

UNIVERSITY OF HOUSTON

INTRODUCTION TO COMPUTER NETWORKS

COSC 6377

Midterm Review

Author

K.M. HOURANI

Based on Notes By

Dr. Omprakash GNAWALI

March 4, 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: Communication 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 <code>getaddrinfo</code>	16
2.2.14	<code>getaddrinfo()</code> [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 \leftrightarrow 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
4.2.11	Some important details	28
5	DNS and P2P	31
5.1	DNS	31
5.1.1	Structure of a DNS Message	31
5.1.1.1	Header Format	31
5.1.2	Other RR Types	32
5.1.3	Inserting a Record in DNS	32
5.1.4	DNS Security	32
5.1.4.1	Cache Poisoning	32
5.1.5	Guessing Query ID	34
5.1.6	Cache Poisoning	35
5.1.7	Hijacking Authority Record	36

5.1.8	Kaminsky Exploit	36
5.1.8.1	Countermeasures	37
5.1.9	Load Balancing using DNS	37
5.2	Peer-to-Peer	37
5.2.1	Client-Server Bottlenecks	37
5.2.2	Peer-to-Peer Systems	37
5.2.2.1	3 Key Requirements	37
5.2.3	Napster	38
5.2.4	Gnutella: Flooding on Overlays (2000)	38
5.2.5	BitTorrent	38
5.2.5.1	BitTorrent Tracker Files	38
5.2.6	Skype	39
6	Structured P2P and the Transport Layer	41
6.1	Structured P2P Systems	41
6.1.1	DHTs	41
6.1.1.1	Consistent Hashing	41
6.1.1.2	Consistent Hashing Properties	42
6.1.1.3	Lookup	42
6.1.1.4	Joining	42
6.2	Transport Layer	43
6.2.1	Network Applications	43
6.2.2	Transport Layer	43
6.2.3	Error Detection	43
6.2.3.1	Parity Bit	43
6.2.3.2	2-D Parity	43
6.2.3.3	Checksum	44
6.2.3.4	How good is it?	44
6.2.3.5	CRC – Error Detection with Polynomials	44
6.2.4	Reliable Delivery	44
6.2.4.1	At Least Once Semantics	44
6.2.4.2	Stop and Wait Problems	45
6.2.4.3	At Most Once Semantics	45
6.2.5	Sliding Window Protocol	45
6.2.5.1	Sliding Window Sender	45

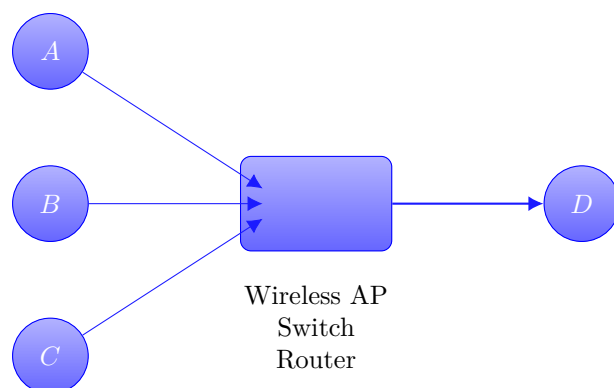
Chapter 1

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

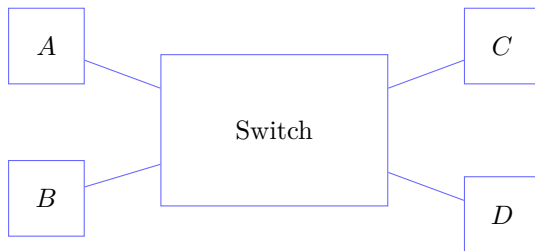
1.2 Packet vs. Circuit Switching



1.2.1 Circuit Switching

- Setup the connection or resource

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



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

- Natural for predictable data rates
- 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

