“Real” Internet delays and routes

What do “real” Internet delay & loss look like?

**Traceroute program**: provides delay measurement from source to router along end-end Internet path towards destination. For all i:

- sends three packets that will reach router i on path towards destination
- router i will return packets to sender
- sender times interval between transmission and reply.

---

Traceroute: Measuring the Forwarding Path

**Time-To-Live field in IP packet header:**
- Source sends a packet with a TTL of n
- Each router along the path decrements the TTL
- “TTL exceeded” sent when TTL reaches 0

**Traceroute tool exploits this TTL behavior**

---

Packet loss

queue (aka buffer) preceding link in buffer has finite capacity when packet arrives to full queue, packet is dropped (aka lost) lost packet may be retransmitted by previous node, by source end system, or not retransmitted at all
IP Packet Structure

Layering in the IP Protocols

Application-Layer Protocols

Sockets
Types of Sockets

Different types of sockets implement different service models

• Stream vs. 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.

Stream/TCP Sockets

socket ()
bind ()
listen ()
accept ()
send ()
recv ()
close ()

initialize
establish
data xfer
terminate

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
• connects to 128.125.1.45 at the well-known SMTP port # (25)
• parcels out your email into packets
• sends the packets out

At the receiver, smtpd
• must make a “receiver” ahead of time:
  • creates a socket
• decides on TCP
• binds the socket to smtp's well-known port #
• listens on the socket
• accepts your smtp connection requests
• receives your email packets

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
• connects to 128.125.1.45 at the well-known SMTP port # (25)
• parcels out your email into packets
• sends 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
• binds the socket to smtp's well-known port #
• listens on the socket
• accepts your smtp connection requests
• receives your email packets
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 a 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)

```c
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)

```c
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)

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

```c
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)

```c
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)

```c
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)

```c
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 listened to socket)

API design question: why not merge listen() and accept()?!
Socket Connection Queues

Receiving Data Stream (Server)

```c
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?

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:
- 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()`
- non-blocking I/O:
  - only checks for device availability
  - by polling or signal driven (not covered)
- asynchronous I/O:
  - process is notified when I/O is completed (not covered)
Non-Blocking I/O: Polling

```c
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;
    }
}
```

Blocking I/O: polling

```c
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;
    }
}
```

Blocking I/O: select()

```c
select(maxfd, readset, writset, 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: polling

```
which of the following would you use? why?
```

```c
loop {
    select(..., timeout);
    recv();
}  till done;
```

or:

```c
loop {
    sleep(seconds)
    recv();
}  till done;
```
### Byte Ordering

```c
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();
}
```

Little-endian: (Intel x86 and Alpha)

<table>
<thead>
<tr>
<th>Actual Value 1</th>
<th>LSB</th>
<th>MSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000001</td>
<td>01000010</td>
<td>00000010</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Actual Value 1</th>
<th>LSB</th>
<th>MSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000010</td>
<td>01000010</td>
<td>00000001</td>
</tr>
</tbody>
</table>

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)

```c
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();
}
```

### 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: `gethostbyaddr()`

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 `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

Browser cache → DNS resolution → TCP open → 1st byte response → Last byte response

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
- an hierarchical name resolution infrastructure:
  - a distributed database storing resource records (RRs)
  - client-server query-reply

DNS consists of:
- 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

<table>
<thead>
<tr>
<th>Generic Domains</th>
<th>Country Domains</th>
<th>Root Name Servers</th>
<th>.com Name Servers</th>
<th>.edu Name Servers</th>
<th>.org Name Servers</th>
</tr>
</thead>
<tbody>
<tr>
<td>.ac</td>
<td>.uk</td>
<td>.zw</td>
<td>.com</td>
<td>.edu</td>
<td>.org</td>
</tr>
<tr>
<td>Generic Domains</td>
<td>Country Domains</td>
<td>Root Name Servers</td>
<td>.com Name Servers</td>
<td>.edu Name Servers</td>
<td>.org Name Servers</td>
</tr>
<tr>
<td>.ac</td>
<td>.uk</td>
<td>.zw</td>
<td>.com</td>
<td>.edu</td>
<td>.org</td>
</tr>
<tr>
<td>Generic Domains</td>
<td>Country Domains</td>
<td>Root Name Servers</td>
<td>.com Name Servers</td>
<td>.edu Name Servers</td>
<td>.org Name Servers</td>
</tr>
<tr>
<td>.ac</td>
<td>.uk</td>
<td>.zw</td>
<td>.com</td>
<td>.edu</td>
<td>.org</td>
</tr>
</tbody>
</table>

Distributed Hierarchical Database (1st Approx)

Root name servers
- .com name servers
- .org name servers
- .edu name servers

Yahoo.com name servers
- Amazon.com name servers
- Pbs.org name servers
- Poly.edu name servers
- Umass.edu name servers

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

DNS Terminology and DNS Name Servers

DNS database is partitioned into zones
A zone holds one or more domains, analogy:

DNS domains zones
<table>
<thead>
<tr>
<th>File System folders volumes</th>
</tr>
</thead>
</table>
| 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)
| 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**: name is hostname
- **Value**: value is IP address

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

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

**Type**: name is alias name for some "canonical" (the real) name

- **Value**: value is canonical name

**Type**: name is none of nameserver associated with name

**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: (networkuptopia.com, dns1.networkuptopia.com, NS) (dns1.networkuptopia.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 networkuptopia.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

**Client Side**

- 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
- performs 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
The diagram illustrates the 13 root name servers worldwide, distributed across various locations:

- **Verisign**, Dulles, VA (and 11 other locations)
- **Cogent**, Herndon, VA (also Los Angeles)
- **NASA**, La Verne, CA (Internet Software Palo Alto, CA and 17 other locations)
- **U Maryland College Park**, MD
- **US DoD Vienna**, VA
- **ARL Aberdeen**, MD
- **Autonomica**, Stockholm (also Amsterdam, Frankfurt)
- **RIPE London** (also Amsterdam, Frankfurt)
- **WIDE Tokyo**
- **US DoD Vienna**, VA
- **ARL Aberdeen**, MD
- **Verisign**, (11 locations)