COP 4610L: Operating Systems Lab

Distributed Applications in the Enterprise

Lecture Set 1

Dr. R. Lent

Objective of the Course

To expose you to the world of heterogeneous enterprise computing architecture with emphasis on networked, distributed applications using objects.

Getting Organized

- Prerequisites: COP3330 (OOP), COP3503 (CS2), EEL 4882 (OS), CGS 2545 (Databases).
- Textbook: Deitel, Deitel, and Santry, *Advanced Java 2 Platform*
- Course web page: http://www.cs.ucf.edu/courses/cop4610 (check it frequently!)
- Final grade = 50% projects + 50% tests

Getting Organized (cont'd)

- Projects are programming (Java) assignments that must show your own independent work.
- Project submissions will be received via WebCT
- Open lab hours will be available
- No food or drinks will be allowed in the lab
- Assignment 0: Install JDK 1.4.x (we will need several additional packages later)

Note: The number of chapters may seem taunting, but many of them get their size due to the embedded examples!

Time Estimate

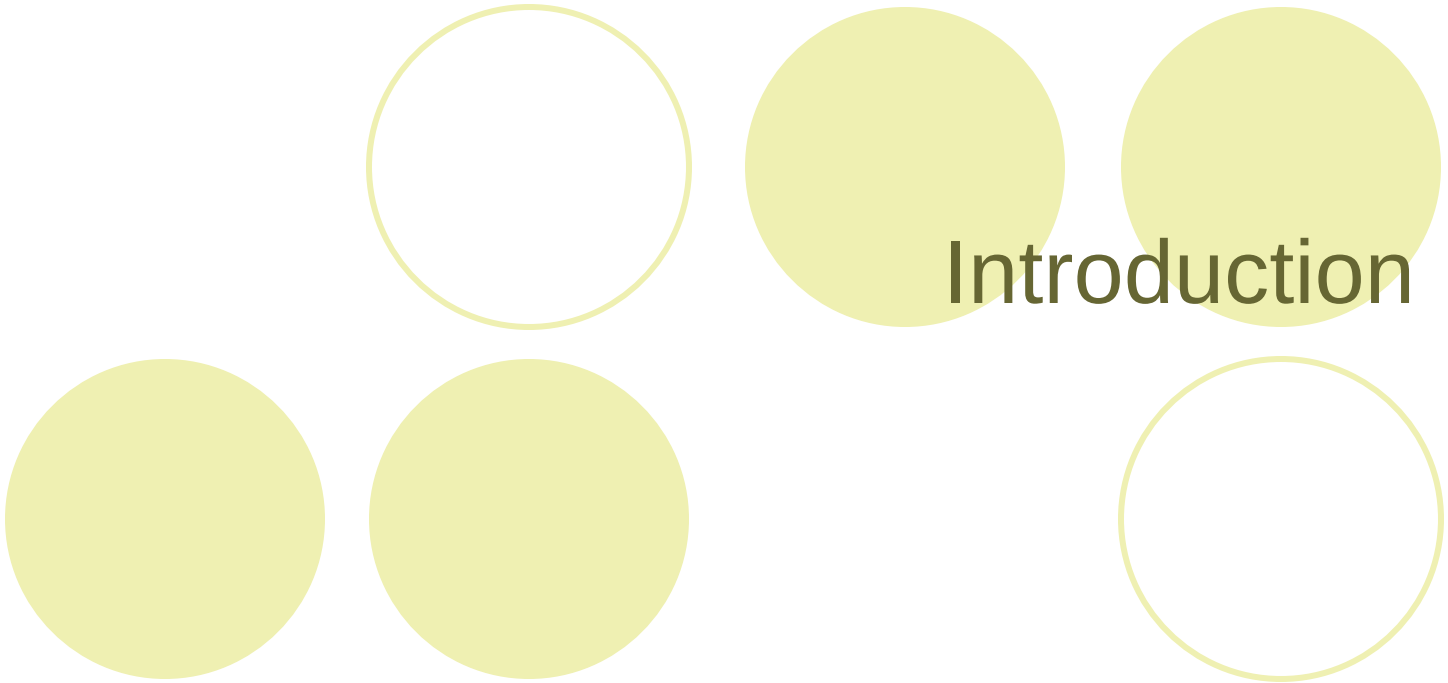
- You should expect to spend an average of 8-12 hours per week

Task	hr./week
class	2
homework	5
reading (exams)	1
total	8

- Your mileage will vary, but if you are spending less than 4 or more than 12 hours per week, there is a problem

Topics to be Covered

- Overview of XML, DTD, DOM, XSLT, XHTML (App. A,B,C,D)
- Networking concepts, socket programming, Web server (notes)
- Concurrency (notes)
- Advanced GUI Swing components, Web browser (Ch.2,3)
- Security (Ch. 7)
- Java Beans (Ch. 6)
- Java Database Connectivity (Ch. 8)
- Servlets and Java Server Pages (Ch. 9,10)
- Remote Method Invocation and CORBA (Ch. 13)
- Enterprise Java Beans (Ch. 14,15)
- Jini and JavaSpaces (Ch. 22, 23)
- SOAP
- *P2P, JXTA,
- *JMS (Ch. 16)



Introduction

Motivation

- Growing demand for Information technology and e-commerce applications
- Constrain: Enterprise applications have to be designed, built, and produced for less money, faster, and with fewer resources than ever before
- Distributed Systems provide a good solution

Distributed Component-based Applications

- A configuration of services provided by different application components on physically independent computers
- Appear to the users of the system as a single application on a single physical machine

Why Distributed Systems?

- Some tasks are inherently distributive. By their nature they require cooperative work from multiple agents
- Reliability. No single point of failure in the system.
- Scalability. By properly designing the system, it should be able to handle more load by adding new services and hardware.
- Performance and economics. Existing distributed, cheap computer power with increased network bandwidth can be used to avoid spending money in new hardware.

Distributed vs. Parallel Computing

Distributed:

- Multiple heterogeneous devices at multiple sites (each independent, with local resource controls)
- Multi-purpose interconnection network
- Shared purpose
- Varied bandwidth; Often high latency; Flexible communication
- Requires more attention to reliability, security and routing

Parallel:

- Multiple, usually similar, devices at a single site (some, perhaps all, resources are centrally controlled)
- Dedicated interconnection network
- Shared purpose
- High bandwidth; Low latency; Inflexible communication

What Should a Distributive Application Provide?

Answer: Transparency (give the illusion of a single unified application on a single machine):

- *Data location*: The user does not need to know where the data is
- *Failure*: The user does not need to worry about consistency of data even if there is a failure in the network of data sources
- *Replication*: The user does not need to know how data replication is done
- *Distribution*: The user does not need to know how computing power and data are distributed across the system.

Transactions

- Groups of statements that represents a unit of work, which must be executed as a unit
- Transactions provide consistent operations on resources (read, write, update)
- Should have the following ACID properties:
  - ***A****tomicity*: “all-or-nothing” property
  - ***C****onsistency*: Map a consistent state of resources to another
  - ***I****solation* (serialization): Reveal no results before commit
  - ***D****urability*: completed transactions cannot be erased due to system failure.
- Transaction management: at local and global levels

Common DS Paradigms

- Channels – send / receive

  Messages to single or collection of recipients

- Distributed Objects – invoke services on remote objects

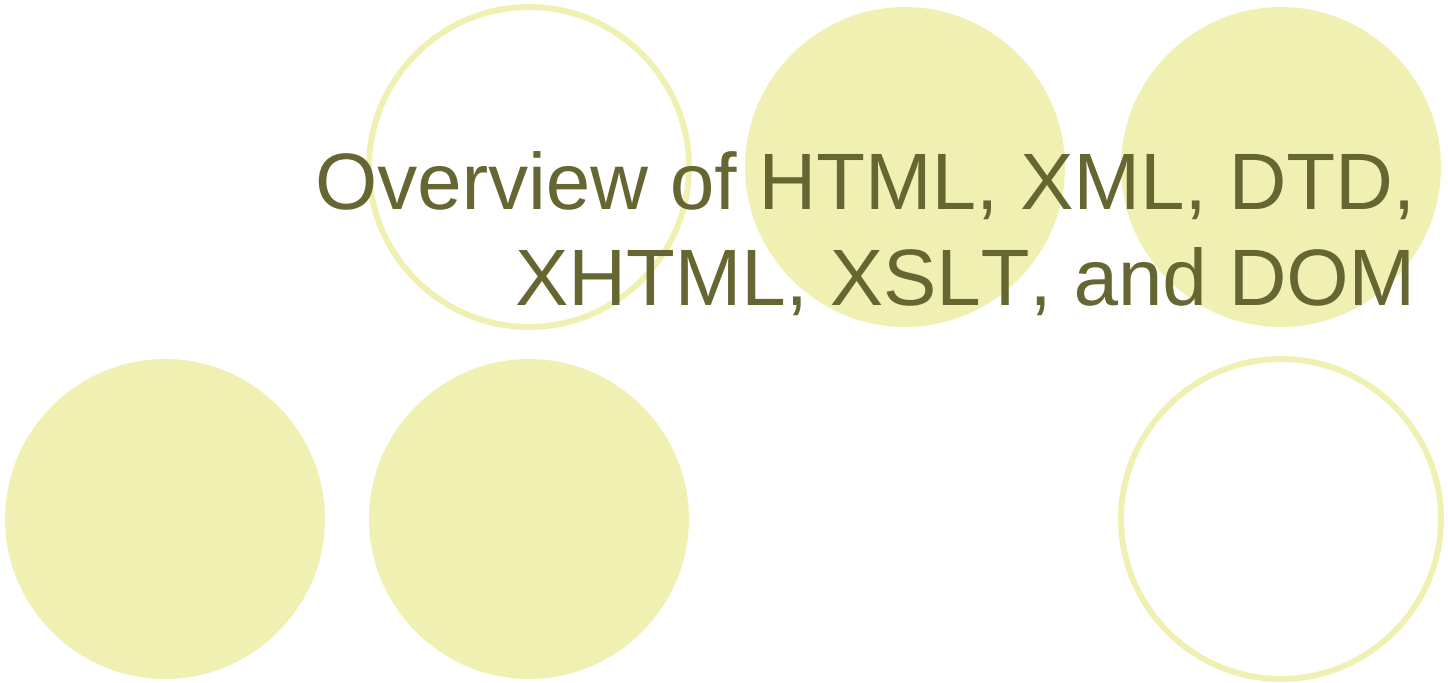
  Requires objects to be transferred over network

  - Serialization (marshalling) / un-serialization

- Tuple space (shared memory) – write, read, take

  A space for reliable communication and coordination

Overview of HTML, XML, DTD, XHTML, XSLT, and DOM



HTML

- HTML = HyperText Markup Language
- Current version: 4.01
- Language for publishing hypertext on the World Wide Web.
- Non-proprietary format (in plain text) based upon SGML.
- HTML uses tags such as <h1> and </h1> to structure text into headings, paragraphs, lists, hypertext links etc.

