HTTP, HTTPS and TCP Networking in Java

Some reminders

Credits

Some material derived from:

- HTTP vs. HTTPS by Eng. T. Aldaldooh

RFC

Request for Comments (RFC) are a type of publication from the Internet Engineering Task Force (IETF) and the Internet Society (ISOC), the principal technical development and standards-setting bodies for the Internet.

- An RFC is authored by engineers and computer scientists in the form of a memorandum describing methods, behaviors, research, or innovations applicable to the working of the Internet and Internet-connected systems. I

Protocol

High-Change in Bond Street, — ou — la Politesse du Grande Monde.

- Synonymous of **Etiquette**

a code of behavior that delineates expectations for social behavior according to contemporary conventional norms within a society, social class, or group.

Communications protocol, a set of rules and regulations that determine how data is transmitted in telecommunications and computer networking

Port

HTTP on port 80
• HTTP with SSL (HTTPS) on port 443
• FTP on port 21
• SMTP on port 25
• POP on port 110
• SSH on port 22

A port is an **endpoint of communication in an operating system**.

A **process** associates its input or output channels, via an Internet socket, with a transport protocol, a port number, and an IP address.

This process is known as binding,

PID	PORT	IP	Protocol
84	21	193.205.196.130	FTP
78	80	193.205.196.130	HTTP
321	8080	193.205.196.130	HTTP
541	25	193.205.196.130	SMTP

Mistranslated into Italian as “Porta” (door)

URL and URI

- URLs used early on by all Internet protocols, including various document retrieval protocols.
- More specifications (both from 1994):
  - URL : Uniform Resource Locators - RFC 1738.
  - URI : Universal Resource Identifiers - RFC 1630.

URL is just one type of a URI.

URLS and URIS

- URL (Uniform Resource Locators)
  - Provides single short string to identify network-accessible resource
  - <scheme>://<host>[:<port>]/<path>[?<query>]
  - http://www.w3.org/Icons/w3c_home.gif
- URI (Uniform Resource Identifier)
  - Identifies a resource either by location or name.
  - The selection of the representation can be determined by the web server through HTTP content negotiation.
  - A superset of URLs
  - http://www.w3.org/Icons/w3c_home.
  - http request line contains a non-URL URI

URL, URN, URC

- **URL**: identify resources by specifying their locations in the context of a particular access protocol, such as HTTP or FTP.
- **URN**: persistent, location-independent identifiers
- **URC**: standardized representation of document properties, such as owner, encoding, access restrictions or cost.

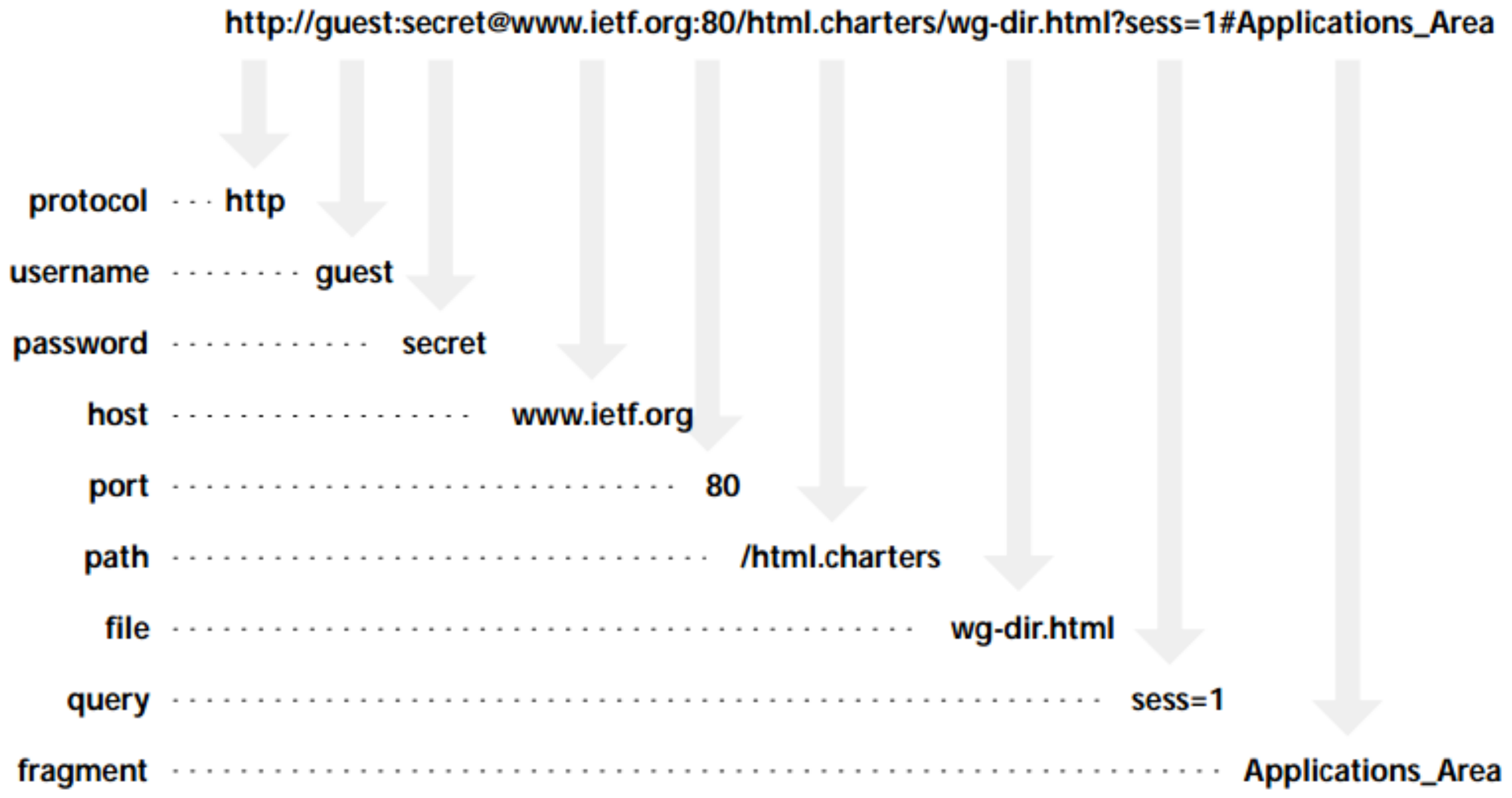
URN

- URN are not locators, are not required to be associated with a particular protocol or access method, and need not be resolvable.

