Network application programming

- Fundamental programming abstraction for network communication: socket
- Two main styles of interaction: packets and streams
- A packet enables a one-shot, unreliable transfer of a *datagram*
- A stream is a higher level abstraction to transfer a sequence of bytes with an end-to-end reliability guarantee

Most major operating systems now support *sockets* as the main programming idiom for interprocess and interhost communication. A socket can be used for two styles of communication. In the datagram style, data is packaged into a single packet which is transferred using an unreliable protocol. In the stream style the socket is regarded as a reliable channel that can transfer a stream of bytes. Reliability is implemented as part of the OS using mechanisms such as timeouts and retransmissions. Usually the stream protocols are implemented on top of the datagram support.

Sockets and ports

- Client
  - Create a socket (socket)
  - Map server name to IP addr (gethostbyname)
  - Connect to a given port on the server address
- Server
  - Create a socket (socket)
  - Bind to one or more port numbers (bind)
  - Listen on the socket (listen)
  - Accept client connections (accept)

Using sockets follows a simple sequence of actions at the client and server end. Sockets are like phone numbers; computers are like houses. Sockets are assigned integer identifiers. Together, the IP address and socket number specify the requested connection. Many server sockets are open by default on UNIX computers, e.g., port 23 for telnet, port 80 for HTTP. The server creates a socket and binds it to a desired port number if free. Then it listens on the socket for client connection requests. If there is a request the server accepts it.

A client first creates a socket. This need not be bound to a port number. Then the client locates a server, translating from name to IP address if needed. Then it connects to the desired port number on the specified IP address.

Once the client and server sockets are associated with each other, datagram or stream communication can commence. (We will see example codes in the lab session.) At the end of the communication there are API calls to “tear down” the communication channel.

Stream server example

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

public class daytimeServer {
  public final static int daytimePort = 13;
  public static void main(String[] args) throws Exception {
    ServerSocket theServer;
    Socket theConnection;
    PrintStream p;
    theServer = new ServerSocket(daytimePort);
    while (true) {
      theConnection = theServer.accept();
      p = new PrintStream(theConnection.getOutputStream());
      p.println(new Date());
      theConnection.close();
    }
  }
}
```


Here is a server that serves out the current date and time whenever contacted by a client. It first opens a `ServerSocket` and waits in an infinite loop to accept client requests. The accept method returns a `Socket` which can be used to send data to the client.

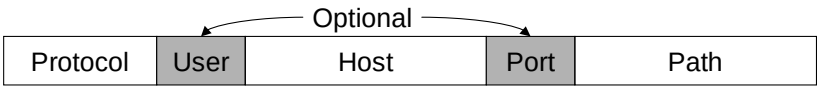
Stream client example

```
import java.net.*;
import java.io.*;
public class daytimeClient {
  public static void main(String[] args) throws Exception {
    Socket theSocket;
    String hostname;
    DataInputStream theTimeStream;

