CS118 Discussion 1B, Week 2

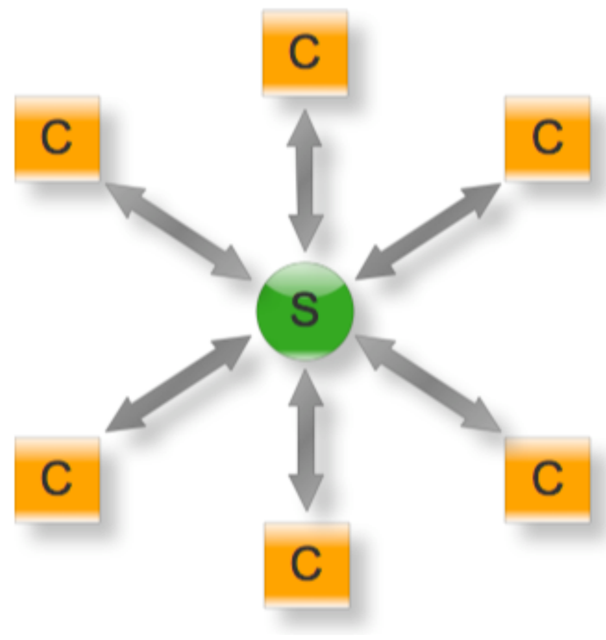
Boyan Ding

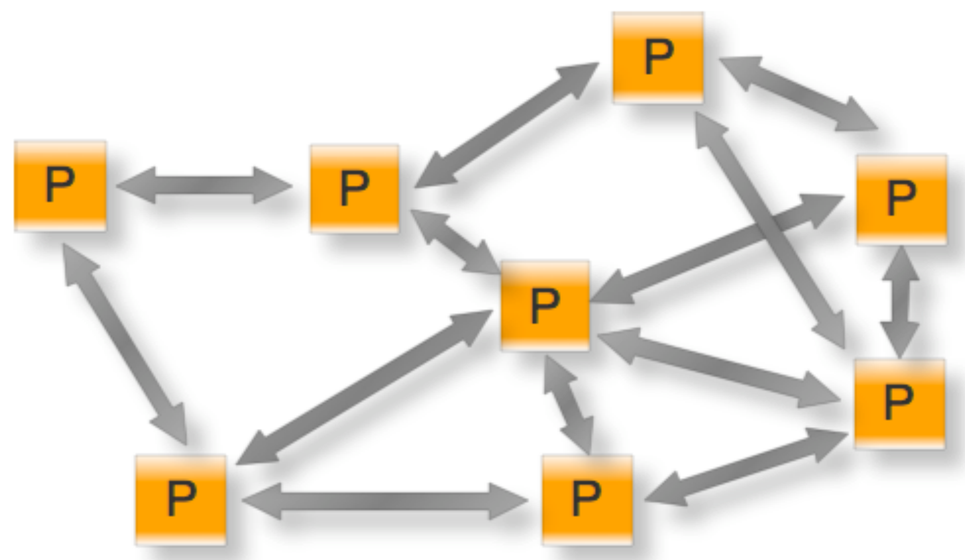
Outline

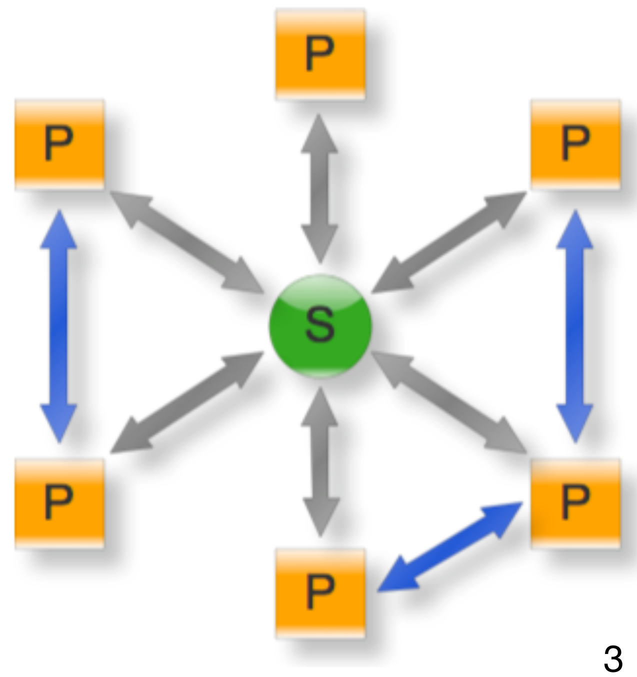
- Lecture Review
    - Packet delay, loss, throughput
    - Applications
        - HTTP (persistent/parallel), SMTP, DNS, etc
- Socket programming review

Application Layer: Models

- Application Architectures
  - Client-server model: Web (TCP), FTP (TCP), E-mail (TCP), DNS (UDP/TCP), RTP
  - Peer-to-Peer (P2P): BitTorrent (TCP), Tor (aka Onion Routing, TCP)
  - Hybrid model: Skype (TCP&UDP), GTalk (TCP&UDP)