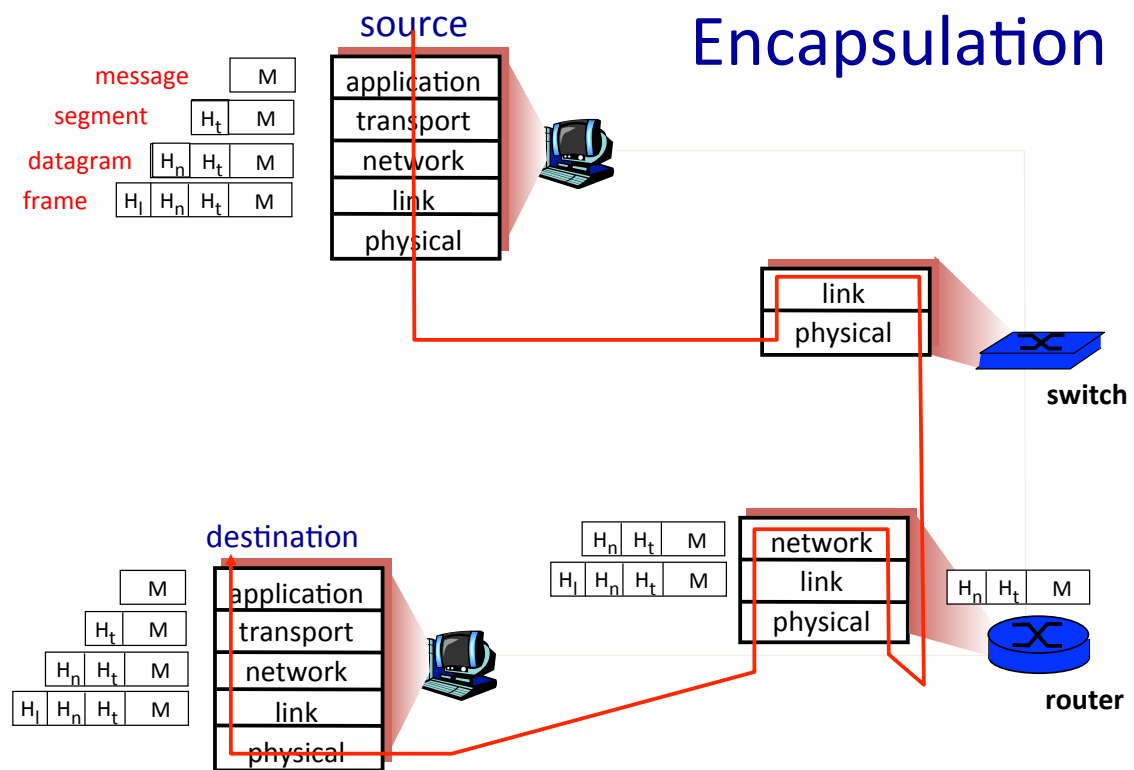
1.3.3 Reliability

- 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:
 - Hello
 - How are you?
 - Fine.

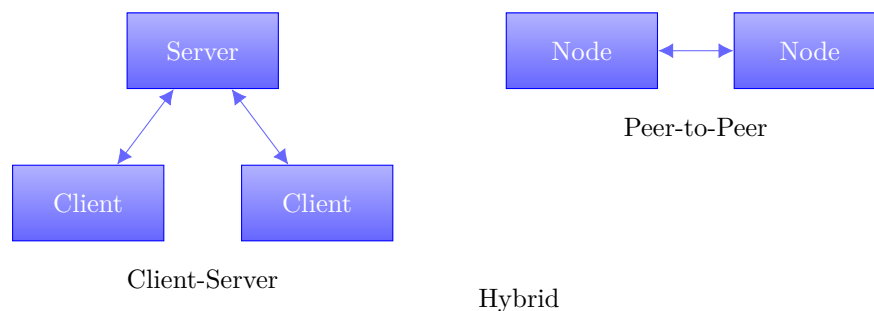
https://en.wikipedia.org/wiki/Communication_protocol



Chapter 2

Network Applications and Socket Programming

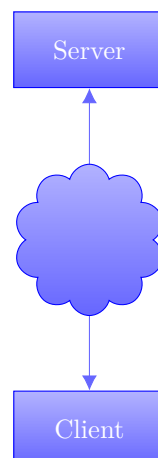
2.1 Network Applications



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

2.1.1 Inter-Application Communication

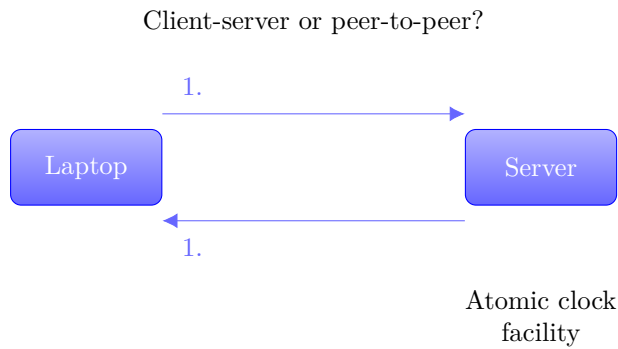
- Need a way to send and receive messages
- Inter-process communication
- Need naming, routing, transport
- Transport using TCP and UDP
 - On top of IP