    hostname = args[0];
    theSocket = new Socket(hostname, 13);
    theTimeStream = new
        DataInputStream(theSocket.getInputStream());
    String theTime = theTimeStream.readLine();
    System.out.println("It is " + theTime + " at " + hostname);
  }  // end main
}
```

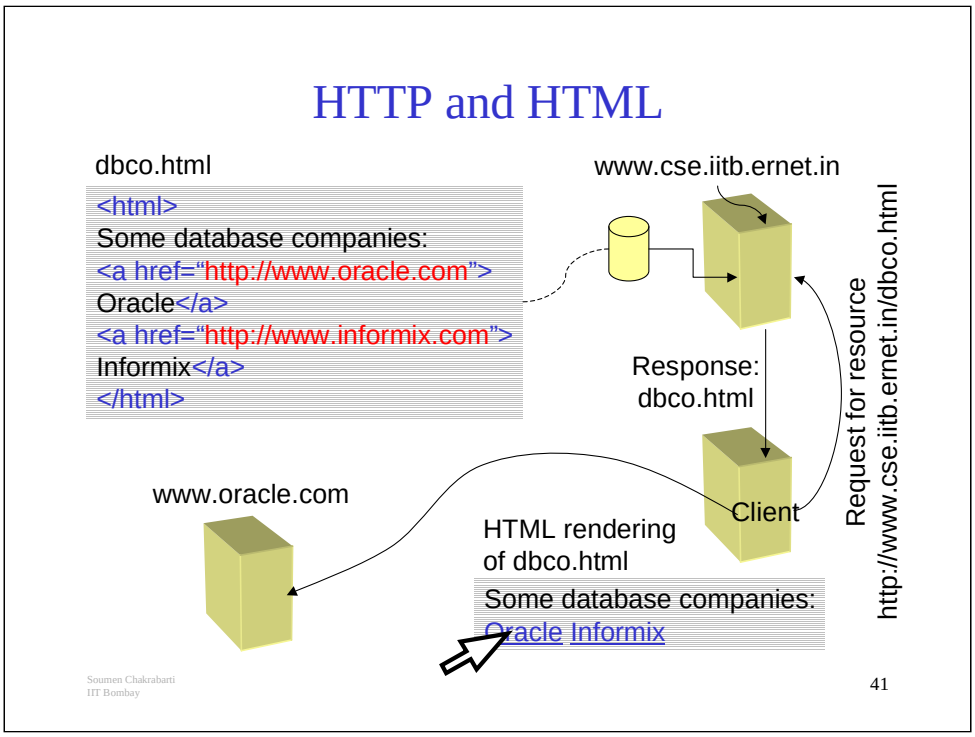

Meanwhile a client creates a `Socket` and connects it to a pre-arranged port number on the server host (passed in as a command line argument). Then it reads one line of text from the resulting `Socket` and prints it.

Naming distributed resources



Protocol	Application	Example
http	Hypertext (HTML)	http://www.cse.iitb.ernet.in/dbco.html
ftp	FTP	ftp://soumen@lcs.mit.edu/home/soumen
file	Local file	file:/users/fac/soumen
news	News group	news:comp.std.unix
news	News article	news:EL8Gur.x4@bhishma.cse.iitb.ernet.in
gopher	Gopher	gopher://gopher.tc.umn.edu/11/Library
mailto	Sendmail	mailto:billg@microsoft.com
telnet	Telnet	telnet:w3.org:80

Resources are named on the Web using a *Universal Resource Locator* or URL. A URL has many fields. The first field specifies a *protocol* such as HTTP (HyperText Transfer Protocol), FTP (File Transfer Protocol) etc. The next optional field is a user name. This field is required, e.g., for sending mail, but not for HTTP. The third field is a host name or IP address. This field is used to locate the computer to connect to. The next field is an optional port number. (You already know what a port is.) Each protocol has a pre-assigned default port number, but this can be overridden. The last field is called a *path*. This indicates a directory-like hierarchical namespace local to the server. For a Web-server serving static pages, a directory on disk is usually designated as the 'root' of that portion of the file system that is exported to the Web. The path is relative to this root position. E.g., if the designated root on a server is `/home/httpd/html`, then the path `/java/manual/index.html` refers to the absolute path `/home/httpd/html/java/manual/index.html` on the server's file system. The 'root' exported to the Web is never the root of the file system for security reasons.



Here we show a brief overview of how Web pages composed using HTML (HyperText Markup Language) are delivered over the network using the HTTP (HyperText Transport Protocol). Note the distinction between the *formatting language* and the *transport protocol*. We will go into details later.

HTML looks almost like plain ASCII text, except you can create *links* to other HTML pages on remote servers using the `HREF`*tag*. Suppose the HTML file `dbco.html` is accessible at the HTTP server `www.cse.iitb.ernet.in`. A *browser* client will ask the server for this resource using a URL (explained earlier). The server will send the HTML file shown top left. The browser will format the tagged text so that the `HREF`s become clickable hyperlinks, with the text between `<a>` and `</a>` highlighted as the *anchor text*.

Clicking on the anchor text will lead the browser to interpret the URL contained in the tag and to generate a fresh request to the server `www.oracle.com`.

Thus, using the simple mechanisms of URLs and tagged text, a vast interconnected network of documents can be built in a distributed, decentralized manner.

HTML markup features

- Formatting markups
  - HTML, HEAD, BODY, TITLE, H, B, I, UL, OL, LI, BR, P, HR, PRE, FONT
- Hyperlinking markups
  - A, IMG
- Table markups
  - TABLE, TR, TH, TD

Apart from hyperlinks, HTML has many other tags for typesetting. All tags must be properly nested, e.g., `<html><body></body></html>`. The body text is typically divided into sections nested at various levels. Each section starts with a section header, enclosed in `<h1>Main</h1>` `<h2>Subsection</h2>`, etc. Boldface and italics can be formatted using `<b>bold</b>` and `<i>italics</i>`. More explicit font selection can be made via the `FONT` tag. Itemized lists can be typeset using the `UL` (unnumbered) and `OL` (numbered) tags in the form `<ul><li>first</li>` `<li>second</li></ul>`. Images can be embedded in the HTML text using the `<img src="river.gif">` construct. Tables can be typeset in the form `<table><tr><th>Item</th><th>Price</th></tr>` `<tr><td>Rice</td><td>Rs10</td></tr></table>`; TR means table row, TH means table header, and TD means table cell.

In the lab session we will get quite some experience in HTML authoring. It is good to know about the range of tags, even though you will most likely use a wysiwyg editor such as Netscape Composer or Microsoft Frontpage.

