … downloading embedded image(s) on the page

Domain Name System (DNS)

DNS consists of:

an hierarchical name space: name allocation decentralized to domains

```
host.sub-subdomain.. . ..subdomain.domain[.ROOT]
```

`host`: machine name, can be an alias
`sub-subdomain`: department (engin, eecs, physics, math)
`subdomain`: institution, company, geography, provider (umich, mi, comcast)
`domain`: most significant segment (edu, com, org, net, gov, mil, us, it)

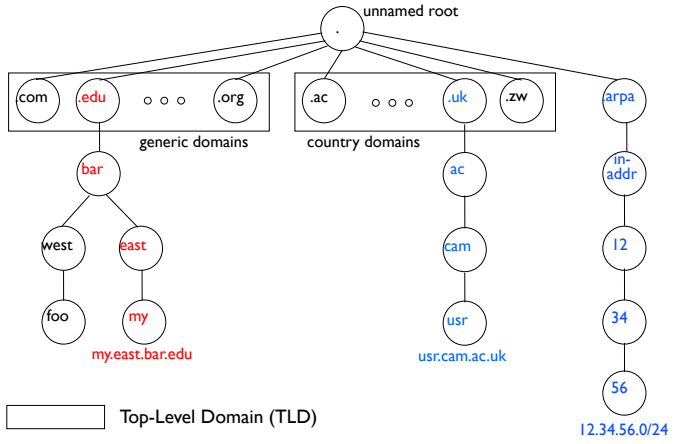
Examples of Fully Qualified Domain Names (FQDNs):
`www.eecs.umich.edu`, `www.cl.cam.ac.uk`, `mlab.t.u-tokyo.ac.jp`

an hierarchical name resolution infrastructure:

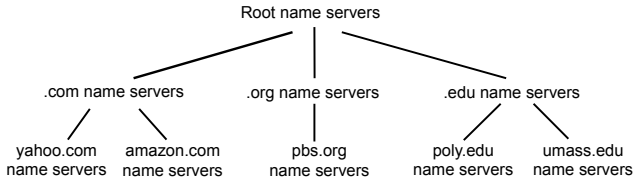
- a distributed database storing resource records (RRs)
- client-server query-reply

Berkeley Internet Name Domain (BIND): the most common implementation of the DNS name resolution architecture

DNS Hierarchical Name Space



Distributed Hierarchical Database (1st Approx)



Client wants IP for `www.amazon.com`:

- Client queries a root server to find `.com` name server
- Client queries `.com` name server to get `amazon.com` name server
- Client queries `amazon.com` name server to get IP address for `www.amazon.com`

BIND Terminology and DNS Name Servers

DNS database is partitioned into *zones*

A *zone* holds one or more domains, analogy:

DNS	File System
domains	folders
zones	volumes

Name server: a process managing a zone

Authoritative or *primary* name server: the "owner" of a zone

- providing authoritative mappings for organization's server names (e.g., Web and mail)
  - can be maintained by an organization or its service provider

Zones may be replicated (Why replicate a zone?)

Secondary servers: replicas

Zone transfer: downloading a zone from the primary server to the replicas

A name server can be the primary server for one or more zones, and the secondary server for one or more zones

DNS Resource Record

DNS: distributed database 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 alias name for some "cannonical" (the real) name
  for example: `www.ibm.com` is really `servereast.backup2.ibm.com`
- **`value`** is cannonical name

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

Try:
```
% dig smtp.eecs.umich.edu MX
```

DNS lookup returns only entries matching type:

Hence when web browswer couldn't find an **A**ddress entry, mail may still find a **M**ail e**X**change entry