Example: Forms with POST or GET

```
<FORM ACTION="http://127.0.0.1/submission" METHOD="POST">
<B>Your Name :</B> <INPUT TYPE="text" NAME="nameVal" SIZE="20"
   MAXLENGTH="80">
<p><I>Email Address :</I><INPUT TYPE="text" NAME="emailVal" SIZE="20"
   MAXLENGTH="80">
<P><U>Are you hungry?</U><P>
<INPUT TYPE="radio" NAME="hungryValY">Yes
<INPUT TYPE="radio" NAME="hungryValN" VALUE="_">No
<P>Describe yourself
<SELECT NAME="yourselfField">
<OPTION>A seeker after truth
<OPTION>Head in the sand
<OPTION>Falling asleep quickly
</SELECT>
<p>How do you like my website?
<p><TEXTAREA NAME="yourComments" ROWS="5" COLS="40" value="place your
   comments here"></TEXTAREA>
<P><INPUT TYPE="submit"> <INPUT TYPE="Reset">
</FORM>
```

XML

- XML stands for EXtensible Markup Language
- XML is a markup language much like HTML
- XML was designed to describe data
- XML tags are not predefined in XML. You must define your own tags
- XML uses a Document Type Definition (DTD) or an XML Schema to describe the data
- XML with a DTD or XML Schema is designed to be self-descriptive

XML does not DO really anything

Example:

```
<note>
 <to>Student</to>
 <from>Professor</from>
 <heading>Reminder</heading>
 <body>Don't forget to install JDK!</body>
</note>
```

The example consists of header, body, sender, and receiver, but still the document does not indicate any action. It just describe information.

XML

- XML is not a replacement for HTML
- XML was designed to describe data and to focus on what data is. HTML was designed to display data and to focus on how data looks
- Use: as a cross-platform, software and hardware independent tool for moving information

XML Markup

- Declaration:

  <?xml version = “1.0”?>

- Comments

  <!-- comment -->

- Data is marked up using *tags*

  <myTag> *character data* **</myTag>**

  <myTag />

- Notes:
  - **myTag** is the element name
  - Element names are case sensitive
  - An *end tag* must follow every *start tag*

XML Markup (cont'd)

- Attributes

  <myTag id="COP4610"> data </myTag>

  - Elements can have any number of attributes
- Characters
  - Any, except '&' and '<'
  - *Entity references*:
    - Ampersand = **&**
    - Left-angle bracket = **<**
    - Right-angle bracket = **>**
    - Apostrophe = **'**
    - Quotation mark = **"**

XML (cont'd)

- CDATA define sections not processed by a XML parser
- Admits any character except **]]>**

```
<![CDATA[
  // Test
  if (value == 0 && sum != 0) {
          value = 10;
          return 0;
  }
]]>
```

Name Spaces

- To avoid naming collisions (two different elements with the same name)

```
<text:directory xmlns:text = “ucf:cs:cop4610”>

<text:book> … </text:book>
```

- A commons practice is to use URLs
- A default name space can be specified with:

```
<directory xmlns = “cs.ucf”>

<book> … </book>
```

DTD

- Define XML document’s structure: permitted elements, attributes, etc.
- It is optional (not every XML document is required to have a corresponding DTD)
- A DTD is defined using *Extended Backus-Naur Form (EBNF)* grammar

XML and DTD

Declaration:

- Internal:

```
<!DOCTYPE myMessage [<!ELEMENT myMessage (#PCDATA)> ]>
```

- External

```
<!DOCTYPE myMessage SYSTEM “myDTD.dtd”>
```

or:

```
<!DOCTYPE html PUBLIC “-//W3C//DTD XHTML 1.0 Strict//EN”
   “http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd”>
```

Notes:

- myMessage is the root element
- PCDATA = Parsable character data

Example

```
1  <?xml version = "1.0" standalone = "yes"?>
2
3  <!-- Fig. B.5 : mixed.xml         -->
4  <!-- Mixed content type elements -->
5
6  <!DOCTYPE format [
7     <!ELEMENT format ( #PCDATA | bold | italic )*>
8     <!ELEMENT bold ( #PCDATA )>
9     <!ELEMENT italic ( #PCDATA )>
10 ]>
11
12 <format>
13    Book catalog entry:
14    <bold>XML</bold>
15    <italic>XML How to Program</italic>
16    This book carefully explains XML-based systems development.
17 </format>
```

XHTML

- XHTML is a family of document types and modules that reproduce, subset, and extend HTML 4
- XHTML family document types a small subset of XML, and ultimately are designed to work in conjunction with XML-based user agents.
- Create discipline (referred to as rigor) within the syntax to avoid inconsistencies in browser interpretation and encourage professional coding practices.

Modules

Applet — applet, param
Block phrasal — address, blockquote, pre, h1-h6
Block presentational — center, hr
Block structural — div, p
Inline phrasal — abbr, acronym, cite, code, dfn, em, kbd, q, samp, strong, var
Inline presentational — b, basefont, big, font, i, s, small, strike, sub, sup, tt, u
Inline structural — bdo, br, del, ins, span
Linking — a, base, link
Lists — dir, dl, dt, dd, ol, ul, li, menu
Simple forms — form, input, select, option, textarea
Extended forms — button, fieldset, label, legend, optgroup, option, select, textarea
Simple tables — table, td, th, tr
Extended tables — caption, col, colgroup, tbody, tfoot, thead
Images — img
Image maps — area, map
Objects — object, param
Frames — frameset, frame, iframe, noframes
Events — onclick, ondblclick, onmousedown, onmouseup, onmouseover, onmousemove, onmouseout, onkeypress, onkeydown, onkeyup
Metadata — meta, title
Scripts — noscript, script
Styles — style element and attribute
Structure — html, head, body

XHTML (vs. HTML)

- Tags and attributes must be in lowercase
- All XHTML elements must be closed
- Attribute values must be quoted and minimization is forbidden
- The *id* attribute replaces the *name* attribute
- Documents must conform to XML rules
- XHTML documents have some mandatory elements

XHTML example

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
<title>COP 4610L</title>
</head>
<body>
<p>Moved to <a href="http://www.cs.ucf.edu/">UCF</a>.</p>
</body>
</html>
```

Document Object Model (DOM)

- XML Parsers are of two basic types:
  - Hierarchical tree based (DOM)
  - Event based (SAX)
- XML DOM is a W3C recommendation
- A DOM-based parser exposes a programmatic library (DOM API) that allows access to data in an XML document
- Sun Microsystems: JAXP (Java API for XML Processing)

DOM example: XML document

```
1  <?xml version = "1.0"?>
2
3  <!-- Fig. C.1: article.xml        -->
4  <!-- Article formatted with XML   -->
5
6  <article>
7
8     <title>Simple XML</title>
9
10    <date>July 31, 2001</date>
11
12    <author>
13       <fname>Tem</fname>
14       <lname>Nieto</lname>
15    </author>
16
17    <summary>XML is easy.</summary>
18
19    <content>Once you have mastered XHTML, you can easily learn
20       XML. You must remember that XML is not for
21       displaying information but for managing information.
22    </content>
23
24 </article>
```

DOM example: Parser 1/4

```
1  // Fig C.2 : XMLInfo.java
2  // Outputs node information
3
4  // Java core libraries
5  import java.io.*;
6
7  // Java standard extensions
8  import javax.xml.parsers.*;
9
10 // third-party libraries
11 import org.w3c.dom.*;
12 import org.xml.sax.*;
13
14 public class XMLInfo {
15
```

DOM example: Parser 2/4

```
16      public static void main( String args[] )
17      {
18
19         if ( args.length != 1 ) {
20            System.err.println( "Usage: java XMLInfo input.xml" );
21            System.exit( 1 );
22         }
23
24         try {
25
26            // create DocumentBuilderFactory
27            DocumentBuilderFactory factory =
28               DocumentBuilderFactory.newInstance();
29
30            // create DocumentBuilder
31            DocumentBuilder builder = factory.newDocumentBuilder();
32
33            // obtain document object from XML document
34            Document document = builder.parse(
35               new File( args[ 0 ] ) );
36
37            // get root node
38            Node root = document.getDocumentElement();
39
40            System.out.print( "Here is the document's root node:" );
41            System.out.println( " " + root.getNodeName() );
```

DOM example: Parser 3/4