Active content features

- Embedding calls to server-side programs
  - FORM
  - CGI and servlets
- Embedding client-side programs
  - SCRIPT, APPLET, PARAM

HTML also has a number of tags for active interaction between the client browser and the server. This has three common forms, CGI, Javascript, and and Java applets.

Using the FORM and related tags, you can write an HTML page which shows various text fields to fill in, radio buttons and selections to choose from, and submit the form. The Common Gateway Interface is an architecture standard for submitting such forms to the server side where they are interpreted by a CGI script.

Javascript is a scripting language designed originally by Netscape. It can be embedded in HTML pages to perform actions like click redirection, image highlighting, checking arguments in a form, etc. We won't deal with Javascript in this module.

Java is a portable object oriented language whose binary code can be run by browsers. These *applets* can be sent from the server to the browser client where they are executed. These applets can contact a Java *servlet* at the server end. Servlets follow the same architecture convention as CGI scripts, but are supported by a well-designed language and runtime library.

We will study forms, CGI scripts, applets and servlets in detail in the lab sessions.

Popular browsers

- Microsoft Internet Explorer
- Netscape Navigator and Communicator
  - Started from NCSA Mosaic
- HotJava
  - JavaSoft's all-Java implementation
- Lynx (for text-only terminals)
- Opera
  - The only browser you have to pay for!

Mosaic was the first graphical browser interface. It evolved into Netscape, the most popular browser in use on both Windows and Unix platforms. Microsoft played a good catch-up game, creating another popular browser called Internet Explorer. Java support for both of these is unreliable. HotJava from JavaSoft (Sun) is a browser implemented completely in Java, hence it automatically has excellent Java support. For text-only situations (owing to low bandwidth, user disability) Lynx is a very popular non-graphic browser. A new browser which is supposedly much more reliable than Netscape and IE and not free (!) is Opera.

It is quite important to remember that visitors at your site will be using a variety of browsers (mostly from the above list). You will benefit from double checking that your site has a good presentation, look and feel across the board for all these browsers. Do not over-optimize the size for a particular browser, and make sure text-only users can also get significant utility from your site.

Browser features

- Multimedia: images, video, audio, data
- Secure communication e.g. for banking
- Forms to invoke server-side scripts
- Java, Javascript and ActiveX
  - Embed client-side programs in Web page
- 'Cookies' to remember client identity
- Style sheets to separate form and content
- Frames (avoid!)

Apart from text and images, browsers can also render video and audio with the help of numerous *plug-in*s that are freely available for download. (If you use such a feature on your site, make sure you provide a link to a site where the necessary plug-in can be found.) Browsers also support secure versions of the HTTP protocol where the client-server communication is *encrypted* so that an eavesdropper cannot tap the message or interfere with the protocol without being detected. You will learn more about network security in another lecture of this module.

As mentioned before popular browsers support HTML extensions which enable forms and client-side active content.

Browsers also support *cookies* (a feature discussed later in this lecture) so that servers can enter into extended sessions with clients, e.g. for a online store browsing and shopping application.

Recent versions of HTML separate form and content: the document specifies *logical* tags such as SECTION and a *style sheet* specifies how to map logical tags to a specification for how to render the content. This is similar to the use of templates in Microsoft Word documents, or *style* files in the LaTeX typesetting system.

Some features like frames are a result of hasty development and should be avoided because they upset the association between a URL and its content.

Hypertext transfer protocol (HTTP)

- RFC1945, RFC2068
- Stateless client-server dialogue
- One request per connection
- HTTP requests
- Request headers
- Status codes
- Responses