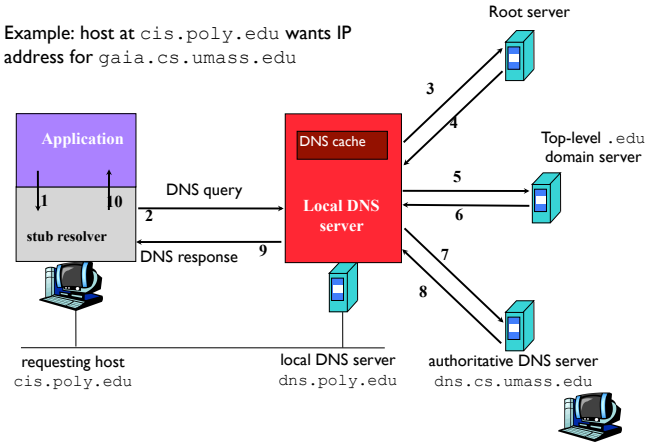
Adding Records to DNS

- Example: just created startup "Network Utopia"
- Register name `networkuptopia.com` at a registrar (e.g., Network Solutions)
  - provide registrar with names and IP addresses of your authoritative name servers (primary and secondary)
  - registrar inserts two RRs into the `.com` top-level domain (TLD) server:
    `(networkutopia.com, dns1.networkutopia.com, NS)`
    `(dns1.networkutopia.com, 212.212.212.1, A)`
    - TLD name servers are responsible for `.com`, `.org`, `.net`, `.edu`, etc, and all top-level country domains `.uk`, `.fr`, `.cn`, `.jp`
      - Network Solutions maintains servers for `.com` TLD
- Add authoritative server Type A record for `www.networkuptopia.com` and Type MX record for `networkutopia.com`

How do people get the IP address of your Web site?

DNS Name Resolution

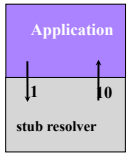
Example: host at `cis.poly.edu` wants IP address for `gaia.cs.umass.edu`



DNS Name Resolution: Client Side

Client:

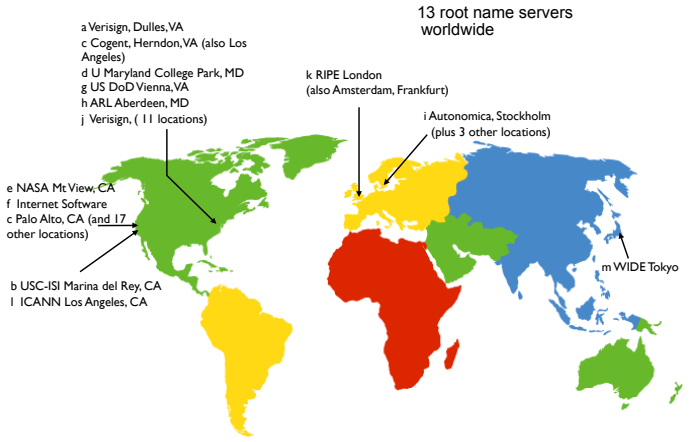
- has stub resolver linked in
- consults `/etc/resolv.conf` to find local name server
- forms FQDN
- queries up to 3 local name servers in turn
- if no response, double timeout and retry for 4 rounds



Local name server:

- when a host makes a DNS query, query is sent to its local name server
- each ISP (residential ISP, company, university) has one
  - also called "default name server"
- acts as a proxy, forwards query into hierarchy
- parses FQDN from right to left
  - ➡ always goes to ROOT first
- consults `/etc/named.conf`, `named.root`, and `zonefile` to find root name servers
- caches resolved name

DNS Root Name Servers



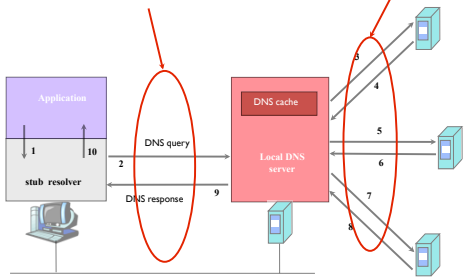
Recursive vs. Iterative Query

Recursive query:

- local name server must resolve the name (or return "not found"), if necessary asking other name servers for resolution
- puts burden of name resolution on contacted name server

Iterative query:

- contacted server replies with the name of server address of sub-domain
- "I don't know this name, but ask this other name server"
- requesting name server visits each name server referred to



Why not always do recursive resolution?

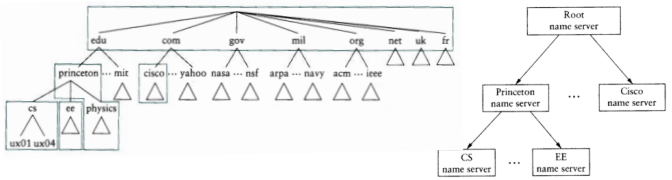
DNS Caching

- Once a (any) name server learns mapping, it caches mapping
  - to reduce latency in DNS translation
- Cache entries timeout (disappear) after some time (TTL)
  - TTL assigned by the authoritative server responsible for the host name
- Local name servers typically also cache
  - TLD name servers to reduce visits to root name servers
  - all other name server referrals
  - both positive and negative results

DNS Name Resolution Exercises

Show the DNS resolution paths, assuming the DNS hierarchy shown in the figures and assuming caching:

- `thumper.cisco.com` looks up `bas.cs.princeton.edu`
- `thumper.cisco.com` looks up `opt.cs.princeton.edu`
- `thumper.cisco.com` looks up `cat.ee.princeton.edu`
- `thumper.cisco.com` looks up `ket.physics.princeton.edu`
- `bas.cs.princeton.edu` looks up `dog.ee.princeton.edu`
- `opt.cs.princeton.edu` looks up `cat.ee.princeton.edu`



Peterson & Davie 2nd. ed., pp. 627, 628

DNS Design Points

DNS serves a core Internet function

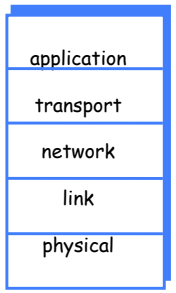
At which protocol layer does the DNS operate?