- They should be assigned by a procedure which provides some assurance that they will **remain unique** and **identify the same resource** persistently over a prolonged period.

- A typical URN namespace is urn:isbn, for International Standard Book Numbers.

URLs cont.



http://guest:secret@www.ietf.org:80/html.charters/wg-dir.html?sess=1#Applications_Area
protocol http
username guest
password secret
host www.ietf.org
port 80
path /html.charters
file wg-dir.html
query sess=1
fragment Applications_Area

URLs cont.

- **Protocol**: Identifies the application protocol needed to access the resource, in this case HTTP.
- **Username** : If the protocol supports the concept of user names, this provides a user name that has access to the resource; in the example "guest."
- **Password**: The password associated with the user name, "secret" in the example.
- **Host** : The communication system that has the resource; for HTTP this is the Web server, www.ietf.org in the example.
- **Port** : The TCP port that the application protocols should use to access the resource; many protocols have an implied TCP port (for HTTP that port is 80
- **Path** : The path through a hierarchical organization under which the resource is located, often a file system's directory structure or equivalent.
- **File**: The resource itself.
- **Query**: Additional information about the resource or the client.
- **Fragment**: A particular location within a resource.

URL and MIME type

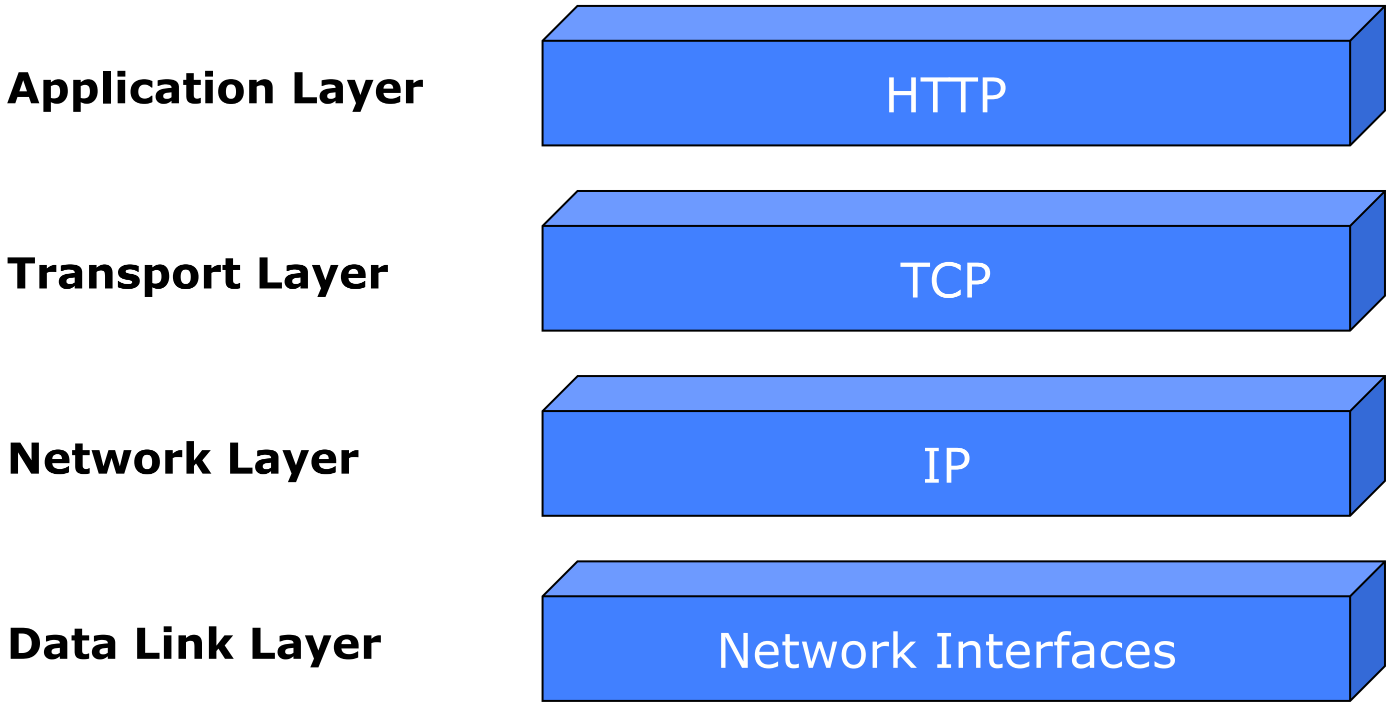
- **URLs point to resources (“content”).**
- Resources are represented using different **Internet Media Types** (MIME Types)
  - Multipurpose Internet Mail Extensions RFC 2045,6
- MIME Type tells how content should be handled
  - File extensions are mapped to certain MIME Types
    - .html usually means a MIME Type of text/html
    - .jpg usually means a MIME Type of image/jpeg
- The most common MIME Types used on the Web come from the text, image and application top-level groups
    - text/html, text/css
    - image/gif, image/jpeg, image/png
    - application/pdf, application/octet-stream
    - application/x-javascript, application/x-shockwave-flash

An Introduction to HTTP

- Hyper Text Transfer Protocol
- One of the application layer protocols that make up the Internet
    - HTTP over TCP/IP
    - Like SMTP, POP, IMAP, NNTP, FTP, etc.
- The underlying language of the Web
- Three versions have been used, two are in common use and have been specified:
    - RFC 1945 HTTP 1.0 (1996)
    - RFC 2616 HTTP 1.1 (1999)

HTTP and TCP/IP

HTTP sits atop the TCP/IP Protocol Stack



HTTP

- The Hypertext Transfer Protocol
- distributed, collaborative, hypermedia information systems.
- HTTP functions as a request-response protocol in the client-server computing model.
- Actors:
  - Internet Engineering Task Force (IETF)
  - World Wide Web Consortium (W3C)