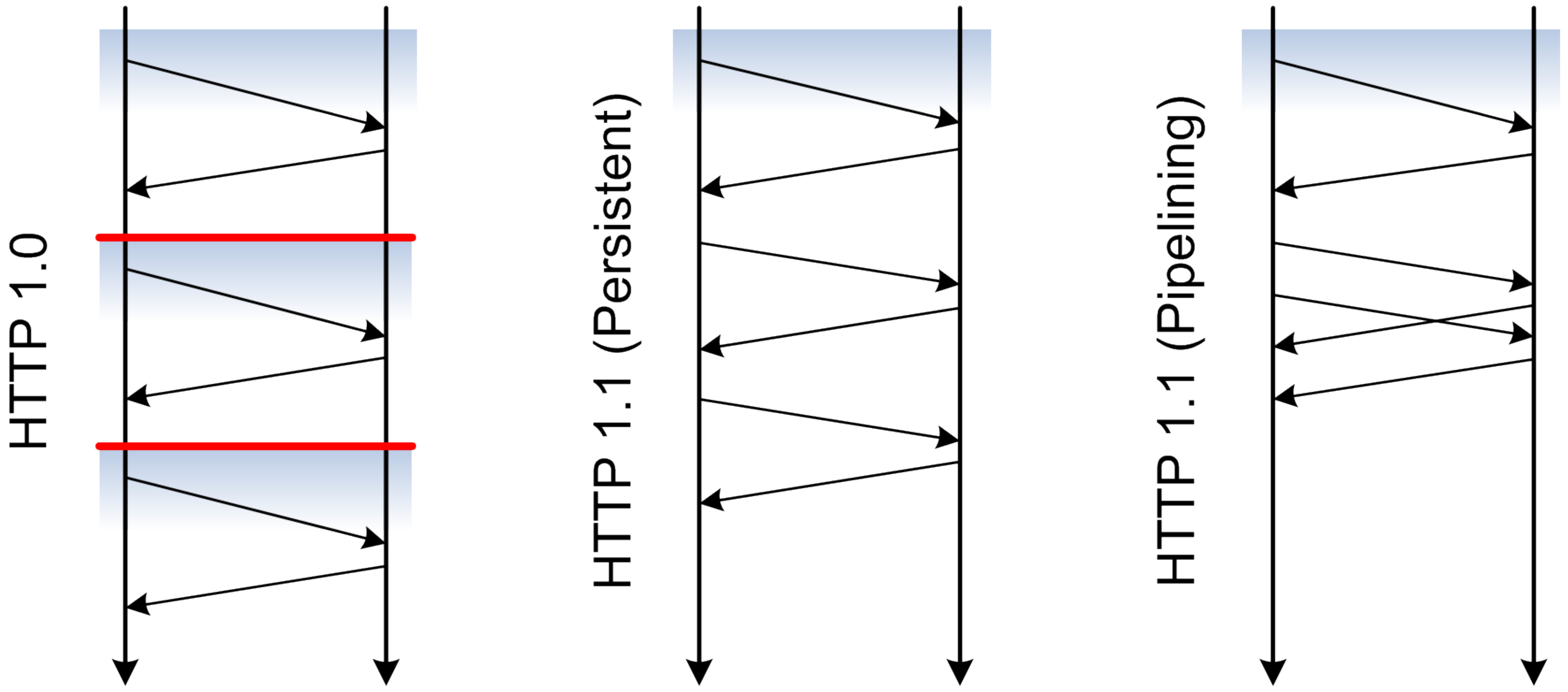




Application Layer: Protocols

- HTTP: a stateless protocol on top of TCP
  - HTTP is based on pull model
  - Persistent HTTP V.S. Non-persistent HTTP
  - Method Types: GET, HEAD, POST, PUT, DELETE, Conditional GET
  - What if we want stateful service (e.g. shopping cart)?
  - Web Caches (proxy server)

Non-persistent v.s. Persistent v.s. Pipelining



Question

- How many TCP connections do we need to get one HTML file with 5 embedded images? How many RTTs shall we need?

- Demo (note: calculate of transmission delay in this demo when there are parallel TCP connection is wrong!)
    - Transmission delay of one object in one TCP connection is 0.25 RTT -> transmission delay of two objects in two TCP connection would be 0.5 RTT (Link rate for one connection reduce by half since two connections are sharing the link!)

HTTP Header: request

- Request message elements:
    - Method
    - URL
    - HTTP Version
    - Header lines
    - CRLF (carriage return and line feed)

HTTP Header: response

- Response message elements:
    - HTTP Version
    - Status line
    - Header lines
    - CRLF
    - Data requested

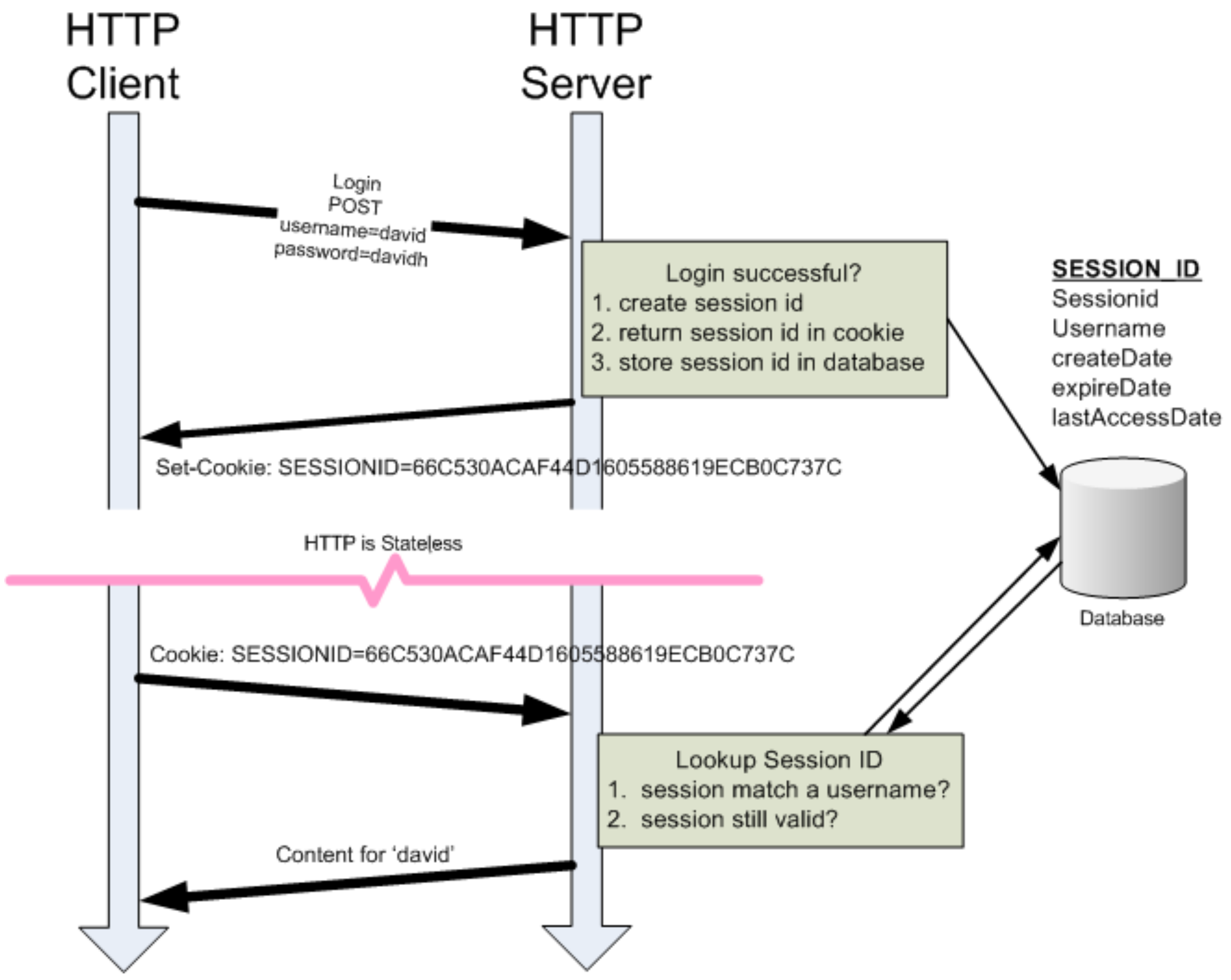
Try HTTP GET yourself

- telnet google.com 80
    - Get / HTTP/1.1
    - Host: google.com
    - <Enter>
    - <Enter>