```
43            System.out.println( "Here are its child elements: " );
44            NodeList childNodes = root.getChildNodes();
45            Node currentNode;
46
47            for ( int i = 0; i < childNodes.getLength(); i++ ) {
48
49               currentNode = childNodes.item( i );
50
51               // print node name of each child element
52               System.out.println( currentNode.getNodeName() );
53            }
54
55            // get first child of root element
56            currentNode = root.getFirstChild();
57
58            System.out.print( "The first child of root node is: " );
59            System.out.println( currentNode.getNodeName() );
60
61            // get next sibling of first child
62            System.out.print( "whose next sibling is: " );
63            currentNode = currentNode.getNextSibling();
64            System.out.println( currentNode.getNodeName() );
65
66            // print value of first child's next sibling
67            System.out.println( "value of " +
68               currentNode.getNodeName() + " element is: " +
```

DOM example: Parser 4/4

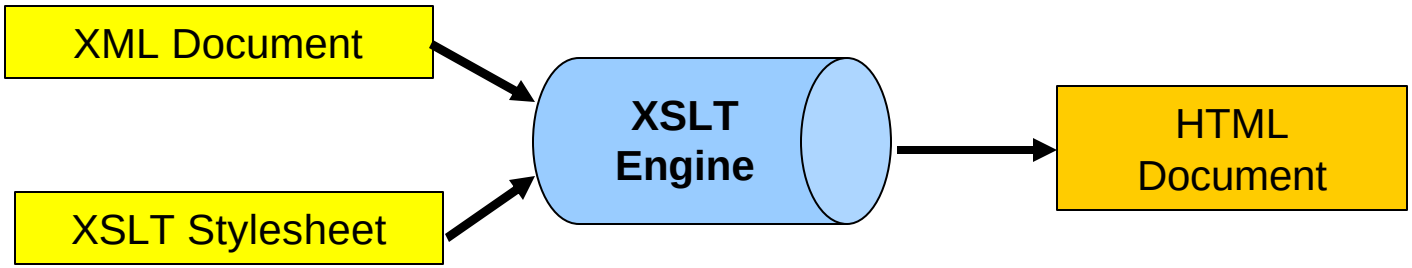
```
69                currentNode.getFirstChild().getNodeValue() );
70
71             // print name of next sibling's parent
72             System.out.print( "Parent node of " +
73                currentNode.getNodeName() + " is: " +
74                currentNode.getParentNode().getNodeName() );
75          }
76
77          // handle exception creating DocumentBuilder
78          catch ( ParserConfigurationException parserError ) {
79             System.err.println( "Parser Configuration Error" );
80             parserError.printStackTrace();
81          }
82
83          // handle exception reading data from file
84          catch ( IOException fileException ) {
85             System.err.println( "File IO Error" );
86             fileException.printStackTrace();
87          }
88
89          // handle exception parsing XML document
90          catch ( SAXException parseException ) {
91             System.err.println( "Error Parsing Document" );
92             parseException.printStackTrace();
93          }
94       }
95    }
```

DOM example: output

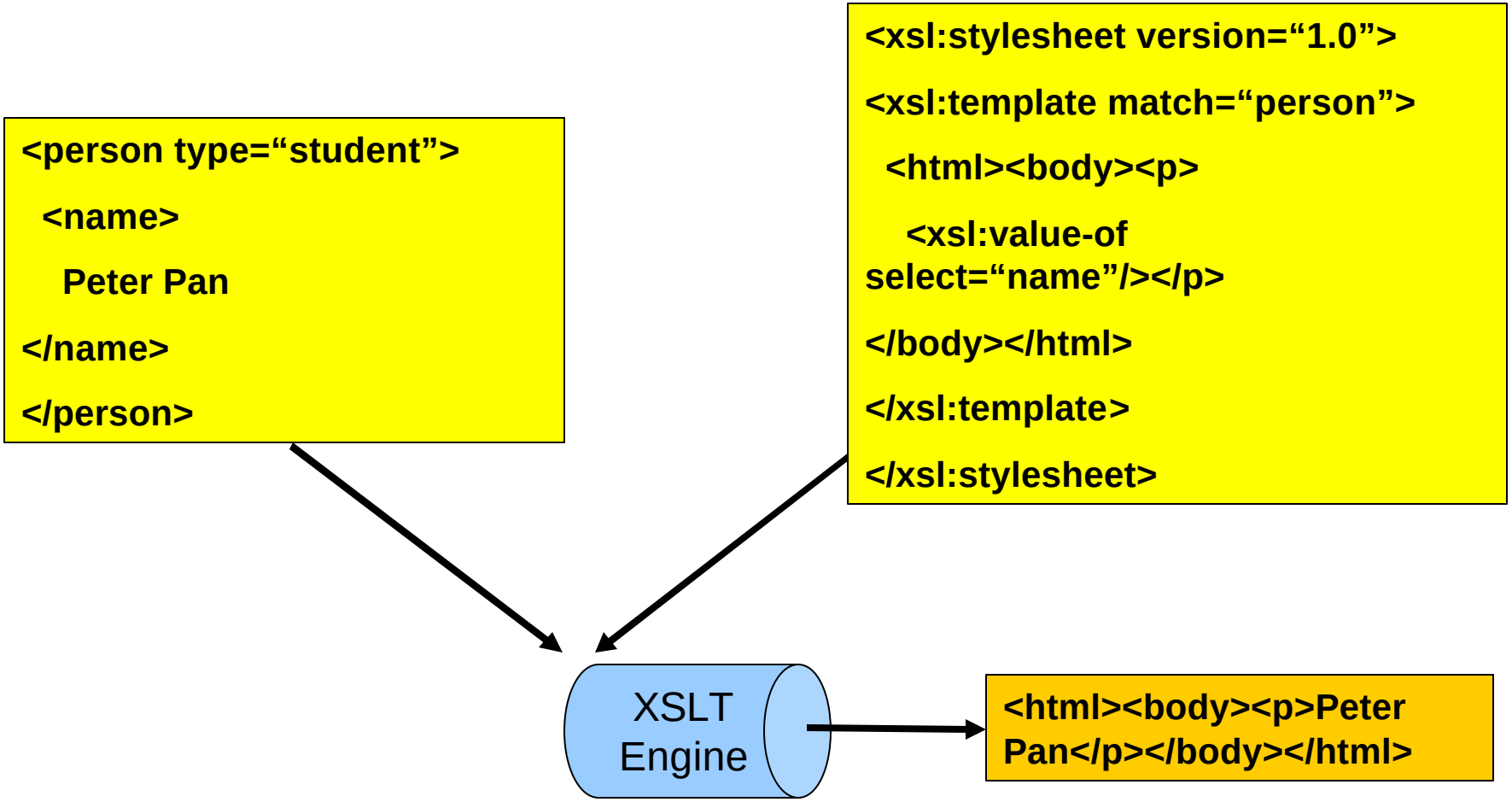
```
Here is the document's root node: article
Here are its child elements:
title
date
author
summary
content
The first child of root node is: title
whose next sibling is: date
value of date element is: July 31, 2001
Parent node of date is: article
```

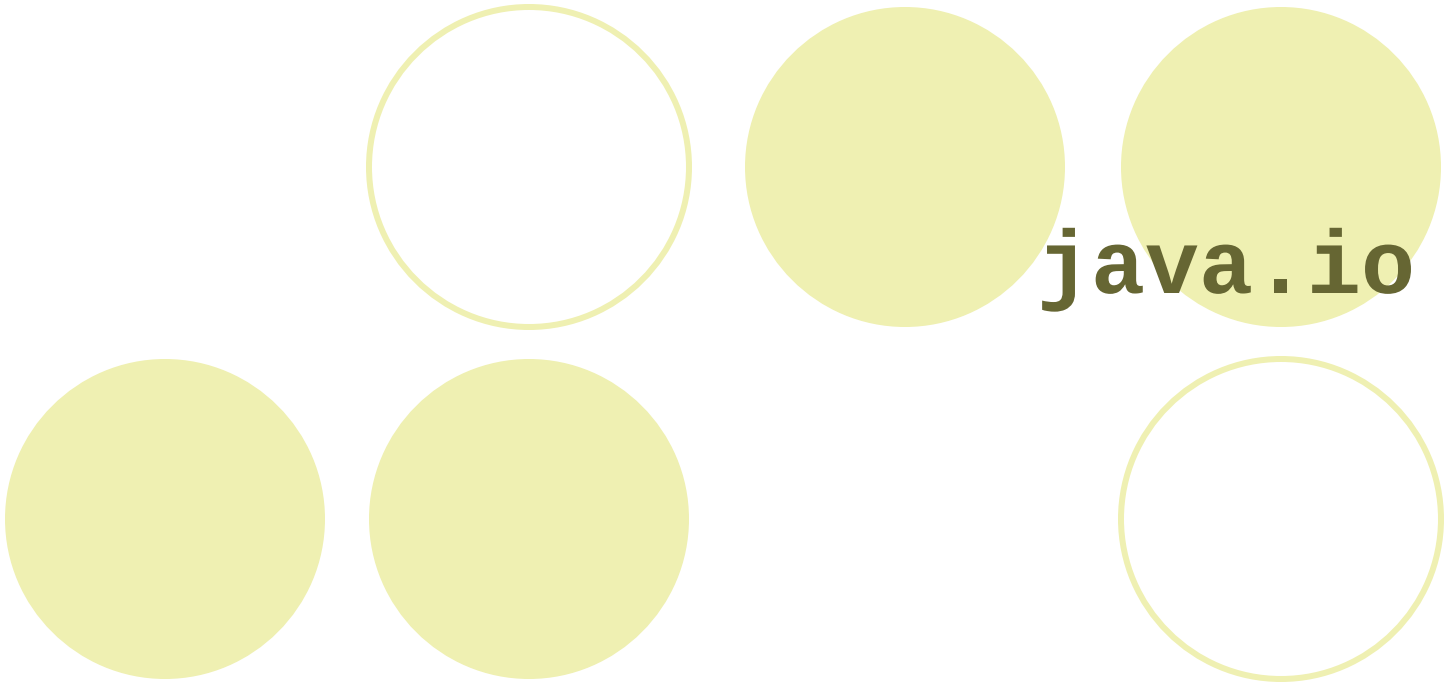
XSLT

- XSL = Extensible Stylesheet Language
- XSLT = XSL Transformations
- Provides rules for formatting XML documents



XSLT Example





java.io

Input/Output

- Java views files and devices as a stream of bytes
  - Example: System.in, System.out, and System.err
  - Streams can be redirected
- A file ends with end-of-file marker or a specific byte number
- Abstract classes:
  - Byte-based streams
    - **`InputStream`**
    - **`OutputStream`**
  - Character-based streams (Unicode two-byte character streams)
    - **`Reader`**
    - **`Writer`**

I/O Streams: File Streams

- File processing with classes in package java.io
  - **`FileInputStream`** for byte-based input from a file
  - **`FileOutputStream`** for byte-based output to a file
  - **`FileReader`** for character-based input from a file
  - **`FileWriter`** for character-based output to a file

