Part I: Network protocols: HTTP and beyond

Tanenbaum Ch. 12.3

Distributed Software Systems

CS 707

The Web: some jargon

- Web page:
  - consists of "objects"
  - addressed by a URL
- Most Web pages consist of:
  - base HTML page, and
  - several referenced objects.
- URL has two components: host name and path name:

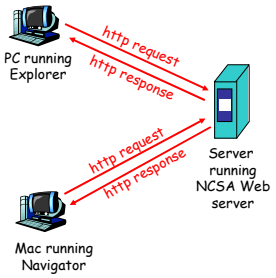
`www.someSchool.edu/someDept/pic.gif`

- User agent for Web is called a browser:
  - MS Internet Explorer
  - Netscape Communicator
- Server for Web is called Web server:
  - Apache (public domain)
  - MS Internet Information Server

The Web: the http protocol

http: hypertext transfer protocol

- Web's application layer protocol
- client/server model
  - *client:* browser that requests, receives, "displays" Web objects
  - *server:* Web server sends objects in response to requests
- http1.0: RFC 1945
- http1.1: RFC 2068



The http protocol: more

http: TCP transport service:

- client initiates TCP connection (creates socket) to server, port 80
- server accepts TCP connection from client
- http messages (application-layer protocol messages) exchanged between browser (http client) and Web server (http server)
- TCP connection closed

http is "stateless"

- server maintains no information about past client requests

aside

Protocols that maintain "state" are complex!

- past history (state) must be maintained
- if server/client crashes, their views of "state" may be inconsistent, must be reconciled

http example

(contains text, references to 10 jpeg images)

Suppose user enters URL
www.someSchool.edu/someDepartment/home.index

1a. http client initiates TCP connection to http server (process) at www.someSchool.edu. Port 80 is default for http server.

1b. http server at host www.someSchool.edu waiting for TCP connection at port 80. "accepts" connection, notifying client

2. http client sends http *request message* (containing URL) into TCP connection socket

3. http server receives request message, forms *response message* containing requested object (someDepartment/home.index), sends message into socket

time

http example (cont.)

4. http server closes TCP connection.

5. http client receives response message containing html file, displays html. Parsing html file, finds 10 referenced jpeg objects

6. Steps 1-5 repeated for each of 10 jpeg objects

time

Non-persistent and persistent connections

Non-persistent

- HTTP/1.0
- server parses request, responds, and closes TCP connection
- 2 RTTs to fetch each object
- Each object transfer suffers from slow start

But most 1.0 browsers use parallel TCP connections.

Persistent

- default for HTTP/1.1
- on same TCP connection: server, parses request, responds, parses new request,..
- Client sends requests for all referenced objects as soon as it receives base HTML.
- Fewer RTTs and less slow start.

http message format: request

- two types of http messages: *request*, *response*
- http request message:
  - ASCII (human-readable format)

request line (GET, POST, HEAD commands)

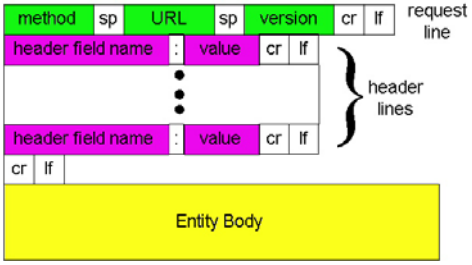
```
GET /somedir/page.html HTTP/1.0
User-agent: Mozilla/4.0
Accept: text/html, image/gif,image/jpeg
Accept-language:fr
```

header lines

(extra carriage return, line feed)

Carriage return, line feed indicates end of message

http request message: general format



Examples of a complete client request

Example1:

```
GET / HTTP/1.1
<blank line>
```

Example2:

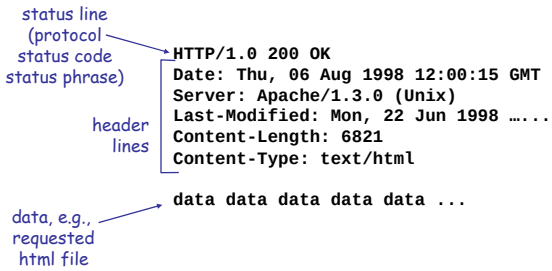
```
HEAD / HTTP/1.1
Accept: */*
Connection: Keep-Alive
Host: somehost.com
User-Agent: Generic
<blank line>
```

Examples of a complete client request

Example3:

```
POST  /servlet/myServer.servlet HTTP/1.0
Accept: */*
Connection: Keep-Alive
Host: somehost.com
User-Agent: Generic
<blank line>
Name=donald&email=donald@someU.edu
```

http message format: response



http response status codes

In first line in server->client response message.