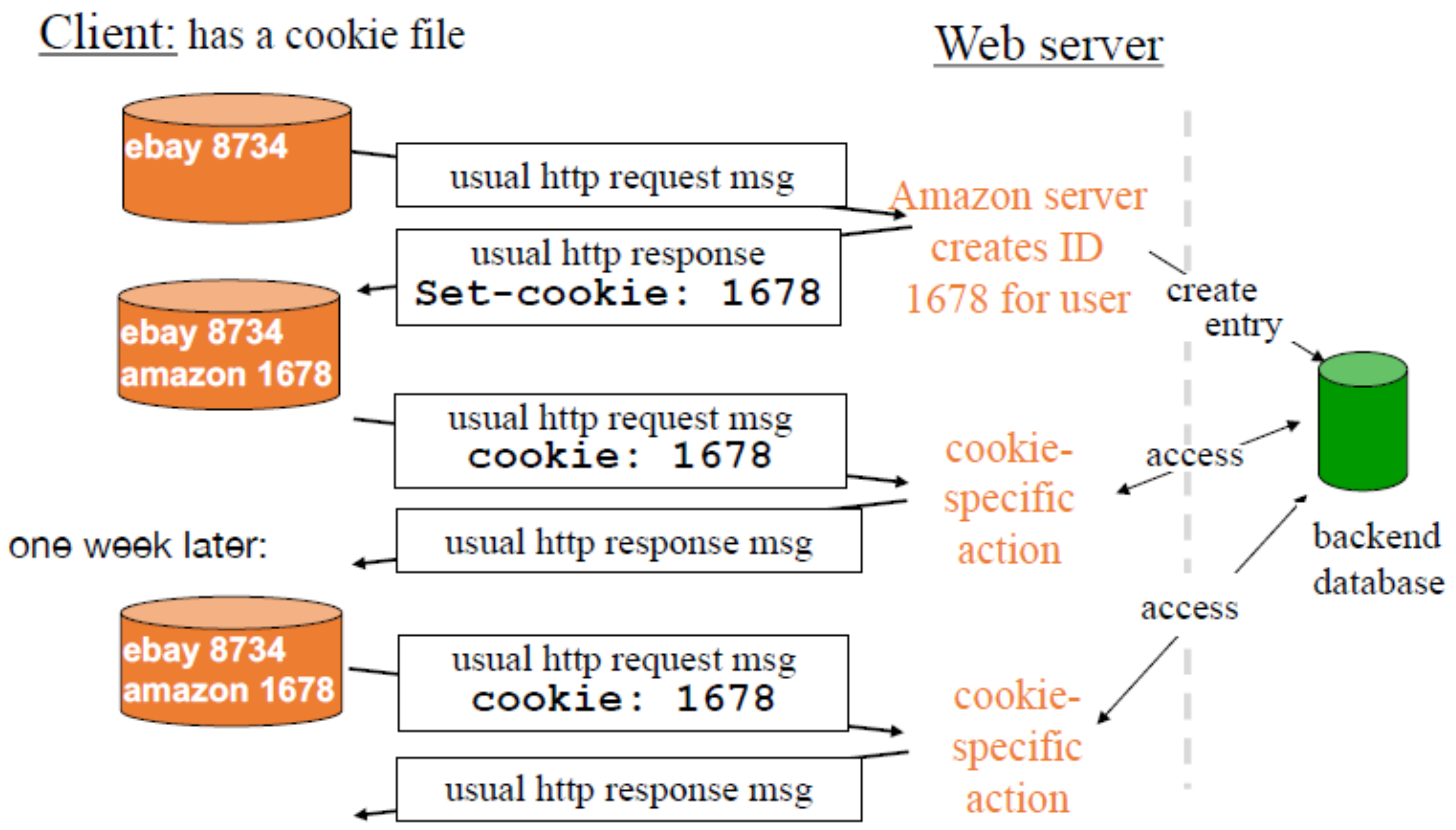
Cookie

- Bring statefulness into HTTP
- Components
  - Cookie header line of HTTP response message
  - Cookie header line of HTTP request message
  - Cookie file on the browser
  - Back-end database at web-site

Cookie: make HTTP stateful



Cookie: operations



Web caching: Proxy v.s. CDN

- Proxy acts both as client and server
  - What if cache is stale?
    - HTTP conditional GET
- CDN: Content Distribution Network
  - Globally distributed network of web servers
  - Stores and replicates images, videos and other files

HTTP conditional GET

Request:

```
GET /sample.html HTTP/1.1
Host: example.com
```

Response:

```
HTTP/1.x 200 OK
Via: The-proxy-name
Content-Length: 32859
Expires: Tue, 27 Dec 2005 11:25:11 GMT
Date: Tue, 27 Dec 2005 05:25:11 GMT
Content-Type: text/html; charset=iso-8859-1
Server: Apache/1.3.33 (Unix) PHP/4.3.10
Cache-Control: max-age=21600
Last-Modified: Wed, 01 Sep 2004 13:24:52 GMT
Etag: “4135cda4”
```

Cache-Control: It tells the client the maximum time in seconds to cache the document.

Last-Modified: The document’s last modified date

Etag: A unique hash for the document.

