Digital Communication in the Modern World
Lesson 2
SMTP, Sockets, Threads

http://www.cs.huji.ac.il/~com1

com1@cs.huji.ac.il

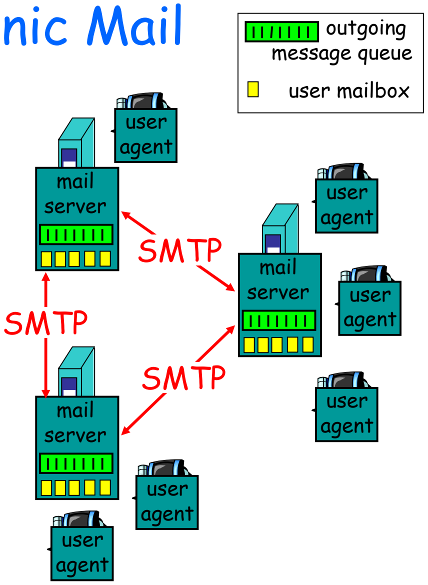
Electronic Mail

Three major components:

- user agents (clients)
- mail servers
- simple mail transfer protocol: SMTP

User Agent

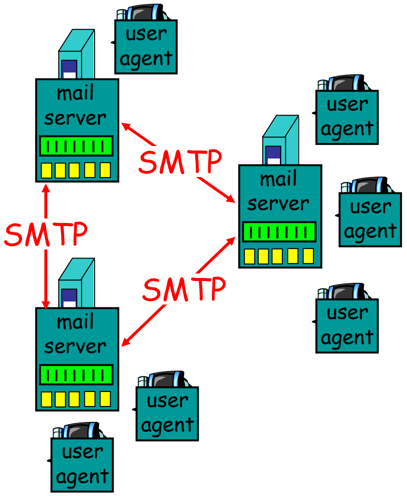
- a.k.a. "mail reader"
- composing, editing, reading mail messages
- e.g., Eudora, Outlook, elm, Netscape Messenger, PINE
- outgoing, incoming messages stored on server



Electronic Mail: mail servers

Mail Servers

- mailbox contains incoming messages 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 2821]

- 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

Scenario: Alice sends message to Bob

1) Alice uses UA to compose message and "to" bob@someschool.edu

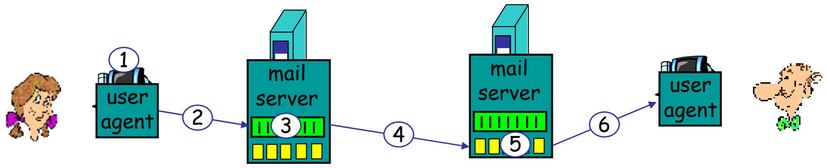
2) Alice's UA sends message to her mail server; message placed in message queue

3) Client side of SMTP opens TCP connection with Bob's mail server

4) SMTP client sends Alice's message over the TCP connection

5) Bob's mail server places the message in Bob's mailbox

6) Bob invokes his user agent to read message



Sample SMTP interaction

```
S: 220 mail.cs.huji.ac.il
C: HELO mail.cs.huji.ac.il
S: 250  Hello mail.cs.ac.il, pleased to meet you
C: MAIL FROM: <falafel@cs.huji.ac.il>
S: 250 falafel@cs.huji.ac.il... Sender ok
C: RCPT TO: <sabih@pita.com>
S: 250 sabih@pita.co ... Recipient ok
C: DATA
S: 354 Enter mail, end with "." on a line by itself
C: From: me@something
C: To: you@somewhere
C: Subject: lunch…
C:     Do you want with hilbe?
C: .
S: 250 Message accepted for delivery
C: QUIT
S: 221 mail.cs.huji.ac.il 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 message (header & body) to be in 7-bit ASCII
- SMTP server uses `CRLF.CRLF` to determine end of message

Comparison with HTTP:

- HTTP: pull
- SMTP: push
- both have ASCII command/response interaction, status codes
- HTTP: each object encapsulated in its own response msg
- SMTP: multiple objects sent in multipart msg

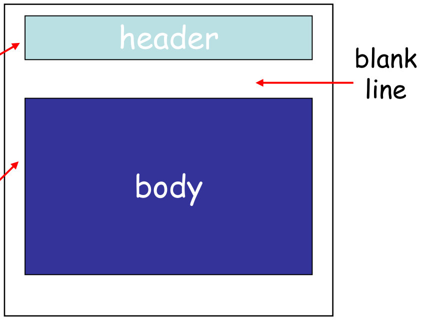
Mail message format

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

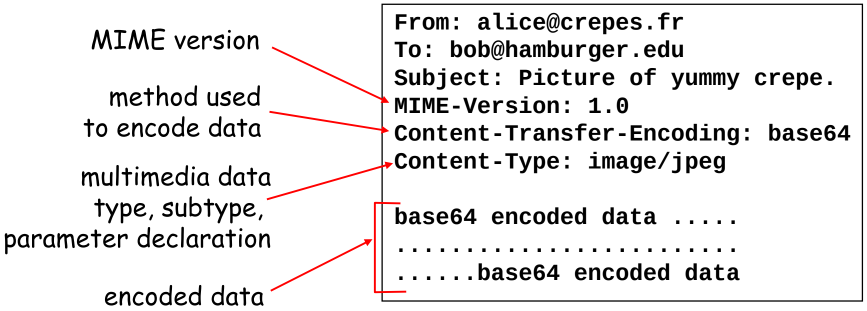


MIME types

- Similar to file extensions but more universally accepted "MIME types" are used to identify the type of information that a file contains. While the file extension .html is informally understood to mean that the file is an HTML page, there is no requirement that it mean this, and many HTML pages have different file extensions. In the HTTP protocol used by web browsers to talk to web servers, the "file extension" of the URL is *not* used to determine the type of information that the server will return. Indeed, there may be no file extension at all at the end of the URL.
- Instead, the web server specifies the correct MIME type using a *Content-type:* header when it responds to the web browser's HTTP request.
- MIME stands for "Multimedia Internet Mail Extensions." MIME was originally invented to solve a similar problem for email attachments where the client is responsible for correctly displaying the requested file.

Message format: multimedia extensions

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



MIME version

method used to encode data

multimedia data type, subtype, parameter declaration

encoded data