RFC

RFC 2616 (June 1999) defined HTTP/1.1

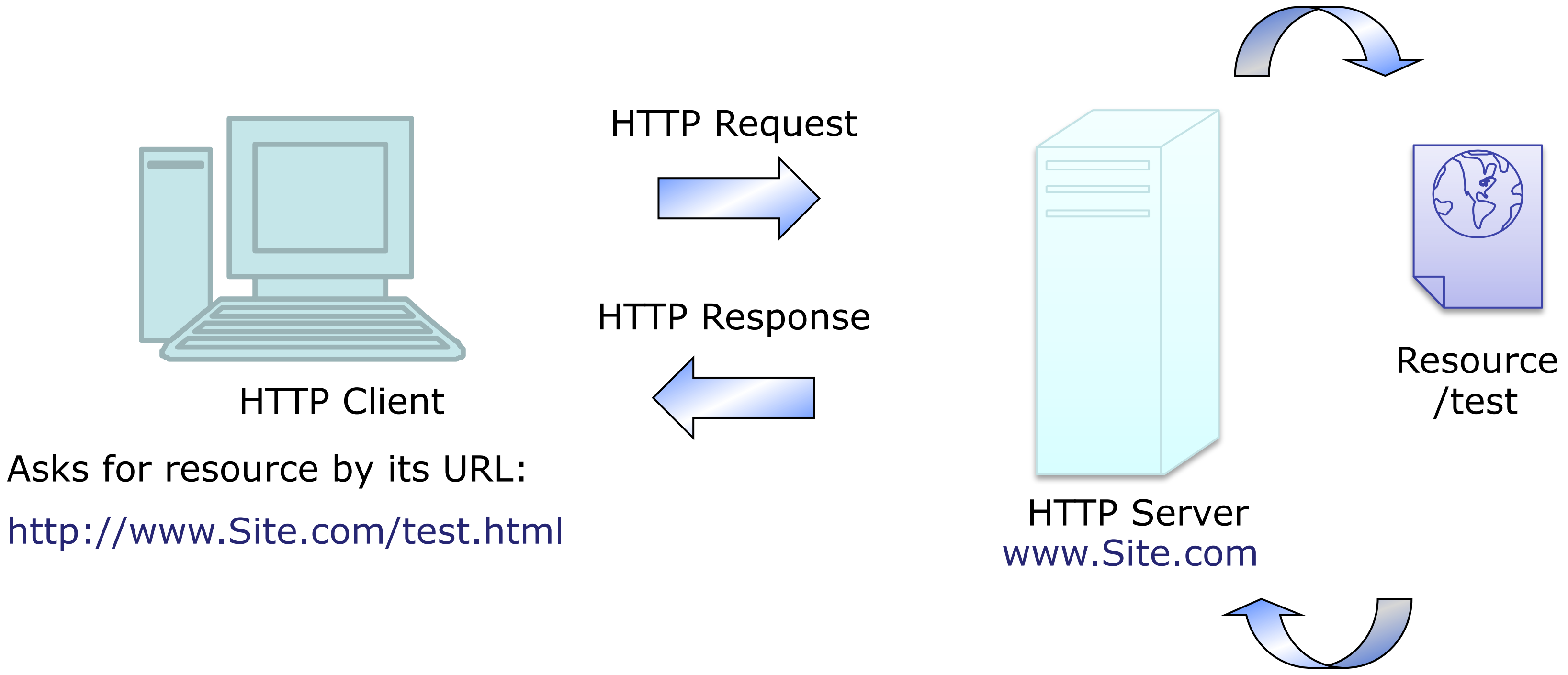
In June 2014, RFC 2616 was retired and HTTP/1.1 was redefined by

- RFC 7230 - HTTP/1.1: Message Syntax and Routing
- RFC 7231 - HTTP/1.1: Semantics and Content
- RFC 7232 - HTTP/1.1: Conditional Requests
- RFC 7233 - HTTP/1.1: Range Requests
- RFC 7234 - HTTP/1.1: Caching
- RFC 7235 - HTTP/1.1: Authentication

HTTP 2.0 Current Status

- May 2015 RFC 7540
- May 2015 RFC 7541 (HPACK)

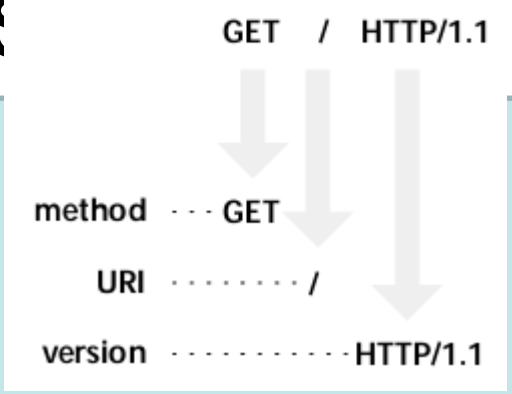
HTTP servers turn URLs into resources through a request-response cycle



HTTP requests and responses Messages

- HTTP requests and responses are both types of Internet Messages (RFC 822), and share a general format:
  - **A Start Line, followed by a CRLF**
    - Request Line for requests
    - Status Line for responses
  - **Zero or more Message Headers**
    - field-name ":" [field-value] CRLF
  - **An empty line**
    - Two CRLFs mark the end of the Headers
  - **An optional Message Body if there is a payload**
    - All or part of the "Entity Body" or "Entity"

HTTP Requests



```
GET / HTTP/1.1[CRLF]
Host: www.iugaza.edu.ps[CRLF]
Connection: close[CRLF]
User-Agent: Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1)[CRLF]
Accept-Encoding: gzip[CRLF]
Accept-Charset: ISO-8859-1,UTF-8;q=0.7,*;q=0.7[CRLF]
Cache-Control: no-cache[CRLF]
Accept-Language: de,en;q=0.7,en-us;q=0.3[CRLF]
Referer: http://web-sniffer.net/[CRLF] [CRLF]
```

A Closer Look at the Request Methods

- **GET**
  - By far most common method
  - Retrieves a resource from the server
  - Supports passing of query string arguments
- **HEAD**
  - Retrieves only the Headers associated with a resource but not the entity itself
  - Highly useful for protocol analysis, diagnostics
- **POST**
  - Allows passing of data in entity rather than URL
  - Can transmit of far larger arguments that GET
  - Arguments not displayed on the URL

