University of Houston

Introduction to Computer Networks

COSC 6377

Midterm Review

Author
K.M. Hourani

Based on Notes By
Dr. Omprakash Gnawali

March 5, 2021

Contents

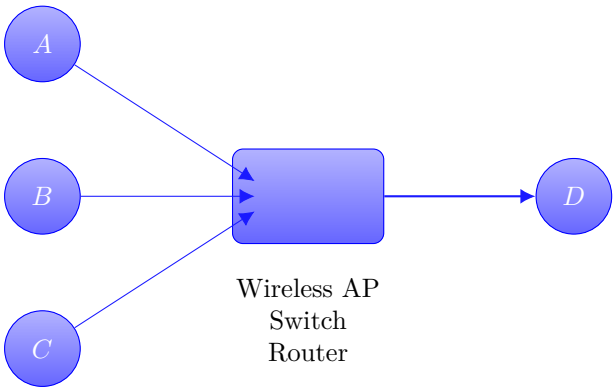
Chapter 1

Intro

1.1 The Internet

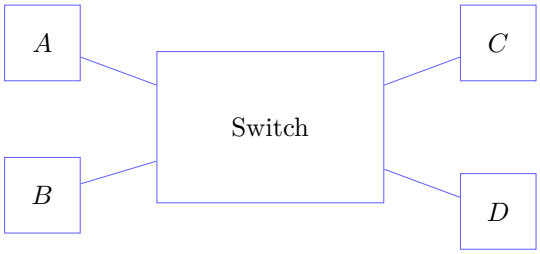
- Collection of nodes, wired and wireless technology connecting these nodes, applications and services
- Types of nodes
  - Desktops and Laptop
  - Servers
  - TV/Refrigerator
  - Cellphones
- Goal: Connect all the nodes to each other
- Solutions
  - $\binom{n}{2} = \mathcal{O}(n^2)$ cables
  - Sharing 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

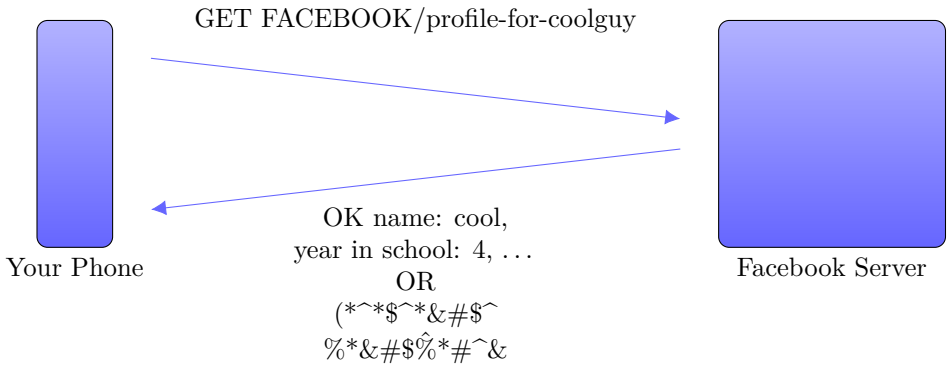
- How often does a network fail?
- How often do packets drop?
  - Damage (corruption)
  - Drops in the queues
- How persistent are failures?
- Typical metrics
  - uptime percentage
  - packet or bit loss rate

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?

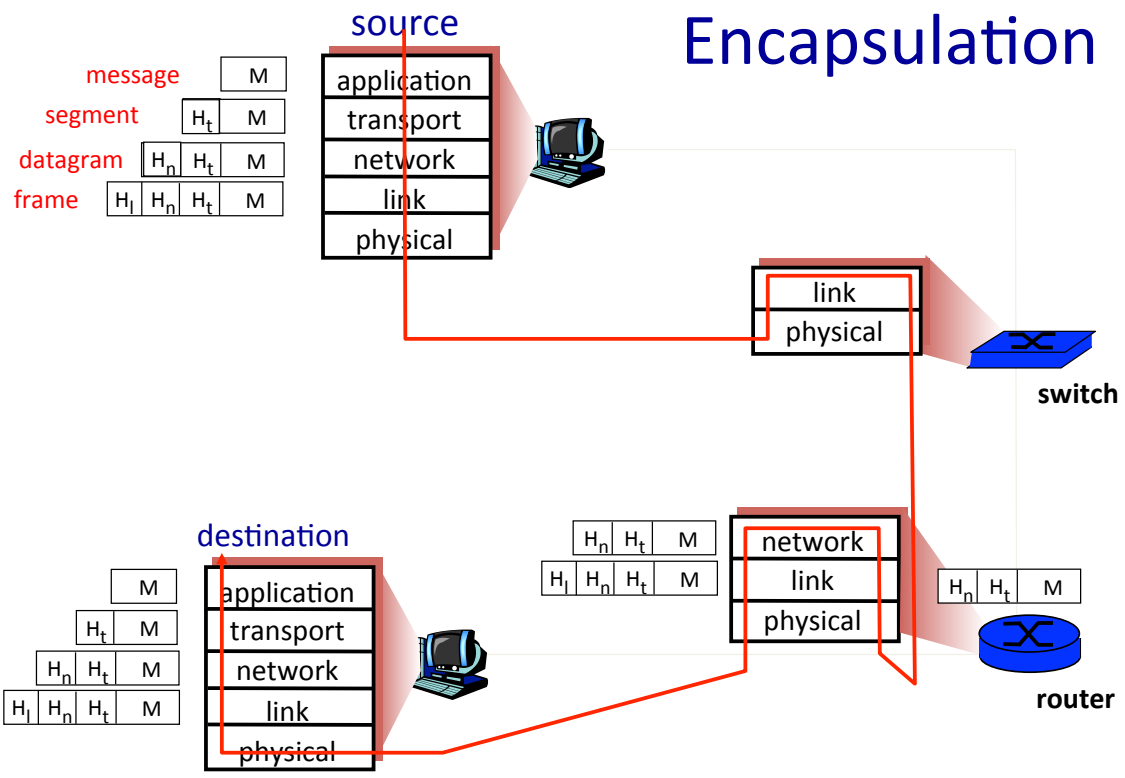
1.5.1.1 Protocol Layers

- 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

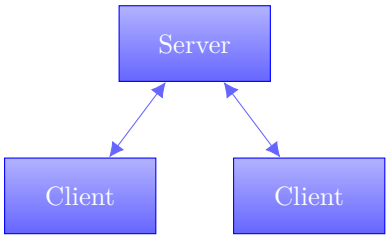
- Think of how paperwork is processed in a university
  - Each person processes and adds some information to it and passes it along
- On the transmitter, the lower layers include the message from upper layers, add their own information, and send it along
- On the receiver: reverse


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

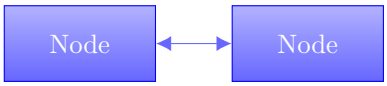
Chapter 2

Network Applications and Socket Programming

2.1 Network Applications



Client-Server



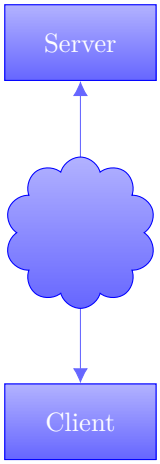
Peer-to-Peer

Hybrid

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

2.1.1 Inter-Application Communication

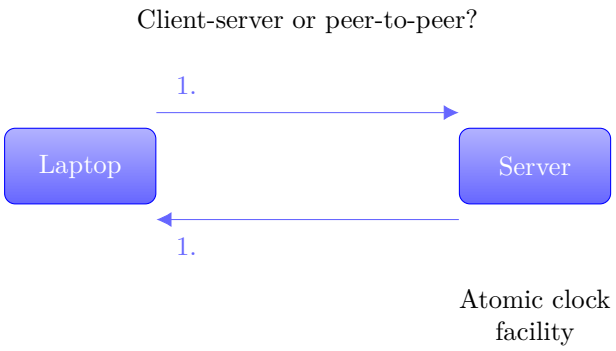
- Need a way to send and receive messages
- Inter-process communication
- Need naming, routing, transport
- Transport using TCP and UDP
  - On top of IP



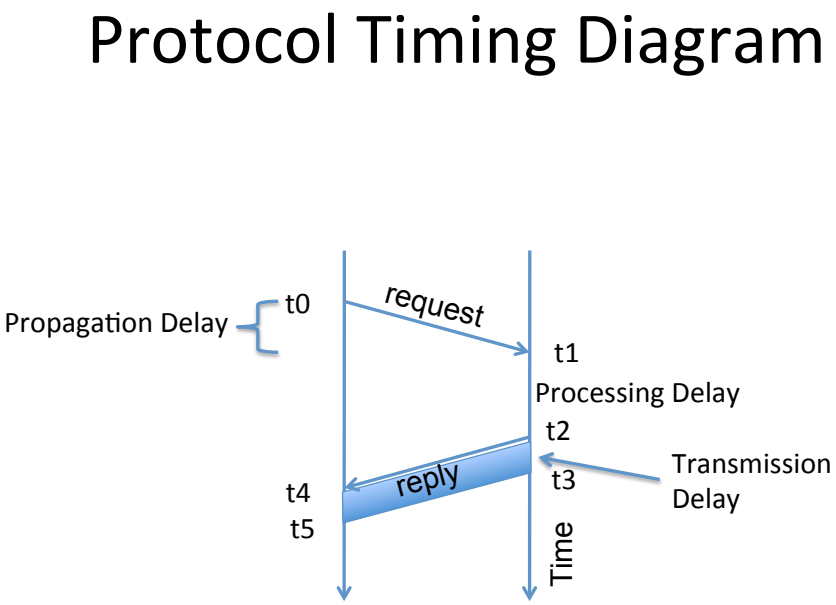
2.1.2 Application Protocols

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

2.1.3 Network Time Service

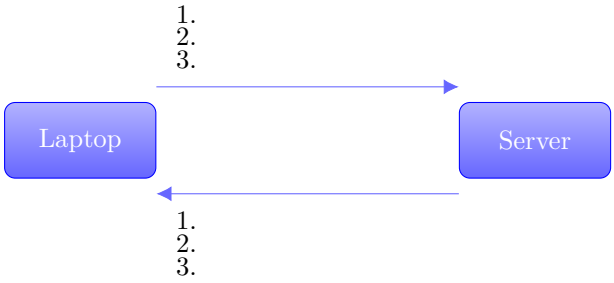


2.1.3.1 Protocol Timing Diagram



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?

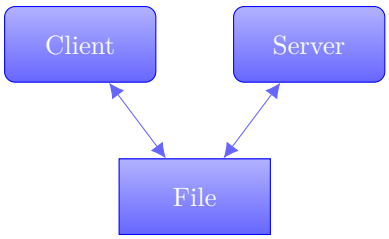


2.2 Socket Programming

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

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.