2.1.2 Application Protocols

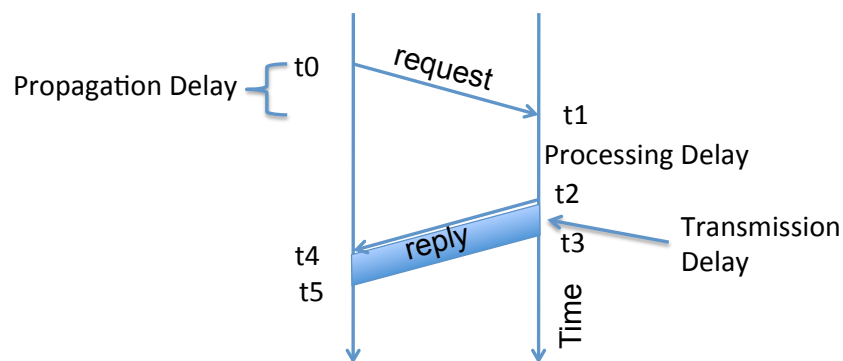
- Messages between processes, typically encapsulated within TCP or UDP
- Need agreement between
 - Sending process
 - Receiving process

2.1.3 Network Time Service



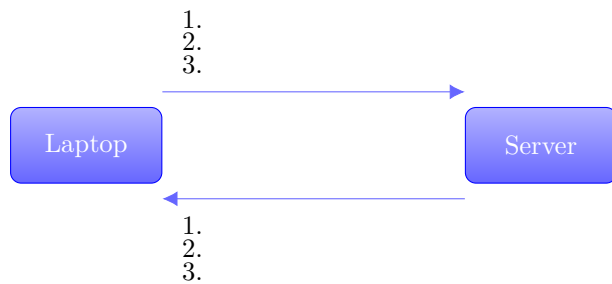
2.1.3.1 Protocol Timing Diagram

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?

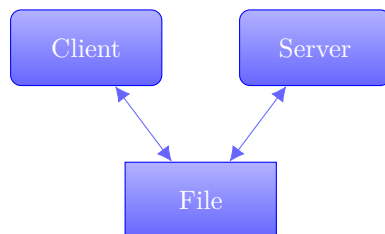


2.2 Socket Programming

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 whence);`
 - `whence`: `SEEK_SET`, `SEEK_CUR`, `SEEK_END`
 - returns new offset, or `-1` on error
- `int close(int fd);`

2.2.6 Sockets: Communication 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

<u>Client</u>	<u>Server</u>
1.	<code>socket</code> – make socket
2.	<code>bind</code> – assign address, port
3.	<code>listen</code> – listen for clients
4. <code>socket</code> – make socket	
5. <code>bind</code> – assign address ¹	
6. <code>connect</code> – connect to listening socket	
7.	<code>accept</code> – 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

¹This 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", gai_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` $\approx 100\mu\text{sec}$
 - 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 through 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>