```
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 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

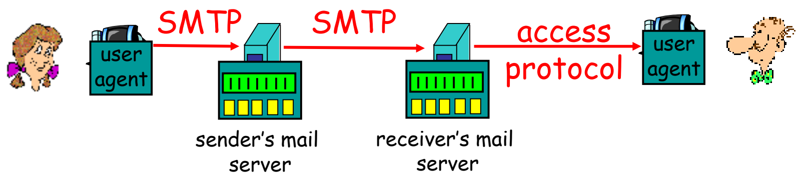
- other data that must be processed by reader before “viewable”
- example subtypes: **msword**, **octet-stream**

Multipart Type

```
From: alice@crepes.fr
To: bob@hamburger.edu
Subject: Picture of yummy crepe.
MIME-Version: 1.0
Content-Type: multipart/mixed; boundary=StartOfNextPart

--StartOfNextPart
Dear Bob, Please find a picture of a crepe.
--StartOfNextPart
Content-Transfer-Encoding: base64
Content-Type: image/jpeg
base64 encoded data .....
.........................
......base64 encoded data
--StartOfNextPart
Do you want the recipe?
```

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, Gmail, 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 bob
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
```

POP3 (more) and IMAP

More about POP3

- Previous example uses "download and delete" mode.
- Bob cannot re-read e-mail if he changes client
- "Download-and-keep": copies of messages on different clients
- POP3 is stateless across sessions

IMAP

- Keep all messages in one place: the server
- Allows user to organize messages in folders
- IMAP keeps user state across sessions:
  - names of folders and mappings between message IDs and folder name

Socket programming

Goal: learn how to build application that communicate using sockets

Socket API

- introduced in BSD4.1 UNIX, 1981
- explicitly created, used, released by apps
- client/server paradigm
- two types of transport service via socket API:
  - unreliable datagram
  - reliable, byte stream-oriented

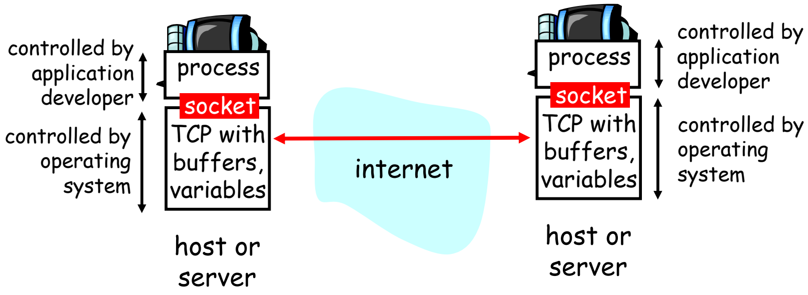
socket

a *host-local*, *application-created/owned*, *OS-controlled* interface (a "door") into which application process can both send and receive messages to/from another (remote or local) application process

Socket-programming using TCP

Socket: a door between application process and end-end-transport protocol (UDP or TCP)

TCP service: reliable transfer of bytes from one process to another


controlled by application developer
controlled by operating system
process
socket
TCP with buffers, variables
host or server
internet
process
socket
TCP with buffers, variables
host or server
controlled by application developer
controlled by operating system

Socket programming with TCP

Client must contact server

- server process must first be running
- server must have created socket (door) that welcomes client's contact

Client contacts server by:

- creating client-local TCP socket
- specifying IP address, port number of server process
- When client creates socket: client TCP establishes connection to server TCP
- When contacted by client, server TCP creates new socket for server process to communicate with client
  - allows server to talk with multiple clients

application viewpoint