- host, routers, and name servers communicate to **resolve** names (name to address translation)
- complexity at network's "edge"

Why not centralize DNS?

- single point of failure
- traffic volume
- performance: distant centralized database
- maintenance
- ➡doesn't scale!

DNS is "exploited" for server load balancing, how?

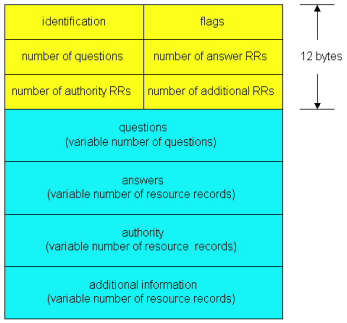


DNS protocol, messages

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



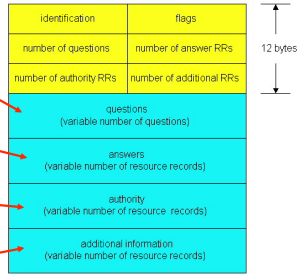
DNS protocol, messages

Name, type fields for a query

RRs in reponse to query

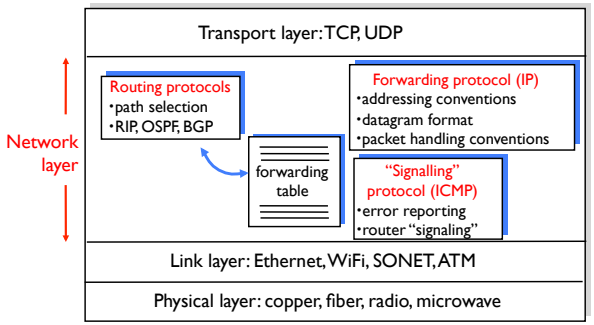
records for authoritative servers

additional "helpful" info that may be used



The Internet Network Layer

Host, router network layer functions:



Packet and Packet Header

Previously . . . the Internet is a **packet switched network**:
- data is parceled into packets
- each packet carries a destination address
- each packet is routed independently
- packets can arrive out of order
- packets may not arrive at all

Just as with the postal system, the "content" you want to send must be put into an envelope and the envelope must be addressed

The "envelope" in this case is the **packet header**

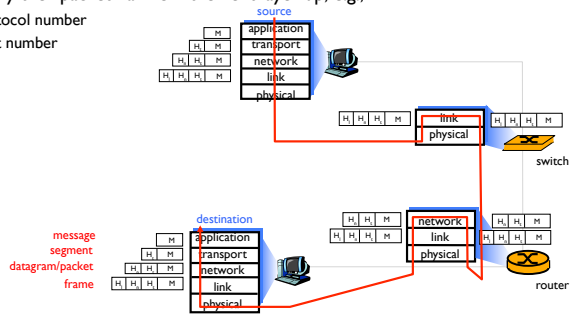
Recall: **protocols** are rules ("syntax" and "grammar") governing communication between nodes

The **format** of a packet header is part of a protocol

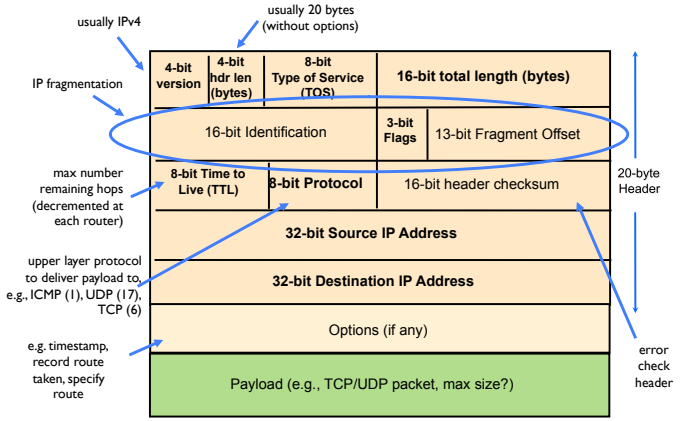
For packet forwarding on the Internet, the protocol is the **Internet Protocol (IP)**