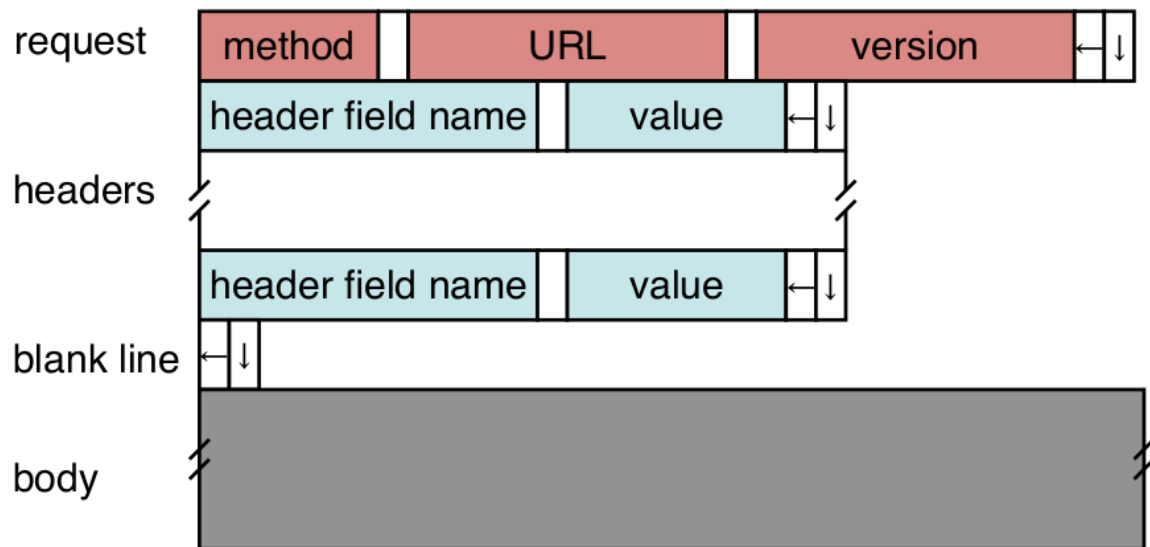
3.2 HTTP

- Important properties
 - Client-server protocol
 - Protocol (but not data) in ASCII
 - Stateless
 - 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!

HTTP Request



- Method:
 - **GET**: current value of resource, run program
 - **HEAD**: return metadata associated 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

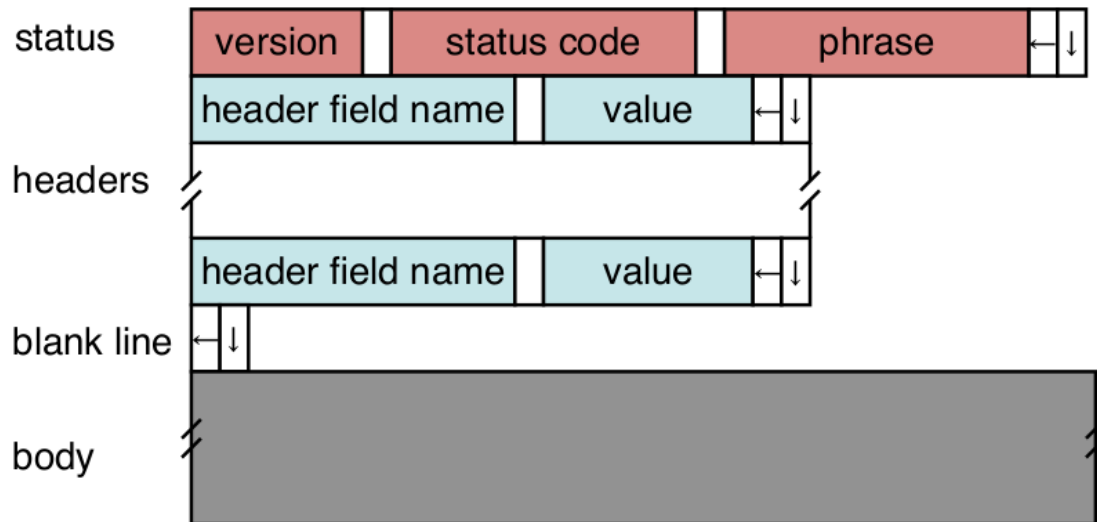
```
GET / HTTP/1.1
Host: localhost:8000
User-Agent: Mozilla/5.0 (Macinto ...
Accept: text/xml,application/xml ...
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
(empty line)
```

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...>
```

HTTP Response



- 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 specification?

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
```

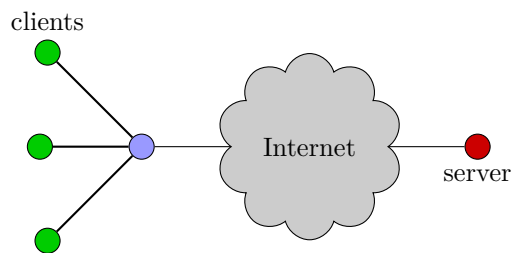
```
Host: localhost:8000
User-Agent: Mozilla/5.0 (Macinto ...
Accept: text/xml,application/xml ...
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection: keep-alive
```

- 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.uk`
- 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 ↔ 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.)

