University of Houston

Introduction to Computer Networks

COSC 6377

Midterm Review

Author
K.M. Hourani

Based on Notes By
Dr. Omprakash Gnawali

March 3, 2021

Contents

1 Intro **5**
- 1.1 The Internet . . . 5
- 1.2 Packet vs. Circuit Switching . . . 5
  - 1.2.1 Circuit Switching . . . 5
  - 1.2.2 Some Circuit Switching Techniques . . . 6
  - 1.2.3 Packet Switching . . . 6
  - 1.2.4 Summary . . . 6
- 1.3 Describing a Network . . . 6
  - 1.3.1 Throughput . . . 7
  - 1.3.2 Latency . . . 7
    - 1.3.2.1 Relation between Latency and Throughput . . . 7
  - 1.3.3 Reliability . . . 7
- 1.4 Protocols . . . 7
- 1.5 Network Protocols . . . 8
  - 1.5.1 Protocols and Standards . . . 8
    - 1.5.1.1 Protocol Layers . . . 8
- 1.6 Encapsulation . . . 8

2 Network Applications and Socket Programming **11**
- 2.1 Network Applications . . . 11
  - 2.1.1 Inter-Application Communication . . . 11
  - 2.1.2 Application Protocols . . . 11
  - 2.1.3 Network Time Service . . . 12
    - 2.1.3.1 Protocol Timing Diagram . . . 12
    - 2.1.3.2 Cloud-based File Backup Application . . . 12
- 2.2 Socket Programming . . . 13
  - 2.2.1 Using TCP/IP . . . 13
  - 2.2.2 System Calls . . . 13
  - 2.2.3 File Descriptors . . . 13
  - 2.2.4 Error Returns . . . 13
  - 2.2.5 Some operations on File Descriptors . . . 14
  - 2.2.6 Sockets: Communincation Between Machines . . . 14
  - 2.2.7 System calls for using TCP . . . 14
  - 2.2.8 Socket Naming . . . 14
  - 2.2.9 Socket Address Structures . . . 15
  - 2.2.10 Dealing with Address Types . . . 15
  - 2.2.11 Client Skeleton (IPv4) . . . 15
  - 2.2.12 Server Skeleton (IPv4) . . . 15
  - 2.2.13 Looking up socket address with `getaddrinfo` . . . 16
  - 2.2.14 `getaddrinfo()`[RFC3493] . . . 16
  - 2.2.15 EOF in more detail . . . 16
  - 2.2.16 Using UDP . . . 17
  - 2.2.17 Serving Multiple Clients . . . 17

2.2.18 Threads . . . 17
2.2.19 Non-blocking I/O . . . 17
2.2.20 How do you know when to read/write? . . . 18
2.2.21 Event-driven servers . . . 18

3 HTTP and the Web **19**
3.1 Precursors . . . 19
3.1.1 Tim Berners-Lee . . . 19
3.1.2 Components . . . 19
3.1.3 Ingredients . . . 20
3.1.4 URLs . . . 20
3.1.5 Examples of URLs . . . 20
3.2 HTTP . . . 20
3.2.1 Steps in HTTP Request . . . 20
3.2.1.1 Sample Browser Request . . . 21
3.2.1.2 Sample HTTP Response . . . 21
3.2.2 HTTP is Stateless . . . 22
3.2.3 HTTP Cookies . . . 22
3.2.4 Anatomy of a Web Page . . . 23
3.2.5 AJAX . . . 23
3.2.6 HTTP Performance . . . 23
3.2.6.1 Small Requests . . . 23
3.2.6.2 Larger Objects . . . 24

4 Domain Name System **25**
4.1 Host names and IP Addresses . . . 25
4.1.1 Separating Naming and Addressing . . . 25
4.1.2 Scalable Address ↔ Name Mappings . . . 25
4.1.3 Goals for an Internet-scale name system . . . 26
4.1.3.1 The Good News . . . 26
4.2 Domain Name System (DNS) . . . 26
4.2.1 DNS Architecture . . . 26
4.2.2 Resolver Operation . . . 27
4.2.3 DNS Root Server . . . 27
4.2.4 DNS Root Servers . . . 27
4.2.5 TLD and Authoritative DNS Servers . . . 27
4.2.6 Reverse Mapping . . . 27
4.2.7 DNS Caching . . . 27
4.2.8 Negative Caching . . . 28
4.2.9 DNS Protocol . . . 28
4.2.10 Resource Records . . . 28

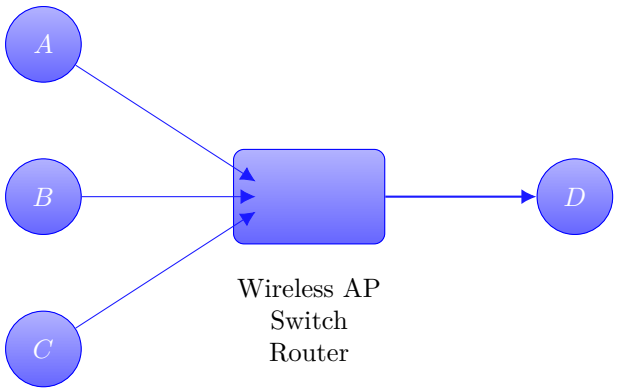
Chapter 1

Intro

1.1 The Internet

- Collection of nodes, wired and wireless technology connecting these nodes, applications and services
- Types of nodes
  - Desktops and Laptop
  - Servers
  - TV/Refrigerator
  - Cellphones
- Goal: Connect all the nodes to each other
- Solutions
  - $\binom{n}{2} = \mathcal{O}(n^2)$ cables
  - Sharing the links
    - Circuit Switching
    - Packet Switching
- Packet
  - Collection of bits to transfer across a network
  - Think: envelope and its contents
- Circuit
  - Pre-allocated path/resource