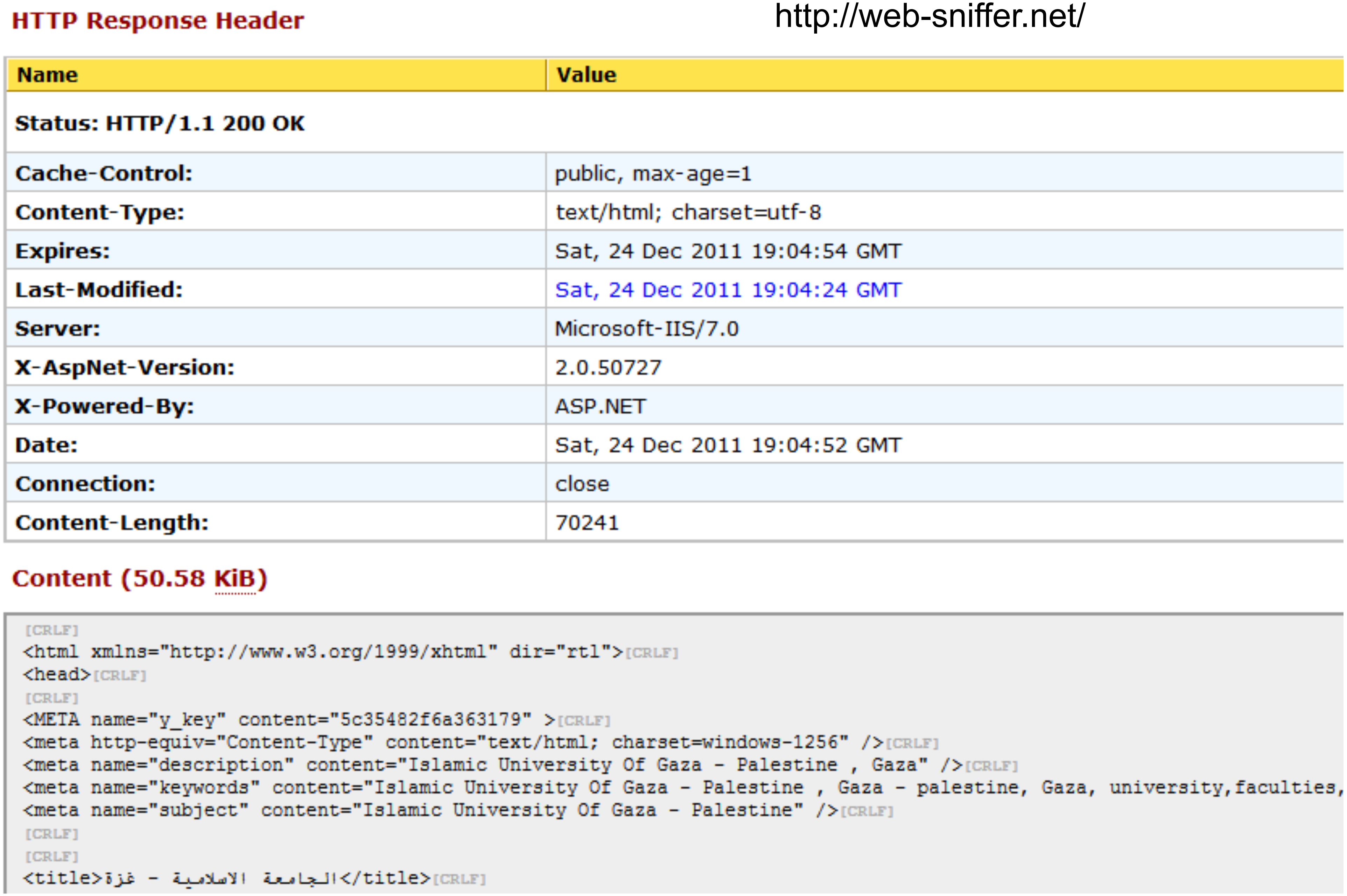
More Request Methods, cont.

- **OPTIONS**
  - Shows methods available for use on the resource (if given a path) or the host (if given a “*”)
- **TRACE**
  - Diagnostic method for assessing the impact of proxies along the request-response chain
- **PUT, DELETE**
  - Used in HTTP publishing (e.g., WebDav)
- **CONNECT**
  - A common extension method for Tunneling other protocols through HTTP

Web-based Distributed Authoring and Versioning (WebDAV) is a set of methods based on the Hypertext Transfer Protocol (HTTP) that facilitates collaboration between users in editing and managing documents and files stored on World Wide Web servers.

HTTP Responses

http://web-sniffer.net/



HTTP Response Header

Name	Value
Status: HTTP/1.1 200 OK	
Cache-Control:	public, max-age=1
Content-Type:	text/html; charset=utf-8
Expires:	Sat, 24 Dec 2011 19:04:54 GMT
Last-Modified:	Sat, 24 Dec 2011 19:04:24 GMT
Server:	Microsoft-IIS/7.0
X-AspNet-Version:	2.0.50727
X-Powered-By:	ASP.NET
Date:	Sat, 24 Dec 2011 19:04:52 GMT
Connection:	close
Content-Length:	70241

Content (50.58 KiB)

```
[CRLF]
<html xmlns="http://www.w3.org/1999/xhtml" dir="rtl">[CRLF]
<head>[CRLF]
[CRLF]
<META name="y_key" content="5c35482f6a363179" >[CRLF]
<meta http-equiv="Content-Type" content="text/html; charset=windows-1256" />[CRLF]
<meta name="description" content="Islamic University Of Gaza - Palestine , Gaza" />[CRLF]
<meta name="keywords" content="Islamic University Of Gaza - Palestine , Gaza - palestine, Gaza, university,faculties,
<meta name="subject" content="Islamic University Of Gaza - Palestine" />[CRLF]
[CRLF]
[CRLF]
<title>الجامعة الاسلامية - غزة</title>[CRLF]
```

A Closer Look at the Status Line

- **Consists of three major parts:**
- The HTTP Version
  - Just like third part of Request Line
- Status Code
  - **5 groups of 3 digit integers indicating the result of the attempt to satisfy the request:**
  - 1xx are informational
  - 2xx are success codes
  - 3xx are for alternate resource locations (redirects)
  - 4xx indicate client side errors
  - 5xx indicate server side errors
- The Reason Phrase followed by the CRLF
  - Short textual description of the status code