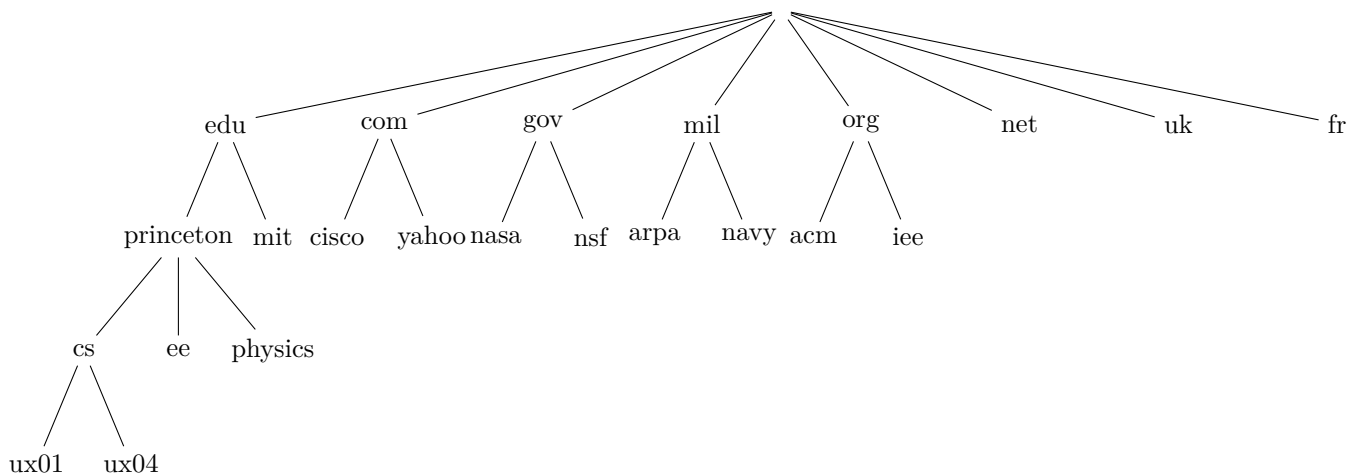
4.1.3 Goals for an Internet-scale name system

- 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 aggressively

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

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
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)
 - Type 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

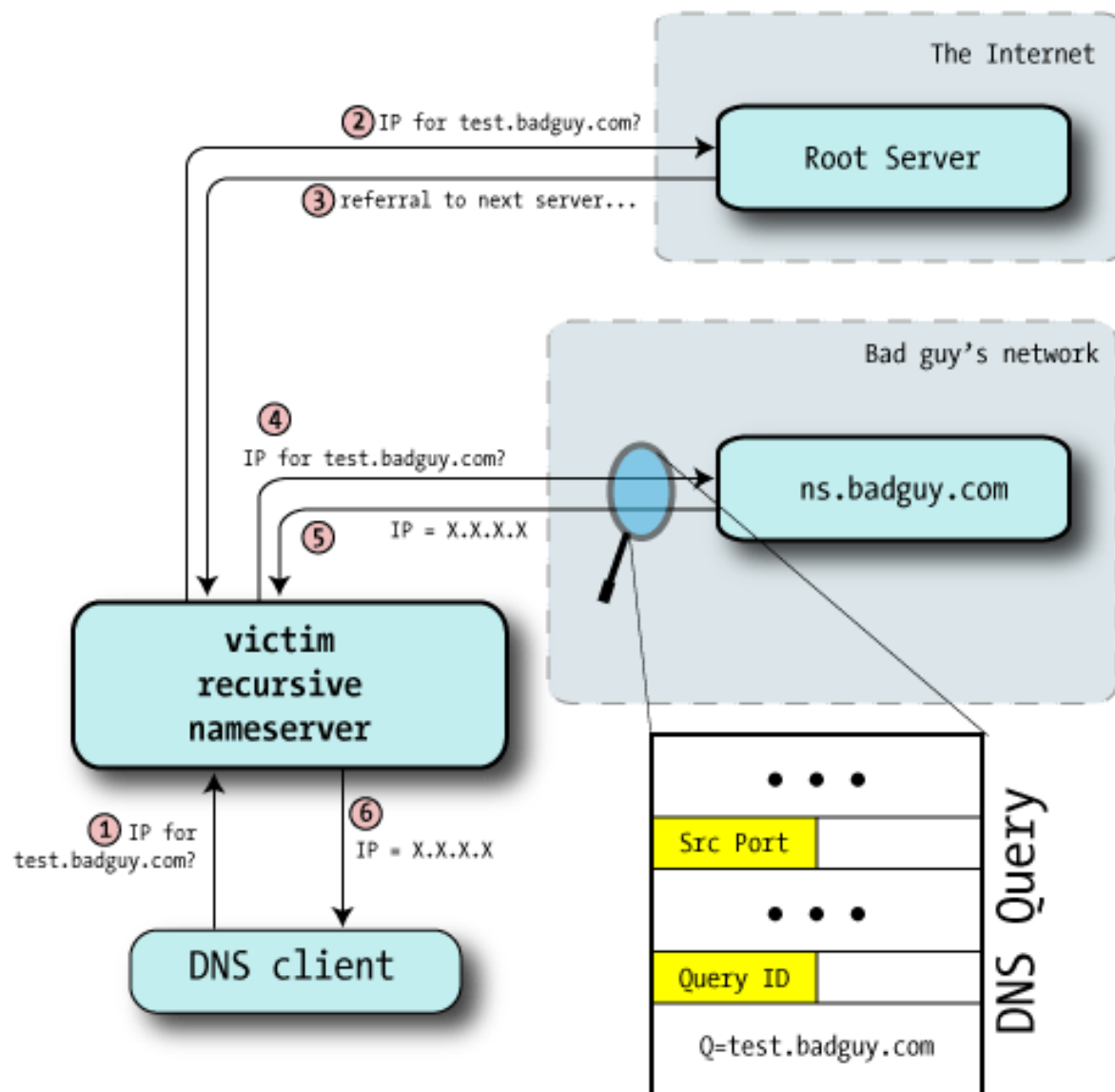
;; AUTHORITY SECTION:
evil.com.                    600     IN      NS      dns1.evil.com.
evil.com.                    600     IN      NS      google.com.