```
import java.io.*;
public class Copy { public static void main(String[] args) throws
   IOException {
    File inputFile = new File("farrago.txt");
    File outputFile = new File("outagain.txt");
    FileReader in = new FileReader(inputFile);
    FileWriter out = new FileWriter(outputFile);
    int c;
    while ((c = in.read()) != -1) out.write(c);
    in.close(); out.close(); }
}
```

I/O Streams: Print Streams

- Define convenient printing methods that are the easiest streams to write to
- You will often see other writable streams wrapped in one of these
- Classes: **`PrintWriter`** and **`PrintStream`**

I/O Streams: Data Conversion

- Read or write primitive data types in a machine-independent format.
- Classes: DataInputStream, DataOutputStream
- Methods: writeDouble(), writeChar(), writeBytes(), etc.

I/O Streams: Converting between bytes and Chars

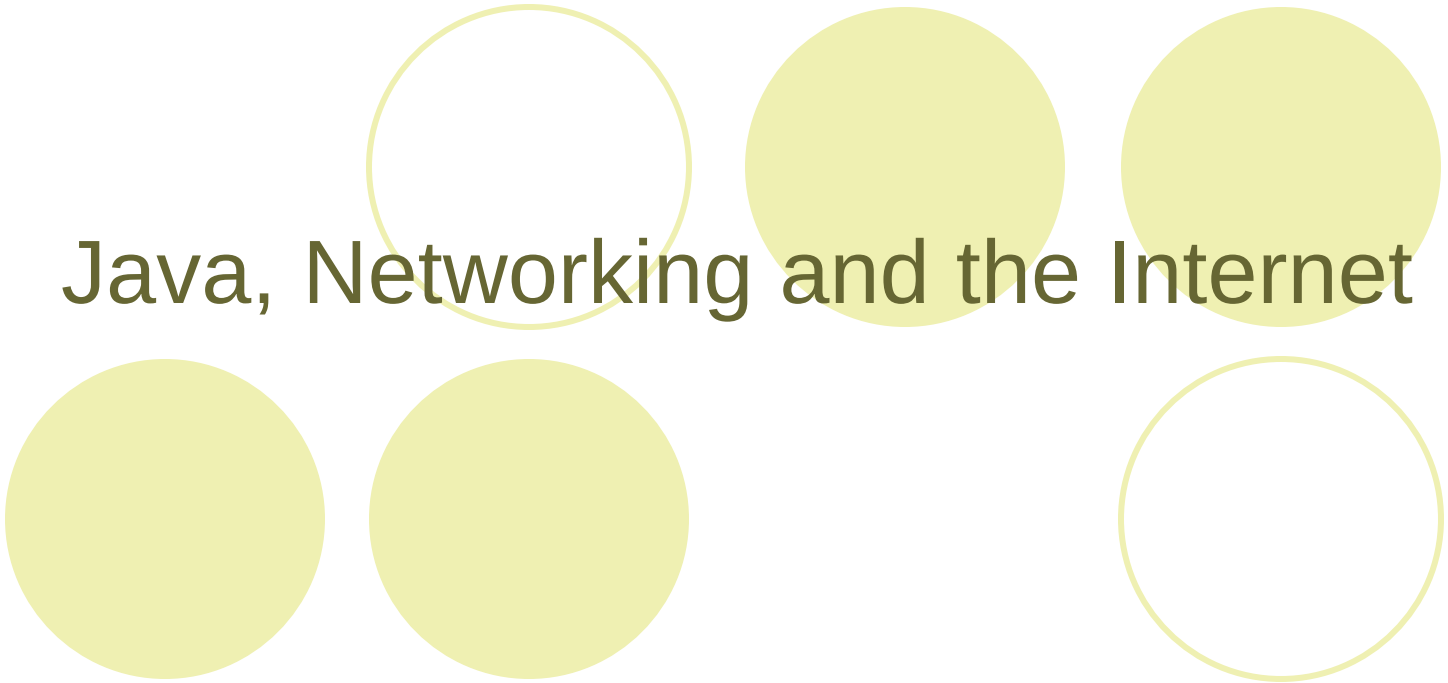
- A reader and writer pair that forms the bridge between byte streams and character streams
- An **`InputStreamReader`** reads bytes from an InputStream and converts them to characters
- An **`OutputStreamWriter`** converts characters to bytes and then writes them to an **`OutputStream`**

I/O Streams: Buffering

- Improves performance of I/O
- Copies each output to a region of memory called a buffer
- Entire buffer output at once

  Use: one long disk access takes less time than many smaller ones
- Classes: **`BufferedOutputStream, BufferedInputStream, BufferedReader, BufferedWriter`**
- Methods: **`readLine(), writeLine()`**

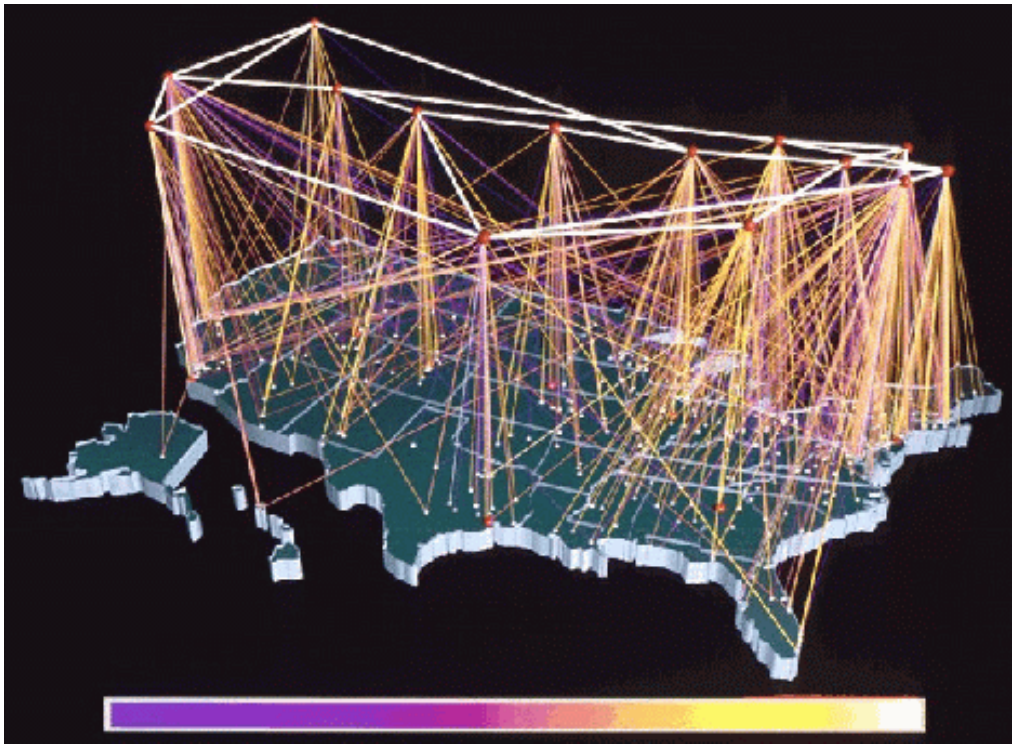
Java, Networking and the Internet



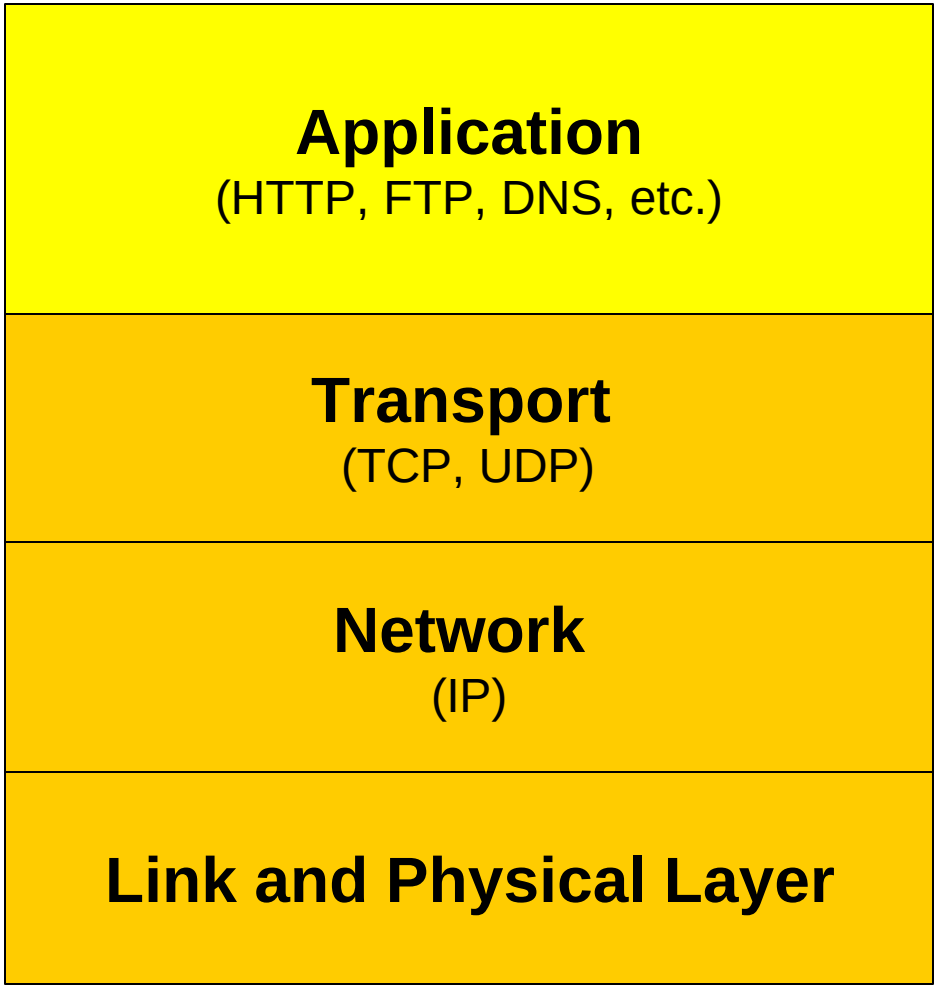
java.net

- “High-level” APIs
  - Implement commonly used protocols (e.g. HTML, FTP)
- “Low-Level” APIs
  - Socket-based communications
    - Applications view networking as streams of data
    - Connection-based protocol
    - Uses TCP (Transmission Control Protocol)
  - Packet-based communications
    - Individual packets transmitted
    - Connectionless service
    - Uses UDP (User Datagram Protocol)

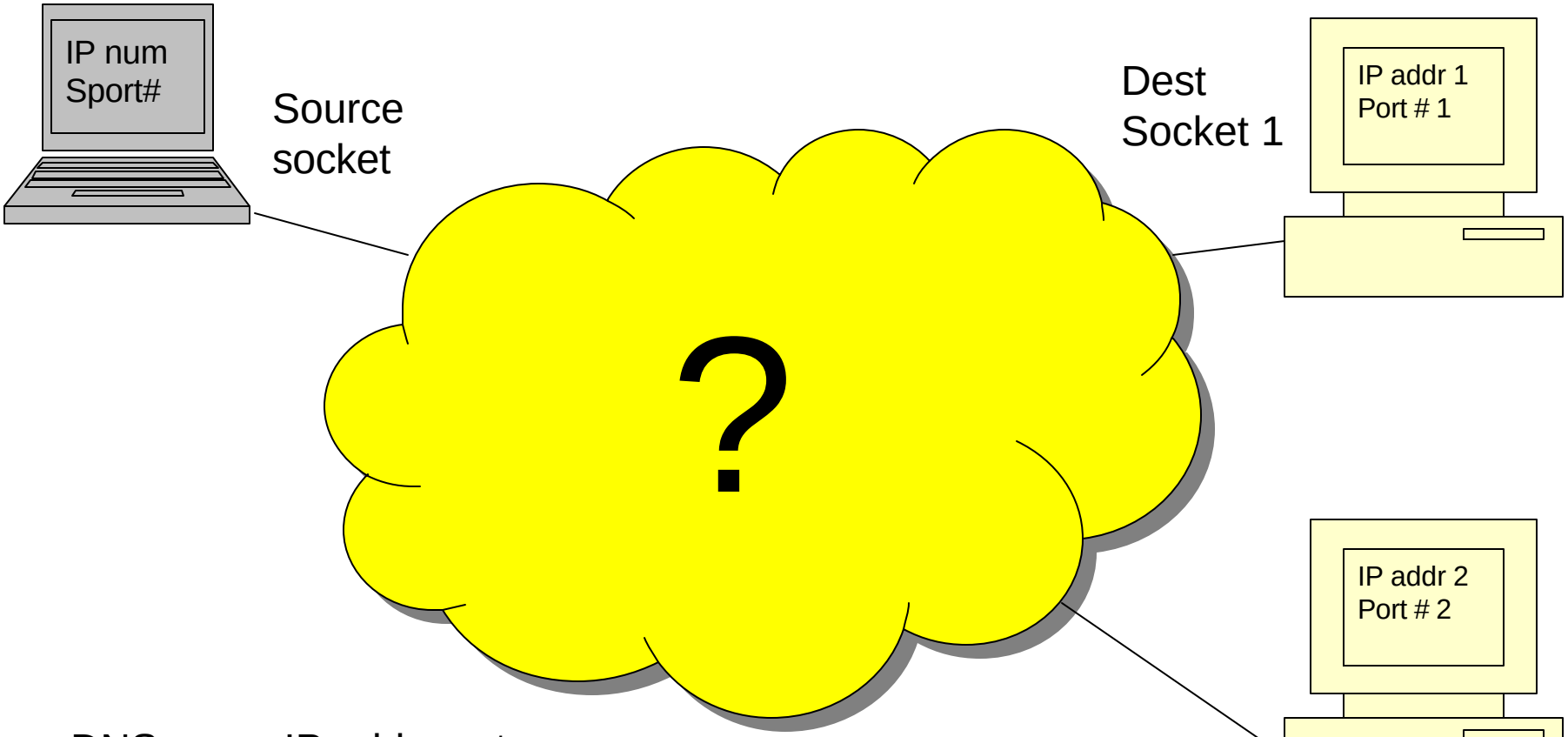
The Internet in the USA



Internet Reference Model



Application's View of the Net



- DNS maps IP address to names
- Special IP address: 127.0.0.1 = localhost

Type of Services

- connection-oriented service
  - TCP - Transmission Control Protocol
  - *reliable, in-order* byte-stream data transfer
- connectionless service
  - UDP - User Datagram Protocol
  - Unreliable

Socket API

- introduced in BSD4.1 UNIX, 1981
- explicitly created, used, released by apps
- client/server paradigm

Socket programming *with UDP*

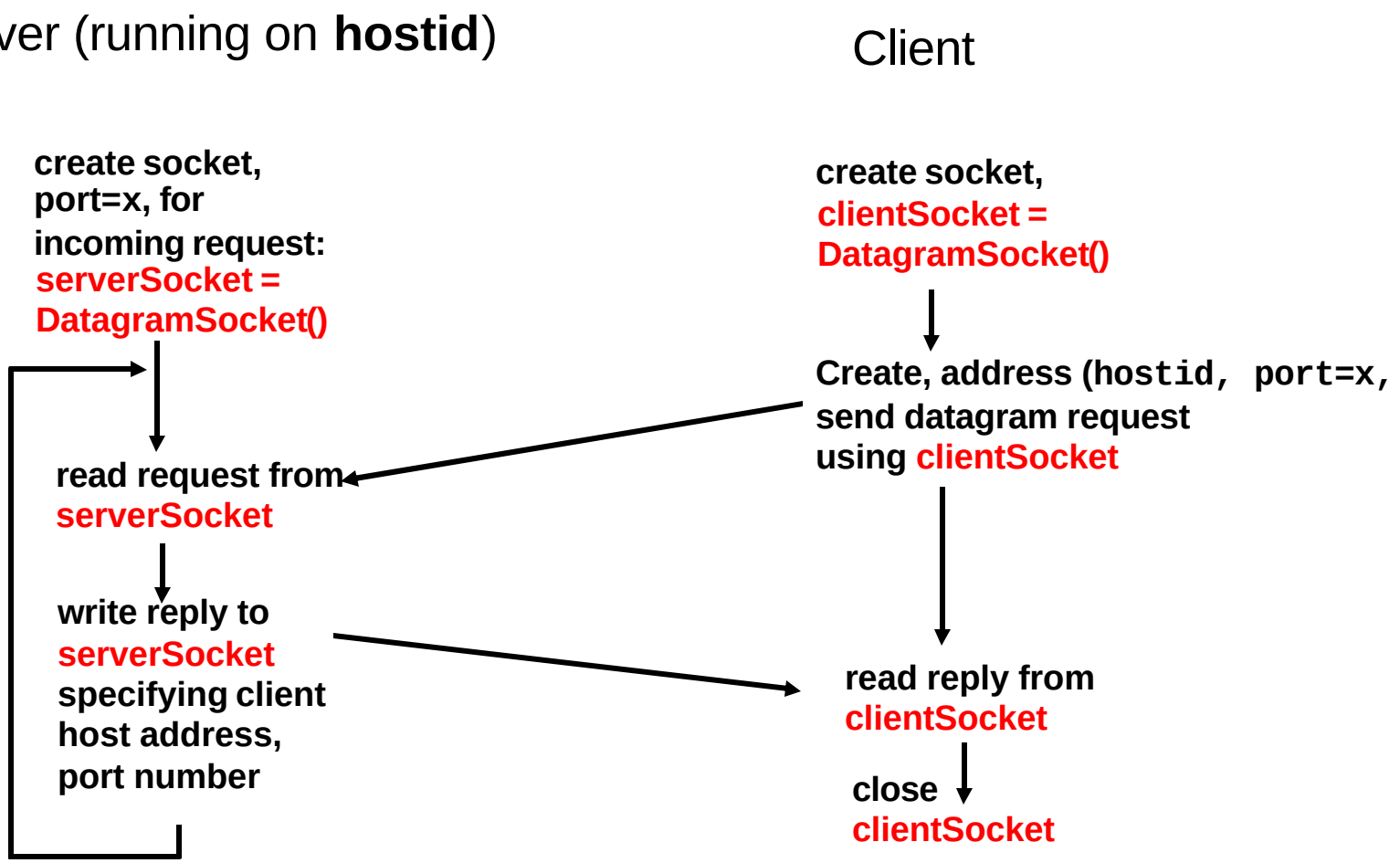
UDP: no “connection” between client and server

- no handshaking
- sender explicitly attaches IP address and port of destination to each packet
- server must extract IP address, port of sender from received packet

UDP: transmitted data may be received out of order, or lost

Application viewpoint: UDP provides unreliable transfer of groups of bytes (“datagrams”) between client and server

Example: client/server socket interaction via UDP



Example: Java client (UDP)

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

class UDPClient {
   public static void main(String args[]) throws Exception
   {

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

      DatagramSocket clientSocket = new DatagramSocket();

      InetAddress IPAddress = InetAddress.getByName("localhost");

      byte[] sendData = new byte[1024];
      byte[] receiveData = new byte[1024];

      String sentence = inFromUser.readLine();

      sendData = sentence.getBytes();
```