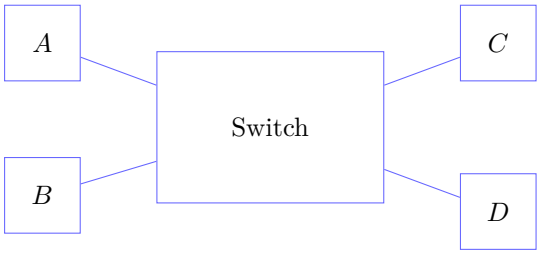
1.2 Packet vs. Circuit Switching



1.2.1 Circuit Switching

- Setup the connection or resource

  - Schedule (e.g., TDMA)
  - State in the network



Time	Circuit
T, $3T$, $5T$, ...	$A-D$
$2T$, $4T$, $6T$, ...	$B-C$

- Natural for predictable data races
- Can guarantee certain level of services
- Can be inefficient for many applications

1.2.2 Some Circuit Switching Techniques

- Time
  - Reserve to use the link at a given schedule
  - Read: https://en.wikipedia.org/wiki/Time-division_multiplexing
- Frequency
  - Reserve to use certain frequencies (channel)
  - Read: https://en.wikipedia.org/wiki/Frequency-division_multiplexing

1.2.3 Packet Switching

- Wire is selected for each packet
- No network **state**
- Supports unpredictable/bursty traffic pattern
- Higher link utilization
- No guarantees but good enough for most applications

https://en.wikipedia.org/wiki/Packet_switching

1.2.4 Summary

- Packet Switching
  - Plus: more sharing (more efficient)
  - Minus: no service guarantee
- Circuit Switching
  - Plus: service guarantee
  - Minus: less sharing (less efficient)
- Every day examples
  - Road network

1.3 Describing a Network

- How to describe how well a network is working?
  - Metrics
- Performance metrics
  - Throughput
  - Latency
  - Reliability

1.3.1 Throughput

- How many bytes can we send through in a given time?
  - Bytes per second
  - How many bits/s in kbps?
  - Read: https://en.wikipedia.org/wiki/Data-rate_units
- Useful bytes transferred vs. overhead
  - Goodput
  - Everyday example: car vs. passenger

https://en.wikipedia.org/wiki/Throughput

1.3.2 Latency

- How long does it take for one bit to travel from one end to the other end?
  - ms, s, minutes, etc.
- Typical latencies
  - Speed of light
  - Why is web browsing latency in seconds?

1.3.2.1 Relation between Latency and Throughput

- Characterize the latency and throughput of
  - Oil Tanker –
  - Aircraft –
  - Car –
  - Tractor Trailer –
- Which metrics matter most for these applications?
  - Netflix
  - Skype
  - Amazon
  - Facebook

1.3.3 Reliability

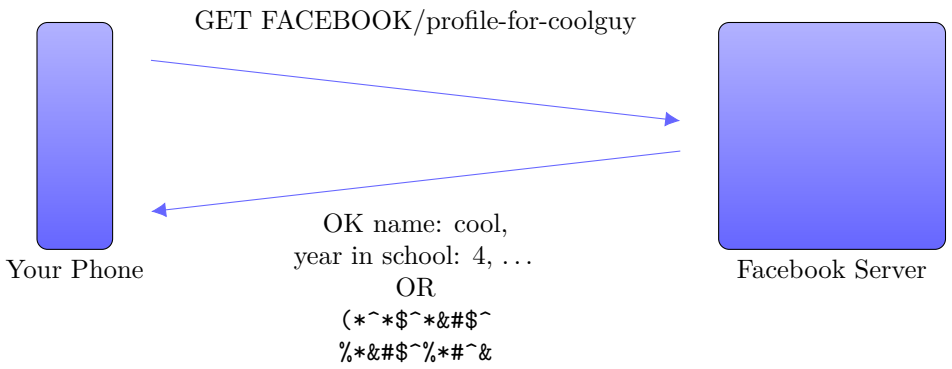
- How often does a network fail?
- How often do packets drop?
  - Damage (corruption)
  - Drops in the queues
- How persistent are failures?
- Typical metrics
  - uptime percentage
  - packet or bit loss rate

1.4 Protocols

- Agreed-upon rules, format, and meaning for message exchange
- Let's examine this sequence:
  - Hellow
  - How are you?
  - Fine.

https://en.wikipedia.org/wiki/Communication_protocol

1.5 Network Protocols



What are the rules, format, and meaning in this message exchange?

1.5.1 Protocols and Standards

- How can your phone (HTC running Android) access Facebook (runs on UNIX-like OS on big servers)?
- Using standard protocol enables interoperation
- Who standardizes the protocols?