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

```
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

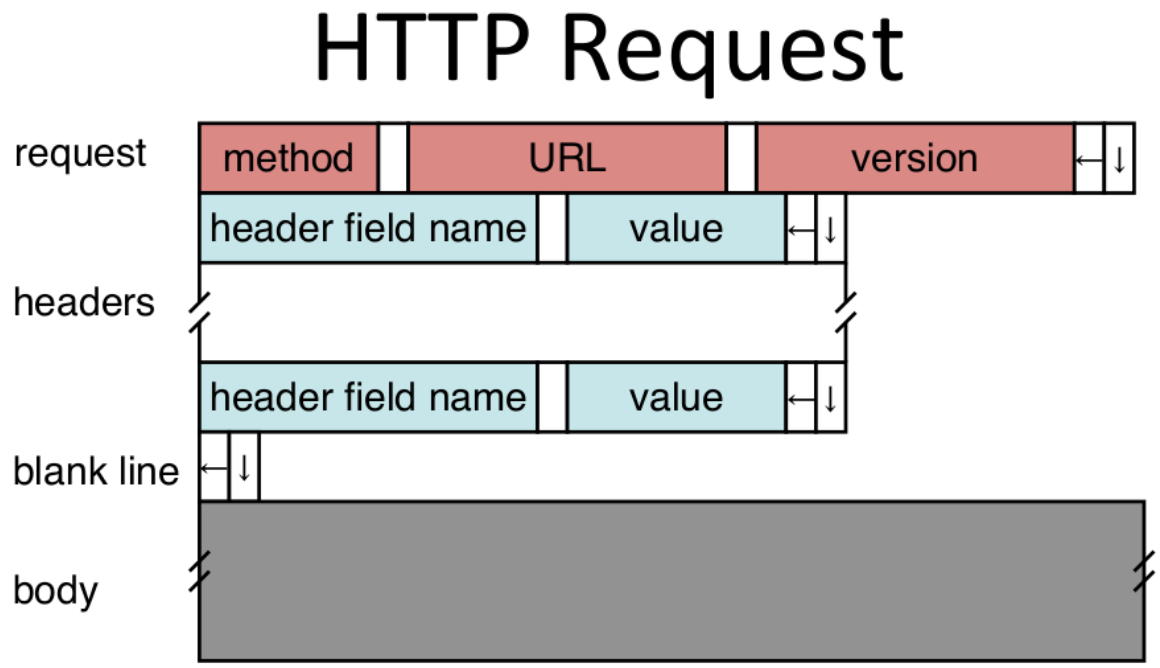
- `http://www2.cs.uh.edu/~gnawali/courses/cosc4377-s12/schedule.html`
- `http://en.wikipedia.org/wiki/Domain_name#Top-level_domains`
- `http://www.uh.edu/search/?q=computer+science&x=0&y=0`

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!



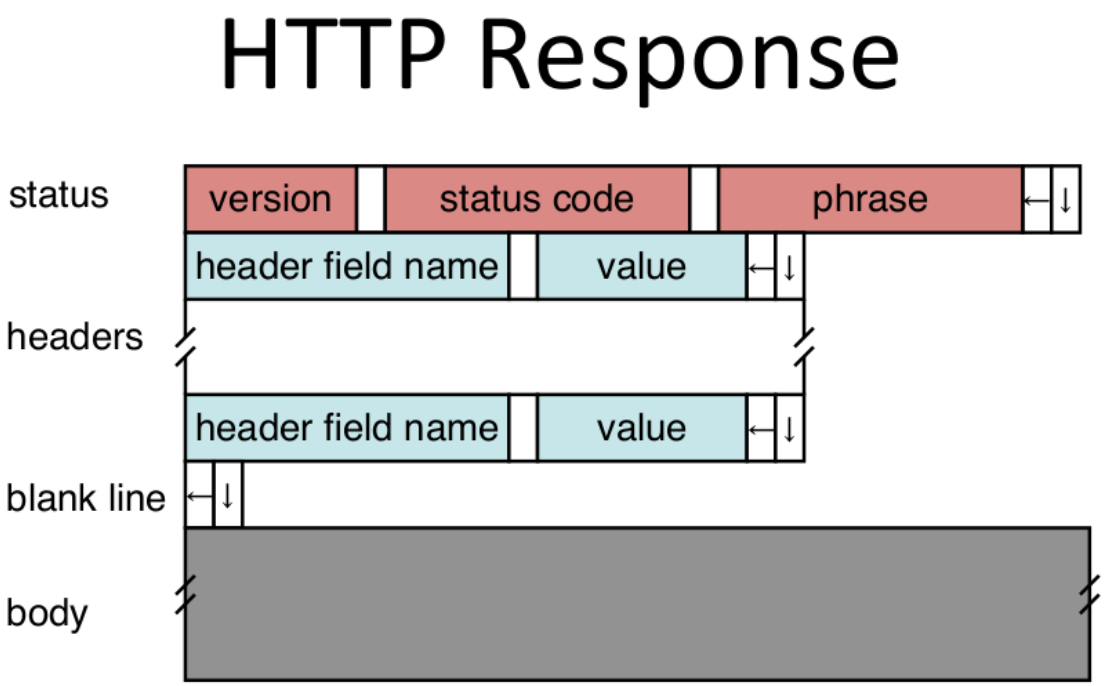
- Method:
  - GET: current value of resource, run program
  - HEAD: return metadata assocated with a resource
  - POST: update a resource, provide input for a program
- Headers: useful info for proxies or the server
  - e.g. desired language

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

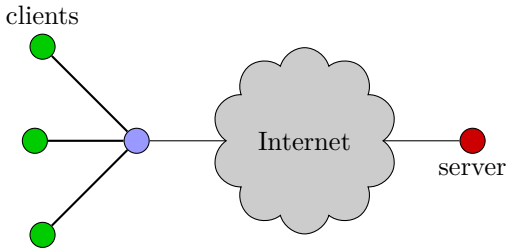
Browser Request

```
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.)

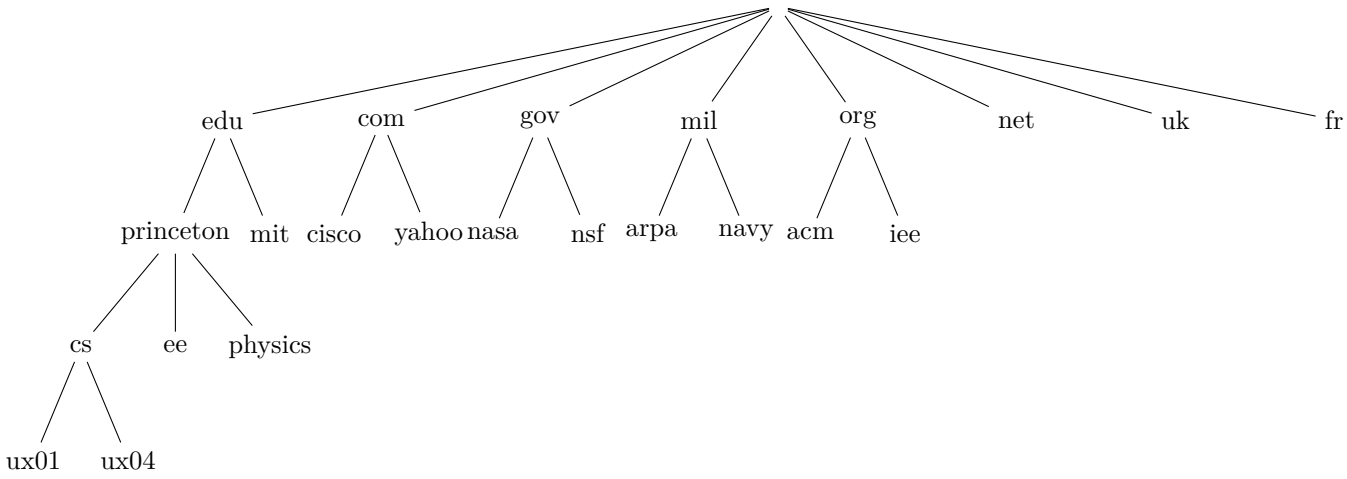
4.1.3 Goals for an Internet-scale name system

- 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.

Chapter 5

DNS and P2P

5.1 DNS

5.1.1 Structure of a DNS Message

```
+---------------------+
|        Header       |
+---------------------+
|       Question      | the question for the name server
+---------------------+
|        Answer       | RRs answering the question
+---------------------+
|      Authority      | RRs pointing toward an authority
+---------------------+
|      Additional     | RRs holding additional information
+---------------------+
```

- Same format for queries and replies
  - Query has 0 RRs in Answer/Authority/Additional
  - Reply includes question, plus has RRs
- Authority allows for delegation
- Additional for glue, other RRs client might need

5.1.1.1 Header Format

```
                                1  1  1  1  1  1
  0  1  2  3  4  5  6  7  8  9  0  1  2  3  4  5
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                      ID                       |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|QR|   Opcode  |AA|TC|RD|RA|   Z    |   RCODE   |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                    QDCOUNT                    |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                    ANCOUNT                    |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                    NSCOUNT                    |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                    ARCOUNT                    |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

- ID: match response to query; QR: 0 query/1 response
- RCODE: error code
- AA: authoritative answer, TC: truncated

- RD: recursion desired, RA: recursion available

5.1.2 Other RR Types

- CNAME (canonical name): specifies an alias