Create input stream

Create client socket

Translate hostname to IP address using DNS

Example: Java client (UDP), cont.

Create datagram with data-to-send, length, IP addr, port

Send datagram to server

Read datagram from server

```
DatagramPacket sendPacket =
  new DatagramPacket(sendData, sendData.length, IPAddress,
                                                  9876);
  clientSocket.send(sendPacket);

DatagramPacket receivePacket =
   new DatagramPacket(receiveData, receiveData.length);

clientSocket.receive(receivePacket);

String modifiedSentence =
    new String(receivePacket.getData());

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

}
```

Example: Java server (UDP)

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

class UDPServer {
  public static void main(String args[]) throws Exception
   {

      DatagramSocket serverSocket = new DatagramSocket(9876);

      byte[] receiveData = new byte[1024];
      byte[] sendData  = new byte[1024];

      while(true)
       {

        DatagramPacket receivePacket =
          new DatagramPacket(receiveData, receiveData.length);

        serverSocket.receive(receivePacket);
```

Create datagram socket at port 9876

Create space for received datagram

Receive datagram

Example: Java server (UDP), cont

```
String sentence = new String(receivePacket.getData());

InetAddress IPAddress = receivePacket.getAddress();

int port = receivePacket.getPort();

String capitalizedSentence = sentence.toUpperCase();

sendData = capitalizedSentence.getBytes();

DatagramPacket sendPacket =
   new DatagramPacket(sendData, sendData.length, IPAddress,
                      port);