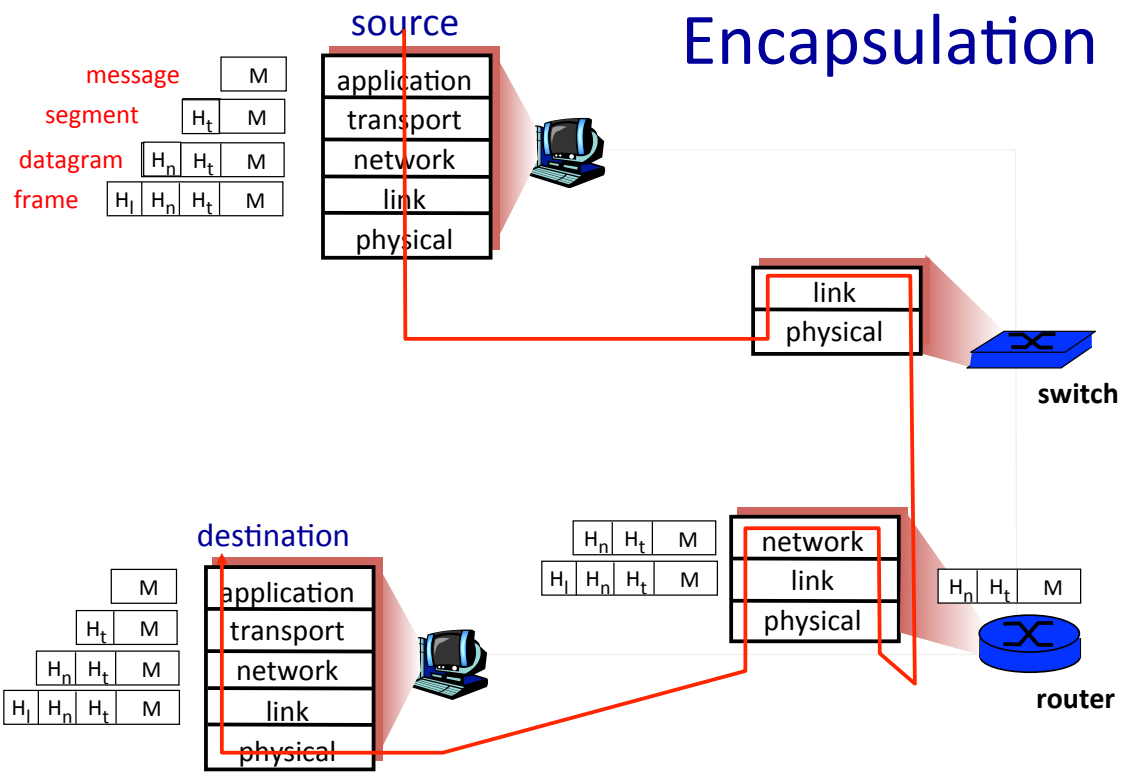
1.5.1.1 Protocol Layers

- Lower level to higher level message exchange
  - Organize the functionalities
  - Abstractions in services used and provided
- 5-7 layers depending on who you talk to
  - Physical, Link, Network, Transport, Application
- Should a smartphone app developer worry about
  - Voltages being applied on the wire
  - If the underlying media uses packet or circuit switching

https://en.wikipedia.org/wiki/Protocol_stack

1.6 Encapsulation

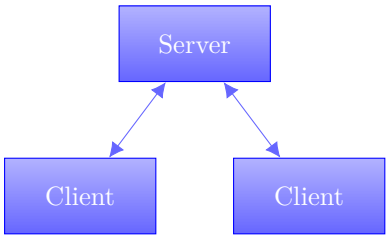
- Think of how paperwork is processed in a university
  - Each person processes and adds some information to it and passes it along
- On the transmitter, the lower layers include the message from upper layers, add their own information, and send it along
- On the receiver: reverse


Encapsulation
source
message
M
segment
H_t M
datagram
H_n H_t M
frame
H_l H_n H_t M
application
transport
network
link
physical
link
physical
switch
destination
M
H_t M
H_n H_t M
H_l H_n H_t M
application
transport
network
link
physical
H_n H_t M
H_l H_n H_t M
network
link
physical
H_n H_t M
router

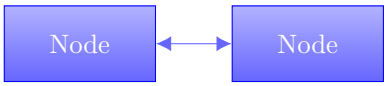
Chapter 2

Network Applications and Socket Programming

2.1 Network Applications



Client-Server



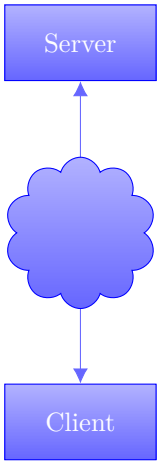
Peer-to-Peer

Hybrid

https://en.wikipedia.org/wiki/Peer-to-peer

2.1.1 Inter-Application Communication

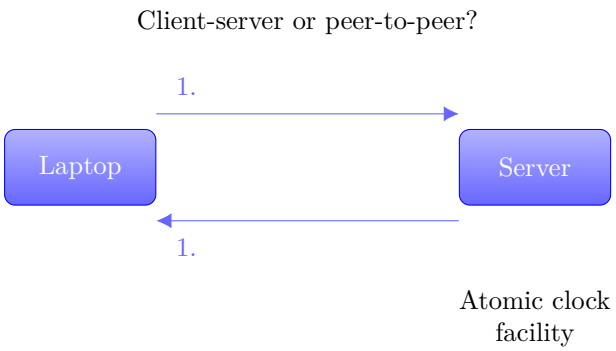
- Need a way to send and receive messages
- Inter-process communication
- Need naming, routing, transport
- Transport using TCP and UDP
  - On top of IP



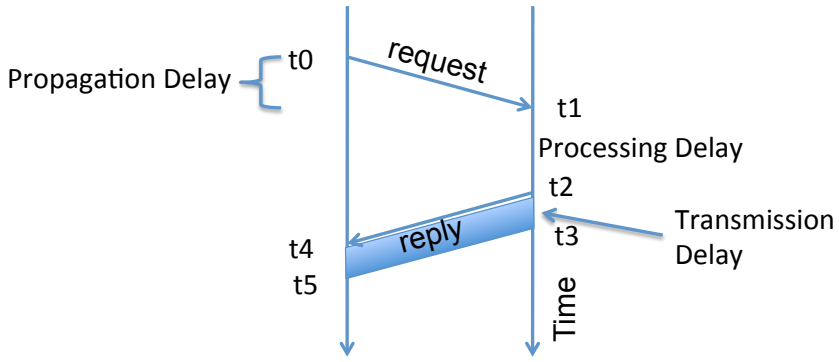
2.1.2 Application Protocols

- Messages between processes, typically encapsulated within TCP or UDP
- Need agreement between
  - Sending process
  - Receiving process

2.1.3 Network Time Service

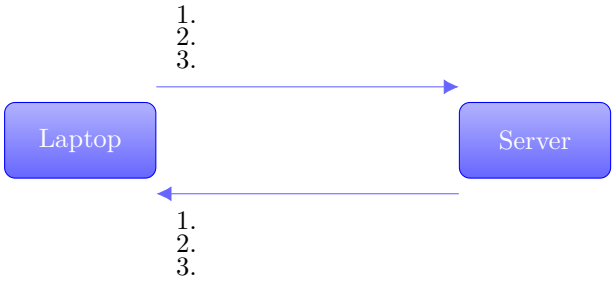


2.1.3.1 Protocol Timing Diagram



2.1.3.2 Cloud-based File Backup Application

- Client-server or peer-to-peer?
- Where do the applications run?
- Who/how to run these applications?
- What messages are exchanged?