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

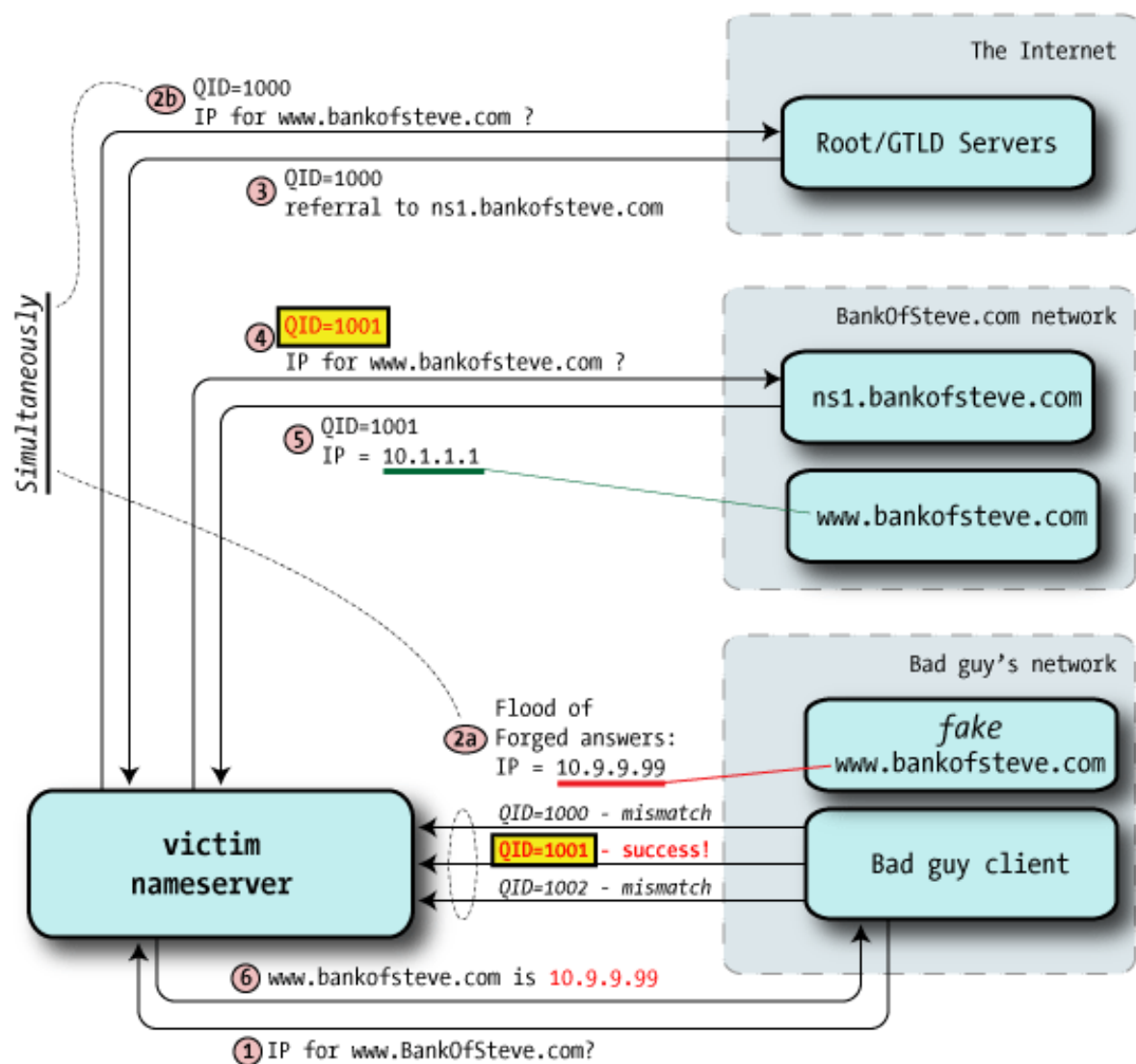
```

- 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

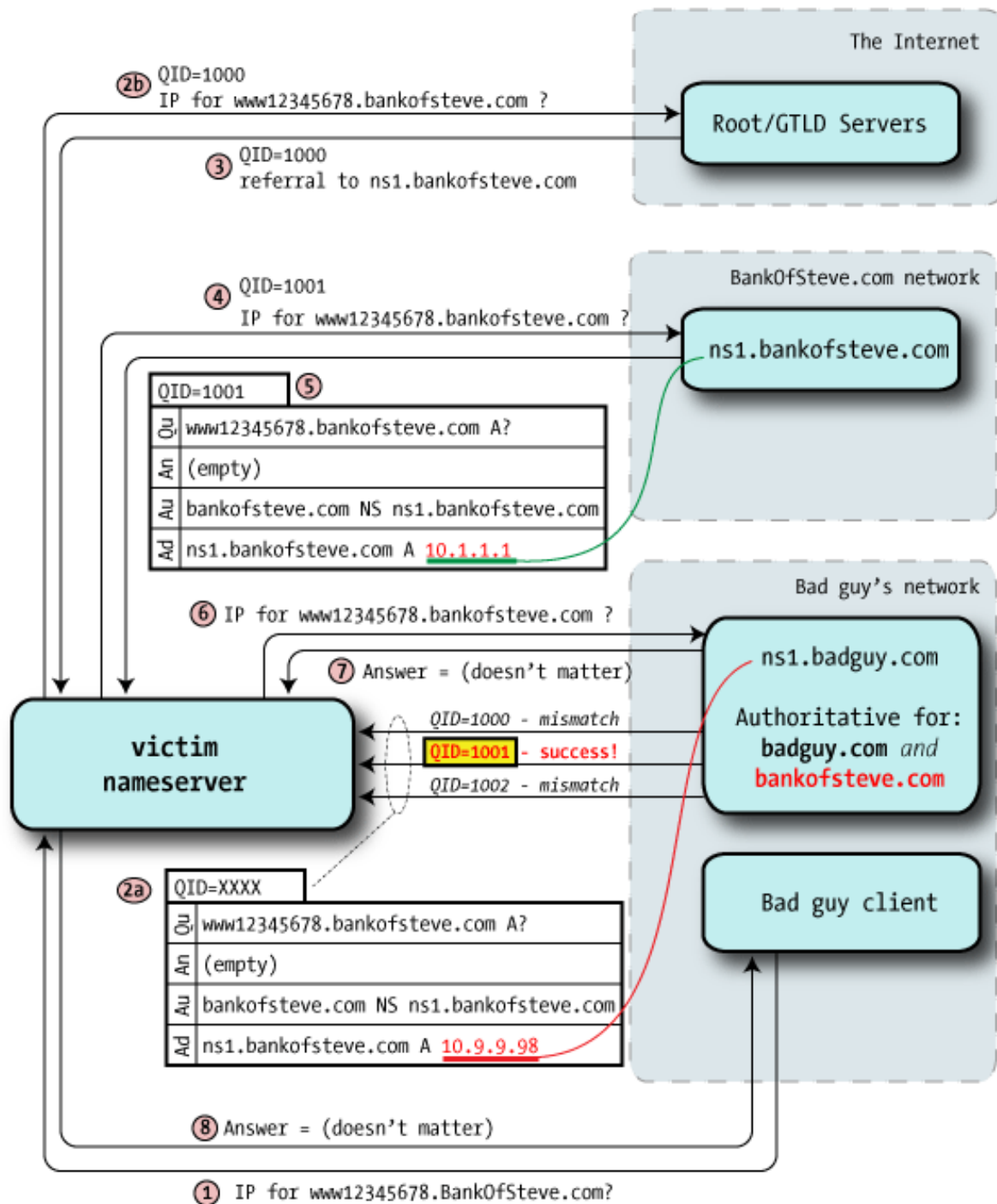
5.1.5 Guessing Query ID



5.1.6 Cache Poisoning



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-
list1139: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:...
```