HTTP conditional GET

Request:

```
GET /sample.html HTTP/1.1
Host: example.com
```

Response:

```
HTTP/1.x 200 OK
Via: The-proxy-name
Content-Length: 32859
Expires: Tue, 27 Dec 2005 11:25:11 GMT
Date: Tue, 27 Dec 2005 05:25:11 GMT
Content-Type: text/html; charset=iso-8859-1
Server: Apache/1.3.33 (Unix) PHP/4.3.10
Cache-Control: max-age=21600
Last-Modified: Wed, 01 Sep 2004 13:24:52 GMT
Etag: “4135cda4”
```

Request:

```
GET /sample.html HTTP/1.1
Host: example.com
If-Modified-Since: Wed, 01 Sep 2004 13:24:52 GMT
If-None-Match: “4135cda4”
```

Response:

```
HTTP/1.x 304 Not Modified
Via: The-proxy-server
Expires: Tue, 27 Dec 2005 11:25:19 GMT
Date: Tue, 27 Dec 2005 05:25:19 GMT
Server: Apache/1.3.33 (Unix) PHP/4.3.10
Keep-Alive: timeout=2, max=99
Etag: “4135cda4”
Cache-Control: max-age=21600
```

Question

Consider the following string of ASCII characters that were captured by Wireshark when the browser sent an HTTP GET message.

a. What is the URL of the document requested by the browser?

b. What version of HTTP is the browser running?

c. Does the browser request a non-persistent or a persistent connection?

```
GET /118/index.html HTTP/1.1<cr><lf>Host: gai
a.cs.umass.edu<cr><lf>User-Agent: Mozilla/5.0 (
Windows;U; Windows NT 5.1; en-US; rv:1.7.2) Gec
ko/20040804 Netscape/7.2 (ax) <cr><lf>Accept:ex
t/xml, application/xml, application/xhtml+xml, text
/html;q=0.9, text/plain;q=0.8,image/png,*/*;q=0.5
<cr><lf>Accept-Language: en-us,en;q=0.5<cr><lf>
AcceptEncoding: zip,deflate<cr><lf>Accept-Charset: ISO
-8859-1,utf-8;q=0.7,*;q=0.7<cr><lf>Keep-Alive: 300<cr>
<lf>Connection:keep-alive<cr><lf><cr><lf>
```

Question

Consider the following string of ASCII characters that were captured by Wireshark when the browser sent an HTTP GET message.

d. What type of browser initiates this message? Why is the browser type needed in an HTTP request message?

```
GET /118/index.html HTTP/1.1<cr><lf>Host: gai
a.cs.umass.edu<cr><lf>User-Agent: Mozilla/5.0 (
Windows;U; Windows NT 5.1; en-US; rv:1.7.2) Gec
ko/20040804 Netscape/7.2 (ax) <cr><lf>Accept:ex
t/xml, application/xml, application/xhtml+xml, text
/html;q=0.9, text/plain;q=0.8,image/png,*/*;q=0.5
<cr><lf>Accept-Language: en-us,en;q=0.5<cr><lf>
AcceptEncoding: zip,deflate<cr><lf>Accept-Charset: ISO
-8859-1,utf-8;q=0.7,*;q=0.7<cr><lf>Keep-Alive: 300<cr>
<lf>Connection:keep-alive<cr><lf><cr><lf>
```

Question

The text below shows the reply sent from the server in response to the HTTP GET message in the question above.

a. As the server able to successfully find the document or not? What time was the document reply provided?

b. When was the document last modified?

c. How many bytes are there in the document being returned?

d. What are the first 5 bytes of the document being returned? Did the server agree to a persistent connection?

```
HTTP/1.1 200 OK<cr><lf>Date: Tue, 07 Mar 2008
12:39:45GMT<cr><lf>Server: Apache/2.0.52 (Fedora)
<cr><lf>Last-Modified: Sat, 10 Dec2005 18:27:46
GMT<cr><lf>ETag: "526c3-f22-
a88a4c80"<cr><lf>Accept- Ranges:
bytes<cr><lf>Content-Length: 3874<cr><lf> Keep-Alive:
timeout=max=100<cr><lf>Connection: Keep-
Alive<cr><lf>Content-Type: text/html; charset =ISO-
8859-1<cr><lf><cr><lf><!doctype html public "-
//w3c//dtd html 4.0 transitional//en"><lf><html><lf>
<head><lf> <meta http-equiv="Content-Type"
content="text/html; charset=iso-8859-1"><lf><meta
name="GENERATOR" content="Mozilla/4.79 [en]
(Windows NT 5.0; U) Netscape]"><lf> <title>CMPSCI
453 / 591 / NTU-ST550A Spring 2005
homepage</title><lf></head><lf> <much more
document text following here (not shown)>
```

Application Layer: Protocols

- FTP: separate control from data transmission (“out-of-band”)
- SMTP: protocol for email exchange between email servers
  - SMTP is based on push model
  - Mail access protocol: POP, IMAP, HTTP-based
- P2P: no always-on server, peers are intermittently connected
  - BitTorrent: tracker and torrent. Files are divided into multiple chunks.

Application Layer: Protocols

- DNS: convert hostname to IP address (and more)
- A distributed and hierarchical database
  - Root DNS servers
  - Top-level domain (TLD) servers
  - Authoritative DNS servers
  - local DNS server (aka, DNS resolver)