```
www.google.com.      446199 in CNAME    www.l.google.com
www.l.google.com.  300 IN  A    72.14.204.147
```

- MX record: specifies servers to handle mail for a domain (the part after the @ in email address)
- SOA (start of authority)
  - Information about a DNS zone and the server responsible for the zone
- PTR (reverse lookup)
  `18.240.7.129.in-addr.arpa. 3600 IN PTR bayou.cs.uh.edu.`
  https://en.wikipedia.org/wiki/List_of_DNS_record_types

5.1.3 Inserting a Record in DNS

- Your new startup httpserver.com
- Get a block of addresses from ISP
  - say 212.44.9.128/25
- Register helpme.com at GoDaddy.com (for example)
  - Provide name and address of your authoritative name server (primary and secondary)
  - Registrar inserts RR pair into the com TLD server:
    * helpme.com NS dns1.httpserver.com
    * dns1.helpme.com A 212.44.9.129
- Configure your authoritative server (dns1.helpme.com)
  - Tyep A record for www.httpserver.com
  - Type MX record for httpserver.com
- Need to provide reverse PTR bindings
  - e.g. 212.44.9.129 → dns1.httpserver.com
- Normally, these would go into 9.44.212.in-addr.arpa zone
- Problem: you can't run the name server for that domain. Why not?
  - Your block is 212.44.9.128/25, not 212.44.9.0/24
  - Whoever has 212.44.9.0/24 would not be happy with you setting their PTR records
- Solution: [RFC2317, Classless Delegation]
  - Install CNAME records in parent zone, e.g. 129.9.44.212.in--addr.arpa CNAME 129.ptr.httpserver.com

5.1.4 DNS Security

- You go to Starbucks, how does your browser find www.google.com?
  - ask local name server, obtained from DHCP
  - you implicitly trust this server
  - can return any answer for google.com, including a malicious IP that poses as a man in the middle
- How can you know you are getting correct data?
  - today, you can't
  - HTTPS can help
  - DNSSEC extension will allow you to verify

5.1.4.1 Cache Poisoning

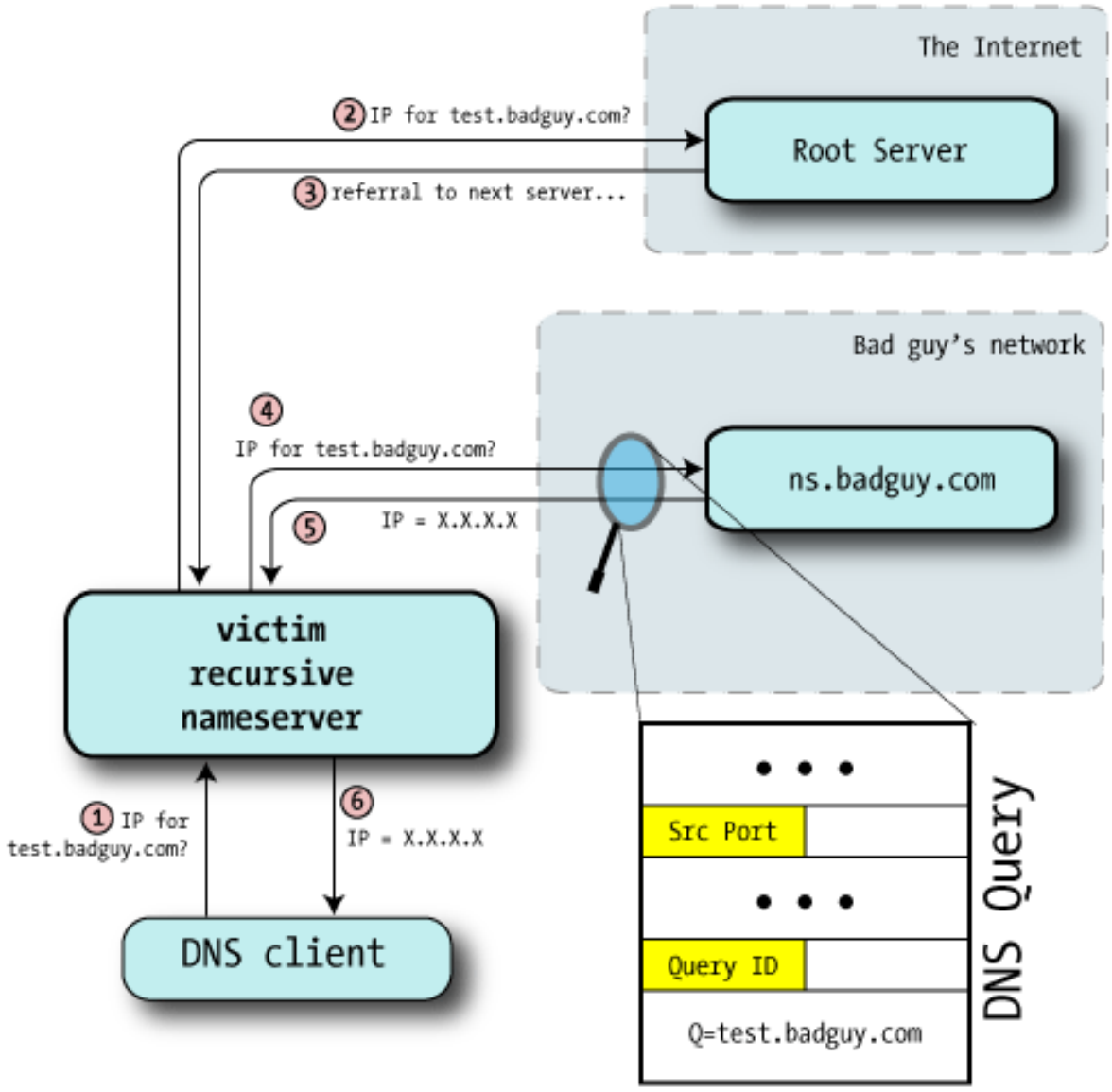
- Suppose you can tronl evil.com. You receive a query for www.evil.com and reply

```
;; QUESTION SECTION:
;www.evil.com.                  IN     A

;; ANSWER SECTION:
www.evil.com.           300     IN     A     212.44.9.144

;; AUTHORITY SECTION:
evil.com.               600     IN     NS    dns1.evil.com.
evil.com.               600     IN     NS    google.com.
```