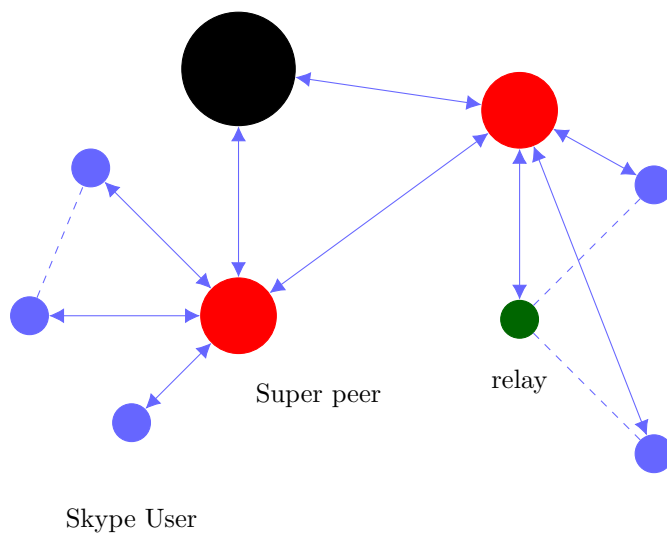
Example tracker from ubuntu.com

- 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

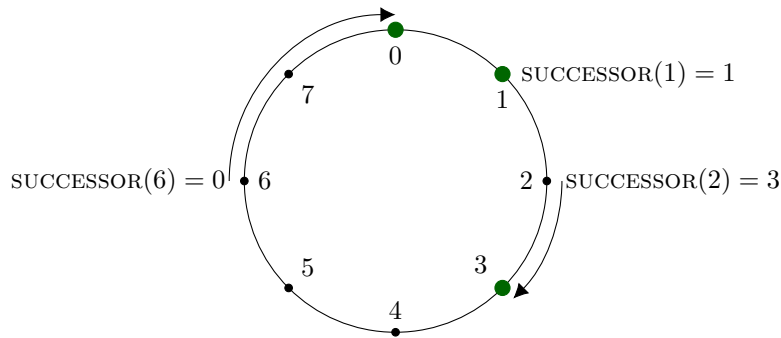
- Distributed Hash Table (DHT)
 - Efficient (**Key**, **Value**) storage
 - Approach: map the ID to a host
- Challenges
 - Scale to millions of nodes
 - Churn
 - Heterogeneity

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

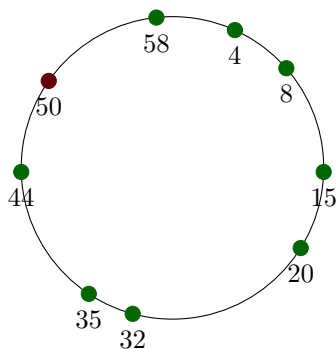
- Designed for node join/leave with minimal churn in key mapping
- k/n keys per node
- k/n keys change hands during join/leave

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 its successor to 50
- Node 44 sends a stabilize message to its new successor, node 50
- Node 50 sets its predecessor to node 44



6.2 Transport Layer

6.2.1 Network Applications

- Centralized and Peer-to-peer architectures
- 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

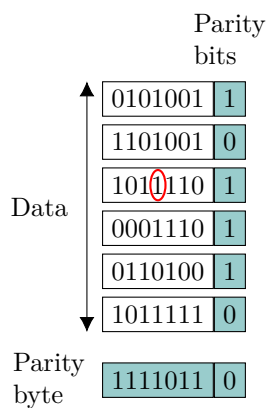
6.2.3.1 Parity Bit

- Can detect odd number of bit errors
- No correction

Data 1101101
 Parity **1**
 Transmit 1101101**1**

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: $LFS - LAR \leq SWS$
- 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: $LAF - LFR \leq RWS$
- Frame SeqNum arrives:
 - if $LFR < SeqNum \leq LAF$, accept
 - if $SeqNum \leq LFR$ or $SeqNum > LAF$, discard
- Send *cumulative* ACKs