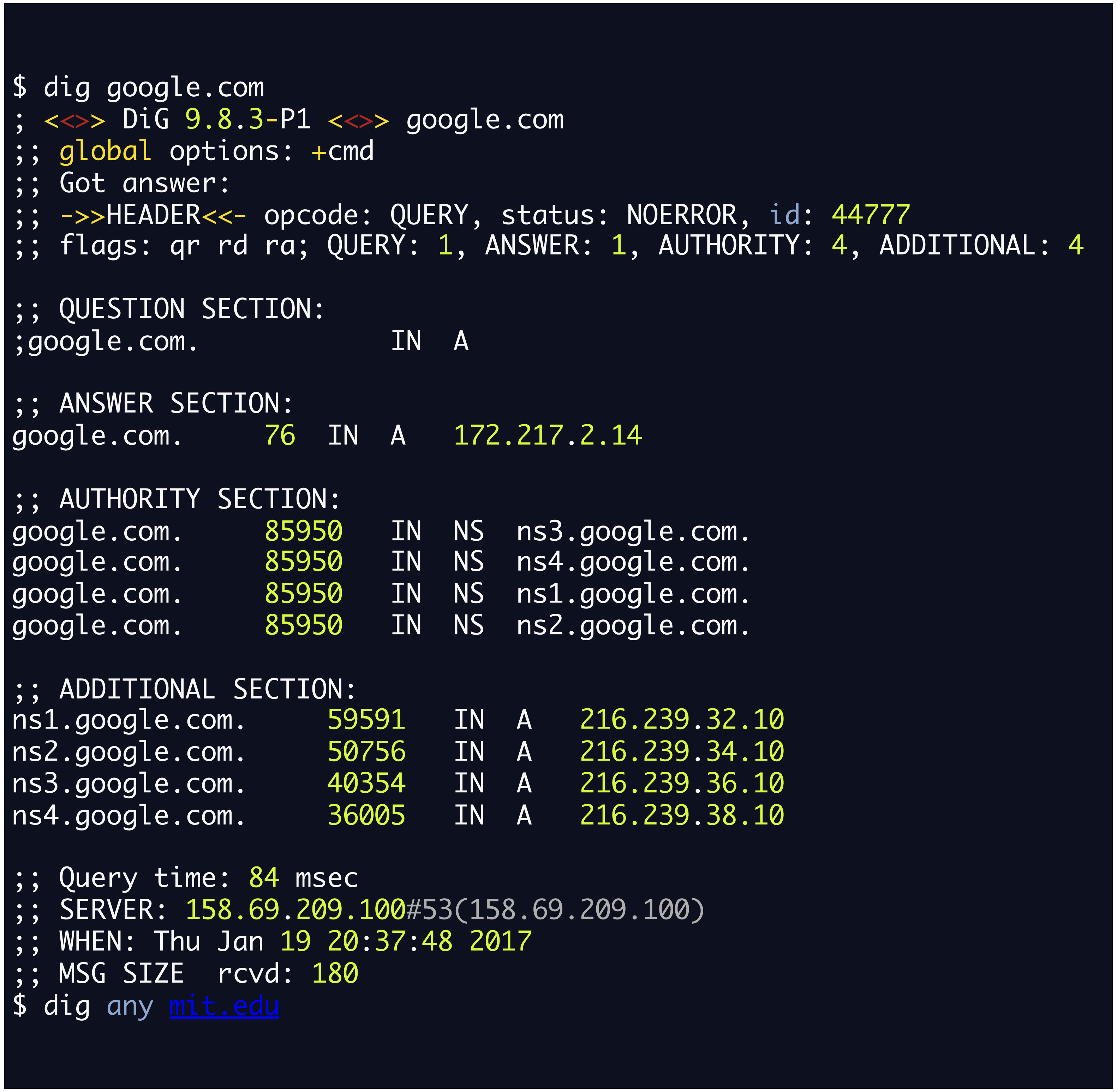
Application Layer: protocols

- DNS:
  - What is the transport layer protocol?
  - How the scalability is achieved?
  - Who will use iterative/recursive query?
  - Why is DNS resolver needed?

Application Layer: protocols

- DNS:
  - What is the transport layer protocol? UDP
  - How the scalability is achieved? Distributed records
  - Who will use iterative/recursive query? DNS resolver
  - Why is DNS resolver needed? Reduce latency