```
$ telnet www.cse.iitb.ernet.in 80
Trying 144.16.111.14...
Connected to surya.cse.iitb.ernet.in.
Escape character is '^]'.
GET / HTTP/1.0

HTTP/1.1 200 OK
Date: Fri, 11 Jun 1999 06:26:16 GMT
Server: Apache/1.3.0 (Unix) PHP/3.0.4
Last-Modified: Tue, 04 May 1999 11:47:42 GMT
ETag: "2f70a-1645-372ede5e"
Accept-Ranges: bytes
Content-Length: 5701
Connection: close
Content-Type: text/html
X-Pad: avoid browser bug

<HTML>
<HEAD><TITLE>IIT Bombay
CSE Department Home Page</TITLE></HEAD>
<BODY> … </BODY> </HTML>
Connection closed by foreign host.
```


HTTP is a stateless network protocol (at the application layer) specified in technical notes called “Request For Comments” or RFCs, available at `http://www.w3.org`. Early versions of HTTP allowed only one request-response per socket connection; later versions allow many transfers to be made over one socket session for efficiency, but the semantics of interaction are still stateless. In the example we show how to access a Web server via the `telnet` program. It also shows the essence of how a browser interacts with the server. (In practice the browser sends many other pieces of information which we will study soon.)

The reply from the server typically has two parts: the *header* and the *body*. The reply header has various important meta-data fields used in various ways by the client. Some important fields are the ones that specify the *type* of the reply object (text, image, audio) called a MIME-type, the length of the body of the reply in bytes, and the date it was last modified. Most of these fields are optional and clients must be designed to tolerate missing fields in robust applications.

HTTP requests

- GET fetches the specified resource
- HEAD requests meta-information about specified resource
- PUT places a client document on the server
- POST sends user-specified form data to a server script and returns results
- DELETE erases a document from the server

HTTP permits several commands besides the GET command illustrated in the previous slide.

In the example we showed an example of the GET operation, for which the reply consists of meta-data and content. If only the meta-data is desired, a HEAD command can be used. We shall see that HTML can be used to render *forms* at the client browser which can be filled in and submitted by the browser to the server. The mechanism for doing this is the POST command.

Less frequently used commands are PUT, which transfers a document from the client browser for storage to the server side; and DELETE, which destroys a document stored at the server. For security reasons, most servers have these features unimplemented or disabled.

HTTP request headers

- User-agent: Mozilla (Netscape)
- Accept, Accept-encoding, Accept-language
- Authorization: user name and password
- Pragma: no-cache
- If-modified-since
- Content-type
  - E.g. application/x-www-form-urlencoded
- Content-length: number of bytes in POST

In the simple GET example, all that the client needed to specify was the URL to fetch. In general, the client includes more information in the request after the command line (GET, HEAD etc.). The entire request is terminated by two carriage-return-line-feed sequences ("`\r\n\r\n`"). For GET and HEAD, the body of the request is empty, whereas for PUT and POST it is not.

The client identifies its software lineage, e.g. what brand of a browser it is. It also has to specify the preferred response document type, character sets, and languages. This lets a suitably configured server send different responses for the same URL request, based on client configuration.

Some resources may be protected by passwords; in such cases, the client has to send suitable authentication in the request header.

Sometimes content may be fetched via a *proxy* which is responsible for caching so as to improve bandwidth utilization. Suitable HTTP request headers exist to control caching behavior; caching can be disabled, a bound can be established on the staleness of a page in cache, and the server can be requested for a page on condition that it has been modified since a given time. The last option is very useful for crawlers.