A Closer Look at the Status Line

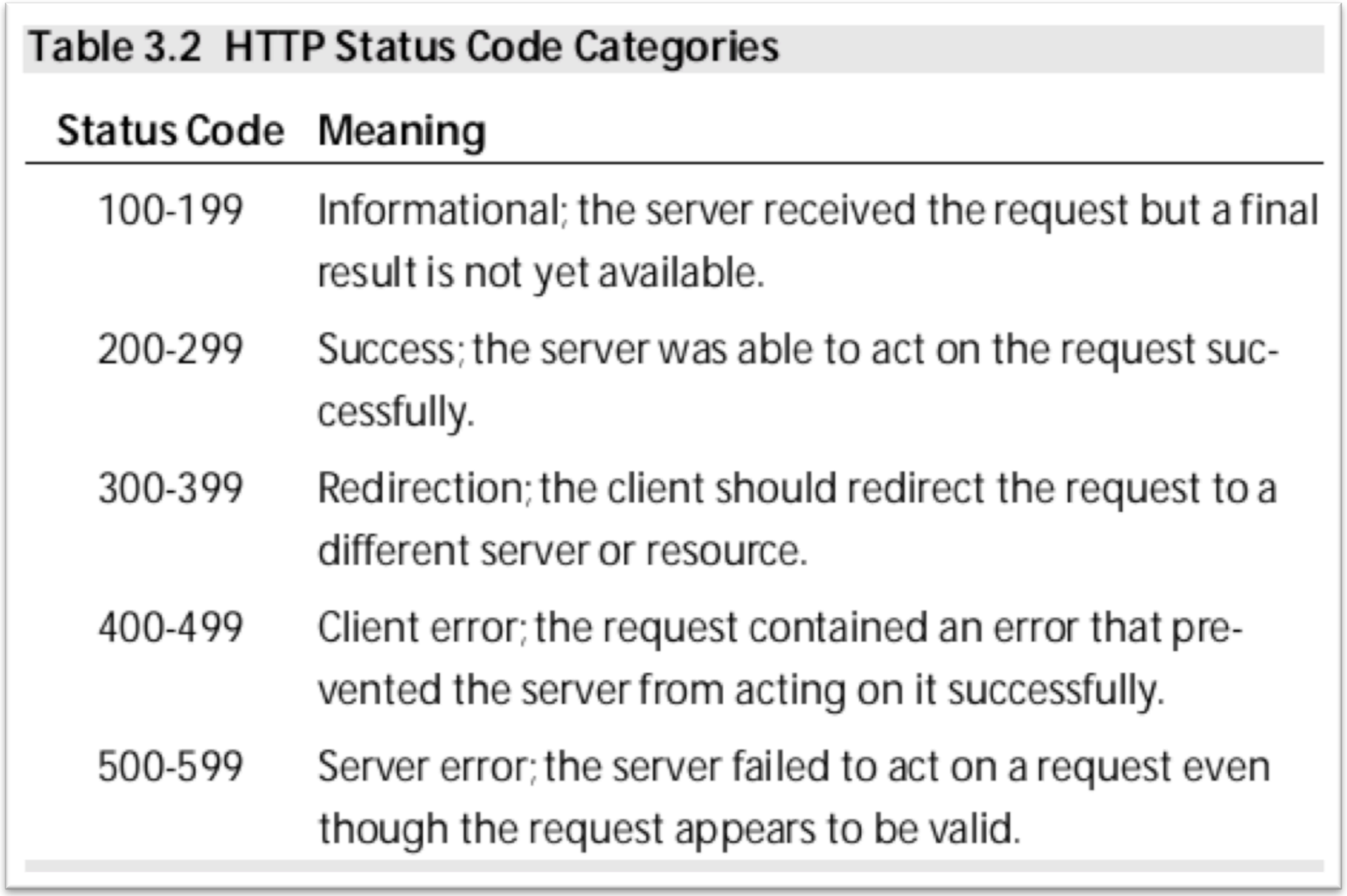
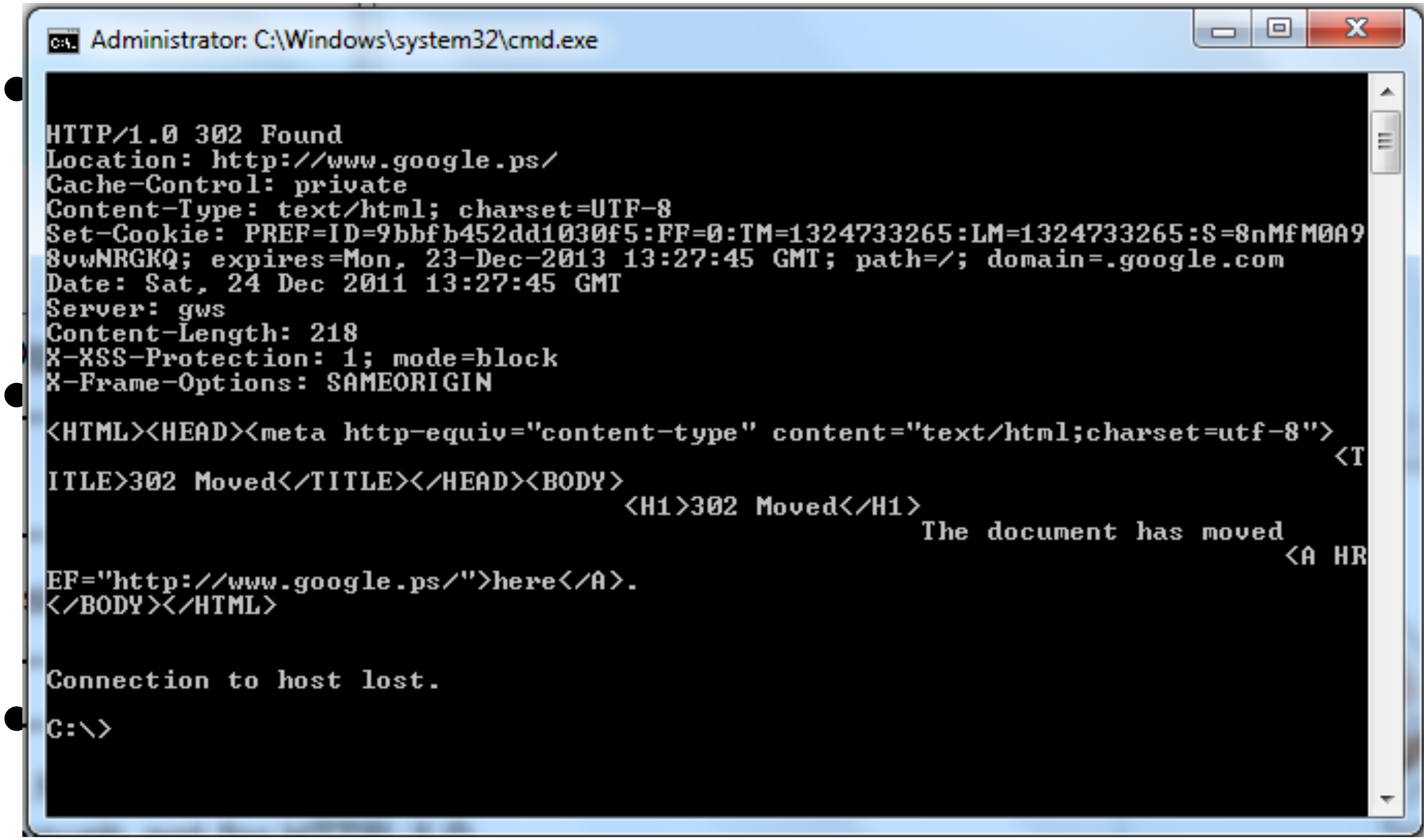