```
;; ADDITIONAL SECTION:
google.com.             5       IN      A       212.44.9.155
```

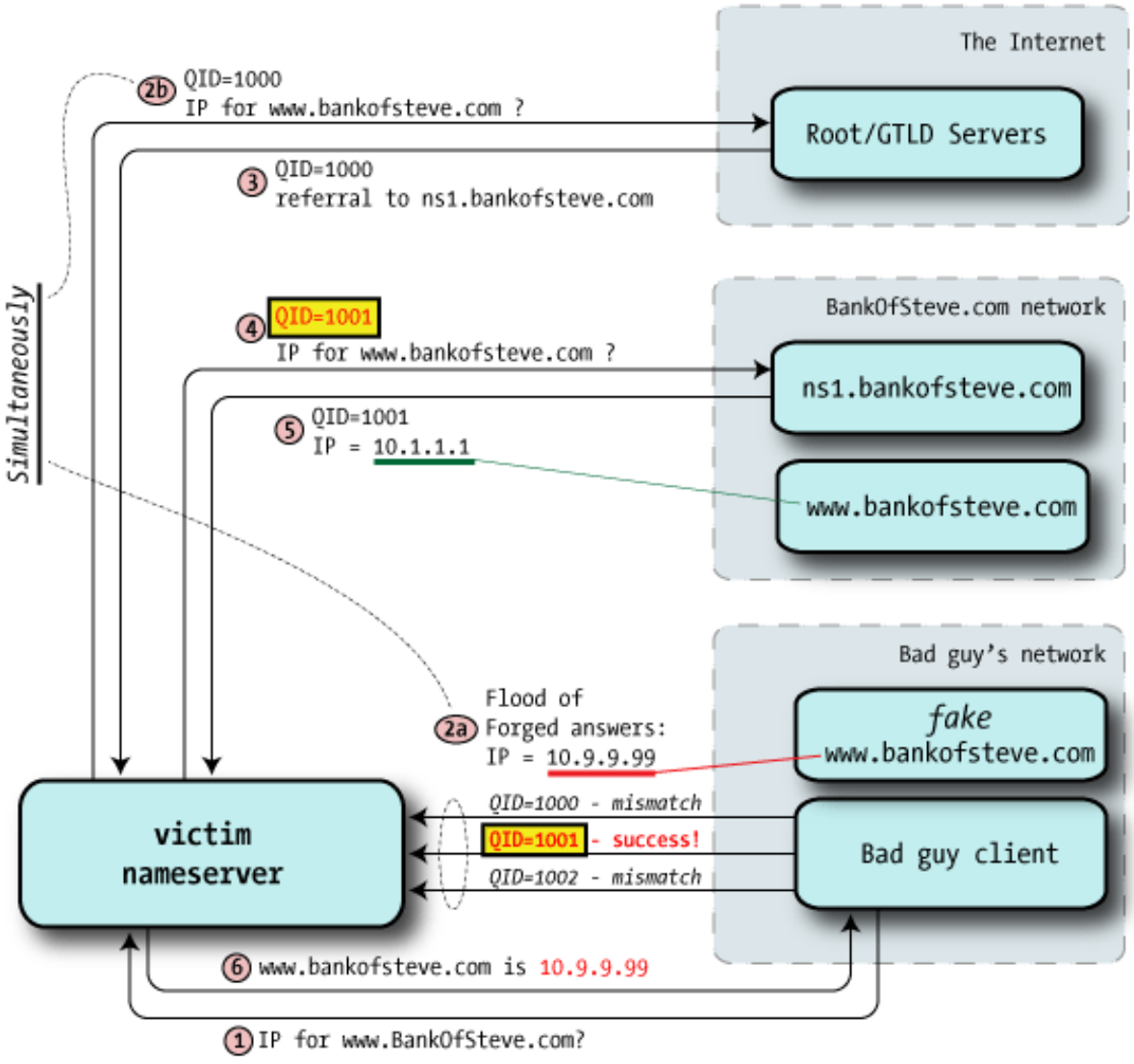
- Glue record pointing to your IP, not Google's
- Gets cached!
- But how do you get a vimctim to look up evil.com?
- You might connect to their mail server and send
  - HELO www.evil.com
  - Which their mail server then looks up to see if it corresponds to your IP address (SPAM filtering)
- Mitigation (bailiwick checking)
  - Only accepts glue records from the domain you asked for
- Bad guy at Starbucks can sniff or *guess* the ID field the local server will use
  - Not hard if DNS server generates ID numbers sequentially
  - Can be done if you force the DNS server to look up something in *your* name server
  - Guess has 1 in 65535 chance (or does it?)
- Now:
  - Ask the local server to lookup google.com
  - Spoof the response from google.com using the correct ID
  - Bogus response arrives before legit one (maybe)
- Local server caches first response it receives
  - Attacker can set a long TTL

5.1.5 Guessing Query ID



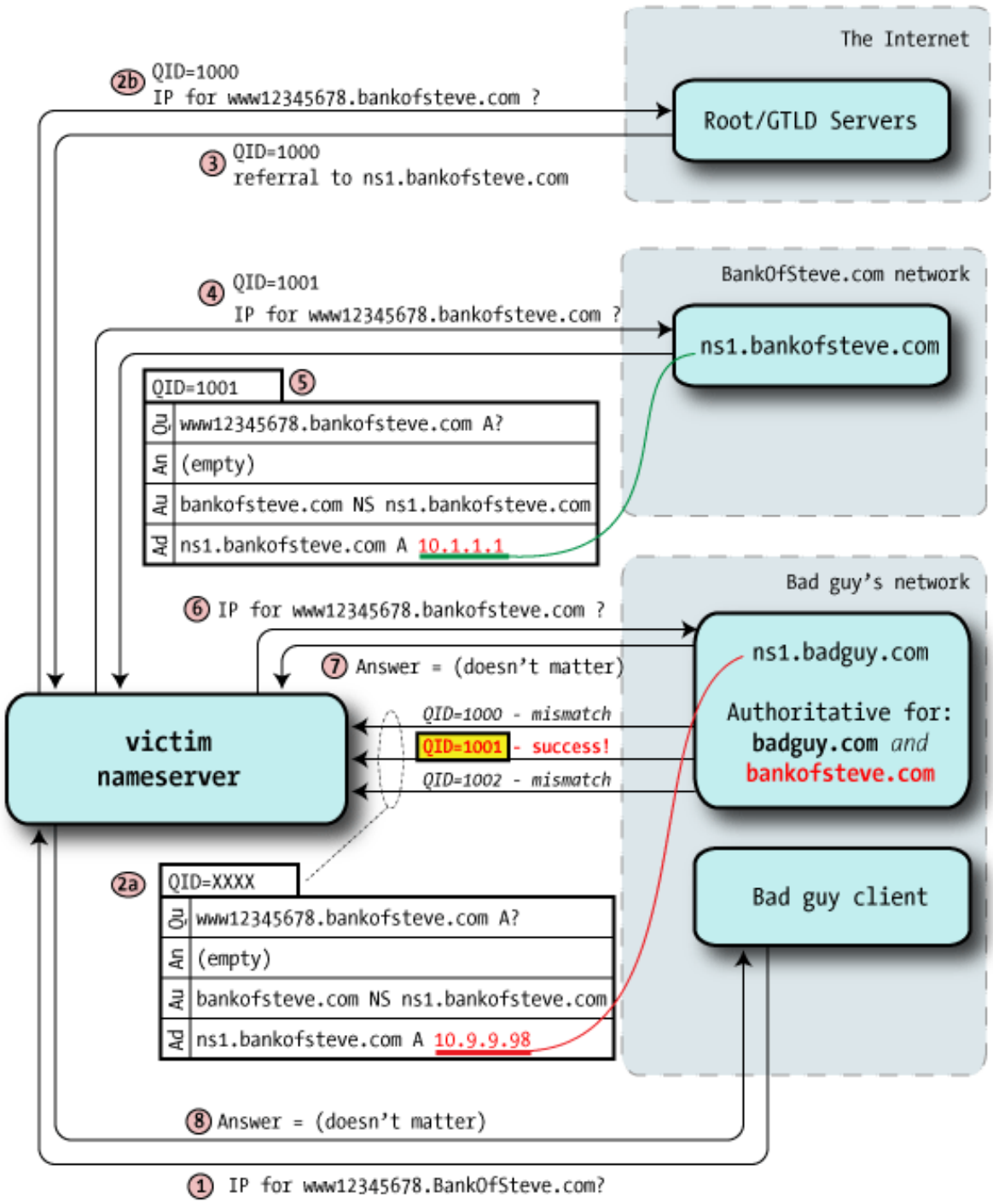
http://www.unixwiz.net/techtips/iguide-kaminsky-dns-vuln.html

5.1.6 Cache Poisoning



http://www.unixwiz.net/techtips/iguide-kaminsky-dns-vuln.html

5.1.7 Hijacking Authority Record



http://www.unixwiz.net/techtips/iguide-kaminsky-dns-vuln.html

5.1.8 Kaminsky Exploit

- If good guy wins the race, you have to wait until the TTL to race again
- But...
  - What if you start a new race for AAAA.google.com, AAAB.google.com, ...?
  - Forge CNAME responses for each
  - Circumvents bailiwick checking

5.1.8.1 Countermeasures

- Randomize ID
  - Used to be sequential
- Randomize source port number
  - Used to be the same for all requests from the server
- Offers some protection, but attack still possible

5.1.9 Load Balancing using DNS

- Return multiple IP addresses ("A" records) for a name
- Benefits
  - Spread the load evenly across the IP addresses
- Problems
  - Caching, no standard on which address to use, ...
- How to solve these problems?
  - Poll load to compute return list
  - https://en.wikipedia.org/wiki/Round-robin_DNS

5.2 Peer-to-Peer

5.2.1 Client-Server Bottlenecks

- Download time can scale linearly ($\mathcal{O}(n)$ with n clients)
- Scaling up server bandwidth can be expensive
- Too expensive to provision for flash crowds

5.2.2 Peer-to-Peer Systems

- How did it start?
  - A killer application: file distribution
  - Free music over the internet (not exactly legal...)
- Key idea: share storage, content, and bandwidth of individual users
  - Lots of them
- Big challenge: coordinate all of these users
  - In a scalable way (not $n \times n = n^2$)
  - With changing population (aka *churn*)
  - With no central administration
  - With no trust
  - With large heterogeneity (content, storage, bandwidth, ...)

5.2.2.1 3 Key Requirements

- P2P Systems do Three things:
  1. Help users *determine what they want*
     - Some form of search
     - P2P version of Google
  2. *Locate* that content
     - Which node(s) hold the content?
     - P2P version of DNS (map name to location)
  3. *Download* the content
     - Should be efficient
     - P2P form of Akamai

5.2.3 Napster

- Search & Location: central server
- Download: contact a peer, transfer directly
- Advantages:
  - Simple, advanced search possible
- Disadvantages:

  - Single point of failure (technical and . . . legal!)
  - The latter is what got Napster killed

5.2.4 Gnutella: Flooding on Overlays (2000)

- Search & Location: flooding (with TTL)
- Download: direct

5.2.5 BitTorrent

- One big problem with previous approaches
  - Asymmetric bandwidth
- BitTorrent
  - Search: independent search engines (e.g. PirateBay, isoHunt)
    - Maps keywords → .torrent file
  - Location: centralized *tracker* node per file
  - Download: chunked
    - File split into many pieces
    - Can download from many peers
- How does it work?
  - Split files into large pieces (245KB - 1MB)
  - Split pieces into subpieces
  - Get peers from tracker, exchange info on pieces
- Three phases in download
  - Start: get a piece as soon as possible (random)
  - Middle: spread pieces fast (rarest piece)
  - End: don't get stuck (parallel downloads of last pieces)

5.2.5.1 BitTorrent Tracker Files

- Torrent file (.torrent) describes files to download
  - Names tracker, server tracking who is participating
  - File length, piece length, SHA1 hash of pieces
  - Additional metadata
- Client contacts tracker, starts communicating with peers

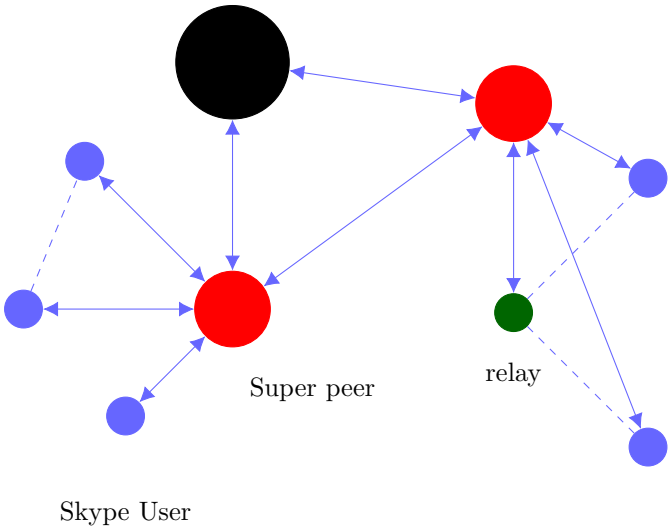
```
d8:announce39:http://torrent.ubuntu.com:6969/announce13:announce-
listll39:http://torrent.ubuntu.com:6969/announceel44:http://
ipv6.torrent.ubuntu.com:6969/announceee7:comment29:Ubuntu CD
releases.ubuntu.com13:creation
datei1272557944e4:infod6:lengthi733837312e4:name29:ubuntu-10.04-
netbook-i386.iso12:piece lengthi524288e6:pieces28000:...
```

  Example tracker from ubuntu.com
- Self-scaling: incentivize sharing
  - If people upload as much as they download, system scales with number of users (no free-loading)
- Uses tit-for-tat: only upload to those who give you data
  - *Choke* most of your peers (don't upload to them)
  - Order peers by download rate, choke all but P best
  - Occasionally unchoke a random peer (might become a nice uploader)

5.2.6 Skype

- Real-time communication
- Two major challenges:
  - Finding what host a user is on
  - Being able to communicate with those hosts
- Uses Superpeers for registering presence, searching for where you are
  - Need bootstrap super-peers
- Those Superpeers organize index of users

- Making a call
  - Many nodes don't allow incoming connections
  - Uses regular nodes, outside of NATs, as decentralized relays



Chapter 6

Structured P2P and the Transport Layer

6.1 Structured P2P Systems

- Distributed Hash Table (DHT)
  - Efficient (`Key`, `Value`) storage
  - Approach: map the ID to a host
- Challenges
  - Scale to millions of nodes
  - Churn
  - Heterogeneity

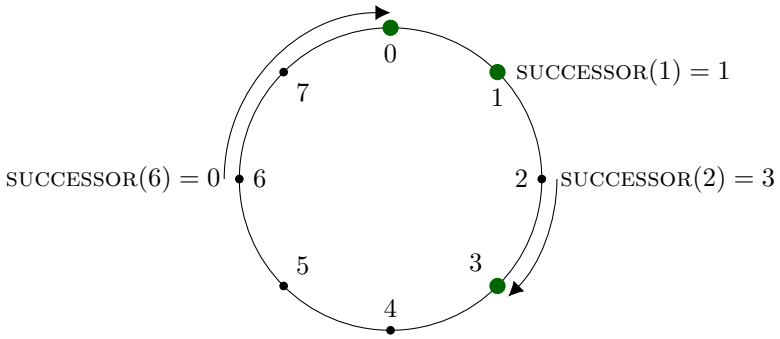