POST requests also need to notify the server that the MIME type is special and also specify the length of the POST body.

HTTP status codes

- Success codes (200-299)
  - 200=OK, 204=No Response
- Redirection (300-399)
  - 301=Moved, 302=Found, 304=Not modified
- Client errors (400-499)
  - 400=Bad Request, 401=Unauthorized, 403=Forbidden, 404=Not Found
- Server errors (500-599)
  - 500=Internal Error, 503=Server Overloaded

As shown in the example, the first line of response from the server is always a status code line. Status codes are 3-digit numbers divided into various ranges with corresponding significance. Some of the most common response codes and their verbals meanings are listed above.

A *redirection* is a pointer from an old URL to a new URL that is automatically followed by most browsers. A redirection is used mostly in the interim period of a major site reorganization.

HTTP response headers

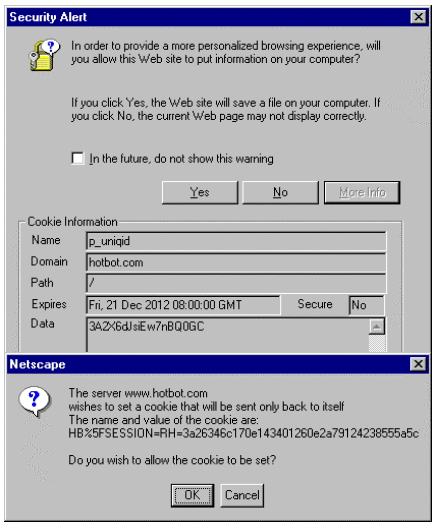
Header name	Meaning
Date	The current date
Last-Modified	The last time the requested resource was modified
Expires	The time at which the requested resource expires
URI/Location	The new location of a redirected resource (301/302)
MIME-Version	Multipurpose Internet Mail Extensions version number
Content-Type	The MIME type of the returned resource, used by browser to render resource
Content-Length	Resource size in bytes; if unspecified, client reads until server disconnects

Like the request, the response header also has numerous fields; we saw some of them in the telnet-based example. The most important and frequent header fields are listed above with their meanings alongside.

'Expires' is used for pages with a known lifetime, e.g., newspaper sites.

Cookies

- Keep track of a client across stateless request-reply connections
- Server gives client browser a text string to store locally
- Client attaches cookie string to each subsequent request
- Privacy?



So far we have seen that HTTP is a connectionless protocol: the client asks for data which is named using a URL, the server dishes it out and terminates the session. (HTTP1.1 and beyond permit persistent TCP connections to piggyback multiple transfers, but there is no support for an extended logical session within HTTP.)

Suppose you are hosting an online bookstore where you want to keep track of the books browsed or bought thus far by a customer. You cannot do this using HTTP alone. The way out is to use *cookies*. A cookie is a short segment of text that the Web server sends to the browser in the response header. The browser saves the cookie in a database stored on the user's file system. When the browser makes another connection to the server, it sends back the cookie in the request header. The server keeps a mapping of cookies to customer records in a database and can use the cookie to identify the customer. Typically the cookie is an arbitrary string generated randomly or by hashing a combination of request headers.

Bear in mind that identification is not necessarily a desirable thing on the Web. Browsers have options to control acceptance, storage and expiry of cookies that you should carefully configure. We will go over this in the lab session.

Popular HTTP servers

- Apache
  - Descended from NCSA httpd
  - 53% usage in January 1999
  - Open source from www.apache.org
  - Support for scripts, Java
- Microsoft Internet Information Server IIS
- Netscape Server

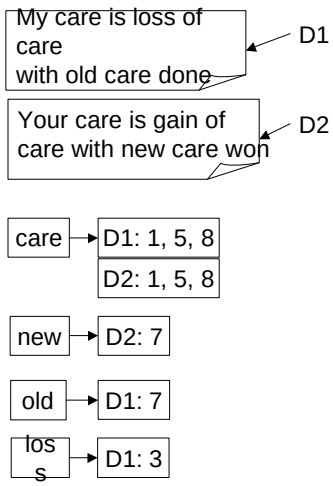
The most popular Web server is the free, open-source Apache server. Descended from the original NCSA httpd server, it is expertly built with robust performance even on desktops. There are both UNIX and Windows ports, although the UNIX port is far more reliable.