Table 3.2 HTTP Status Code Categories

Status Code	Meaning
100-199	Informational; the server received the request but a final result is not yet available.
200-299	Success; the server was able to act on the request successfully.
300-399	Redirection; the client should redirect the request to a different server or resource.
400-499	Client error; the request contained an error that prevented the server from acting on it successfully.
500-599	Server error; the server failed to act on a request even though the request appears to be valid.

Making a simple HTTP request using Telnet



Administrator: C:\Windows\system32\cmd.exe

```
HTTP/1.0 302 Found
Location: http://www.google.ps/
Cache-Control: private
Content-Type: text/html; charset=UTF-8
Set-Cookie: PREF=ID=9bbfb452dd1030f5:FF=0:TM=1324733265:LM=1324733265:S=8nMfM0A9
8vwNRGKQ; expires=Mon, 23-Dec-2013 13:27:45 GMT; path=/; domain=.google.com
Date: Sat, 24 Dec 2011 13:27:45 GMT
Server: gws
Content-Length: 218
X-XSS-Protection: 1; mode=block
X-Frame-Options: SAMEORIGIN

<HTML><HEAD><meta http-equiv="content-type" content="text/html;charset=utf-8">
                                                                              <T
ITLE>302 Moved</TITLE></HEAD><BODY>
                                   <H1>302 Moved</H1>
                                                     The document has moved
                                                                           <A HR
EF="http://www.google.ps/">here</A>.
</BODY></HTML>


Connection to host lost.

C:\>
```

A Closer Look at HTTP Headers

Headers come in four major types, some for requests, some for responses, some for both:

- **General Headers**
  - Provide info about messages of both kinds
- **Request Headers**
  - Provide request-specific info
- **Response Headers**
  - Provide response-specific info
- **Entity Headers**
  - Provide info about request and response entities
- Extension headers are also possible

General Headers

- **Connection** – lets clients and servers manage connection state
  - Connection: Keep-Alive
  - Connection: close
- **Date** – when the message was created
  - Date: Sat, 31-May-03 15:00:00 GMT
- **Via** – shows proxies that handled message
  - Via: 1.1 www.myproxy.com (Squid/1.4)
- **Cache-Control** – Among the most complex of headers, enables caching directives
  - Cache-Control: no-cache

Request Headers

- **Host** – The hostname (and optionally port) of server to which request is being sent
- **Referer** – The URL of the resource from which the current request URI came
  - Referer: http://www.host.com/login.asp
- **User-Agent** – Name of the requesting application, used in browser sensing
  - User-Agent: Mozilla/4.0 (Compatible; MSIE 6.0)
- **Accept** and its variants – Inform servers of client's capabilities and preferences
  - Enables content negotiation
  - Accept: image/gif, image/jpeg;q=0.5
  - Accept- variants for Language, Encoding, Charset
- **Cookie** How clients pass cookies back to the servers that set them
  - Cookie: id=23432;level=3

Response Headers

- **Server** – The server’s name and version
  - Server: Microsoft-IIS/5.0
  - Can be problematic for security reasons

- **Set-Cookie** – This is how a server sets a cookie on a client
  - Set-Cookie: id=234; path=/shop; expires=Sat, 31-May-03 15:00:00 GMT; secure

Entity Headers

- **Allow** – Lists the request methods that can be used on the entity
  - Allow: GET, HEAD, POST
- **Location** – Gives the alternate or new location of the entity
  - Used with 3xx response codes (redirects)
  - Location: http://www.iugaza.edu.ps/ar/
- **Content-Encoding** – specifies encoding performed on the body of the response
  - Used with HTTP compression
  - Corresponds to Accept-Encoding request header
  - Content-Encoding: gzip
- **Content-Length** – The size of the entity body in bytes
- **Content-Location** – The actual if different than its request URL
- **Content-Type** – specifies Media (MIME) type of the entity body

HTTP Overview

HTTP Requests

An HTTP request consists of
a request method, (“subprotocol” specification)
a request URL, (location)
header fields, (metadata)
a body. (data)

HTTP 1.1 defines the following request methods:

- GET: Retrieves the resource identified by the request URL
- HEAD: Returns the headers identified by the request URL
- POST: Sends data of unlimited length to the Web server
- PUT: Stores a resource under the request URL
- DELETE: Removes the resource identified by the request URL
- OPTIONS: Returns the HTTP methods the server supports
- TRACE: Returns the header fields sent with the TRACE request
- CONNECT request connection to a transparent TCP/IP tunnel,
- PATCH apply partial modifications to a resource.

HTTP 1.0 includes only the GET, HEAD, and POST methods.

Clients and Servers

- The *client* is the actor that requests to talk.
- The *server* is the actor that accepts to talk.

The client can create a socket to start a conversation to a server app anytime.
The server must be repared in aadvance to accept an incoming conversation.