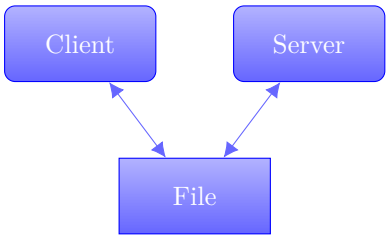


2.2 Socket Programming

2.2.1 Using TCP/IP

- How can applications use the network?
- *Sockets* API
  - Originall from BS, widely implemented (*BSD, Linux, Mac OS X, Windows, . . .)
  - Higher-level APIs build on them
- After basic setup, much like files

One could test network protocols with read/write on a file



2.2.2 System Calls

- Problem: how to access resources other than the CPU
  - Disk, netowrk, terminal, other processes
  - CPU prohibits instructions that would access devices
  - Only privileged OS kernel can access devices
- Kernel supplies well-defined system call interface
  - Applications request I/O oeprations through syscalls
  - Set up syscall arguments and trap to kernel
  - Kernel performs operation and returns result
- Higher-level functions built on syscall interface
  - `printf`, `scanf`, `gets`, all user-level code

2.2.3 File Descriptors

- Most I/O in Unix done through *file descriptors*
  - Integer *handles* to per-process table in kernel
- `int open(char *path, int flags, ...);`
- Returns file descriptor, used for all I/O to file

https://en.wikipedia.org/wiki/File_descriptor

2.2.4 Error Returns

- What if `open` fails? Return `-1` (invalid file descriptor)
- Most system calls return `-1` on failure
  - Specific type of error in gobal `int errno`

- `#include <sys/errno.h>` for possible values
  - `2` = `ENOENT` "no such file or directory"
  - `13` = `EACCES` "permission denied"

2.2.5 Some operations on File Descriptors

- `ssize_t read(int fd, void* buf, int nbytes);`
  - Returns number of bytes read
  - Returns `0` bytes at end of file, or `-1` on error
- `ssize_t write(int fd, void* buf, int nbytes);`
  - Returns number of bytes written, `-1` on error
- `off_t lseek(int fd, off_t offset, int whences);`
  - `whence`: `SEEK_SET`, `SEEK_CUR`, `SEEK_END`
  - returns new offset, or `-1` on error
- `int close(int fd);`

2.2.6 Sockets: Communincation Between Machines

- Network sockets are file descriptors too
- Datagram sockets: unreliable message delivery
  - With IP, gives you UDP
  - Send atomic messages, which may be reordered or lost
  - Special system calls to read/write: `send`/`recv`
- Stream sockets: bi-directional pipes
  - With IP, gives you TCP
  - Bytes written on one end read on another
  - Reads may not return full amount requested, must reread

2.2.7 System calls for using TCP

	Client	Server
1.		`socket` – make socket
2.		`bind` – assign address, port
3.		`listen` – listen for clients
4.	`socket` – make socket	
5.	`bind` – assign address[1]	
6.	`connect` – connect to listening socket	
7.		`accept` – accept connection

2.2.8 Socket Naming

- Naming of TCP and UDP communication endpoints
  - IP address specifies host (129.7.240.18)
  - 16-bit port number demultiplexes within host
  - Well-known services listen on standard ports (e.g. ssh – 22, http – 8, see `/etc/services` for list)
  - Clients connect from arbitrary ports to well-known ports
- A connection is named by 5 components
  - Protocol, local IP, local port, remote IP, remote port
  - TCP requires connected sockets, but not UDP

[1] This call to bind is optional, connect can choose address and port

2.2.9 Socket Address Structures

- Socket interface supports multiple network types
- Most calls take a generic `sockaddr`:

```
struct sockaddr {
    uint16_t sa_family;   /* address family */
    char     sa_data[14]; /* protocol-specific addr */
};
```

- e.g. `int connect(int s, struct sockaddr* srv, socklen_t addrlen);`
- Cast `sockaddr*` from protocol-specific struct, e.g.

```
struct addr_in {
    short   sin_family;       /* = AF_INET */
    u_short sin_port;         /* = htons (PORT) */
    struct  in_addr sin_addr; /*32-bit IPV4 addr */
    char    in_zero[8];
};
```

2.2.10 Dealing with Address Types

- All values in network byte order (Big Endian)
  - `htonl()`, `htons()`: host to network, 32 and 16 bits
  - `ntohl()`, `ntohs()`: network to host, 32 and 16 bits
  - Remember to always convert!
- All address types begin with family
  - `sa_family` in `sockaddr` tells you the actual type
- Not all addresses are the same size
  - e.g. `struct sockaddr_in6` is typically 28 bytes, yet generic `struct sockaddr` is only 16 bytes
  - so most calls require passing around socket length
  - new `sockaddr_storage` is big enough

2.2.11 Client Skeleton (IPv4)

```
struct sockaddr_in {
    short   sin_family; /* = AF_INET */
    u_short sin_port;   /* = htons (PORT) */
    struct  in_addr sin_addr;
    char    sin_zero[8];
} sin;

int s = socket (AF_INET, SOCK_STREAM, 0);
memset(&sin, sizeof(sin), 0);
sin.sin_family = AF_INET;
sin.sin_port = htons(13); /* daytime port */
sin.sin_addr.s_addr = htonl(IP_ADDRESS);
connect(s, (sockaddr*)&sin, sizeof(sin));
while ((n = read(s, buf, sizeof(buf))) > 0) {
    write(1, buf, n);
}
```

2.2.12 Server Skeleton (IPv4)

```
int s = socket(AF_INET, SOCK_STREAM, 0);
struct sockaddr_in sin;
```

```
    memset(&sin, sizeof(sin), 0);
    sin.sin_family = AF_INET;
    sin.sin_port = htons(9999);
    sin.sin_addr.s_addr = htonl(INADDR_ANY);
    bind(s, (struct sockaddr*)&sin, sizeof(sin));
    listen(s, 5);
    while (true) {
        socklen_t len = sizeof (sin);
        int cfd = accept(s, (struct sockaddr*)&sin, &len);
        /* cfd is new connection; you never read/write s */
        do_something_with(cfd);
        close(cfd);
    }
```