A few sample codes:

200 OK

- request succeeded, requested object later in this message

301 Moved Permanently

- requested object moved, new location specified later in this message (Location:)

400 Bad Request

- request message not understood by server

404 Not Found

- requested document not found on this server

505 HTTP Version Not Supported

Trying out http (client side) for yourself

1. Telnet to your favorite Web server:

```
telnet www.eurecom.fr 80
```

Opens TCP connection to port 80 (default http server port) at www.eurecom.fr. Anything typed in sent to port 80 at www.eurecom.fr

2. Type in a GET http request:

```
GET /~ross/index.html HTTP/1.0
```

By typing this in (hit carriage return twice), you send this minimal (but complete) GET request to http server

3. Look at response message sent by http server!

Same thing in C

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
void error(char *msg)
{  perror(msg);
   exit(0);  }
int main(int argc, char *argv[])
{  int sockfd, portno, n;
   struct sockaddr_in serv_addr;
   struct hostent *server;
   char buffer[1024];
   portno = 80;
   sockfd = socket(AF_INET, SOCK_STREAM, 0);
   if (sockfd < 0)   error("ERROR opening socket");
   server = gethostbyname("osf1.gmu.edu");
   if (server == NULL) { fprintf(stderr,"ERROR, no such host\n");  exit(0); }
   bzero((char *) &serv_addr, sizeof(serv_addr));
   serv_addr.sin_family = AF_INET;
   bcopy((char *)server->h_addr, (char *)&serv_addr.sin_addr.s_addr,  server->h_length);
   serv_addr.sin_port = htons(portno);
   if (connect(sockfd,&serv_addr,sizeof(serv_addr)) < 0)   error("ERROR connecting");
   bzero(buffer,1024);
   sprintf(buffer,"GET /~white/ HTTP/1.0\n\n");
   n = write(sockfd,buffer,strlen(buffer));
   if (n < 0)   error("ERROR writing to socket");
   bzero(buffer,1024);
   n = read(sockfd,buffer,1023);
   if (n < 0)   error("ERROR reading from socket");
   printf("%s\n",buffer);
   close(sockfd);
   return 0;
}
```

A Basic Java HTTP Client implementation

```
InetAddress host =
     InetAddress.getByName(args[0]);
int port = Integer.parseInt(args[1]);
String fileName = args[2].trim();
String request =
     "GET " + fileName + " HTTP/1.0\n\n";
MyStreamSocket mySocket =
     new MyStreamSocket(host, port);
mySocket.sendMessage(request);
// now receive the response from the HTTP server
String response = mySocket.receiveMessage();
// read and display one line at a time
while (response != null) {
   System.out.println(response);
   response = mySocket.receiveMessage();
}
```

Content Type and the Mime Protocol

- One of the header lines returned in a server response is the Contents Type of the object requested.
- Specification of the contents type follows the scheme established in a protocol known as MIME (Multipurpose Internet Mail Extension.)
- Originally used for Email, MIME is now widely used for describing the content of a document sent over a network.
- It supports a large number and evolving set of predefined content types, specified in the format `Type/Subtype`.

MIME types

`Content-Type: type/subtype; parameters`

Text
- example subtypes: **`plain, html`**

Image
- example subtypes: **`jpeg, gif`**

Audio
- exampe subtypes: **`basic`** (8-bit mu-law encoded), **`32kadpcm`** **(32 kbps coding)**

Video
- example subtypes: **`mpeg, quicktime`**

Application
- other data that must be processed by reader before "viewable"
- example subtypes: **`msword, octet-stream`**

Message format: multimedia extensions

- MIME: multimedia mail extension, RFC 2045, 2056
- additional lines in msg header declare MIME content type