TCP provides reliable, in-order transfer of bytes ("pipe") between client and server

Stream jargon

- A stream is a sequence of characters that flow into or out of a process.
- An input stream is attached to some input source for the process, e.g., keyboard or socket.
- An output stream is attached to an output source, e.g., monitor or socket.

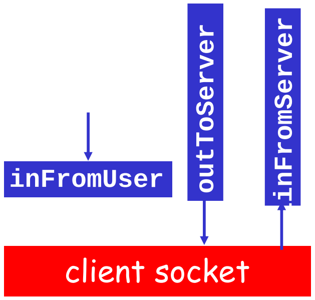
Socket programming with TCP

Example client-server app:

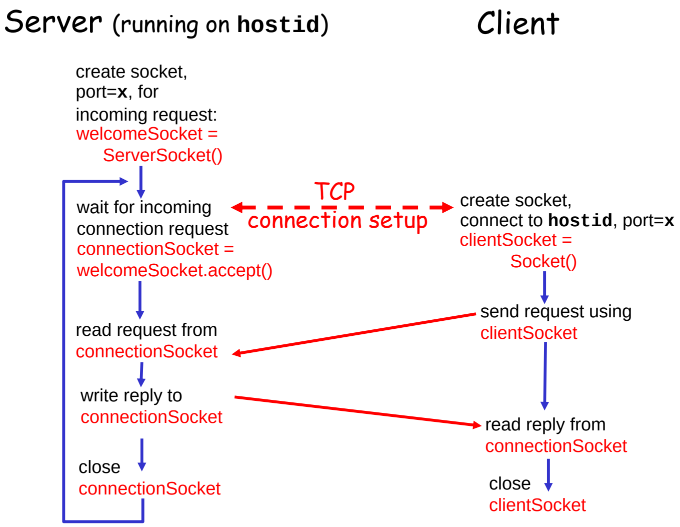
- client reads line from standard input (**inFromUser** stream) , sends to server via socket (**outToServer** stream)
- server reads line from socket
- server converts line to uppercase, sends back to client
- client reads, prints modified line from socket (**inFromServer** stream)

Input stream: sequence of bytes into process

Output stream: sequence of bytes out of process



Client/server socket interaction: TCP


Server (running on **hostid**)

Client

create socket, port=**x**, for incoming request: welcomeSocket = ServerSocket()

wait for incoming connection request connectionSocket = welcomeSocket.accept()

TCP connection setup

create socket, connect to **hostid**, port=**x** clientSocket = Socket()

send request using clientSocket

read request from connectionSocket

write reply to connectionSocket

read reply from connectionSocket

close connectionSocket

close clientSocket

Example: Java client (TCP)

```
import java.io.*;
import java.net.*;
class TCPClient {

  public static void main(String argv[]) throws Exception
  {
    String sentence;
    String modifiedSentence;
```

Create input stream

```
    BufferedReader inFromUser =
      new BufferedReader(new InputStreamReader(System.in));
```

Create client socket, connect to server

```
    Socket clientSocket = new Socket("hostname", 6789);
```

Create output stream attached to socket

```
    DataOutputStream outToServer =
      new DataOutputStream(clientSocket.getOutputStream());
```

Example: Java client (TCP), cont.

Create input stream attached to socket

```
BufferedReader inFromServer =
  new BufferedReader(new
  InputStreamReader(clientSocket.getInputStream()));

sentence = inFromUser.readLine();
```

Send line to server

```
outToServer.writeBytes(sentence + '\n');
```

Read line from server

```
modifiedSentence = inFromServer.readLine();

System.out.println("FROM SERVER: " + modifiedSentence);

clientSocket.close();

  }
}
```

Example: Java server (TCP)

```
import java.io.*;
import java.net.*;

class TCPServer {

  public static void main(String argv[]) throws Exception
   {
     String clientSentence;
     String capitalizedSentence;
```

Create welcoming socket at port 6789

```
     ServerSocket welcomeSocket = new ServerSocket(6789);

     while(true) {
```

Wait, on welcoming socket for contact by client

```
        Socket connectionSocket = welcomeSocket.accept();
```

Create input stream, attached to socket