6.1.1 DHTs

- IDs from a *flat* namespace
  - Contrast with hierarchical IP, DNS
- Metaphor: hash table, but distributed
- Interface
  - GET(`key`)
  - PUT(`key`, `value`)

- How?
  - Every node supports a single operation:
    Given a `key`, route messages to node holding `key`

6.1.1.1 Consistent Hashing

- Map keys to nodes
- `nodeID` = HASH(`nodeIP`)
- k mapped to SUCCESSOR(k)
- SUCCESSOR(k) is the first active node beginning at k



6.1.1.2 Consistent Hashing Properties

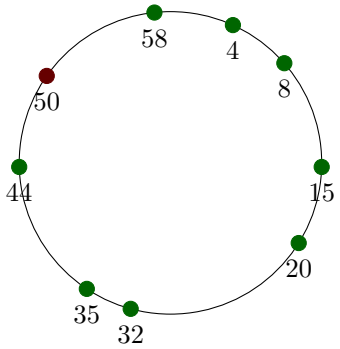
- Designed for node join/leave with minimal churn in key mapping
- k/n keys per node
- k/n keys change hands during join/leave

6.1.1.3 Lookup

- Each node maintains its successor
- Route packet (ID, data) to the node responsible for ID using successor pointers

6.1.1.4 Joining

- Node with ID 50 joins the ring
- Node 50 needs to know at least one node already in the system
  - Assume known node is 15
- Node 50: send JOIN(50) to node 15
- Node 44: returns node 58
- Node 50: updates its successor to 58
- Node 50: send stabilize to node 58
- Node 58:
  - update predecessor to 50
  - send NOTIFY() back
- Node 44 sends a stabilize message to its successor, node 58
- Node 58 replies with a notify message
- Node 44 updates it successor to 50
- Node 44 sends a stabilize message to its new succcesor, node 50
- Node 50 sets its predecessor to node 44



6.2 Transport Layer

6.2.1 Network Applications

- Centralized and Peer-to-peer arhictectures
- How to design and write network applications
- Case studies
  - HTTP
  - DNS
  - P2P applications
- These applications need a *reliable method to send information across the network*
- Transport Layer provides that service

6.2.2 Transport Layer

- Transport protocols sit on top of network layer and provide
  - Application-level multiplexing (“ports”)
  - Error detection, reliability, etc.

6.2.3 Error Detection

- Idea: add redundant information to catch errors in packet
- Three examples
  - Parity
  - Internet Checksum
  - CRC

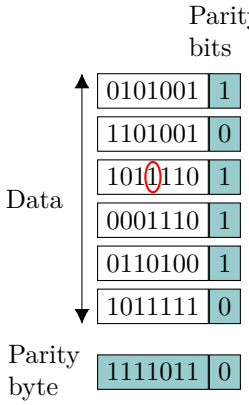
6.2.3.1 Parity Bit

- Can detect odd number of bit errors
- No correction

Data	1101101
Parity	1
Transmit	11011011

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

6.2.3.2 2-D Parity



- Add 1 parity bit for each 7 bits
- Add 1 parity bit for each bit position across the frame
  - Can correct single-bit errors
  - Can detect 2- and 3-bit errors, most 4-bit errors

6.2.3.3 Checksum

- Algorithm
  - Set `checksum` field to 0
  - Sum all 16-bit words, adding any carry bits to the LSB (one’s complement sum)

  - Flip bits to get checksum (one's complement)
- Transmit: data + checksum
- To check: sum whole packet, including sum, should get `0xffff`

https://tools.ietf.org/html/rfc1071

6.2.3.4 How good is it?

- 16 bits is not very long
  - Probability 1-bit error not detected?
- Checksum does catch any 1-bit error
- But not any 2-bit error
  - e.g. increment word ending 0, decrement one ending in 1

6.2.3.5 CRC – Error Detection with Polynomials

- Consider message to be a polynomial in $\mathbb{Z}_2[x]$
  - Each bit is one coefficient
  - e.g. message 10101001 $\rightarrow m(x) = x^7 + x^5 + x^3 + 1$
- Can reduce one polynomial modulo another
  - Select a degree k *irreducible* polynomial $C(x)$ in $\mathbb{Z}_2[x]$
  - Let $n(x) = m(x) \cdot x^k$
  - Compute $r(x) = n(x) \bmod C(x)$
  - Compute $n(x) - r(x)$
- Checking CRC is easy
  - Reduce message by $C(x)$, make sure remainder is 0

6.2.4 Reliable Delivery

- Error detection can discard bad packets
- Problem: if bad packets are lost, how can we ensure reliable delivery?
  - Exactly-once semantics = at least once + at most once

6.2.4.1 At Least Once Semantics

- How can the sender know the packet arrived *at least once*?
  - Acknowledgements + Timeout
- Stop and Wait Protocol
  - S: Sent packet, wait
  - R: Receive packet, send ACK
  - S: Receive ACK, send next packet
  - S: No ACK, timeout and retransmit

6.2.4.2 Stop and Wait Problems

- Duplicate Data
- Duplicate ACKs
- Can't fill pipe
- Difficult to set the timeout value

6.2.4.3 At Most Once Semantics

- How to avoid duplicates?
  - Uniquely identify each packet
  - Have receiver and sender remember
- Stop and wait: add 1 bit to the header
  - Why is it enough?

6.2.5 Sliding Window Protocol

- Still have the problem of keeping pipe full.
  - Generalize approach > 1-bit counter
  - Allow multiple outstanding (unACKed) frames
  - Upper bound on unACKed frames, called *window*

6.2.5.1 Sliding Window Sender

- Assign sequence number (`SeqNum`) to each frame
- Maintain three state variables
  - send window size (`SWS`)
  - last acknowledgement received (`LAR`)
  - last frame send (`LFS`)
- Maintain invariant: $\texttt{LFS} - \texttt{LAR} \leq \texttt{SWS}$
- Advance `LAR` when ACK arrives
- Buffer up to `SWS` frames

6.2.5.2 Sliding Window Receiver

- Maintain three state variables
  - receive window size (`RWS`)
  - largest acceptable frame (`LAF`)
  - last frame received (`LFR`)
- Maintain invariant: $\texttt{LAF} - \texttt{LFR} \leq \texttt{RWS}$
- Frame `SeqNum` arrives:
  - if $\texttt{LFR} < \texttt{SeqNum} \leq \texttt{LAF}$, accept
  - if $\texttt{SeqNum} \leq \texttt{LFR}$ or $\texttt{SeqNum} > \texttt{LAF}$, discard
- Send *cumulative* ACKs

Chapter 7

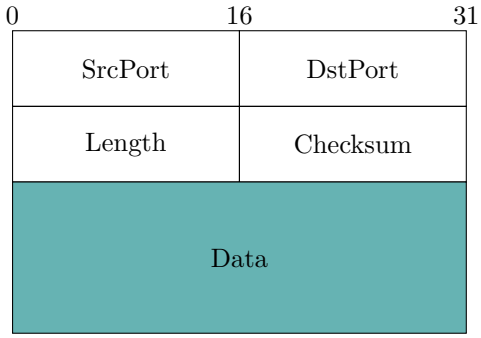
Transport Protocols

7.1 UDP – User Datagram Protocol

- Unreliable, unordered datagram service
- Adds multiplexing checksum
- End points identified by *ports*
  - Scope is an IP address (interface)
- Checksum aids in error detection

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

7.1.1 UDP Header



7.1.1.1 UDP Checksum

- Uses the same algorithm as the IP checksum
  - Set checksum field to 0
  - Sum all 16-bit words, adding any carry bits to the LSB
  - Flip bits to get checksum (except `0xffff` → `0xffff`)
  - To check: sum whole packet, including sum, should get `0xffff`
- How many errors?
  - Catches any 1-bit error
  - Not all 2-bit errors
- Optional in IPv4: not checked if value is 0

7.1.1.2 Pseudo Header