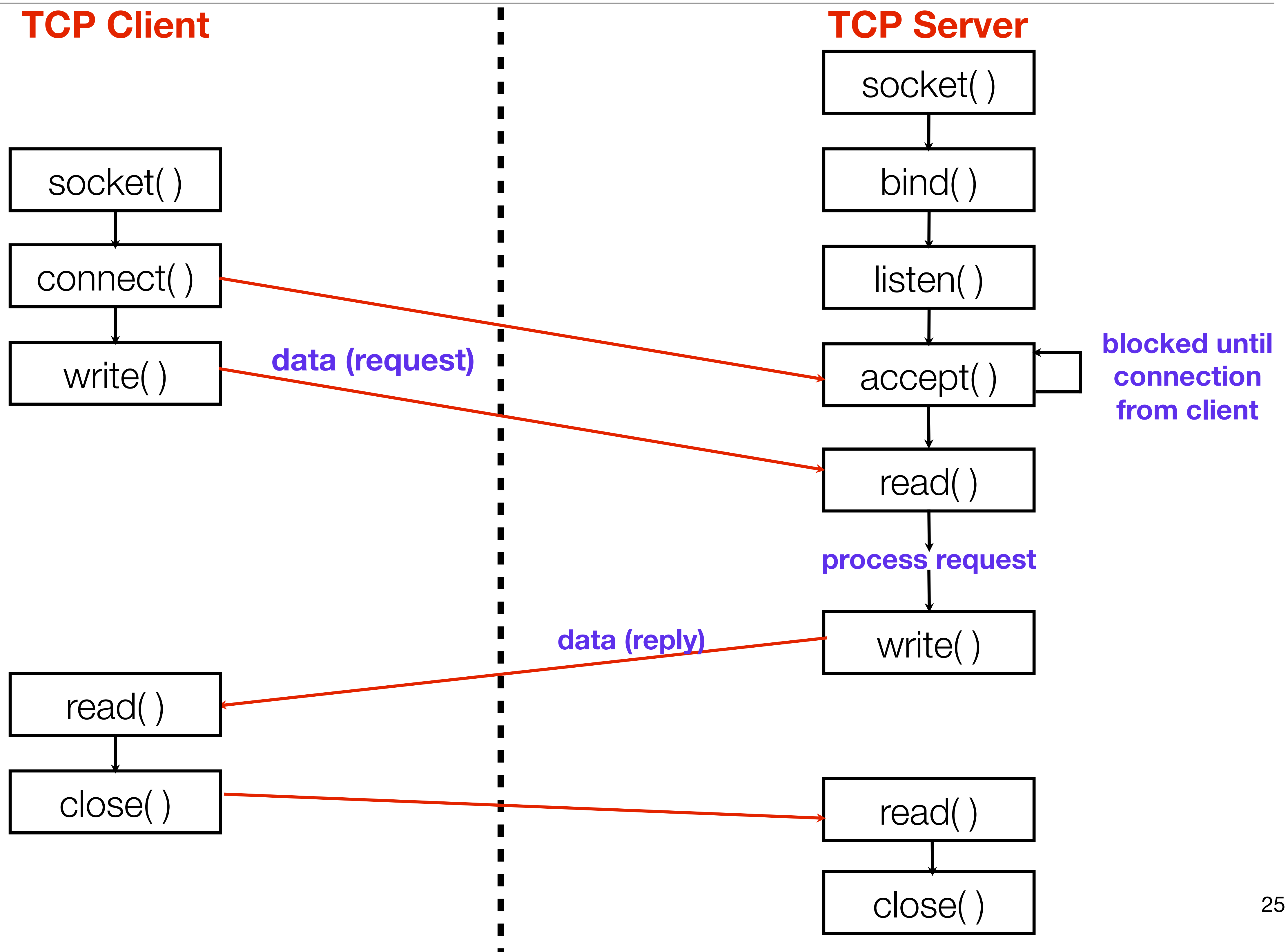
A fun experiment: DNS query



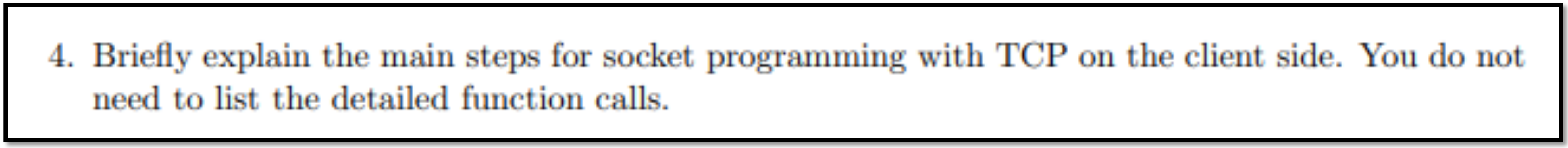
```
$ dig google.com
; <<>> DiG 9.8.3-P1 <<>> google.com
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 44777
;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 4, ADDITIONAL: 4

;; QUESTION SECTION:
;google.com.            IN  A

;; ANSWER SECTION:
google.com.     76  IN  A   172.217.2.14

;; AUTHORITY SECTION:
google.com.     85950   IN  NS  ns3.google.com.
google.com.     85950   IN  NS  ns4.google.com.
google.com.     85950   IN  NS  ns1.google.com.
google.com.     85950   IN  NS  ns2.google.com.

;; ADDITIONAL SECTION:
ns1.google.com.     59591   IN  A   216.239.32.10
ns2.google.com.     50756   IN  A   216.239.34.10
ns3.google.com.     40354   IN  A   216.239.36.10
ns4.google.com.     36005   IN  A   216.239.38.10

;; Query time: 84 msec
;; SERVER: 158.69.209.100#53(158.69.209.100)
;; WHEN: Thu Jan 19 20:37:48 2017
;; MSG SIZE  rcvd: 180
$ dig any mit.edu
```