```
       BufferedReader inFromClient =
         new BufferedReader(new
         InputStreamReader(connectionSocket.getInputStream()));
```

Example: Java server (TCP), cont

Create output stream, attached to socket

```
DataOutputStream  outToClient =
  new DataOutputStream(connectionSocket.getOutputStream());
```

Read in line from socket

```
clientSentence = inFromClient.readLine();

capitalizedSentence = clientSentence.toUpperCase() + '\n';
```

Write out line to socket

```
outToClient.writeBytes(capitalizedSentence);
      }
    }
}
```

End of while loop, loop back and wait for another client connection

Socket programming: references

C-language tutorial (audio/slides):

- "Unix Network Programming" (J. Kurose),

http://manic.cs.umass.edu/~amldemo/courseware/intro.

Java-tutorials:

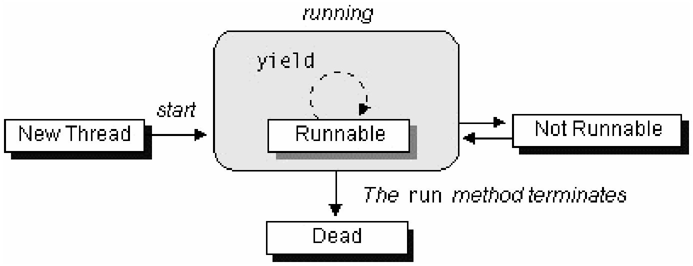
- "All About Sockets" (Sun tutorial), http://www.javaworld.com/javaworld/jw-12-1996/jw-12-sockets.html
- "Socket Programming in Java: a tutorial," http://www.javaworld.com/javaworld/jw-12-1996/jw-12-sockets.html

What is a Thread?

- **Definition**: A thread is a single sequential flow of control within a program
- Multi-thread programming allows to perform several tasks "at the same time".

Java Threads (1)

- java.lang.Thread or interface Runnable
- http://java.sun.com/docs/books/tutorial/essential/threads/
- To implement a thread using the Thread class, you need to provide it with a **run** method that performs the thread's task



Java Threads (2)

```
class PrimeThread extends Thread {
     long minPrime;
        PrimeThread(long minPrime) {
            this.minPrime = minPrime;
         }
         public void run() {
             // compute primes larger than minPrime  . . .
        }
}
         PrimeThread p = new PrimeThread(143);
         p.start();
```

Customizing a Thread's run Method

- The **run** method gives a thread something to do.
- There are two techniques for providing a run method for a thread:
  - Subclassing Thread
  - Overriding run Implementing the Runnable Interface

Synchronizing Threads

- Sometimes threads that run concurrently share data and must consider the state and activities of other threads.
- Because the threads share a common resource, they must be synchronized in some way.

Locking an Object

- The code segments within a program that access the same object from separate, concurrent threads are called *critical sections*.
- In the Java language, a critical section can be a block or a method and is identified with the **synchronized** keyword.
- The Java platform then associates a lock with every object that has **synchronized** code.

notify, notifyAll and wait Methods

- **wait(timeout)** Waits for notification OR until the timeout period has elapsed
- **notify** arbitrarily wakes up one of the threads waiting on this object.
- **notifyAll** method wakes up all threads waiting on the object in question
  - The awakened threads compete for the lock. One thread gets it, and the others go back to waiting. The Object class also defines the notify method, which arbitrarily wakes up one of the threads waiting on this object.

Timer and TimerTask Classes

- Whenever possible, you should use high-level thread API such as the java.util.Timer and its companion class, TimerTask are useful when your program must perform a task repeatedly or after a delay.

Simple demo that uses java.util.Timer
to schedule a task to execute once 5 seconds have passed

```
public class Reminder {
   Timer timer;
   public Reminder(int seconds) {
      timer = new Timer();
      timer.schedule(new RemindTask(),seconds*1000);
   }
```

class RemindTask

```
class RemindTask extends TimerTask {
        public void run() {
            System.out.println("Time's up!");
            timer.cancel(); // Terminate the timer thread
        }
}
    public static void main(String args[]) {
        System.out.println("About to schedule task.");
        new Reminder(5);
        System.out.println("Task scheduled.");
 }
```

Four ways to stop Timer Threads

- Invoke cancel on the timer. You can do this from anywhere in the program, such as from a timer task's run method.
- Make the timer's thread a "daemon" by creating the timer like this: new Timer(true). If the only threads left in the program are daemon threads, the program exits.
- After all the timer's scheduled tasks have finished executing, remove all references to the Timer object. Eventually, the timer's thread will terminate.
- Invoke the System.exit method, which makes the entire program (and all its threads) exit.