```
 0      7 8     15 16    23 24    31
+--------+--------+--------+--------+
|          source address           |
+--------+--------+--------+--------+
|        destination address        |
+--------+--------+--------+--------+
|  zero  |protocol|   UDP length    |
+--------+--------+--------+--------+
```

- UDP Checksum is computed over *pseudo-header* prepended to the UDP header
  - For IPv4: IP Source, IP Dest, Protocol (=17), plus UDP length
- Benefits? Problems?
  - Is UDP a layer on top of IP?

http://www.postel.org/pipermail/end2end-interest/2005-February/004616.html

7.1.2 Next Problem: Reliability

Problem	Mechanism
Dropped Packets	Acknowledgements + Timeout
Duplicate Packets	Sequene Numbers
Packets out of Order	Receiver Window
Keeping the pipe full	Sliding Window (Pipelining)

7.1.2.1 Transport Layer Reliability

- Extra difficulties
  - Multiple hosts
  - Multiple hops
  - Multiple potential paths
- Need for connection establishments, tear down

  - Analogy: dialing a number versus a direct line
- Varring RTTs
  - Both across connections and *during* a connection
  - Why do they vary? What do they influence?
- Out of order packets
  - Not only because of drops/retransmissions
  - Can get very old packets (up to 120s), must not get confused
- Unknown resources at other end
  - Must be able to discover receiver buffer: flow control
- Unknown resources in the network
  - Should not overload the network
  - But should use as much as safely possible
  - Congestion Control

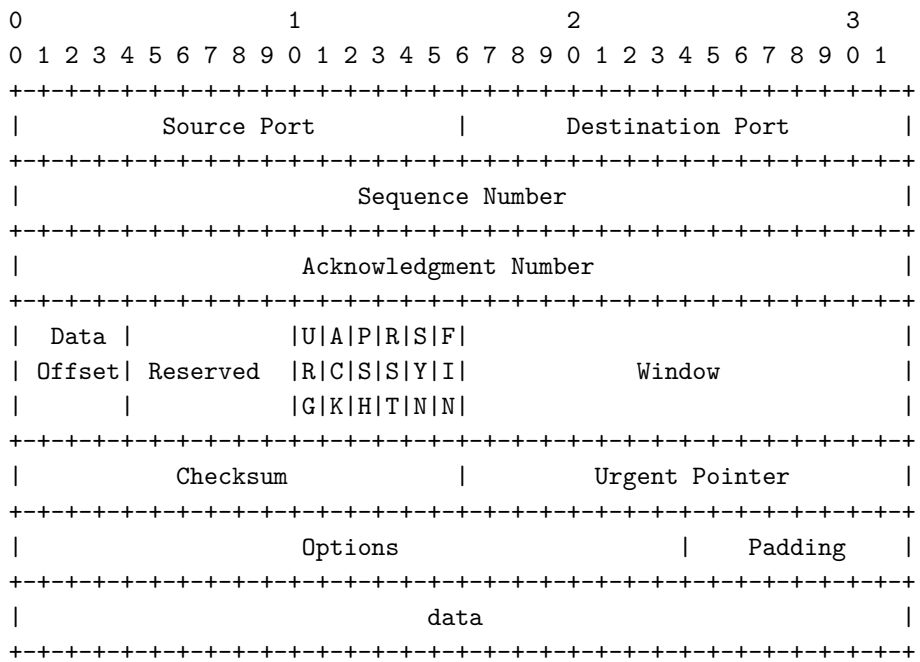
7.1.3 TCP – Transmission Control Protocol

- Service model: "reliable, connection oriented, full duplex byte stream"
  - Endpoints: <IP Address, Port>
- Flow control
  - If one end stops reading, writes at other eventually stop/fail
- Congestion control
  - Keeps sender from overloading the network

7.1.4 TCP

- Specification
  - RFC 793 (1981), RFC 1222 (1989, some corrections), RFC 5681 (2009, congestion control), ...
- Was born coupled with IP, later factored out
- End-to-end protocol
  - Minimal assumptions on the network
  - All mechanisms run on the end points
- Alternative idea:
  - Provide reliability, flow control, etc, link-by-link
  - Does it work?

7.1.4.1 TCP Header



7.1.4.2 Header Fields

- Ports: multiplexing
- Sequence number
  - Correspond to *bytes*, not packets!
- Acknowledgement Number
  - Next expected sequence number
- Window: willing to receive
  - Lets receiver limit SWS (even to 0) for flow control
- Data Offset: number of 4 byte header + option bytes
- Flags, Checksum, Urgen Pointer

7.1.4.3 Header Flags

- URG: whether there is urgent data
- ACK: ack no. valid (all but first segment)
- PSH: push data to the application immediately
- RST: reset connection
- SYN: synchronize, establishes connection
- FIN: close connection

7.1.4.4 Establishing a Connection

- Three-way handshake
- Two sides agree on respective initial sequence nums
- If no one is listening on port: server sends RST
- If server is overloaded: ignore SYN
- If no SYN-ACK: retry, timeout

7.1.4.5 Connection Termination

- FIN bit says no more data to send
  - Caused by close or shutdown
  - Both sides must send FIN to close a connection
- Typical close

7.1.4.6 TIME_WAIT

- Why do yo have to wait for 2MSL in TIME_WAIT?
  - What if last ACK is severely delayed, *and*
  - Same port pair is immediately reused for a new connection?
- Solution: active closer goes into TIME_WAIT
  - Waits for 2MSL (Maximum Segment Lifetime)
- Can be problematic for active servers
  - OS has too many sockets in TIME_WAIT, can accept fewer connections
    - Hack: send RST and dlete socket, SO_LINGER = 0
  - OS won't let you restart server because port in use
    - SO_REUSEADDR lets you rebind

7.1.4.7 Reliable Delivery

- TCP retransmits if data corrupted or dropped
  - Also retransmit if ACK lost
- When should TCP retransmit?
- Challenges in estimating RTT
  - Dynamic
  - No additional traffic

7.1.4.8 Smoothing RTT

- RTT measurement can have large variation
- Need to smooth the samples
  - One RTT measurement = one sample
- Some ways to smooth the sample
  - Average of the whole sequence
  - Windowed Mean

- Problems?

7.1.4.9 EWMA

- EWMA: Exponentially Weighted Moving Average
- Give greater weight to recent samples.
  - Why?

  https://en.wikipedia.org/wiki/Moving_average#Exponential_moving_average
- Estimate RTT
- $\text{RTT}(t) = \alpha\text{RTT}(t-1) + (1-\alpha)\texttt{newEst}$
- More generally, for a dataset $Y = Y_1, Y_2, \ldots$

$$S(t) = \begin{cases} Y_1 & t = 1 \\ \alpha Y_t + (1-\alpha)S(t-1) & t > 1 \end{cases}$$

Chapter 8

Flow and Congestion Control

8.1 Flow Control

8.1.1 First Goal

- We should not send more data than the receiver can take: *flow control*
- Data is sent in MSS-sized segments
  - Chosen to avoid fragmentation
- Sender can delay sends to get larger segments
- When to send data?
- How much data to send?

8.1.2 Flow Control

- Part of TCP specification (even before 1988)
- Goal: don't send more data than the receiver can handle
- Sliding window protocol
- Receiver uses window header field to tell sender how much space it has
- Receiver:

$$\texttt{AdvertisedWindow} = \texttt{MaxRcvBuffer} - ((\texttt{NextByteExpected} - 1) - \texttt{LastByteRead})$$

- Sender:

$$\texttt{LastByteSent} - \texttt{LastByteAcked} \leq \texttt{AdvertisedWindow}$$

$$\texttt{EffectiveWindow} = \texttt{AdvertisedWindow} - \texttt{BytesInFlight}$$

$$\texttt{LastByteWritten} - \texttt{LastByteAcked} \leq \texttt{MaxSendBuffer}$$

- Advertised window can fall to 0
  - How?
  - Sender eventually stops sending, blocks application
- Sender keeps sendnig 1-byte segments until windows comes back > 0
- 50 students have ssh window open to bayou and are typing 1 character per second
- How many packets are read and written by bayou per second?
  - Consider minimum frame size

8.1.2.1 When to Transmit?

Algorithm Nagle's Algorithm – reduce the overhead of small packets