2.2.13 Looking up socket address with `getaddrinfo`

```
struct addrinfo hints, *ai;
int err;
memset(&hints, 0, sizeof(hints));
hints.ai_family = AF_UNSPEC;     /* or AF_INET or AF_INET6 */
hints.ai_socktype = SOCK_STREAM; /* or SOCK_DGRAM for UDP */

err = getaddrinfo("www.brown.edu", "http", &hints, &ai);
if (err) {
    fprintf (stderr, "%s\n", gia_strerror (err));
} else {
    /* ai->ai_family = address type (AF_INET or AF_INET6) */
    /* ai->ai_addr = actual address cast to (sockaddr *) */
    /* ai->ai_addrlen = length of actual address */
    freeaddrinfo (ai); /* must free when done! */
}
```

2.2.14 `getaddrinfo()`[RFC3493]

- Protocol-independent node name to address translation
  - Can specify port as a service name or number
  - May return multiple addresses
  - You must free the structure with `freeaddrinfo`
- Other useful functions to know about
  - `getnameinfo` – lookup hostname based on address
  - `inet_ntop` – convert IPv4 or 6 address to printable
  - `inet_prton` – convert string to IPv4 or 6 address

2.2.15 EOF in more detail

- What happens at the end of store?
  - Server receives EOF, renames file, responds OK
  - Client reads OK, *after* sending EOF: didn't close `fd`
- `int shutdown(int fd, int how);`
  - Shuts down a socket without closing the file descriptor
  - how: `0` = read, `1` = write, `2` = both
  - Note 1: applies to *socket*, not descriptor, so copies of descriptor (through fork or dup) affected
  - Note 2: with TCP, can't detect if other side shuts down for reading

2.2.16 Using UDP

- Call socket with `SOCK_DGRAM`, bind as before
- New calls for sending/receiving individual packets
  - `sendto(int s, const void* msg, int len, int flags, const struct sockaddr* to, socklen_t tolen);`
  - `recvfrom(int s, void* buf, int len, int flags, struct sockaddr *from, socklen_t* fromlen);`
  - Must send/get peer address with each packet
- Can use UDP in connected mode (why?)
  - connect assigns remote address
  - `send`/`recv` syscalls, like `sendto`/`recvfrom`, without last two arguments

2.2.17 Serving Multiple Clients

- A server may block when talking to a client
  - Read or write of a socket connected to a slow client can block
  - Server may be busy with CPU
  - Server might be blocked waiting for disk I/O
- Concurrency through multiple processes
  - Accept, fork, close in parent; child services request
- Advantages of one process per client
  - Doesn't block on slow clients
  - May use multiple cores
  - Can keep disk queues full for disk-heavy workloads

2.2.18 Threads

- One process per client has disadvantages:
  - High overhead – fork + exit $\approx 100\mu$sec
  - Hard to share state across clients
  - Maximum number of processes limited
- Can use threads for concurrency
  - Data races and deadlocks make programming tricky
  - Must allocate one stack per request
  - Many thread implementations block on some I/O or have heavy thread-switch overhead

Rough equivalents to `fork()`, `waitpid()`, `exit()`, `kill()`, plus locking primitives.

2.2.19 Non-blocking I/O

- `fcntl` sets `O_NONBLOCK` flag on descriptor

```
int n;
if ((n = fcntl(s, F_GETFL)) >= 0) {
    fcntl(s, F_SETFL, n | O_NONBLOCK);
}
```

- Non-blocking semantics of system calls:
  - read immediately returns `-1` with errno `EAGAIN` if no data
  - write may not write all data, or may return `EAGAIN`
  - connect may fail with `EINPROGRESS` (or may succeed, or may fail with a real error like `ECONNREFUSED`)
  - accept may fail with `EAGAIN` or `EWOULDBLOCK` if no connections present to be accepted

2.2.20 How do you know when to read/write?

```
struct timeval {
    long tv_sec;  /* seconds */
    long tv_usec; /* and microseconds */
};
int select(int nfds, fd_set* readfds, fd_set* writefds,
           fd_set* exceptfds, struct timeval* timeout);
FD_SET(fd, &fdset);
FD_CLR(fd, &fdset);
FD_ISSET(fd, &fdset);
FD_ZERO(&fdset);
```

- Entire program runs in an *event loop*

2.2.21 Event-driven servers

- Quite different from processes/threads
  - Race conditions, deadlocks rare
  - Often more efficient
- But...
  - Unusual programming model
  - Sometimes difficult to avoid blocking
  - Scaling to more CPUs is more complex

Chapter 3

HTTP and the Web

3.1 Precursors

- 1945, Vannevar Bush, Memex:
  - "*a device in which an individual stores all his books, records, and communications, and which is mechanized so that it may be consulted with exceeding speed and flexibility*"
- Precursors to hypertext
  - "*The human mind [...] operates by association. WIth one item in its grasp, it snaps instantly to the next that is suggested by the association of thoughts, in accordance with some intricate web of trails carried by the cells of the brain*"
- Read his 1945 essay, "As we may think"
  - https://www.theatlantic.com/magazine/archive/1945/07/as-we-may-think/303881/

3.1.1 Tim Berners-Lee

- Physicist at CERN, trying to solve real problem
  - Distributed access to data
- WWW: distributed database of pages linked thruogh the Hypertext Transfer Protocol
  - First HTTP implementation: 1990
  - HTTP/0.9 – 1991
    - Simple `GET` command
  - HTTP/1.0 – 1992
    - Client/server information, simple caching
  - HTTP/1.1 – 1996
    - Extensive caching support
    - Host identification
    - Pipelined, persistent connections, ...