serverSocket.send(sendPacket);
    }
  }
}
```

Get IP addr, port #, of sender

Create datagram to send to client

Write out datagram to socket

End of while loop, loop back and wait for another datagram

Socket programming *with TCP*

- Server process must first be running (must have created a socket)
- Client contacts server by creating client-local TCP socket specifying IP address and port number of server process. 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
  - source port numbers used to distinguish clients

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

Establishing a Simple Server Using Stream Sockets

Five steps to create a simple server in Java:

1. `ServerSocket` object

   Registers an available port and a maximum number of clients

2. Each client connection handled with `Socket` object

   Server blocks until client connects

3. Sending and receiving data

   `OutputStream` to send and `InputStream` to receive data

   Methods `getInputStream` and `getOutputstream` (use on Socket object)

4. Process phase

   Server and Client communicate via streams

5. Close streams and connections

Establishing a Simple Client Using Stream Sockets

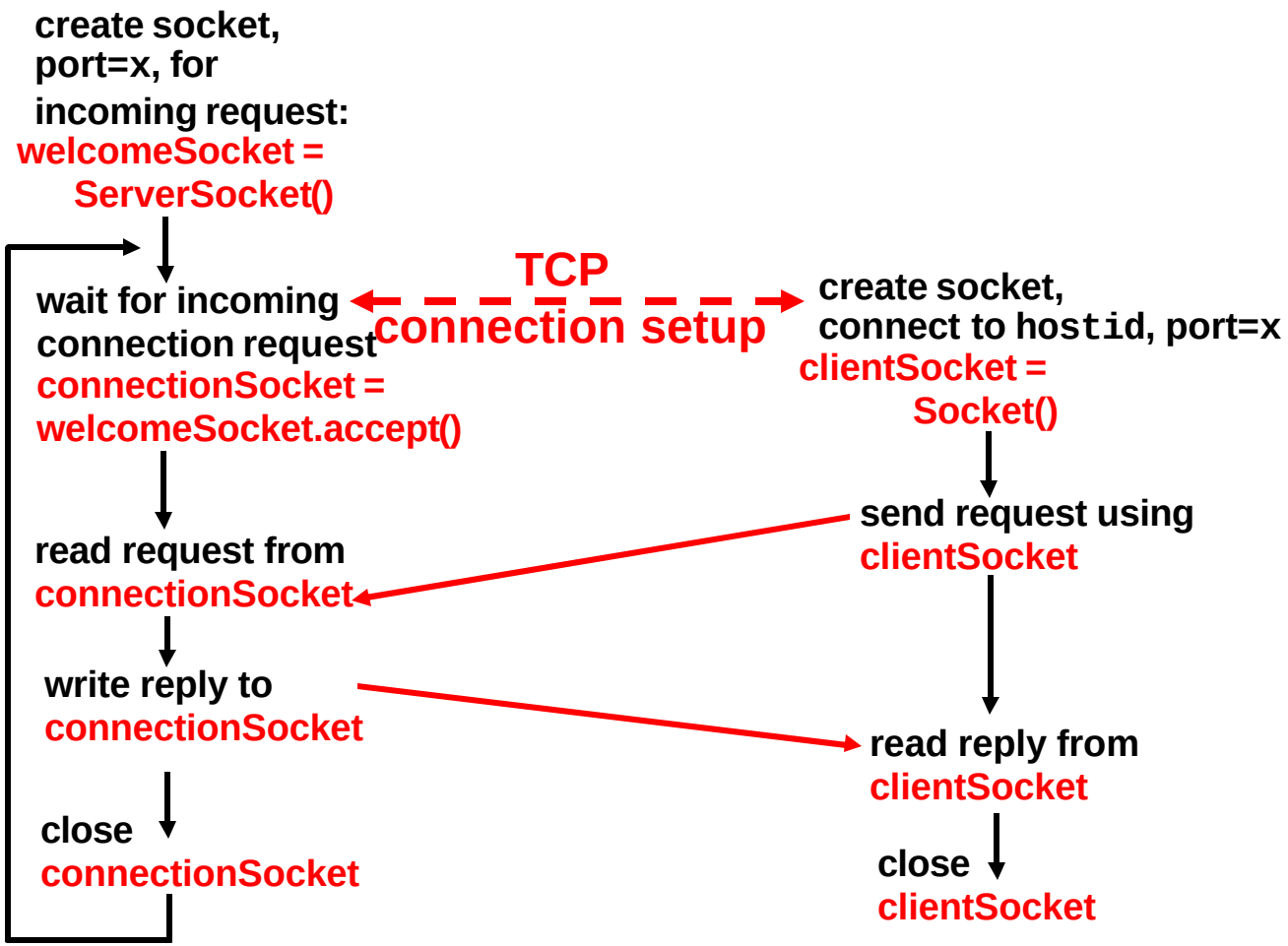
Four steps to create a simple client in Java

1. Create a `Socket` object for the client
2. Obtain `Socket`'s `InputStream` and `Outputstream`
3. Process information communicated
4. Close streams and `Socket`

Example: client/server socket interaction via TCP

Server (running on **hostid**)

Client



Example: Java client (TCP)

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

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

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

        Socket clientSocket = new Socket("localhost", 6789);

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

Create input stream

Create client socket, connect to server

Create output stream attached to socket

Example: Java client (TCP), cont'd

Create input stream attached to socket

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

sentence = inFromUser.readLine();

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

modifiedSentence = inFromServer.readLine();

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

clientSocket.close();

    }
}
```

Send line to server

Read line from server

Example: Java server (TCP)

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

class TCPServer {

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

     ServerSocket welcomeSocket = new ServerSocket(6789);

     while(true) {

       Socket connectionSocket = welcomeSocket.accept();

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

Create welcoming socket at port 6789

Wait, on welcoming socket for contact by client

Create input stream, attached to socket

Example: Java server (TCP), cont'd

Create output stream, attached to socket

Read in line from socket

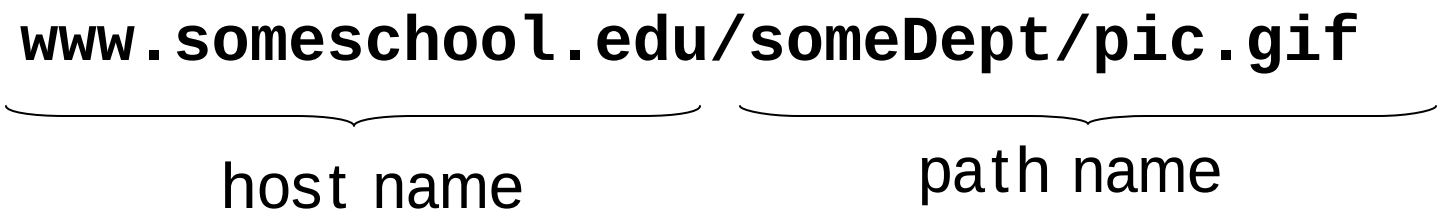
Write out line to socket

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

clientSentence = inFromClient.readLine();

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

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

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

Web and HTTP

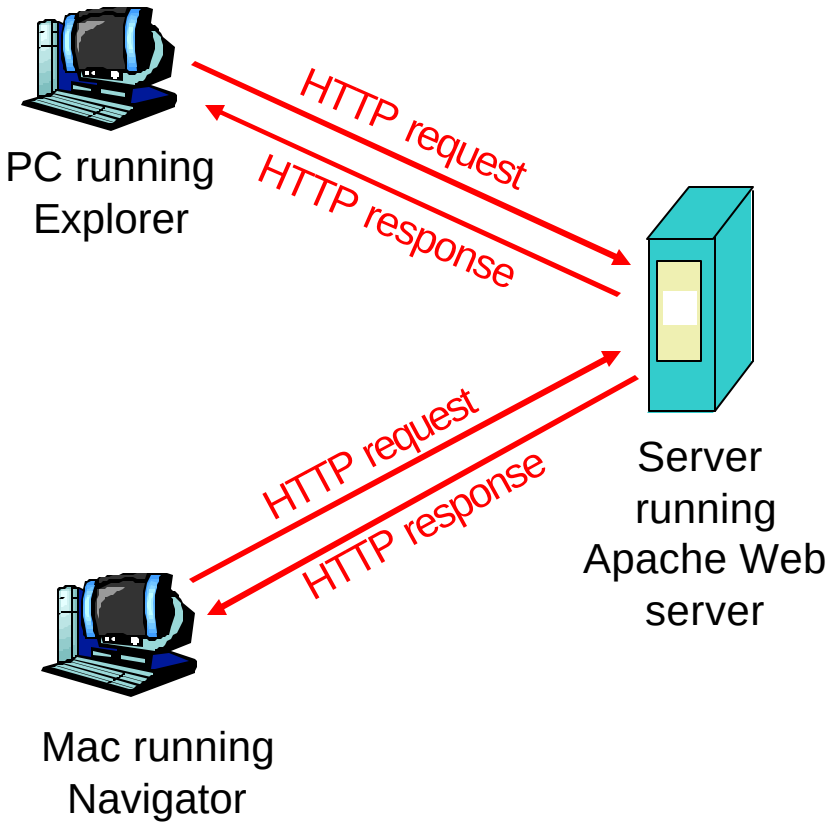
- Web page consists of objects
- Object can be HTML file, JPEG image, Java applet, audio file,…
- Web page consists of base HTML-file which includes several referenced objects
- Each object is addressable by a URL
- Example URL:



HTTP overview

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
- HTTP 1.0: RFC 1945
- HTTP 1.1: RFC 2068



HTTP overview (continued)

Uses TCP:

- 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

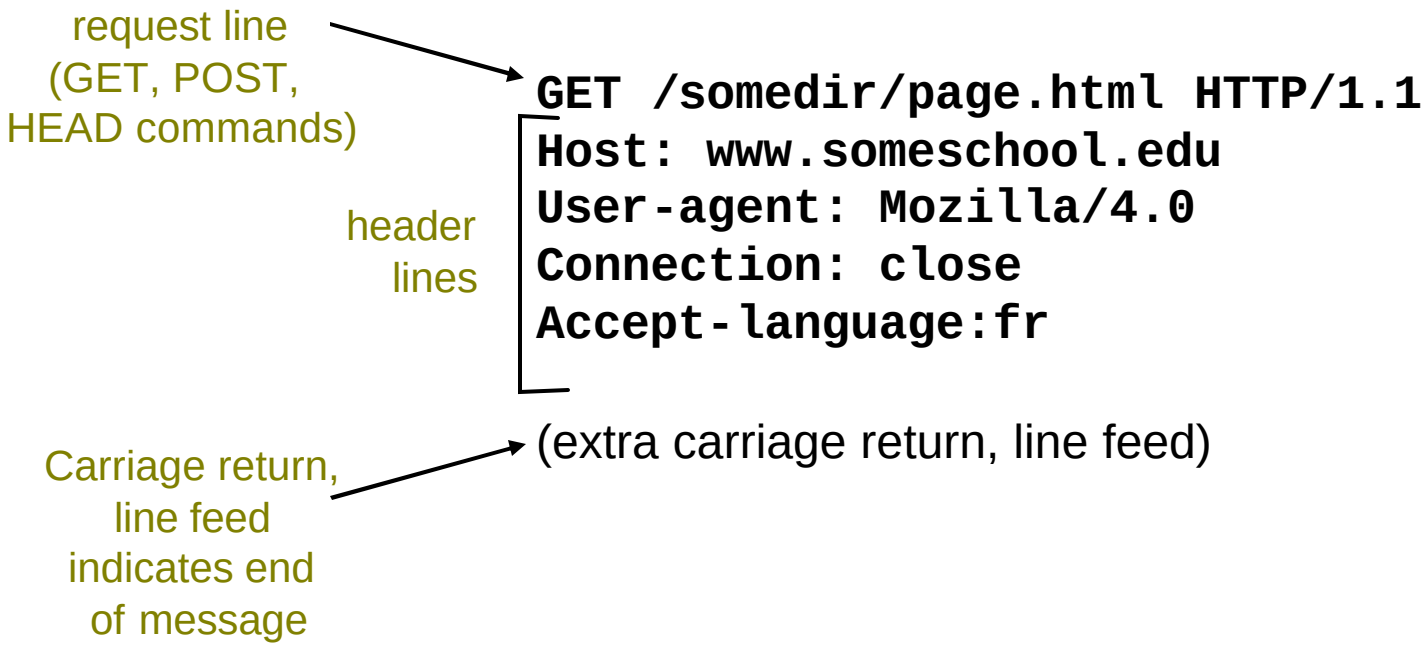
aside

Protocols that maintain “state” are complex!