Apache does much more than serving static HTML pages. It has a modular construction with a core that can dynamically load shared object *modules*; these modules can support server-side scripts, servlets, etc. Following a simple API, you can easily develop and integrate additional modules into Apache. Apache can also handle authentication, proxy caching, and virtual hosting (one machine pretending to be many Web sites).

There are several other commercial choices but in price and performance it is hard to match Apache.

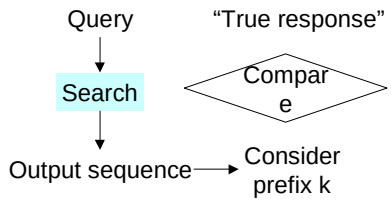
Keyword indexing

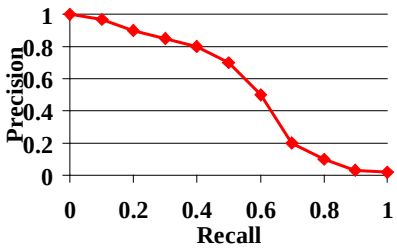
- Boolean search
  - care AND NOT old
- Stemming
  - gain*
- Phrases and proximity
  - “new care”
  - loss NEAR/5 care
  - <SENTENCE>



Relevance ranking

- Recall = coverage
  - What fraction of relevant documents were reported
- Precision = accuracy
  - What fraction of reported documents were relevant
- Trade-off





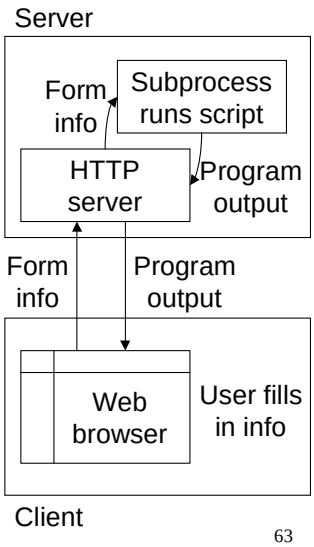
Web server = database

- All but the smallest sites use a database
- Reliability of content storage, versions, backups, and updates
- Providing dynamic views of content, e.g., products by price, vendor, store location
- Perform transactions on-line, e.g. air ticket purchase, fund transfers
- Present electronic ‘store’ customized by each visitor’s behavior

We know how static HTML pages (with audio, images, etc.) are transmitted using the HTTP protocol. Static content is not adequate for a variety of applications. Two kinds of support are therefore needed: a *database* to store data in a form more structured than files and directories, and a mechanism for clients to *execute* programs on the server side. A variety of systems can be used for data storage, ranging from files to full-fledged relational databases. As for executing programs, we will discuss the Common Gateway Interface (CGI) protocol and Java servlets.

Forms and CGI

- User fills out HTML form
- Browser passes info to Common Gateway Interface script
- Script processes info and sends answer to STDOUT
- Server relays output back to browser
- Fast CGI



HTML supports tags to render forms with text fields, choice menus, buttons, etc. When these forms are filled in and submitted to the server, the server typically uses a child process to execute a script. A CGI script may be written in a shell command language, a scripting language such as Perl, or even a C or C++ program. The script reads the form information either from an environment variable set by the HTTP server or from its standard input, and sends output to standard out. This output is relayed by the server to the client browser.

Isolating the script process is nice because a bug in the script cannot crash the Web server. However spawning a new process for each client request is too slow; hence some servers pre-spawn a pool of children processes which are dynamically assigned to execute scripts. More controlled and safer languages like Java enable the server to execute scripts within its process.

Two methods for calling scripts

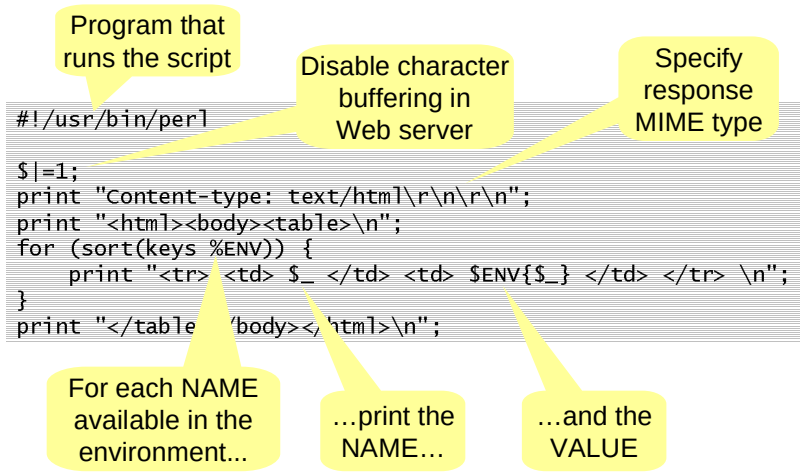
- GET
  - URL of CGI file is extended with arguments
  - http://acme.com/search.cgi?author=gates
- POST
  - Server relays browser form information to the standard input (STDIN=‘keyboard’) of process running script
  - Script process reads a series of NAME=VALUE strings on STDIN

A CGI script can be invoked either by the GET or the POST method. GET is used for short arguments. In GET, the arguments are tagged on to the end of the URL of the CGI script file itself as shown. The server knows how to strip off this suffix and set suitable environment variables before the script it executed.

There are limits to the length of a URL in most implementations. A POST gets around this problem by sending the name=value string via the script process’s standard input.

A caching proxy may cache the contents of a GET URL but not a POST URL since the form argument is not visible in the URL.

Example of GET: trial.cgi