```
1: if available data and window ≥ MSS:
2:     send an MSS segment
3: else:
4:     if there is unAcked data in flight:
5:         buffer the new data until ACK arrives
6:     else:
7:         send all new data now
```

- Receiver should avoid advertising a window $\leq$ `MSS` after advertising a window of 0

https://tools.ietf.org/html/rfc896

8.1.3 Delayed Acknowledgements

- Goal: piggy-back ACKs on data
  - Delay ACK for 200ms in case application sends data
  - If more data received, immediately ACK second segment
  - Note: never delay duplicate ACKs (if missing a segment)
- Warning: can interact *very* badly with Nagle
  - Temporary deadlock
  - Can disable Nagle with TCP_NODELAY
  - Application can also avoid many small writes

https://en.wikipedia.org/wiki/TCP_delayed_acknowledgment
https://developers.slashdot.org/comments.pl?sid=174457&cid=14515105

8.1.3.1 Turning off Nagle's Algorithm

"In general, since Nagle's algorithm is only a defense against careless applications, disabling Nagle's algorithm will not benefit most carefully written applications that take proper care of buffering. Disabling Nagle's algorithm will enable the application to have many small packets in flight on the network at once, instead of a smaller number of large packets, which may increase load on the network, and may or may not benefit the application performance."

- Who wants to turn the algorithm off?
  - Search on Google and find out.

https://en.wikipedia.org/wiki/Nagle's_algorithm

8.1.4 Limitations of Flow Control

- Network may be the bottleneck
- Signal from receiver not enough!
- Sending too fast will cause queue overflows, heavy packet loss
- Flow control provides *correctness*
- Need more for performance: congestion control

8.1.5 A Short History of TCP

- 1974: 3-way handshake
- 1978: IP and TCP split
- 1983: January 1st, ARPAnet switches to TCP/IP
- 1984: Nagle predicts congestion collapses
- 1986: Internet begins to suffer congestion collapses
  - LBL to Berkeley drops from 32Kbps to 40bps
- 1987/8: Van Jacobsen fixes TCP, publishes sminal paper: (TCP Tahoe)
- 1990: Fast transmit and fast recovery added (TCP Reno)

8.1.6 Second Goal

- We should not send more data than the network can take: *congestion control*

8.2 TCP Congestion Control

- 3 Key Challenges

  - Determining the available capacity in the first place
  - Adjusting to changes in the available capacity
  - Sharing capacity between flows
- Idea
  - Each source determines network capacity for itself
  - Rate is determined by window size
  - Uses implicit feedback (drops, delay)
  - ACKs page transmission (self-clocking)

8.2.1 Dealing with Congestion

- TCP keeps congestion and flow control windows
  - Max packets in flight is lesser of two
- Sending rate: $\approx$ `Window/RTT`
- The key here is how to set the congestion window to respond to congestion signals

8.2.2 Starting Up

- Before TCP Tahoe
  - On connection, nodes send full (rcv) window of packets
  - Retransmit packet immediately after its timer expires
- Result: window-sized bursts of packets in network

8.2.3 Determining Initial Capacity

- Question: how to set w initially?
  - Should start at 1MSS (to avoid overloading the network)
  - Could increase additively until we hit congestion
  - May be too slow on fast network
- Start by doubling with each RTT
  - Then will dump at most one extra window int network
  - This is called *slow start*
- *Slow start*, this sounds quite fast!
  - In contrast to initial algorithm: sender would dump entire *control flow* window at once

Chapter 9

Flow and Congestion Control (continued)

9.1 Congestion Control

9.1.1 Slow Start Implementation

- Let w be the size of the window in *bytes*
  - We have $w/$`MSS` segments per RTT
- We are doubling w after each RTT
  - We receive $w/$`MSS` ACKs each RTT
  - So we can set $w = w +$ `MSS` on every ACK
- At some point, we hit the network limit

  - Experience loss
  - We are at most one window size above the limit
  - Remember this: `ssthresh` and reduce window

9.1.2 Slow Start

- We double `cwnd` every round trip
- We are still sending min(`cwnd`, `rcvwnd`) packets
- Continue until `ssthresh` estimate or packet drop

9.1.3 Dealing with Congestion

- Assume losses are due to congestion
- After a loss, reduce congestion window
  - *How much to reduce?*
- Idea: conservation of packets at equilibrium
  - Want to keep roughly the same number of packets network
  - Analogy with water in fixed-size pipe
  - Put new packet into network when one exits

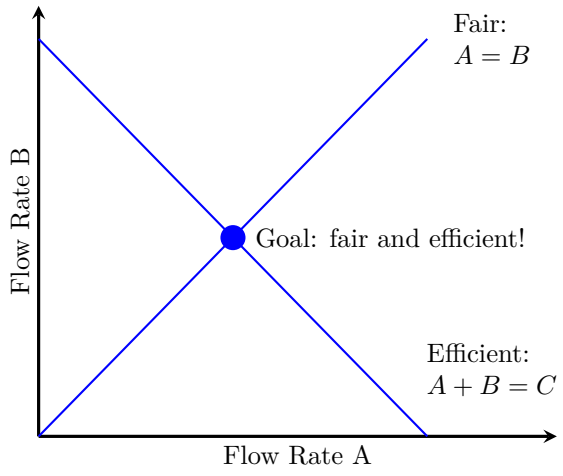
9.1.3.1 How much to reduce window?

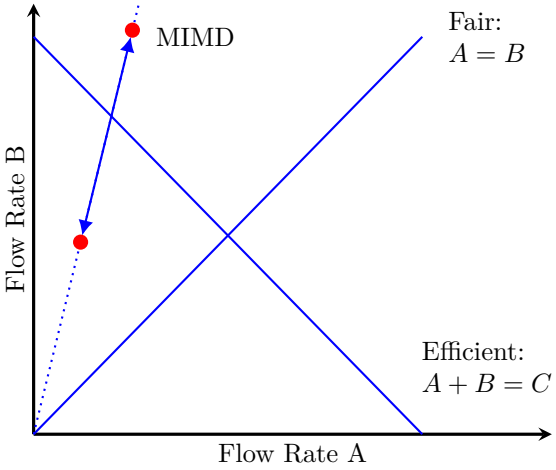
- What happens under congestion?
  - Exponential increase in congestion
- Sources must decrease offered rate exponentially
  - i.e., multiplicative decrease in window size
  - TCP chooses to cut window in half

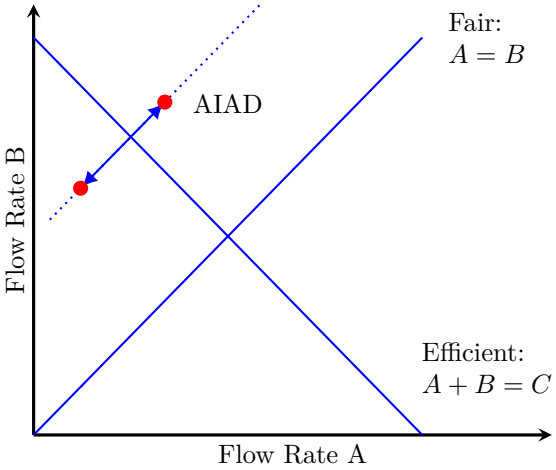
9.1.3.2 How to use extra capacity?

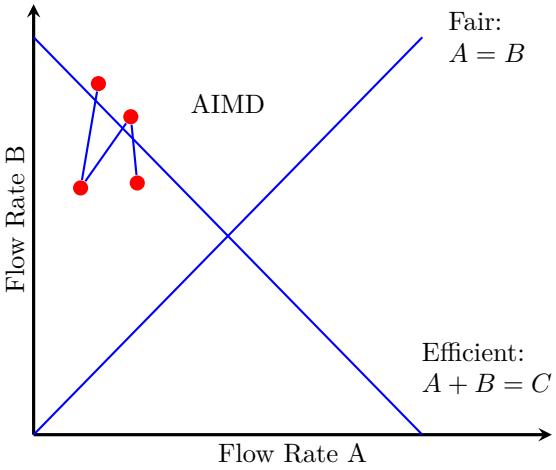
- Network signals congestion, but says nothing of underitilization
  - Senders constantly try to send faster, see if it works
  - So, increase window if no losses... By how much?
- Multiplicative increase?
  - Easier to saturate the network than to recover
  - Too fast, will lead to saturation, wild fluctuations
- Additive Increase?
  - Won't saturate the network

9.1.3.3 Chiu Jain Phase Plots









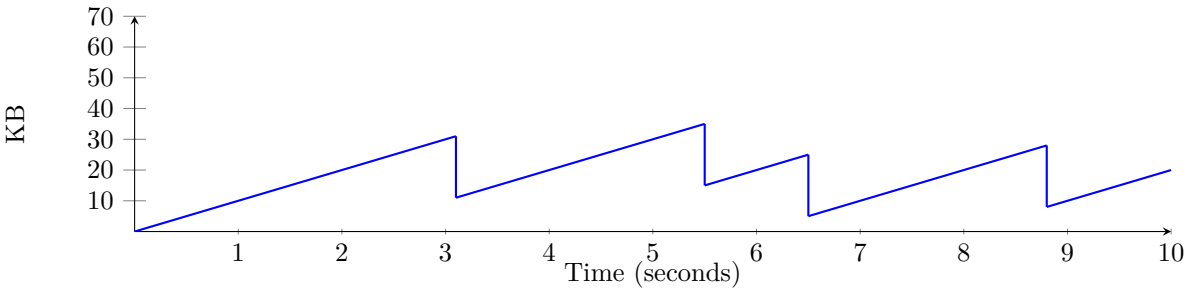
9.1.3.4 AIMD Implementation

- In practice, send `MSS`-sized segments
  - Let window size in bytes be w (a multiple of `MSS`)
- Increase:
  - After w bytes ACKed, could set $w = w + \texttt{MSS}$
  - Smoother to increment on each ACK
    - $w = w + \texttt{MSS} \times \texttt{MSS}/w$
    - (receive $w/\texttt{MSS}$ ACKs per RTT, increase by $\frac{\texttt{MSS}}{w/\texttt{MSS}}$ for each)

  - Decrease:
    * After a packet loss, $w = w/2$
    * But dont want $w <$ `MSS`
    * So react differently to multiple consecutive losses
    * Back off exponentially (pause with no packets in flight)

9.1.3.5 AIMD Trace

- AIMD produces sawtooth pattern of window size
  - Always probing available bandwidth



9.1.3.6 Putting it Together

- TCP has two states: Slow Start (SS) and Congestion Avoidance (CA)
- A window size threshold governs the state transition
  - Window $\leq$ threshold: SS
  - Window $>$ threshold: CA
- States differ in how they respond to ACK
  - Slow start $w = w +$ `MSS`
  - Congestion Avoidance: $w = w + \texttt{MSS}^2/w$ (1 `MSS` per RTT)
- On loss event: set $w = 1$, slow start

9.1.4 How to Detect Loss

- Timeout
- Any other way?
  - Gap in sequence numbers at receiver
  - Receiver uses cumulative ACKs: drops $\Rightarrow$ duplicate ACKs
- 3 duplicate ACKs considered loss

9.1.5 RTT

- We want an estimate of RTT so we can know a packet was likely lost, and not just delayed
- *Key for correct operation*
- Challenge: RTT can be highly variable
  - Both at long and short time-scales!
- Both average and variance increase a lot with load
- Solution
  - Use exponentially weighted moving average (EWMA)
  - Estimate deviation as well as expected value
  - Assume packet is lost when time is well beyond reasonable deviation

9.1.5.1 Originally

- `EstRTT` $= (1 - \alpha) \times$ `EstRTT` $+ \alpha$`SampleRTT`
- `Timeout` $= 2 \times$ `EstRTT`
- Problem 1:
  - in case of retransmission, ACK corresponds to which send?
  - Solution: only sample for segments with no retransmission
- Problem 2:
  - does not take variance into account: too aggressive when there is more load!

9.1.5.2 Jacobson/Karels Algorithm (Taho)

- $\texttt{EstRTT} = (1 - \alpha) \times \texttt{EstRTT} + \alpha \times \texttt{SampleRTT}$
  - Recommended α is 0.125
- $\texttt{DevRTT} = (1 - \beta) \times \texttt{DevRTT} + \beta|\texttt{SampleRTT} - \texttt{EstRTT}|$
  - Recomended β is 0.25
- $\texttt{Timeout} = \texttt{EstRTT} + 4 \cdot \texttt{DevRTT}$
- For successive retransmissions: use exponential backoff

9.1.5.3 Slow start every time?!

- Losses have large effect on throughput
- Fast Recovery (TCP Reno)
  - Same as TCP Tahoe on Timeout: $w = 1$, slow start
  - On triple duplicate ACKs: $w = w/2$
  - Retransmit missing segment (fast retransmit)
  - Stay in Congestion Avoidance mode

9.1.5.4 3 Challenges Revisited

- Determining the available capacity in the first place
  - Exponential increase in congestion window
- Adjusting to changes in the available capacity
  - Slow probing, AIMD
- Sharing capacity between flows
  - AIMD
- Detecting Congestion