Encapsulation

Each protocol has its own "envelope"
- each protocol attaches its header to the packet
- so we have a protocol wrapped inside another protocol
- each layer of header contains a protocol demultiplexing field to identify the "packet handler" the next layer up, e.g.,
  - protocol number
  - port number



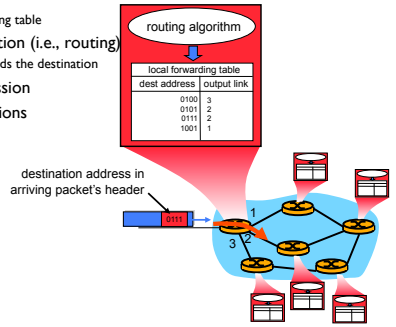
IPv4 Packet Header Format



Packet Forwarding

Goal: deliver packets through routers from source to destination

- source node puts destination address in packet header
- each router node on the Internet:
  - looks up destination address in its routing table
    - we'll study several path selection (i.e., routing)
  - sends the packet to the next hop towards the destination
    - routes may change during session
    - analogy: driving, asking directions

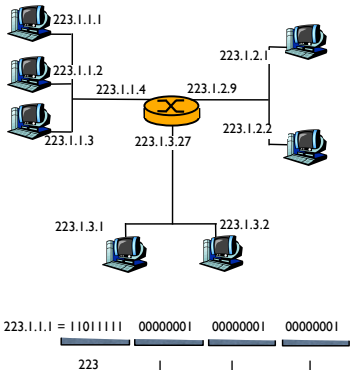


IP Addressing: Introduction

IP address: 32-bit identifier for host/router **interface**

interface: connection between host/router and physical link

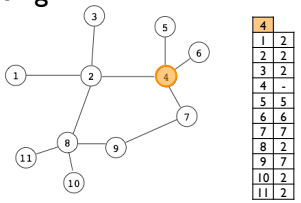
- routers typically have multiple interfaces
- host may have multiple interfaces
- IP address associated with each interface



Flat vs. Hierarchical Addressing

Flat addressing:

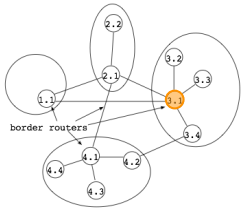
- each router needs 10 entries in its routing table



4	
1	2
2	2
3	2
4	-
5	5
6	6
7	7
8	2
9	7
10	2
11	2

Hierarchical addressing:

- hosts only need to know the default router, usually its border router
- each border router keeps in its routing table:
  - next hop to other networks
  - all hosts within its own network



3.1	
2.*	2.1
1.*	1.1
4.*	2.1
3.2	3.2
3.3	3.3
3.4	3.4

note that for routing table, we store the next hop address instead of the interface number

IPv4 Addressing

Independent of physical hardware address

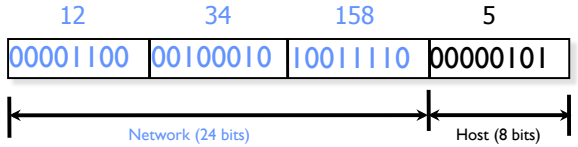
32-bit number represented as dotted decimal:

- for ease of reference
- each # is the decimal representation of an octet

Divided into two parts:

- network prefix, globally assigned
  - route to network first
- host ID, assigned locally

Example: 12.34.158.0/24 is a 24-bit network prefix with 2^8 host addresses

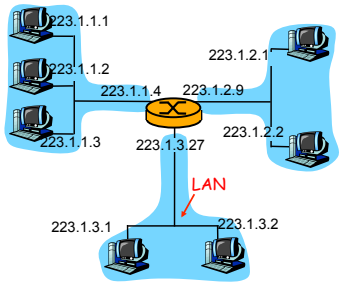


Subnets

A network can be further divided into subnets

What's a subnet ?

- device interfaces with same subnet part of IP address
- can physically reach each other without intervening router



a network consisting of 3 subnets

Classfull Addresses

For the example network prefix: 12.34.158.0/24

- how many hosts can the network have?

What is a good partition of the 32-bit address space between the network and host parts?