Sockets

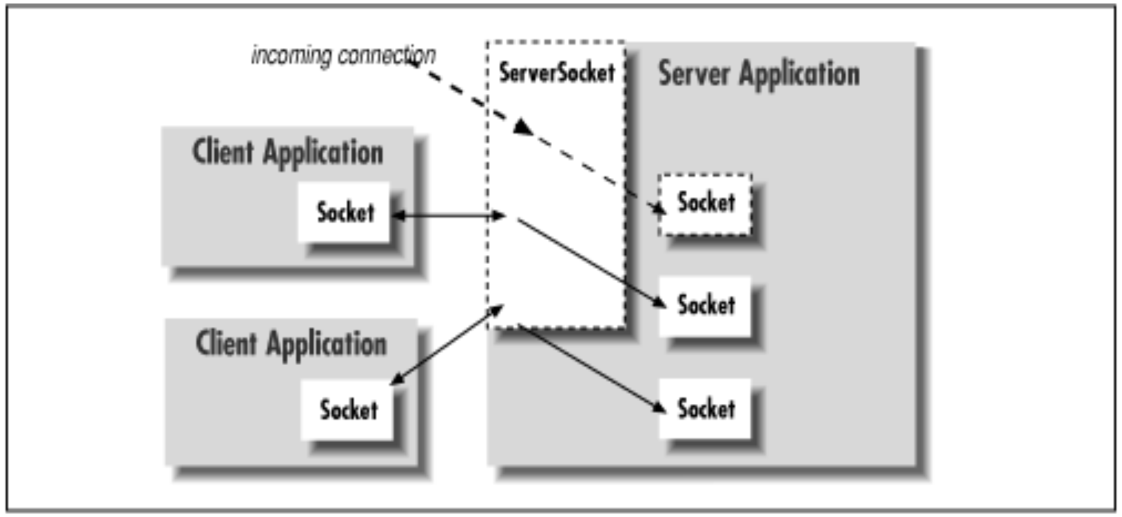
The java.net.Socket class represents a side of connection (regardless if client o or server).

The server uses the java.net.ServerSocket class to wait for incoming conversations. It creates a ServerSocket object and waits, blocked on a accept() call until a connection comes. Then it creates a Socket object to be used to communicate with the client.

Sockets

A server can maintain many conversations simoultaneously.

There is only one ServerSocket, but one Socket for every client.



Server port

The client needs two pieces of info to establish a connection: a hostname (to get the server’s address) and a port number (to identify a process on the server machine).

A server app listens on a predefined port while waiting for a connection.

Port numbers are coded in the RFC (Es. Telnet 23, FTP 21, ecc.), but they can be freely chosen for custom services.

Client port

The client’s port number is generally assigned by the OS, and in general you do not care about it.

When the server responds it opens a new socket whose number is assigned by the OS. It then continues listening on the original port, and serves the particular cliens on the new socket.

Sockets

The first choice is which protocol to use:

connection-oriented (TCP)

or

connectionless (UDP).

The Java Socket class uses TCP

java.net.Socket

This class implements a socket for interprocess communication over the network.

The constructor methods create the socket and connect it to the specified host on the specified port.

java.net.Socket - main methods

The constructor methods create the socket and connect it to specified host and port.

Once the socket is created, getInputStream() e getOutputStream() return InputStream e OutputStream objects (usable as I/O channels).

getInetAddress() e getPort() return address and port to which the socket is connected.

getLocalPort() returns the local port used by the socket .

close() closes la socket.

java.net.ServerSocket

During creation you specify on which port to listen

The accept() starts listening and blocks until there is an incoming call.

At that point, accept() accepts the connection, creates and returns a Socket that the server can use to talk to the client.

java.net.ServerSocket – main methods

getInetAddress() returns the local address

getLocalPort() returns the local port .

close() closes the socket.

Sockets

Clients

```
try {
       Socket sock = new Socket("www.pippo.it", 80);
       //Socket sock = new Socket("128.252.120.1", 80);
} catch ( UnknownHostException e ) {
       System.out.println("Can't find host.");
} catch ( IOException e ) {
       System.out.println("Error connecting to host.");
}
```

Connection-oriented protocol

Server

- Create a ServerSocket object.
- After accepting the connection, create a Socket che object.
- Create InputStream and OutputStream to read/write bytes from/to the connection.
- Optionally create a new thread for every connection, so that the serer can listen for new requests while serving arrived clients.

Reading & Writing raw bytes – Client side

```
try {
        Socket server = new Socket("foo.bar.com", 1234);
        InputStream in = server.getInputStream();
        OutputStream out = server.getOutputStream();
        // Write a byte
        out.write(42);
        // Read a byte
        Byte back = in.read();
        server.close();
} catch (IOException e ) { }
```

Reading & Writing raw bytes – Server side

```
try {
        ServerSocket listener = new ServerSocket( 1234 );
        while ( !finished ) {
                Socket aClient = listener.accept();
                // wait for connection
                InputStream in = aClient.getInputStream();
                OutputStream out = aClient.getOutputStream();
                // Read a byte
                Byte importantByte = in.read();
                // Write a byte
                out.write(43);
                aClient.close();
        }
        listener.close();
} catch (IOException e ) { }
```

Reading & Writing newline delimited strings – Client

Incapsulating InputStream and OutputStream it is possible to access streams in an easier way.

```
try {
        Socket server = new Socket("foo.bar.com", 1234);
        InputStream in = server.getInputStream();
        DataInputStream din = new DataInputStream( in );

        OutputStream out = server.getOutputStream();
        PrintStream pout = new PrintStream( out );

        // Say "Hello" (send newline delimited string)
        pout.println("Hello!");
        // Read a newline delimited string
        String response = din.readLine();
        server.close();
} catch (IOException e ) { }
```

Reading & Writing newline delimited strings – Server

```
try {
        ServerSocket listener = new ServerSocket( 1234 );
        while ( !finished ) {
                Socket aClient = listener.accept();
                // wait for connection
                InputStream in = aClient.getInputStream();
                DataInputStream din = new DataInputStream( in );
                OutputStream out = aClient.getOutputStream();
                PrintStream pout = new PrintStream( out );
                // Read a string
                String request = din.readLine();
                // Say "Goodbye"
                pout.println("Goodbye!");
                aClient.close();
        }
        listener.close();
} catch (IOException e ) { }
```

A concurrent HTTP mini-server - Introduction

TinyHttpd listens on a specified port and services simple HTTP "get file" requests. They look something like this:

```
GET /path/filename [optional stuff]
```

Your Web browser sends one or more as lines for each document it retrieves. Upon reading the request, the server tries to open the specified file and send its contents. If that document contains references to images or other items to be displayed inline, the browser continues with additional GET requests. For best performance (especially in a time-slicing environment), TinyHttpd services each request in its own thread. Therefore, TinyHttpd can service several requests concurrently.