  - Timeout based on RTT
  - Triple duplicate acknowledgements
- Fast retransmit/Fast recovery
  - Reduces slow starts, timeouts

Chapter 10

TCP Friendliness and Getting Help from the Network

10.1 TCP Friendliness

- Can other protocols co-exist with TCP?
  - e.g. if you want to write a video streaming app using UDP, how to do congestion control?
- Equation-based Congestion Control
  - Instead of implementing TCP's CC, estimate the rate at which TCP would send. Function of what?
  - RTT, MSS, Loss
- Measure RTT, Loss, send at that rate!

10.1.1 TCP Throughput

- Assume a TCP connection of window W, rount-trip time of `RTT`, segment size of `MSS`
  - Sending Rate $S = W \times \texttt{MSS}/\texttt{RTT}$ (1)

- Drop $W = W/2$
  - grows by $\texttt{MSS}W/2$ RTTs, until another drop at $W \approx W$
- Average window then $0.75 \times S$
  - From (1), $S = 0.75W\texttt{MSS}/\texttt{RTT}$ (2)
- Loss rate is 1 in number of packets between losses:

$$\begin{aligned}\texttt{Loss} &= \frac{1}{1 + (W/2 + W/2 + 1 + W/2 + 2 + \cdots + W)} \\ &= \frac{1}{3/8W^2} \quad (3)\end{aligned}$$

- $\texttt{Loss} = 8/(3W^2) \implies W = \sqrt{\frac{8}{3 \cdot \texttt{Loss}}}$ (4)
- Substituting (4) in (2), $S = 0.75W\texttt{MSS}/\texttt{RTT}$

$$\texttt{Throughput} \approx 1.22 \times \frac{\texttt{MSS}}{\texttt{RTT} \cdot \sqrt{\texttt{Loss}}}$$

- Equation-based rate control can be TCP friendly and have better properties, e.g., small jitter, fast ramp-up...

$$W = \sqrt{\frac{8}{3p}}$$

Substitute W into the bandwidth equation below:

$$\texttt{BW} = \frac{\text{data per cycle}}{\text{time per cycle}} = \frac{\texttt{MSS} \cdot \frac{3}{8}W^2}{\texttt{RTT} \cdot \frac{W}{2}} = \frac{\texttt{MSS}/p}{\texttt{RTT}\sqrt{\frac{2}{3p}}}$$

Collect the constants in one term, $C = \sqrt{3/2}$, then we arrive at

$$\texttt{BW} = \frac{\texttt{MSS}}{\texttt{RTT}} \frac{C}{\sqrt{p}}$$

10.1.2 What happens when Link is Lossy

- Throughput $= 1/\sqrt{\text{Loss}}$

10.1.2.1 What can we do about it?

- Two types of losses: congestion and corrupt
- One option: mask corruption losses from TCP
  - Retranmissions at the link layer
  - e.g. snoop TCP: intercept duplicate acknowledgments, retransmit locally, filter them from the sender
- Another option:
  - Tell the sender about the cause for the drop
  - Requires modification of the TCP endpoints

10.1.3 Congestion Avoidance

- TCP creates congestion to then back off
  - Queues at bottleneck link are often full: increased delay
  - Sawtooth pattern: jitter
- Alternative strategy
  - Predict when congestion is about to happen
  - Reduce rate early
- Two approaches
  - Host centric: TCP vegas
  - Router-centric: RED, DECBit

10.1.3.1 TCP Vegas

- Idea: source watches for sign that router's queue is building up (e.g. sending rate flattens)
- Compare Actual Rate (A) with Expected Rate (E)
  - If $E - A > \beta$, decrease `cwnd` linearly: A isn't responding
  - If $E - A < \alpha$, increase `cwnd` lienarly: room for A to grow

10.1.3.2 Vegas

- Shorter router queues

- Lower jitter
- Problem:
  - Doesn't compete well with Reno. Why?
  - Reacts earlier, Reno is more aggressive, ends up with higher bandwidth...

10.2 Help from the network

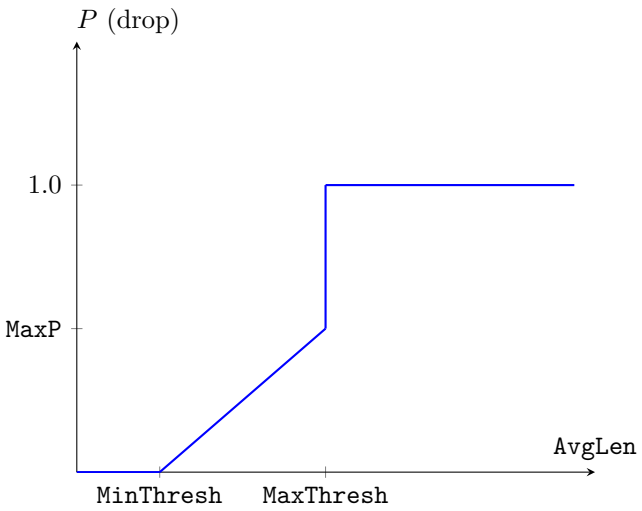
- What if routers could *tell* TCP that congestion is happening?
  - Congestion causes queues to grow: rate mismatch
- TCP responds to drops
  - Idea: Random Early Drop (RED)
    - Rather than wait for queue to become full, drop packet with some probability that increases with queue length
    - TCP will react by reducing `cwnd`
    - Could also mark instead of dropping: ECN

10.2.1 RED Details

- Computer average queue length (EWMA)
  - Don't want to react to very quick fluctuations

10.2.1.1 RED Drop Probability

- Define two thresholsd: `MinThresh`, `MaxThresh`
- Drop probability:



- Improvements to spread drops (see book)

10.2.1.2 RED Avantages

- Probability of dropping a packet of a particular flow is roughly proportional to the share of the bandwidth that flow is currently getting
- Higher network utilization with low delays
- Average queue length small, but can absorb bursts
- ECN
  - Similar to RED, but router sends bit in the packet
  - Must be supported by both ends
  - Avoids retransmissions optionally dropped packets

10.2.2 More help from the network

- Problem: still vulnerable to malicious flows!
  - RED will drop packets from large flows preferentially, but they don't have to respond appropriately
- Idea: Multiple Queues (one per flow)
  - Serve queues in Round-Robin
  - Nagle (1987)

  - Good: protects against misbehaving flows
  - Disadvantage?
  - *Flows with larger packets get higher bandwidth*

Chapter 11

TCP Friendliness and Getting Help from the Network (Continued)

11.1 Help from the network

- Problem: still vulnerable to malicious flows!
  - RED will drop packets from large flows preferentially, but they don't have to respond appropriately
- Idea: Multiple Queues (one per flow)
  - Serve queues in Round-Robin
  - Nagle (1987)
  - Good: protects against misbehaving flows
  - Disadvantage?
  - *Flows with larger packets get higher bandwidth*

11.1.1 Solution

- Bit-by-bit round robin
- Can we do this?
  - No, packets cannot be preempted!
- we can only approximate it...

11.1.2 Fair Queueing

- Define a *fluid flow* system as one where flows are served bit-by-bit
- Simulate *ff* and serve packets in the order in which they would finish in the *ff* system
- Each flow will receive exactly its fair share

11.1.2.1 Implementing Fair Queueing

- Suppose clock ticks with each bit transmitted
  - (RR, among all active flows)
- P_i is the length of the packet
- S_i is packet i's start of transmission time
- F_i is packet i's end of transmission time
- $F_i = S_i + P_i$
- Across all flows
  - Calculate F_i for each packet that arrives on each flow
  - Next packet to transmit is that with the lowest F_i
  - Clock rate depends on the number of flows
- Advantages
  - Achieves *max-min fairness*, independent of sources
  - Work conserving

- Disadvantages
  - Requires non-trivial support from routers
  - Requires reliable identification of flows
  - Not perfect: can't preempt packets

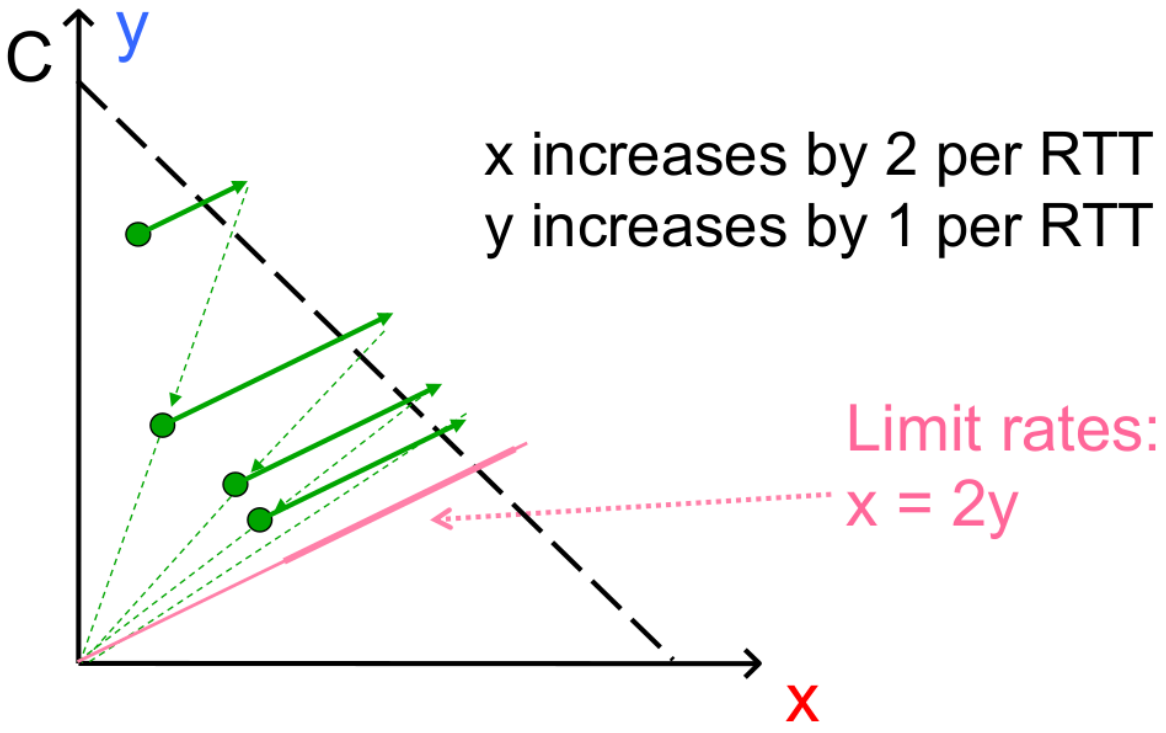
11.1.2.2 Big Picture

- Fair Queueing doesn't eliminate congestion: just manages it
- You need both, ideally:
  - End-host congestion control to adapt
  - Router congestion control to provide isolation

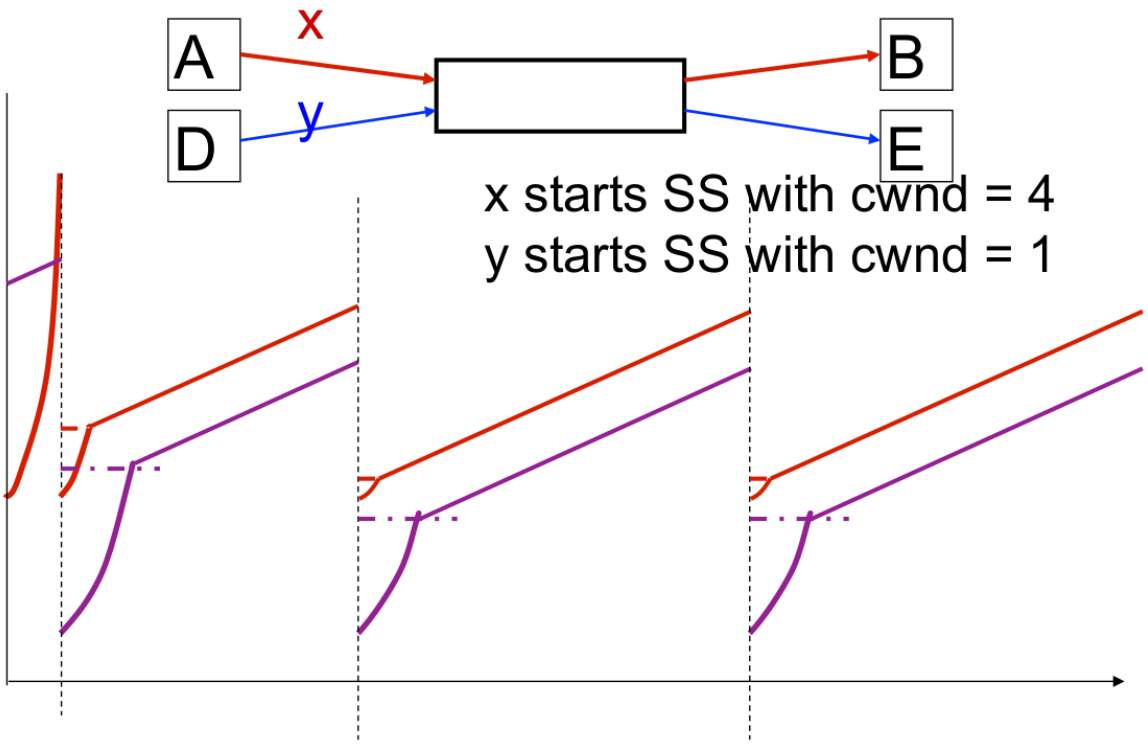
11.1.3 Cheating TCP

- Three possible ways to cheat
  - Increase `cwnd` faster
  - Large initial `cwnd`
  - Opening many connections
  - ACK Division Attack

11.1.3.1 Increasing `cwnd` Faster

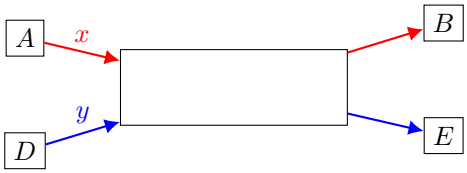


11.1.3.2 Larger Initial Window



11.1.3.3 Open Many Connections

- Web Browser: has to download k objects for a page
  - Open many connections or download sequentially?



  - Assume:
    - A opens 10 connections to B
    - B opens 1 connection to E
  - TCP is fair among connections
    - A gets 10 times more bandwidth than B

11.1.3.4 Exploiting Implicit Assumptions

- Savage, et al., CCR 1999
  - TCP Congestion Control with a Misbehaving Receiver
  - https://cseweb.ucsd.edu/~savage/papers/CCR99.pdf
- Exploits ambiguity of meaning of ACK
  - ACKs can specify any byte range for error control
  - Congestion control assumes ACKs cover entire sent segments

11.1.3.5 ACK Division Attack

- Receiver: *"upon receiving a segment with N bytes, divide the bytes into M groups and ackowledge each group separately"*
- Sender will grow M times faster
- Could cause growth to 4GB in 4 RTTs!
  - $M = N = 1460$

11.1.3.6 Defense

- Appropriate Byte Counting
  - [RFC 3465 (2003), RFC 5681 (2009)]
  - In slow start, `cwnd`+ = min(N, `MSS`) where N is the number of newly acknowledged bytes in the received ACK

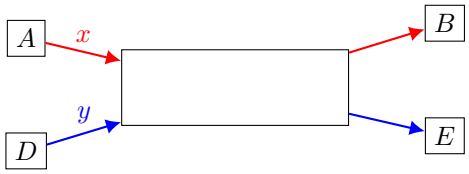
11.1.3.7 DupACK Spoofing

- Receiver: *"Upon receiving a data segment, the receiver sends a long stream of acknowledgments for the last sequence number received"*
- Sender sends at a rate proportional to the ACK rate

11.1.3.8 Optimistic ACKing

- Receiver: *"Upon receiving a data segment, the receiver sends a stream of acknowledgments anticipating data that will be sent by the sender"*

11.1.3.9 Cheating TCP and Game Theory



A \ D	Increases by 1	Increases by 5
Increases by 1	22, 22	10, 35
Increases by 5	35, 10	15, 15

Too aggressive
→Losses
→Throughput falls

Individual incentives: cheating pays
Social incentives: better off without cheating