3.1.2 Components

- Content
  - Objects (may be static or dynamically generated)
- Clients
  - Send requests / receive responses
- Servers
  - Receive requests / send responses
  - Store or generate content
- Proxies

  - Placed between clients and servers
  - Provide extra functions
    * Caching, anonymization, logging, transcoding, filtering access
  - Explicit or transparent

3.1.3 Ingredients

- HTTP
  - Hypertext Transfer Protocol
- HTML
  - Language for description of content
- Names (mostly URLs)

3.1.4 URLs

`protocol://[name@]hostname[:port]/directory/resource?k1=v1&k2=v2#tag`

- *Name* is for possible client identification
- *Hostname* could be an IP address
- *Port* defaults to protocol default (e.g. 80)
- *Directory* is a path to the resource
- *Resource* is the name of the object
- *?parameters* are passed to the server for execution
- *#tag* allows jumps to named tags within document

3.1.5 Examples of URLs

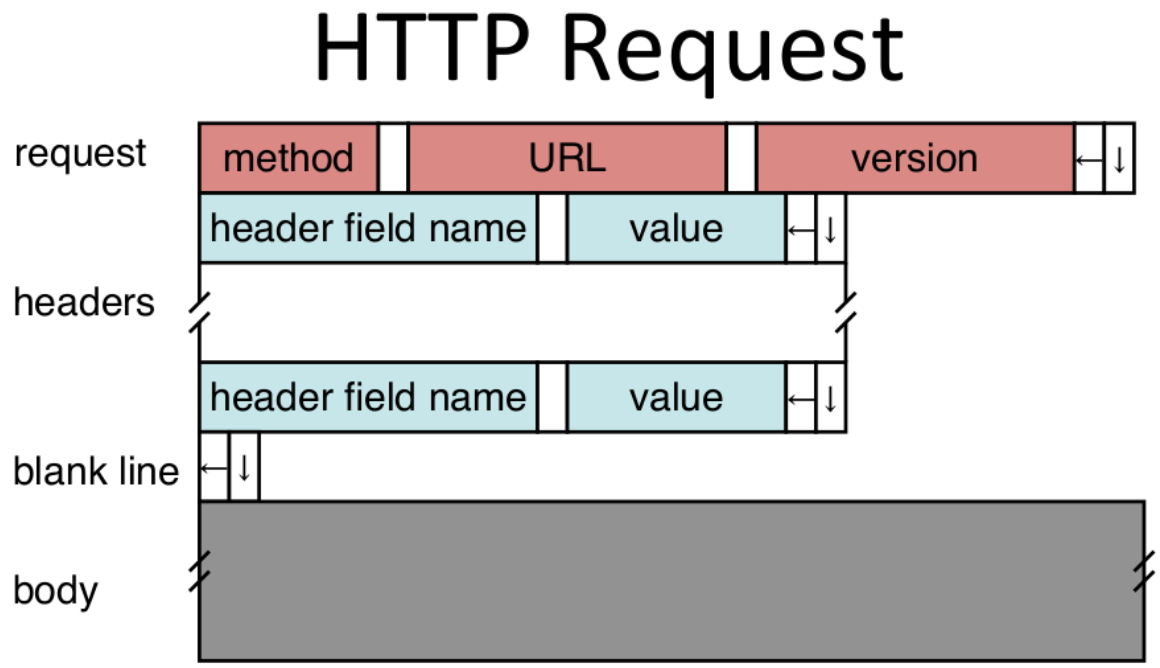
- `http://www2.cs.uh.edu/~gnawali/courses/cosc4377-s12/schedule.html`
- `http://en.wikipedia.org/wiki/Domain_name#Top-level_domains`
- `http://www.uh.edu/search/?q=computer+science&x=0&y=0`

3.2 HTTP

- Important properties
  - Client-server protocol
  - Protocol (but not data) in ASCII
  - Stateles
  - Extensible (header fields)
- Server typically listens on port 80
- Server sends response, may close connection (client may ask it to stay open)
- Version 1.1 in use by less than 45% of websites, version 2 in use by over 45% of websites, version 3 in use by 5.8% of websites

3.2.1 Steps in HTTP Request

- Open TCP connection to server
- Send request
- Receive response
- TCP connection terminates
  - How many RTTs for a single request?
- You may also need to do a DNS lookup first!



- Method:
  - GET: current value of resource, run program
  - HEAD: return metadata assocated with a resource
  - POST: update a resource, provide input for a program
- Headers: useful info for proxies or the server
  - e.g. desired language

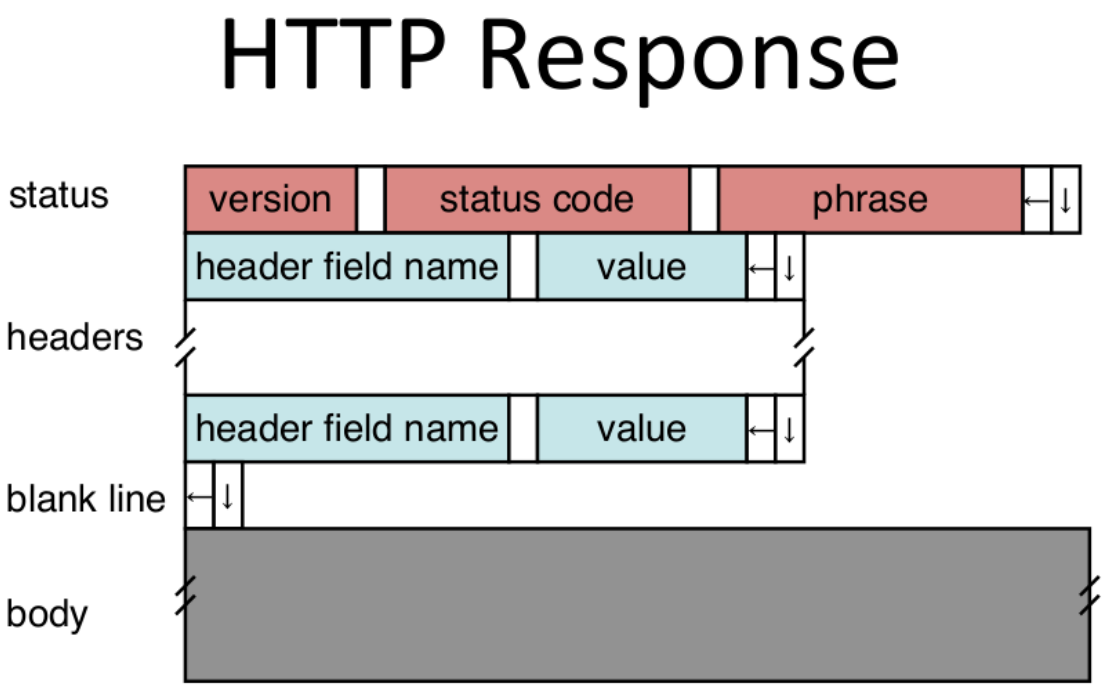
3.2.1.1 Sample Browser Request

```
GET / HTTP/1.1
Host: localhost:8000
User-Agent: Mozilla/5.0 (Macinto ...
Accept: text/xml,application/xm ...
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
(empty line)
```