```
From: alice@crepes.fr
To: bob@hamburger.edu
Subject: Picture of yummy crepe.
MIME-Version: 1.0
Content-Transfer-Encoding: base64
Content-Type: image/jpeg

base64 encoded data .....
.........................
......base64 encoded data
```

MIME version

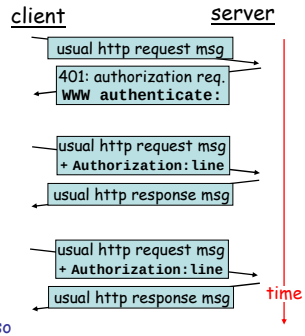
method used to encode data

multimedia data type, subtype, parameter declaration

encoded data

User-server interaction: authentication

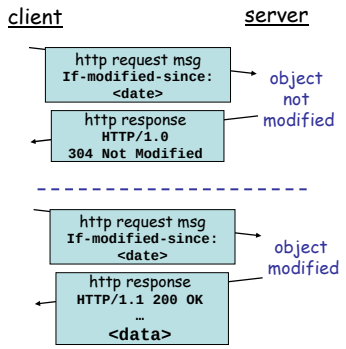
Authentication goal: control access to server documents
- stateless: client must present authorization in each request
- authorization: typically name, password
  - **`authorization:`** header line in request
  - if no authorization presented, server refuses access, sends **`WWW authenticate:`** header line in response



Browser caches name & password so that user does not have to repeatedly enter it.

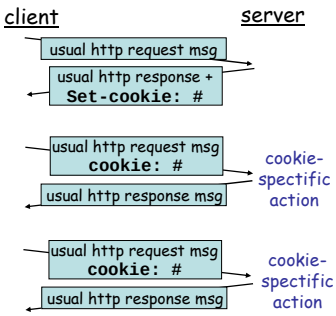
User-server interaction: conditional GET

- Goal: don't send object if client has up-to-date stored (cached) version
- client: specify date of cached copy in http request
  `If-modified-since: <date>`
- server: response contains no object if cached copy up-to-date:
  `HTTP/1.0 304 Not Modified`



User-server interaction: cookies

- server sends "cookie" to client in response msg
  `Set-cookie: 1678453`
- client presents cookie in later requests
  `cookie: 1678453`
- server matches presented-cookie with server-stored info
  - authentication
  - remembering user preferences, previous choices

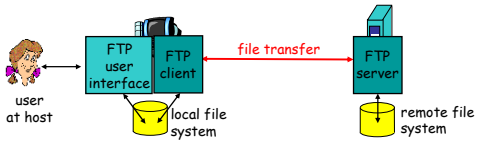


Using cookies for state data

- A scheme for session state data repository on the client side is a mechanism known as a **cookie**, "for no compelling reason".
- The scheme makes use of an extension of the basic HTTP to allow a server's response to contain a piece of state information for which the client will provide storage in an object.
- "Included in that state object is a description of the range of URLs for which that state is valid. Any future HTTP requests made by the client which fall in that range will include a transmittal of the current value of the state object from the client back to the server."

Other Protocols

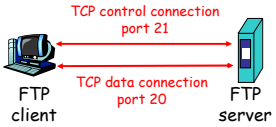
ftp: the file transfer protocol



- transfer file to/from remote host
- client/server model
  - *client:* side that initiates transfer (either to/from remote)
  - *server:* remote host
- ftp: RFC 959
- ftp server: port 21

ftp: separate control, data connections

- ftp client contacts ftp server at port 21, specifying TCP as transport protocol
- two parallel TCP connections opened:
  - control: exchange commands, responses between client, server.
    "out of band control"
  - data: file data to/from server
- ftp server maintains "state": current directory, earlier authentication



ftp commands, responses

Sample commands:

- sent as ASCII text over control channel
- **USER *username***
- **PASS *password***
- **LIST** return list of file in current directory
- **RETR filename** retrieves (gets) file
- **STOR filename** stores (puts) file onto remote host

Sample return codes

- status code and phrase (as in http)
- **331 Username OK, password required**
- **125 data connection already open; transfer starting**
- **425 Can't open data connection**
- **452 Error writing file**

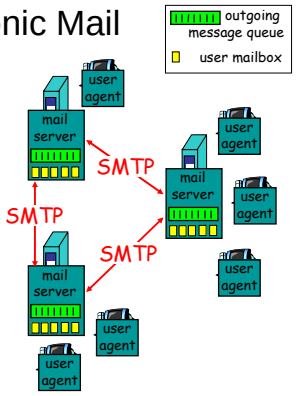
Electronic Mail

Three major components:

- user agents
- mail servers
- simple mail transfer protocol: smtp

User Agent

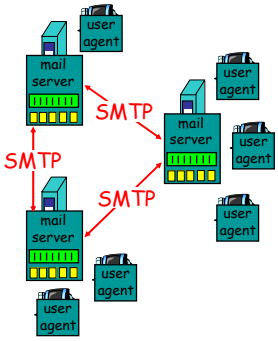
- a.k.a. "mail reader"
- composing, editing, reading mail messages
- e.g., Eudora, Outlook, elm, Netscape Messenger
- outgoing, incoming messages stored on server



Electronic Mail: mail servers

Mail Servers

- mailbox contains incoming messages (yet to be read) for user
- message queue of outgoing (to be sent) mail messages
- smtp protocol between mail servers to send email messages
  - client: sending mail server
  - "server": receiving mail server



Electronic Mail: smtp [RFC 821]

- uses TCP to reliably transfer email message from client to server, port 25
- direct transfer: sending server to receiving server
- three phases of transfer
  - handshaking (greeting)
  - transfer of messages
  - closure
- command/response interaction
  - commands: ASCII text
  - response: status code and phrase
- messages must be in 7-bit ASCII

Sample smtp interaction