DNS parameters note

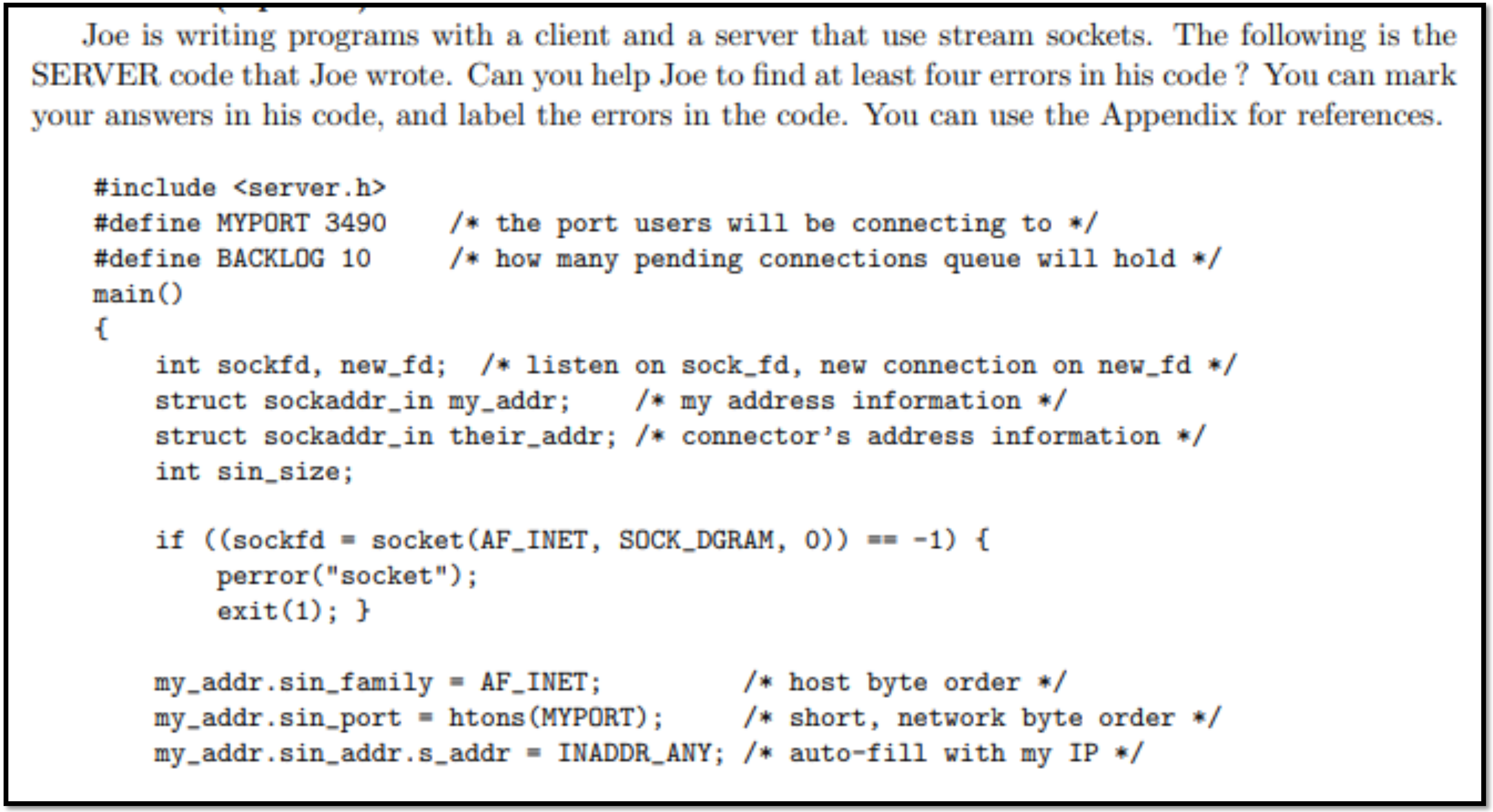
TCP socket: close connection



Socket programming review



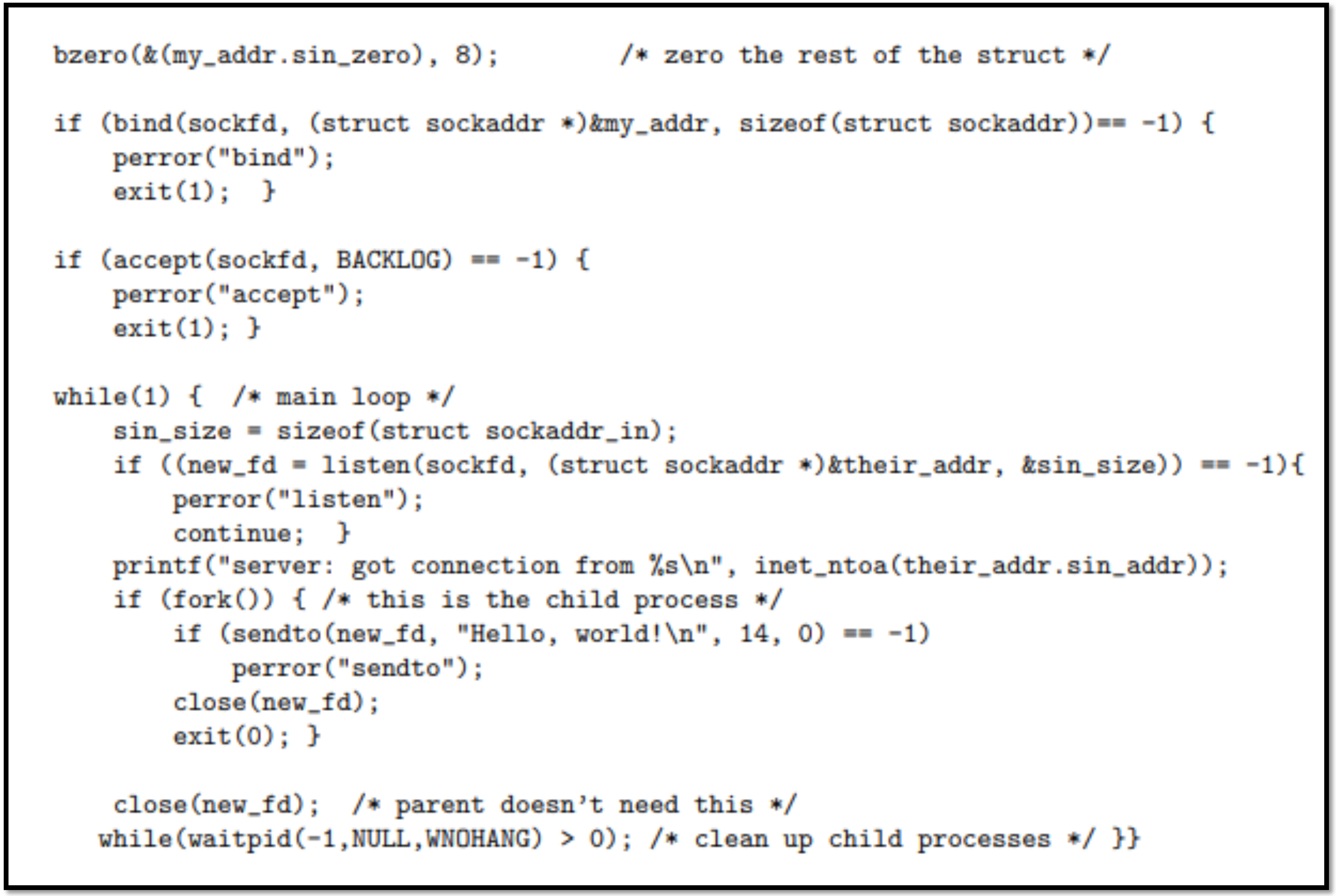
4. Briefly explain the main steps for socket programming with TCP on the client side. You do not need to list the detailed function calls.



Joe is writing programs with a client and a server that use stream sockets. The following is the SERVER code that Joe wrote. Can you help Joe to find at least four errors in his code ? You can mark your answers in his code, and label the errors in the code. You can use the Appendix for references.

```
#include <server.h>
#define MYPORT 3490    /* the port users will be connecting to */
#define BACKLOG 10     /* how many pending connections queue will hold */
main()
{
    int sockfd, new_fd;  /* listen on sock_fd, new connection on new_fd */
    struct sockaddr_in my_addr;    /* my address information */
    struct sockaddr_in their_addr; /* connector's address information */
    int sin_size;

    if ((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) == -1) {
        perror("socket");
        exit(1); }

    my_addr.sin_family = AF_INET;         /* host byte order */
    my_addr.sin_port = htons(MYPORT);     /* short, network byte order */
    my_addr.sin_addr.s_addr = INADDR_ANY; /* auto-fill with my IP */
```