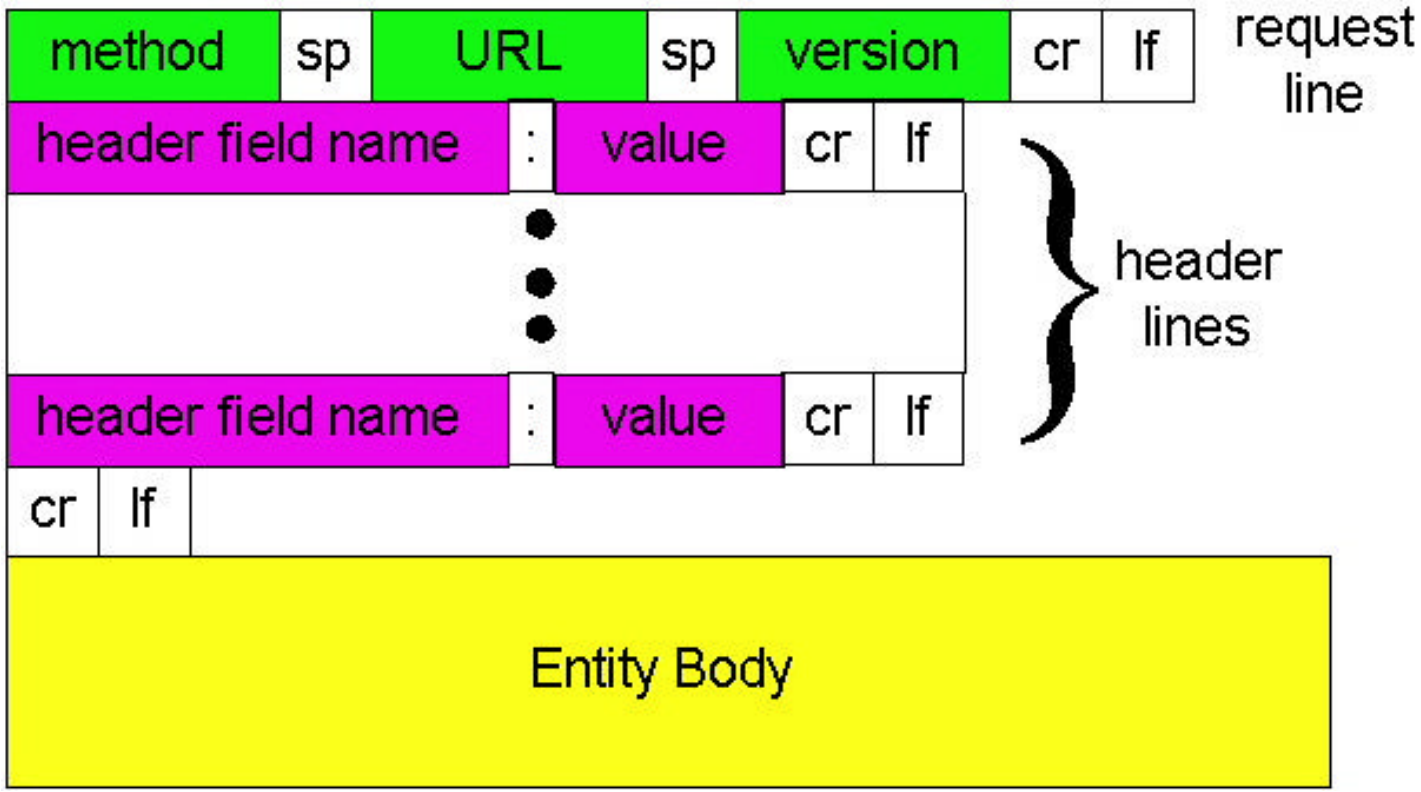
- past history (state) must be maintained
- if server/client crashes, their views of “state” may be inconsistent, must be reconciled

HTTP request message

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



HTTP request message: general format



Uploading form input

Post method:

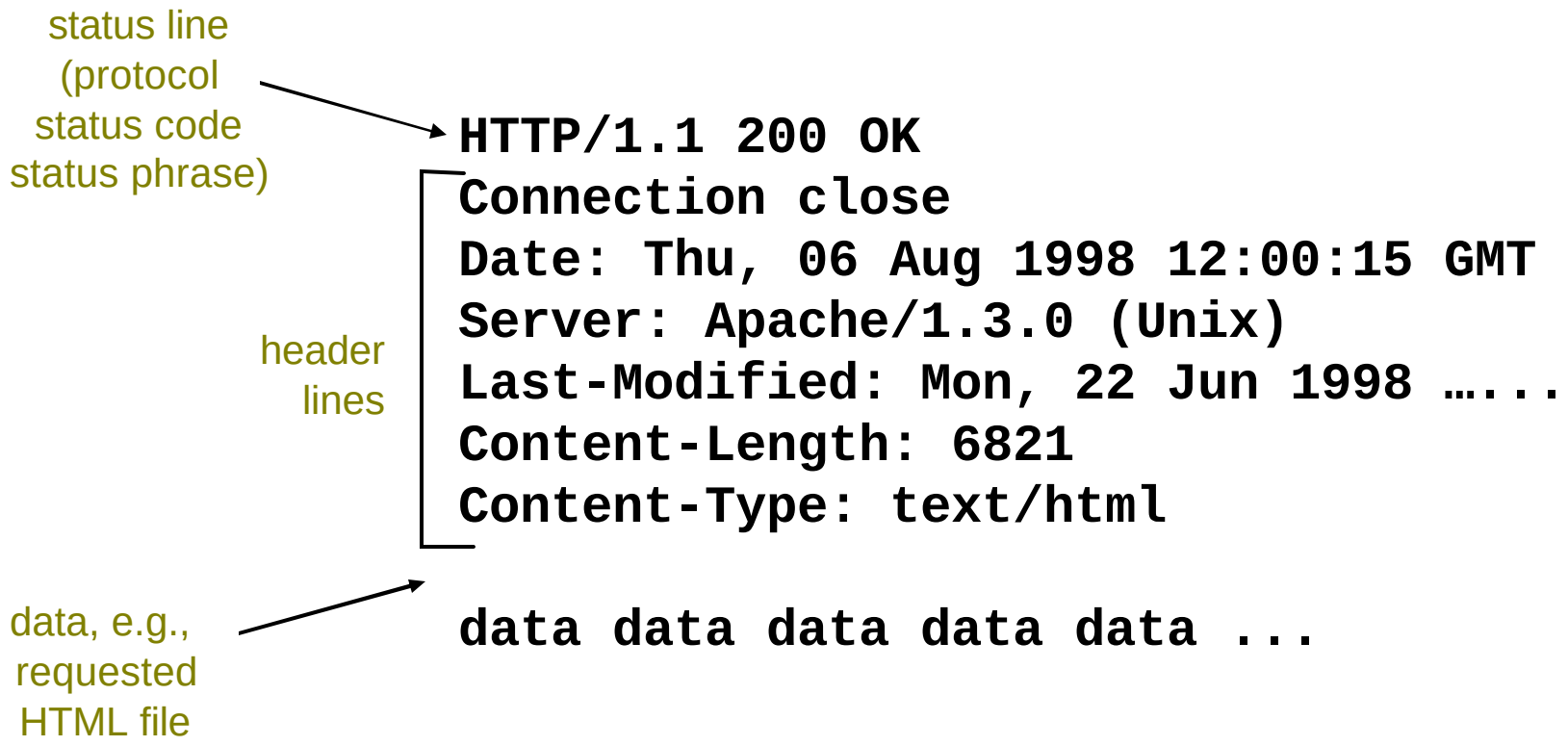
- Web page often includes form input
- Input is uploaded to server in entity body

URL method:

- Uses GET method
- Input is uploaded in URL field of request line:

www.somesite.com/animalsearch?monkeys&banana

HTTP response message



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.cs.ucf.edu 80
```

Opens TCP connection to port 80 (default HTTP server port) at www.cs.ucf.edu
Anything typed in sent to port 80 at www.cs.ucf.edu

2. Type in a GET HTTP request:

```
GET /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!

Building a simple Web server

- handles one HTTP request
- accepts the request
- parses header
- obtains requested file from server's file system
- creates HTTP response message:
  - header lines + file
- sends response to client
- after creating server, you can request file using a browser (e.g. Internet explorer)

WebServer.java (1/3)

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

class WebServer{

    public static void main(String argv[]) throws Exception  {

          String requestMessageLine;
          String fileName;

          ServerSocket listenSocket = new ServerSocket(6789);
          Socket connectionSocket = listenSocket.accept();

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

WebServer.java (2/3)

```
requestMessageLine = inFromClient.readLine();

        StringTokenizer tokenizedLine =
          new StringTokenizer(requestMessageLine);

                    if (tokenizedLine.nextToken().equals("GET")){

        fileName = tokenizedLine.nextToken();

        if (fileName.startsWith("/") == true )
                       fileName  = fileName.substring(1);

                    File file = new File(fileName);
        int numOfBytes = (int) file.length();

        FileInputStream inFile  = new FileInputStream (fileName);

                    byte[] fileInBytes = new byte[numOfBytes];
        inFile.read(fileInBytes);
```

WebServer.java (3/3)

```
            outToClient.writeBytes("HTTP/1.0 200 Document Follows\r\n");

            if (fileName.endsWith(".jpg"))
                    outToClient.writeBytes("Content-Type:
                                                image/jpeg\r\n");
            if (fileName.endsWith(".gif"))
                    outToClient.writeBytes("Content-Type:
                                                image/gif\r\n");

            outToClient.writeBytes("Content-Length: " +
                                            numOfBytes + "\r\n");
            outToClient.writeBytes("\r\n");

            outToClient.write(fileInBytes, 0, numOfBytes);

            connectionSocket.close();
                        }
        else System.out.println("Bad Request Message");
        }
}
```

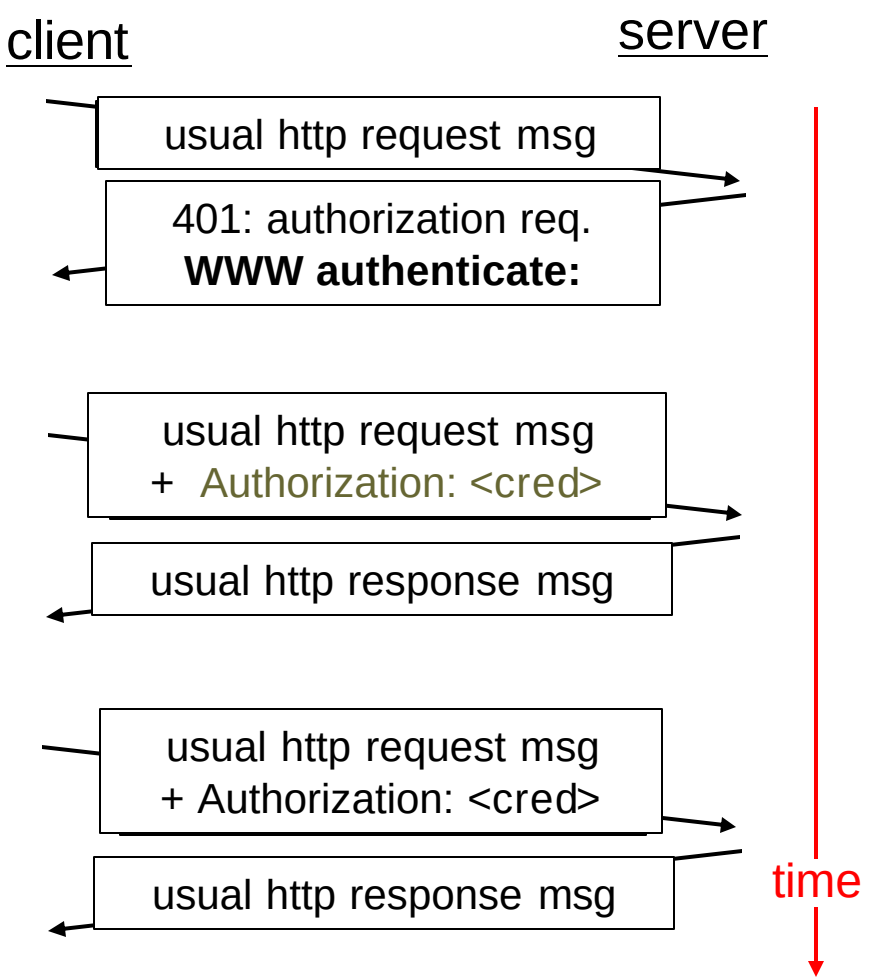
User-server interaction: authorization

Authorization : control access to server content

- authorization credentials: typically name, password
- stateless: client must present authorization in *each* request
  - authorization: header line in each request
  - if no authorization: header, server refuses access, sends