```
S: 220 hamburger.edu
C: HELO crepes.fr
S: 250  Hello crepes.fr, pleased to meet you
C: MAIL FROM: <alice@crepes.fr>
S: 250 alice@crepes.fr... Sender ok
C: RCPT TO: <bob@hamburger.edu>
S: 250 bob@hamburger.edu ... Recipient ok
C: DATA
S: 354 Enter mail, end with "." on a line by itself
C: Do you like ketchup?
C:   How about pickles?
C: .
S: 250 Message accepted for delivery
C: QUIT
S: 221 hamburger.edu closing connection
```

try smtp interaction for yourself:

- **`telnet servername 25`**
- see 220 reply from server
- enter HELO, MAIL FROM, RCPT TO, DATA, QUIT commands

above lets you send email without using email client (reader)

smtp: final words

- smtp uses persistent connections
- smtp requires that message (header & body) be in 7-bit ascii
- certain character strings are not permitted in message (e.g., `CRLF.CRLF`). Thus message has to be encoded (usually into either base-64 or quoted printable)
- smtp server uses `CRLF.CRLF` to determine end of message

Comparison with http

- http: pull
- email: push
- both have ASCII command/response interaction, status codes
- http: each object is encapsulated in its own response message
- smtp: multiple objects message sent in a multipart message

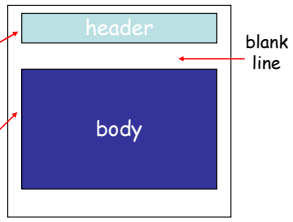
Mail message format

smtp: protocol for exchanging email msgs

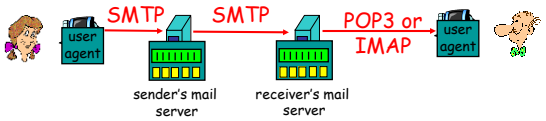
RFC 822: standard for text message format:

- header lines, e.g.,
  - To:
  - From:
  - Subject:

  different from smtp commands!
- body
  - the "message", ASCII characters only



Mail access protocols



- SMTP: delivery/storage to receiver's server
- Mail access protocol: retrieval from server
  - POP: Post Office Protocol [RFC 1939]
    - authorization (agent <-->server) and download
  - IMAP: Internet Mail Access Protocol [RFC 1730]
    - more features (more complex)
    - manipulation of stored msgs on server
  - HTTP: Hotmail , Yahoo! Mail, etc.

POP3 protocol

authorization phase

- client commands:
  - **`user:`** declare username
  - **`pass:`** password
- server responses
  - **`+OK`**
  - **`-ERR`**

transaction phase, client:

- **`list:`** list message numbers
- **`retr:`** retrieve message by number
- **`dele:`** delete
- **`quit`**

```
S: +OK POP3 server ready
C: user alice
S: +OK
C: pass hungry
S: +OK user successfully logged on
```

```
C: list
S: 1 498
S: 2 912
S: .
C: retr 1
S: <message 1 contents>
S: .
C: dele 1
C: retr 2
S: <message 1 contents>
S: .
C: dele 2
C: quit
S: +OK POP3 server signing off
```

Summary

Our study of network apps now complete!

- application service requirements:
  - reliability, bandwidth, delay
- client-server paradigm
- Internet transport service model
  - connection-oriented, reliable: TCP
  - unreliable, datagrams: UDP
- specific protocols:
  - http
  - ftp
  - smtp, pop3
- socket programming
  - client/server implementation
  - using tcp, udp sockets

Summary

Most importantly: learned about *protocols*

- typical request/reply message exchange:
  - client requests info or service
  - server responds with data, status code
- message formats:
  - headers: fields giving info about data
  - data: info being communicated
- control vs. data msgs
  - in-based, out-of-band
- centralized vs. decentralized
- stateless vs. stateful
- reliable vs. unreliable msg transfer
- security: authentication

Part II: Java Threads: A quick tutorial

Distributed Software Systems

CS 707

Java Threads

- 2 ways to create a thread
  - Subclass “Thread”
  - Implement “Runnable” and pass it to Thread constructor, allowing you to add threading to a class that inherits from something other than “Thread”
- In either case: end up with a Thread object
- Call start() to start.
- run() is method that does the work.
- Once run() exits, thread is dead
  - Can’t restart thread, you have to create a new one.

Simple Example: Extending Thread class