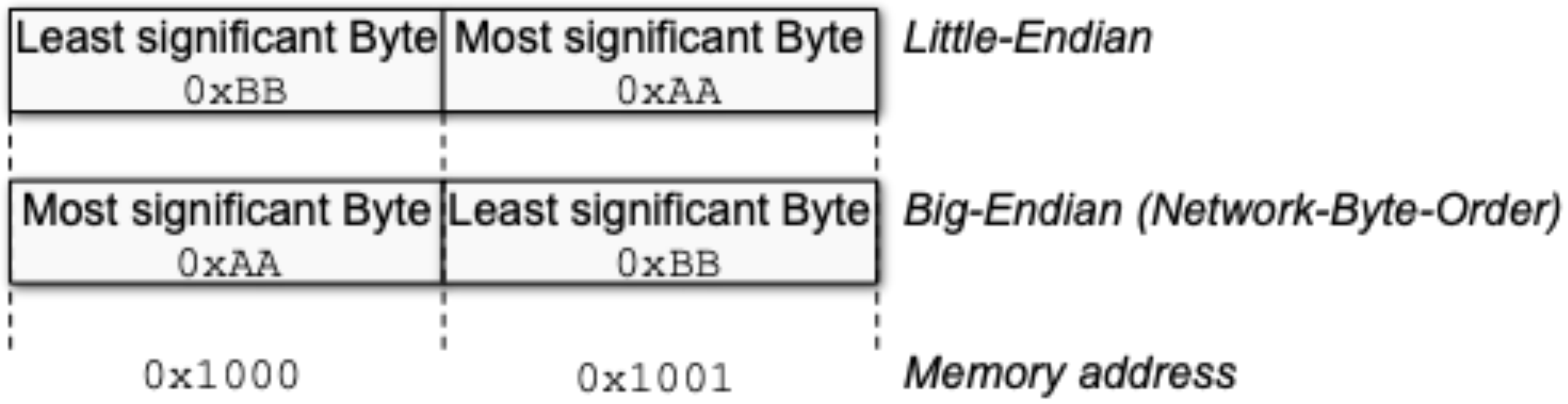
Socket programming review



```
bzero(&(my_addr.sin_zero), 8);        /* zero the rest of the struct */

if (bind(sockfd, (struct sockaddr *)&my_addr, sizeof(struct sockaddr))== -1) {
    perror("bind");
    exit(1);  }

if (accept(sockfd, BACKLOG) == -1) {
    perror("accept");
    exit(1); }

while(1) {  /* main loop */
    sin_size = sizeof(struct sockaddr_in);
    if ((new_fd = listen(sockfd, (struct sockaddr *)&their_addr, &sin_size)) == -1){
        perror("listen");
        continue;  }
    printf("server: got connection from %s\n", inet_ntoa(their_addr.sin_addr));
    if (fork()) { /* this is the child process */
        if (sendto(new_fd, "Hello, world!\n", 14, 0) == -1)
            perror("sendto");
        close(new_fd);
        exit(0); }

    close(new_fd);  /* parent doesn't need this */
   while(waitpid(-1,NULL,WNOHANG) > 0); /* clean up child processes */ }}
```

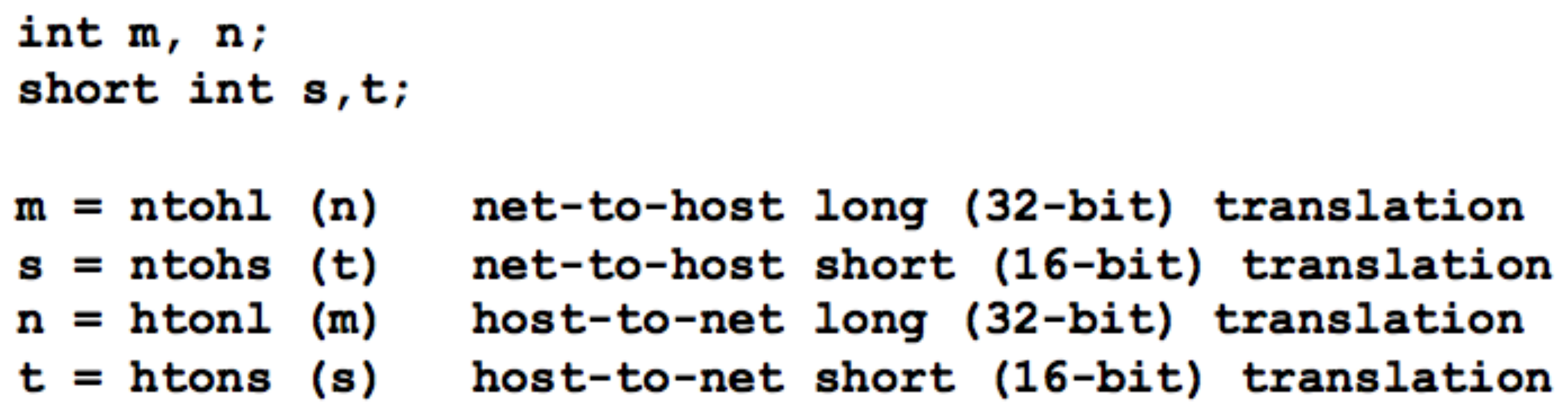
Caveat: byte ordering matters

- Little Endian: least significant byte of word is stored in the lowest address
- Big Endian: most significant byte of word is stored in the lowest address
- Hosts may use different orderings, so we need byte ordering conversion
- **Network Byte Order = Big Endian**



Caveat: byte ordering matters

- Byte ordering functions: used for converting byte ordering
- Example:



```
int m, n;
short int s,t;

m = ntohl (n)   net-to-host long (32-bit) translation
s = ntohs (t)   net-to-host short (16-bit) translation
n = htonl (m)   host-to-net long (32-bit) translation
t = htons (s)   host-to-net short (16-bit) translation
```

- Rule: for every int or short int
- Call htonl( ) or htons( ) before sending data
- Call ntohl( ) or ntohs( ) before reading received data

Address util functions

- All binary values are network byte ordered
- **struct hostent* gethostbyname (const char* hostname);**
  - Translate host name (e.g. “localhost”) to IP address (with DNS working)
- **struct hostent* gethostbyaddr (const char* addr, size_t len, int family);**
  - Translate IP address to host name
- **char* inet_ntoa (struct in_addr inaddr);**
  - Translate IP address to ASCII dotted-decimal notation (e.g. “192.168.0.1”)
- **int gethostname (char* name, size_namelen);**
  - Read local host’s name

FYI: `struct hostent`

`char *h_name`	The real canonical host name.
`char **h_aliases`	A list of aliases that can be accessed with arrays—the last element is `NULL`
`int h_addrtype`	The result's address type, which really should be `AF_INET` for our purposes.
`int length`	The length of the addresses in bytes, which is 4 for IP (version 4) addresses.
`char **h_addr_list`	A list of IP addresses for this host. Although this is a `char**`, it's really an array of `struct in_addr`*s in disguise. The last element is `NULL`.
`h_addr`	A commonly defined alias for `h_addr_list[0]`. If you just want any old IP address for this host (they can have more than one) just use this field.

Address util functions (cont’d)

- in_addr_t inet_addr (const char* strptr);
  - Translate dotted-decimal notation to IP address (network byte order)

```
struct sockaddr_in ina;
ina.sin_addr.s_addr = inet_addr("10.12.110.57");
```

- int inet_aton (const char* strptr, struct in_addr *inaddr);
  - Translate dotted-decimal notation to IP address

```
struct sockaddr_in my_addr;
my_addr.sin_family = AF_INET;          // host byte order
my_addr.sin_port = htons(MYPORT);      // short, network byte order
inet_aton("10.12.110.57",&(my_addr.sin_addr));
memset(&(my_addr.sin_zero), '\0', 8); // zero the rest of the struct
```