    `WWW authenticate:`

    header line in response



Cookies: keeping “state”

Many major Web sites use cookies

Four components:

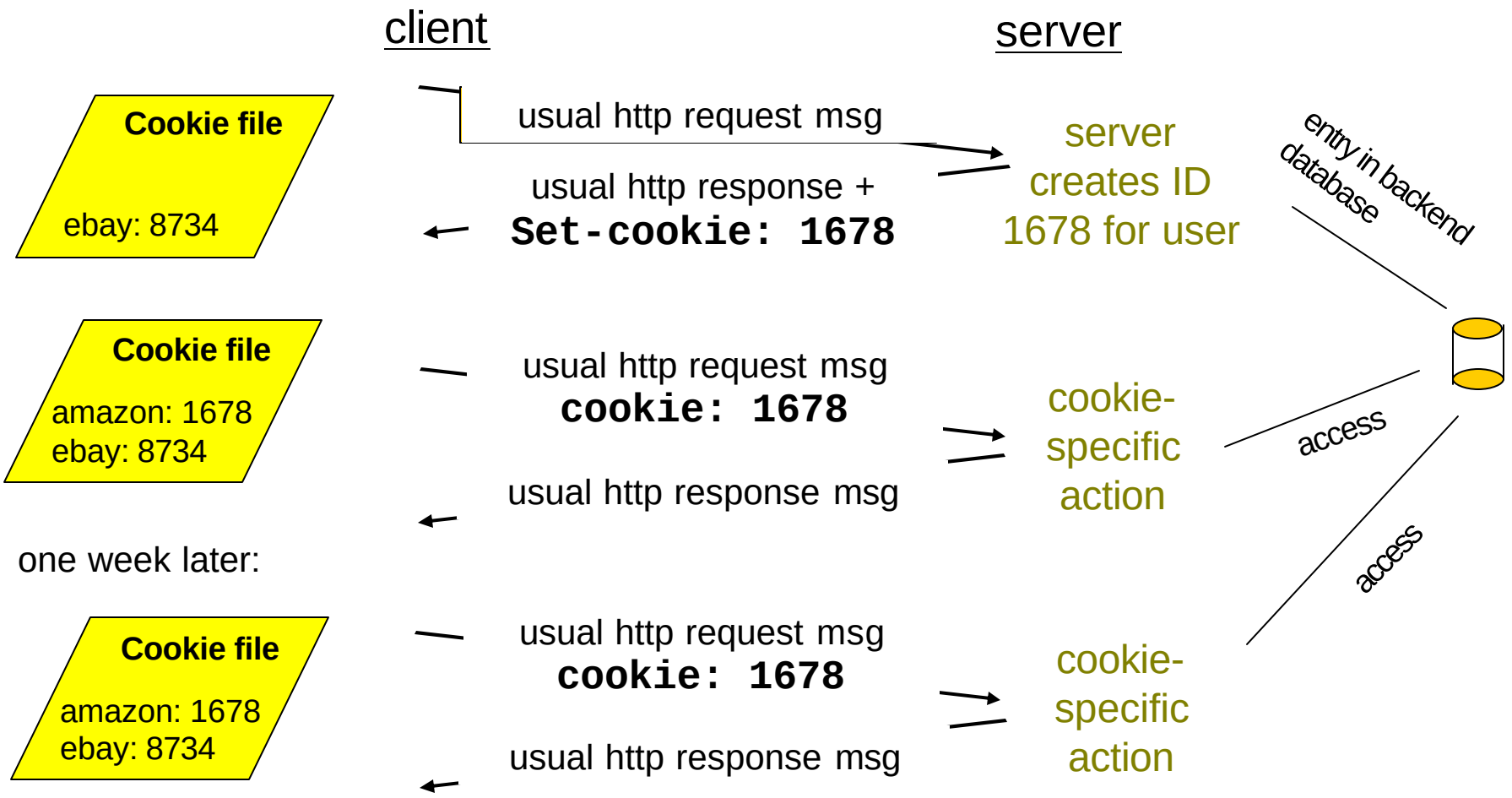
1) cookie header line in the HTTP response message

2) cookie header line in HTTP request message

3) cookie file kept on user’s host and managed by user’s browser

4) back-end database at Web site

Cookies: keeping “state” (cont’d)



Cookies (cont'd)

What cookies can bring:

- authorization
- shopping carts
- recommendations
- user session state (Web e-mail)

aside

Cookies and privacy:

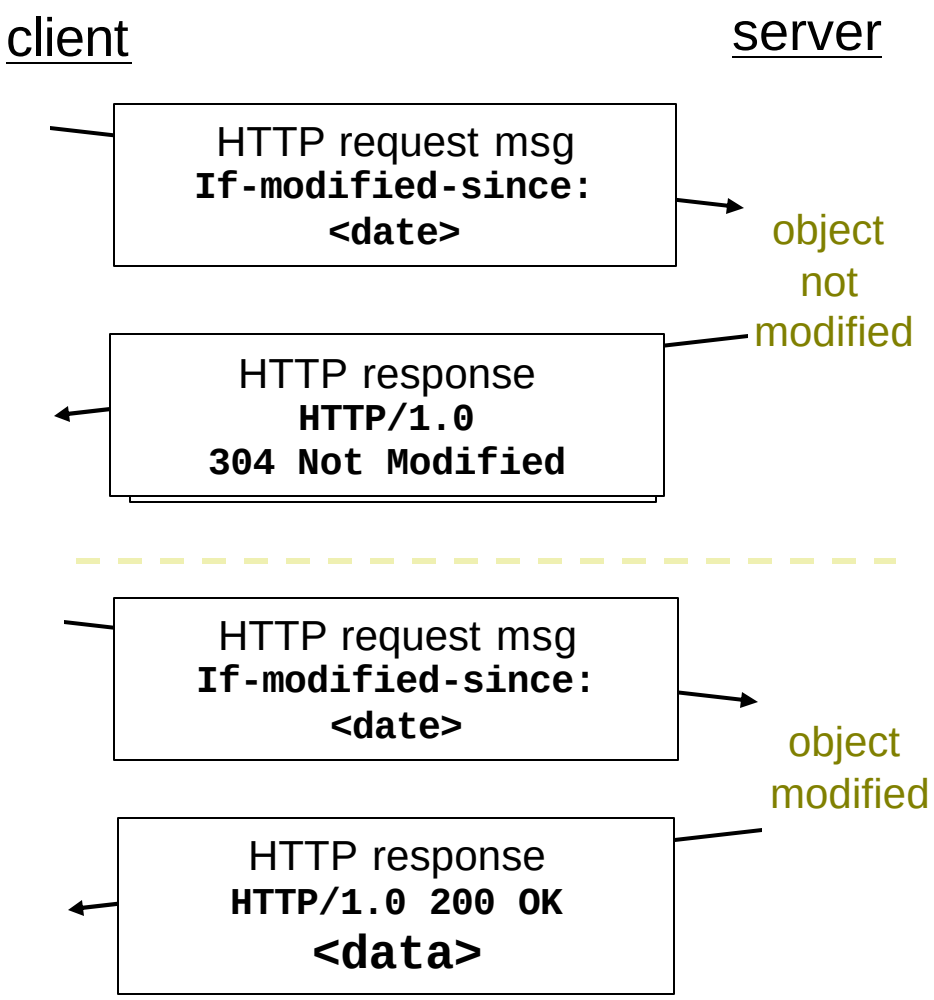
- cookies permit sites to learn a lot about you
- you may supply name and e-mail to sites
- search engines use redirection & cookies to learn yet more
- advertising companies obtain info across sites

Conditional GET: client-side caching

- Goal: don't send object if client has up-to-date cached version
- client: specify date of cached copy in HTTP request

  `If-modified-since: <date>`
- server: response contains no object if cached copy is up-to-date:

  `HTTP/1.0 304 Not Modified`



High-Level Networking API

HTTP-based applications

A few useful classes:

- **URL**
  - Represents the remote object on the WWW
- **URLConnection**
  - Allows finer access to page parameters
- **HttpURLConnection**
  - Extends URLConnection
  - Supports more HTTP-specific features

Example: reading content from URL

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

public class SendReq2 {
 public static void main(String argv[]) throws Exception {

   if(argv.length != 1) {
     System.out.println("Usage: java ReadURL2 <url>");
   System.exit(0); }

   URL url = new URL(argv[0]);
   BufferedReader in = new BufferedReader(new InputStreamReader(
   url.openStream()));
   String line; StringBuffer sb = new StringBuffer();

   while ((line = in.readLine()) != null) {
            sb.append(line);
   }
   in.close();
   System.out.println(sb.toString());
} }
```

Example 2

```
import java.net.*;  import java.io.*;
public class urlTest {
public static void main(String[] args) {
     try {
        URL url = new URL("http://www.google.com/index.html");
        System.out.println("Host: " + url.getHost());
        System.out.println("File: " + url.getPath());

        URLConnection connection = url.openConnection();
        System.out.println("Date: " + connection.getDate());
        System.out.println("Content type:“
                              +connection.getContentType());

        BufferedReader in = new BufferedReader( new InputStreamReader(
                connection.getInputStream()));
        String line;
        while((line=in.readLine())!= null){
                        System.out.println(line);}
        in.close();
     }
     catch(MalformedURLException e){}
     catch(IOException e){}
}}
```