3.2.1.2 Sample HTTP Response

```
HTTP/1.0 200 OK
Date: Wed, 25 Jan 2012 08:11:09 GMT
Expires: -1
Cache-Control: private, max-age=0
Content-Type: text/html; charset=ISO-8859-1
Set-Cookie: PREF=ID....
P3P: CP="This is not a P3P policy! See http://
www.google.com/support/accounts/bin/answer.py?
hl=en&answer=151657 for more info."
Server: gws
X-XSS-Protection: 1; mode=block
X-Frame-Options: SAMEORIGIN
<!doctype html><html><head><meta http-equiv="content-type"
```

```
content="text/html; charset=ISO-8859-1"><meta...>
```



- Status Codes:
  - 1xx: Information, e.g. 100 Continue
  - 2xx: Success, e.g. 200 OK
  - 3xx: Redirection, e.g. 302 Found (elsewhere)
  - 4xx: Client Error, e.g. 404 Not Found
  - 5xx: Server Error, e.g. 503 Service Unavailable

3.2.2 HTTP is Stateless

- Each request/response treated independently
- Servers not required to maintain state
- This is good!
  - Improves server scalability
- This is also bad...
  - Some applications need persistent state
  - Need to uniquely identify user to customize content
  - e.g. shopping cart, web-mail, usage tracking, (most sites today!)

3.2.3 HTTP Cookies

- Client-side state maintenance
  - Client stores small state on behalf of server
  - Sends request in future requests to the server
  - Cookie value is meaningful to the server (e.g. session ID)
- Can provide authentication
- https://en.wikipedia.org/wiki/HTTP_cookie

Where to find official HTTP specificaton?

www.w3.org

3.2.4 Anatomy of a Web Page

- HTML content
- A number of additional resources
  - Images
  - Scripts
  - Frames
- Browser makes one HTTP request for each object
  - Course web page: 4 objects
  - My facebook page this morning: 100 objects

3.2.5 AJAX

- *Asynchronous JavaScript and HTML*
- Based on XMLHttpRequest object in browsers, which allow code in the page to:
  - Issue a new, non-blocking request to the server, without leaving the current page
  - Receive the content
  - Process the content
- Used to add interactivity to web pages
  - XML not always used, HTML fragments, JSON, and plain text also popular

3.2.6 HTTP Performance

- What matters for performance?
- Depends on type of request
  - Lots of small requests (objects in a page)
  - Some big requests (large download or video)

3.2.6.1 Small Requests

- Latency matters
- RTT dominates
- Two major causes:
  - Opening a TCP connection
  - Actually sending the request and receiving response
  - And a third one: DNS lookup!
- Mitigate the first one with persistent connections (HTTP/1.1)
  - Which also means you don't have to "open" the connection each time

Browser Request

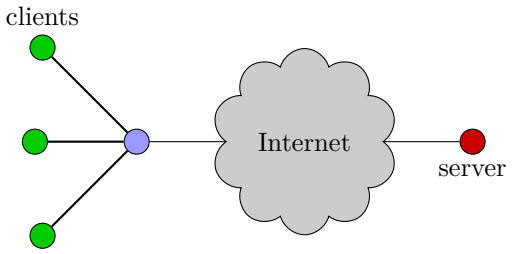
```
GET / HTTP/1.1
Host: localhost:8000
User-Agent: Mozilla/5.0 (Macinto ...
Accept: text/xml,application/xm ...
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection: keep-alive
```

- Second problem is that requests are serialized
  - Similar to stop-and-wait protocols!
- Two solutions

  - Pipelined requests (similar to sliding windows)
  - Parallel Connections
    * HTTP standard says no more than 2 concurrent connections per host name
    * Most browsers use more (up to 8 per host, *approx*35 total)
  - How are these two approaches different?
  - https://en.wikipedia.org/wiki/HTTP_pipelining

3.2.6.2 Larger Objects

- Problem is throughput in bottleneck link
- Solution: HTTP Proxy Caching
  - Also improves latency and reduces server load



Chapter 4

Domain Name System

4.1 Host names and IP Addresses

- Host names
  - Mnemonics appreciated by humans
  - Variable length, ASCII characters
  - Provide little (if any) information about location
  - Examples: www.facebook.com, bbc.co.uh
- IP Addresses
  - Numerical address appreciated by routers
  - Fixed length, binary numbers
  - Hierarchical, related to host location (in the network)
  - Examples: 69.171.228.14, 212.58.241.131

4.1.1 Separating Naming and Addressing

- Names are easier to remember
  - www.cnn.com vs. 157.166.244.26
- Addresses can change underneath
  - e.g. renumbering when changing providers
- Name could map to multiple addresses
  - www.cnn.com maps to at least 6 IP addresses
  - Enables
    - Load balancing
    - Latency reduction
    - Tailoring request based on requester's location/device/identity
  - Multiple names for the same address
    - Aliases: www.cs.brown.edu and cs.brown.edu
    - Multiple servers in the same node (e.g. apache virtual servers)

4.1.2 Scalable Address $\leftrightarrow$ Name Mappings

- Original kept in a local file, `hosts.txt`
  - Flat namespace
  - Central administrator kept master copy (for the internet)
  - To add a host, emailed admin
  - Downloaded file regularly
- Completely impractical today
  - File would be huge (gigabytes)
  - Traffic implosion (lookups and updates)

        * Some names change mappings every few days (dynamic IP)
    - Single point of failure
    - Impractical politics (repeated names, ownership, etc.)