Historically . . . classfull addresses:

Class A: 0*, very large /8 blocks (e.g., MIT has 18.0.0.0/8)
Class B: 10*, large /16 blocks (e.g,. UM has 141.213.0.0/16)
Class C: 110*, small /24 blocks (e.g., AT&T Labs has 192.20.225.0/24)
Class D: 1110*, multicast groups
Class E: 11110*, reserved for future use

Problems:

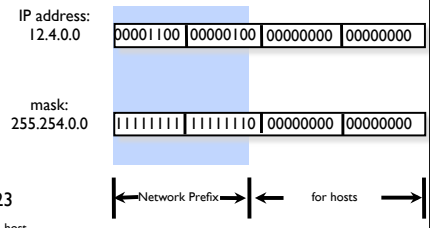
1. the Goldilock problem: everybody wanted a Class B
2. address space usage became inefficient
3. routing table explosion
4. and then, address space became scarce…
   - by 1992, half of Class B has been allocated, would have been exhausted by 3/94

Classless InterDomain Routing (CIDR)

Network portion of address is of arbitrary length, determined by a prefix mask

Uses two 32-bit numbers to represent a network address
network number = IP address + mask

Usually written as a.b.c.d/x, where x is number of bits in the network portion of address: 12.4.0.0/15



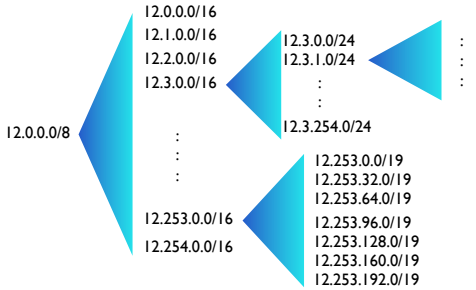
Another example:
200.23.16.0/23

11001000 00010111 0001000**0** 00000000

CIDR: Hierarchical Address Allocation

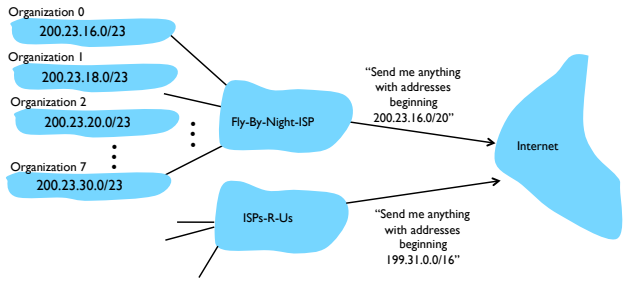
Prefixes are key to Internet routing scalability

- address allocation by ICANN, ARIN/RIPE/APNIC and by ISPs
- routing protocols and packet forwarding based on prefixes
- today, routing tables contain ~150,000-200,000 prefixes



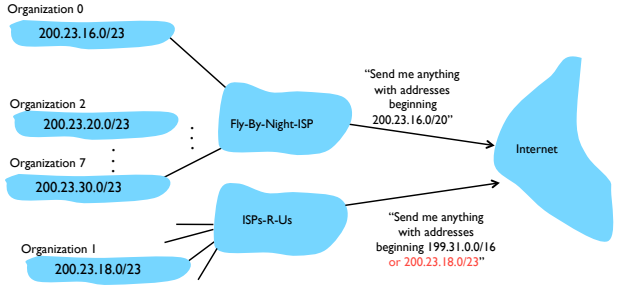
CIDR: Route Aggregation

Hierarchical addressing allows efficient advertisement of routing information:



Longest Prefix Match: More specific routes

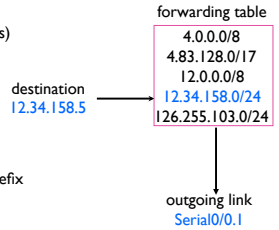
ISPs-R-Us has a more specific route to Organization 1



How are Packets Forwarded?

Routers have forwarding tables
- maps each IP prefix to next-hop link(s)
- entries can be statically configured
  - e.g., "map 12.34.158.0/24 to Serial0/0.1"

Destination-based forwarding
- Packet has a destination address
- Router identifies longest-matching prefix



But, this doesn't adapt
- to failures
- to new equipment
- to the need to balance load
- …

That is where routing protocols come in… [more on this in the next lectures]

Special IPv4 Addresses

- network identification:
  - 0s on host part, e.g. ,`141.212.0.0` (cannot be used to send packets)
- directed broadcast:
  - `0xffff` on host part, e.g., `141.212.255.255`
  - Broadcast to all hosts on network (`141.212`) (Not implemented?)
- limited broadcast:
  - `0xffffffff`, received by all hosts on LAN, not forwarded beyond LAN
- this computer:
  - `0.0.0.0` to be used at startup to ask for one's own IP address (RARP, deprecated)
- loopback address:
  - `127.*.*.*` (usually `127.0.0.1`), named `localhost`
  - pkts sent to `localhost` traverse down the kernel networking code & back up to application without traversing the network, useful for testing networking code