Here is an example of a GET script written in the Perl scripting language. Most of the code and annotations are self-explanatory. Only one aspect needs explanation. `%ENV` is an *associative map* from names to values that is available to all Perl scripts. Both names and values are strings.

The HTTP server fills in various (name, value) pairs in `%ENV` before calling the script. We will see some of these when we run the above script in the lab. The most important value is of a name `QUERY_STRING`. This is accessed as `$ENV{'QUERY_STRING'}`. We shall inspect its value in the lab as well.

The Java language

- Object oriented, strongly typed
  - Similar to C++, stricter rules
- Interpreted by a virtual machine, portable
  - Only virtual machine has to be ported
  - Can also produce optimized native code
- Memory management by virtual machine
  - No malloc/free, no new/delete, no pointers
- Dynamic loading of classes
- Fine-grained security policies

Implementing server-side programs in unsafe languages may crash the Web server. Hence Java libraries have been written to ease the task of writing safer server-side programs.

For some of you who may not be familiar with the Java language, it is syntactically similar to C++. It has no pointers and no user-accessible mechanism for freeing memory. Memory is managed by the runtime system and reclaimed as it becomes inaccessible from the program. Java is usually compiled from source to *byte-code*. Byte-code can be interpreted on Java *virtual machines* (JVMs).

Java can be used to develop both client-side programs, called *applets*, and server-side programs, called *servlets*. Applets are programs that are downloaded from a server and execute in a JVM integrated with the browser. Servlets are server-side programs which are compatible with the CGI protocol specification.

Applets

- A particular Java class and derived subclasses
- No main(), restricted API and behavior
- Compiled binary (‘bytecode’) transmitted from server to browser client
- Browser client includes a Java virtual machine (JVM)
- JVM loads and interprets bytecode

We won’t cover Java in detail in this module, and yet most of our code examples and labs will involve some lightweight Java. So here we give a very brief account of the relevant features.

Applet is a particular class of Java which is inherited by user-designed applets. The Java source is compiled into bytecode and stored as a ‘.class’ file or several .class files collected into a ‘.jar’ file. When a suitable hyperlink is clicked at the client, this .class or .jar files is loaded by a JVM integrated into the browser and executed at the client computer.

Since programs on the Web may not be trustworthy, the execution of the applet code should not assume the effective user ID of the account running the browser process. Therefore the capabilities of applets are restricted in many ways. Ordinary operations for local processes, such as writing and reading files, opening a socket, etc. become *privileges* for applets, which must be explicitly asked for through popup messages.

Applet API

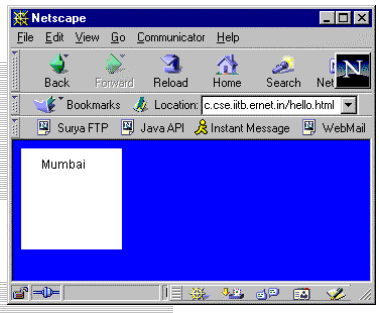
- init
  - Initializations, class construction from params
- getParameter
  - Read parameters from HTML PARAM tags
- start, paint
  - Start the applet, refresh the applet panel
- stop, destroy
  - Suspension and termination

Unlike applications, applets do not have a `main` function. Instead the there are four events in the life of an applet. First the browser JVM loads and initializes the applet. Then the applet is started. If the user removes focus from the browser window, or visits another page, the applet is stopped. If the user backs up from the applet page and visits another page, or the user closes the browser, the applet is stopped and destroyed, releasing its resources from the browser's JVM. Since `init` does not have arguments, applets read their arguments via the `getParameter` method (see next slide).

Applet example

hello.html