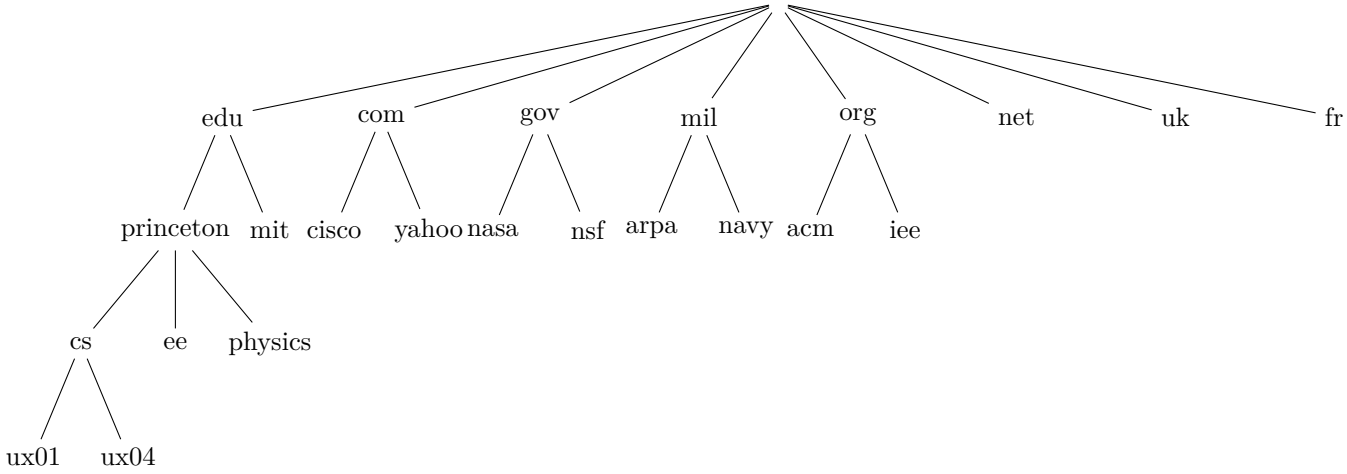
4.1.3 Goals for an Internet-scale name system

- Scalability
    - Must handle a huge number of records
        * With some software synthesizing names on the fly
    - Must sustain update and lookup load
- Distributed Control
    - Let people control their own names
- Fault tolerance
    - Minimize lookup failures in face of other network problems

4.1.3.1 The Good News

- Properties that make these goals easier to achieve
    1. Read-mostly database
        - Lookups *much* more frequent than updates
    2. Loose consistency
        - When adding a machine, not end of the world if it takes minutes or hours to propagate
    3. These suggest aggressive caching
        - Once you've looked up a hostname, remember
        - Don't have to look again in the near future

4.2 Domain Name System (DNS)



- Hierarchical namespace broken into *zones*
    - root (.), edu., princeton.edu, cs.princeton.edu,
    - Zones separately administred :: delegation
    - Parent zone tells you how to find servers for subdomains
- Each zone served from multiple replicated servers

4.2.1 DNS Architecture

- Hierarchy of DNS Servers
    - Root servers

  - Top-level domain (TLD) servers
  - Authoritative DNS servers
- Performing the translation
  - Local DNS servers
  - Resolver software

4.2.2 Resolver Operation

- Apps make recursive queries to local DNS server
  - Ask server to get answer for you
- Server makes iterative queries to remote servers
  - Ask servers who to ask next
  - Cache results aggresively

4.2.3 DNS Root Server

- Located in Virginia, USA
- How do we make the root scale?

4.2.4 DNS Root Servers

- 13 root servers (www.root-servers.org)
  - Labeled A through M (e.g. A.ROOT-SERVERS.NET)
- Does this scale?
- Replication via *anycasting*

4.2.5 TLD and Authoritative DNS Servers

- Top Level Domain (TLD) servers
  - Generic domains (e.g. com, org, edu)
  - Country domains (e.g. uk, br, tv, in, ly)
  - Special domains (e.g. arpa)
  - Typically managed professionally
- Authoritative DNS servers
  - Provides public records for hosts at an organization
    - e.g. for the organization's own servers (www, mail, etc)
  - Can be maintained locally or by a service provider

4.2.6 Reverse Mapping

- How do we get the other direction, IP address to name?
- Addresses have a hierarchy:
  - 128.148.34.7
- But, most significant element comes first
- Idea: reverse the numbers, 7.34.148.128...
  - And look that up in DNS
- Under what TLD?
  - Convention: in-addr.arpa
  - Lookup 7.34.148.128.in-addr.arpa
  - in6.arpa for IPv6

https://en.wikipedia.org/wiki/Reverse_DNS_lookup

4.2.7 DNS Caching

- All these queries take a long time!

  - And could impose tremendous load on root servers
  - This latency happens before any real communication, such as downloading your web page
- Caching greatly reduces overhead
  - Top level servers very rarely change
  - Popular sites visited often
  - Local DNS server caches information from many users
- How long do you store a cached response?
  - Original server tells you: TTL entry
  - Server delete entry after TTL expires

4.2.8 Negative Caching

- Remember things that don't work:
  - Misspellings like www.cnn.comm, ww.cnn.com
- These can take a long time to fail for the first time
  - Good to cache negative results so it will fail faster next time
- But negative caching is optional and not widely implemented

4.2.9 DNS Protocol

- TCP/UDP port 53
- Most traffic uses UDP
  - Lightweight protocol has 512 byte message limit
  - Retry using TCP if UDP fails (e.g. reply truncated)
- TCP requires message boundaries
  - Prefix all messages with 16-bit length
- Bit in query determines if query is recursive

4.2.10 Resource Records

- All DNS info represented as resource records (RR)

  name [ttl] [class] type rdata

  - name: domain name
  - TTL: time to live in seconds
  - class: for extensibility, normally IN (1) "Internet"
  - type: type for the record
  - rdata: resource data dependent on the type
- Two import RR types
  - A – Internet Address (IPv4)
  - NS – name server
- Example RRs

```
bayou.cs.uh.edu. 3600 IN A 129.7.240.18
cs.uh.edu. 3600 IN NS ns2.uh.edu.
cs.uh.edu. 3600 IN NS dns.cs.uh.edu.
```

4.2.11 Some important details

- How do local servers find root servers?
  - DNS lookup on a.root-servers.net?
  - Servers configured with *root cache* file
  - ftp://ftp.rs.internic.net/domain/db.cache
  - Contains root name servers and their addresses

- How do you get addresses of other name servers?
  - To obtain the address of www.cs.brown.edu, ask a.edu-servers.net, says a.root.servers.net
  - How do you find a.edu-servers.net?
  - Glue records: A records in parent zone.