```
public class BytePrinter extends Thread {
  public void run() {
    for (int b = -128; b < 128; b++)
      System.out.println(b);
  }
}
public class ThreadTest {
  public static void main(String[] args) {
    BytePrinter bp1 = new BytePrinter();
    BytePrinter bp2 = new BytePrinter();
    BytePrinter bp3 = new BytePrinter();
    bp1.start();
    bp2.start();
    bp3.start();
  }
}
```

Thread class has three primary methods:
public void start()
public void run()
public final void stop()

create instances

start theads

Simple Example: Using Runnable

```
public class BytePrinter implements Runnable {
  public void run() {
    for (int b = -128; b < 128; b++)
      System.out.println(b);
  }
}
public class ThreadTest {
  public static void main(String[] args) {
    Thread bp1 = new Thread(new BytePrinter());
    Thread bp2 = new Thread(new BytePrinter());
    Thread bp3 = new Thread(new BytePrinter());
    bp1.start();
    bp2.start();
    bp3.start();
  }
}
```

create instances

start theads

Synchronization

Most synchronization can be regarded as either:

- Mutual exclusion (making sure that only one process is executing a CRITICAL SECTION [touching a variable or data structure, for example] at a time),

  or as

- CONDITION SYNCHRONIZATION, which means making sure that a given process does not proceed until some condition holds (e.g. that a variable contains a given value)

Also known as Competition Synchronization

Also known as Cooperation Synchronization

Synchronization with Java Threads

- Mutual Exclusion: A method that includes the **`synchronized`** modifier disallows any other method from running on the object while it is in execution. If only a part of a method must be run without interference, that part can be **`synchronized`**
- Condition: The **`wait`** and **`notify`** methods are defined in **`Object`**, which is the root class in Java, so all objects inherit them. The **`wait`** method must be called in a loop

Mutual Exclusion Synchronization

```
public class Counter {
  private int count = 0;
  public synchronized void count() {
    int limit = count + 100;
    while (count++ != limit)
         System.out.println(count);
   }
}

public class CounterThread extends Thread {
   private Counter c;
   public CounterThread(Counter c) {
       this.c = c;
   }
   public void run() {
      c.count();
   }
}

public class CounterApp2 {
  public static void main(String[] args) {
    Counter c = new Counter();
    CounterThread ct1 = new CounterThread(c);
    CounterThread ct2 = new CounterThread(c);
    ct1.start();
    ct2.start();
  }
}
```

Try this example both with and without the keyword.

In Java, synchronization associates a lock with an item. In order for a thread to access that item, the thread must hold the lock.

See also dining philosophers

Condition Synchronization

```
public class checkpoint {
  boolean here_first = true;
  synchronized void meet_up () {
     if (here_first) {
          here_first = false;
          wait();
     } else {
          notify();
          here_first = true;
     }
  };
};
```

See also bounded buffers

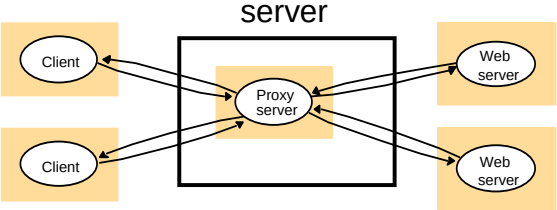
Other Interesting Thread methods

- sleep() – pauses the execution for a given time period
- getPriority() and setPriority(int new_priority)
  - Scheduling done in strict priority ordering
  - Round-robin within equal priority threads.
- join() – makes the caller pause until the thread terminates.

For lots of example code in Java, see:

http://www-dse.doc.ic.ac.uk/concurrency/book_applets/concurrency.html

Your programming assignment: Proxy server



Your server will:

- accept connections from a client
  ex: "`GET http://www.foo.com/bar.html HTTP/1.0`"
- start a new thread for the connection
- the thread will contact the appropriate server using the specified protocol
- once the reply is received from the server, forward it back to the client and log the transaction

Your programming assignment

What do you need to know for this assignment?

- Threads in C/C++ or Java
- Socket connections
- HTTP protocols