```
<html><body bgcolor="blue">
<applet code=hello.class
width=100 height=100>
<param name="text" value="Mumbai">
</applet></body></html>
```



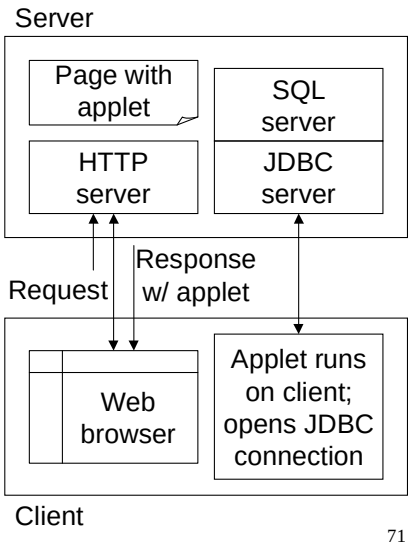
hello.java

```
import java.applet.*;
import java.awt.*;
public class hello extends Applet {
  public void paint(Graphics g) {
    g.drawString(getParameter("text"), 20, 20);
  }
}
```


The developer has to write two files to create a simple Java applet: a '.java' file with the program source code, and a '.html' file which embeds the '.class' file generated by compiling the '.java' file. The example is self-explanatory.

JDBC

- Java client library to connect to remote SQL servers
- De facto Web database connectivity standard, similar to ODBC
- Connection, Statement, ResultSet
- Form-based query, update, transactions



Applets can also connect directly to relational databases via the Java Database Connectivity (JDBC) protocol. The `java.sql` package provides various classes and interfaces to communicate directly with relational databases. We won't cover JDBC programming in detail in this module, but the block diagram gives a broad overview. RDBMS vendors provide a JDBC server and a JDBC client library. The server has to be run at the Web server computer. The client is packaged with the applet or preinstalled at the client computer. Precompiled statements and cursors are supported.

Servlets

- Server side code executed by submitting forms from client browsers
- Written in Java using the servlet class libraries
- Adheres to CGI specifications
- Provides support for sessions and cookies

Finally we will explore how to write servlets. Servlets derive from standard classes in a Java package. Servlets adhere to CGI specifications and have support for reading arguments, processing cookies, and session management.

Hello world servlet example

```
import java.servlet.*;
import java.io.*;
public class HelloWorldServlet extends GenericServlet {
  static String helloString = "Hello, world\r\n";
  public void service(ServletRequest request,
                      ServletResponse response)
  {
    response.setContentLength(helloString.length());
    response.setContentType("text/plain");
    try {
      PrintStream rs = new
                PrintStream(response.getOutputStream());
      rs.print(helloString);
    }
    catch (IOException e) { }
  }
}
```


This is an example of a simple servlet that generates a plain text page when invoked. It shows the basic mechanisms for creating a response and writing to it from the server end. HTML responses can be generated similarly. More advanced examples dealing with FORM parameters, sessions and cookies will be discussed in the lab sessions.