A concurrent HTTP mini-server

```
package tinyhttpd;

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


public class TinyHttpd {
   public static void main( String argv[] )
    throws IOException {
       int port = 8000;
        if (argv.length>0) port=Integer.parseInt(argv[0]);
       ServerSocket ss = new ServerSocket(port);
        System.out.println("Server is ready");
       while ( true )
             new TinyHttpdConnection(ss.accept() );
   }
}
```

A concurrent HTTP mini-server

```
class TinyHttpdConnection extends Thread {

    Socket sock;

    TinyHttpdConnection(Socket s) {
        sock = s;
        setPriority(NORM_PRIORITY - 1);
        start();
    }

    public void run() {
        System.out.println("=========");
        OutputStream out = null;
        try {
            out = sock.getOutputStream();
            BufferedReader d =
              new BufferedReader(new InputStreamReader(
                    sock.getInputStream()));
            String req = d.readLine();;
            System.out.println("Request: " + req);
            StringTokenizer st = new StringTokenizer(req);
```

A concurrent HTTP mini-server - Note

. By lowering its priority to NORM_PRIORITY-1 (just below the default priority), we ensure that the threads servicing established connections won't block TinyHttpd's main thread from accepting new requests.

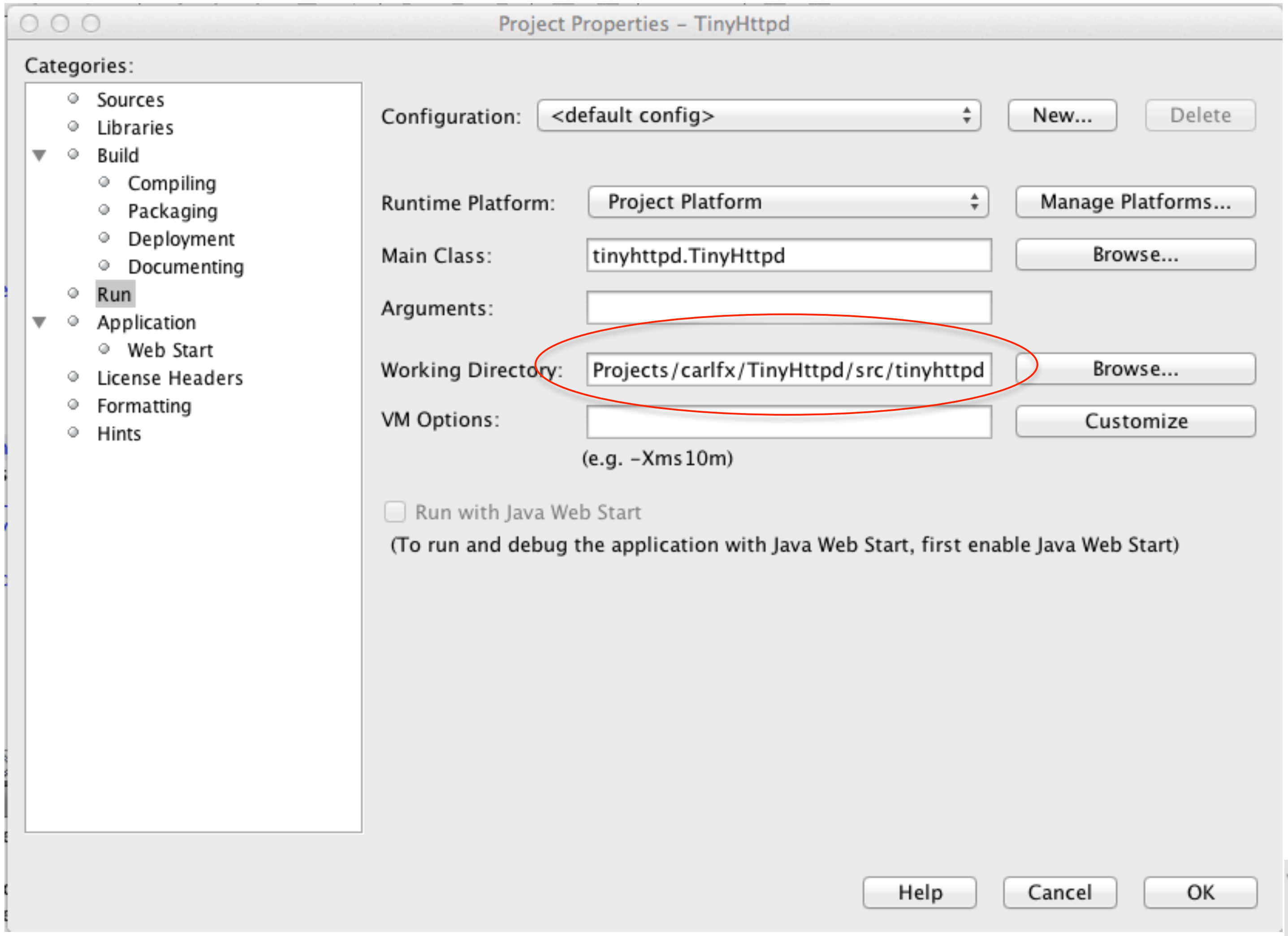
(On a time-slicing system, this is less important.)

Un mini-server concorrente HTTP

```
if ((st.countTokens() >= 2) && st.nextToken().equals("GET")) {
    if ((req = st.nextToken()).startsWith("/")) {
       req = req.substring(1);
    }
    if (req.endsWith("/") || req.equals("")) {
       req = req + "index.html";
    }
    try {
         FileInputStream fis = new FileInputStream(req);
         byte[] data = new byte[fis.available()];
         fis.read(data);
         out.write(data);
    } catch (FileNotFoundException e) {
         new PrintStream(out).println("404 Not Found");
         System.out.println("404 Not Found: " + req);
    }
} else {
    new PrintStream(out).println("400 Bad Request");
    System.out.println("400 Bad Request: " + req);
    sock.close();
}
```

Un mini-server concorrente HTTP

```
} catch (IOException e) {
          System.out.println("Generic I/O error " + e);
      } finally {
          try {
              out.close();
          } catch (IOException ex) {
              System.out.println("I/O error on close" + ex);
          }
      }
  }
}
```



Project Properties – TinyHttpd

Categories:

- Sources
- Libraries
- Build
  - Compiling
  - Packaging
  - Deployment
  - Documenting
- Run
- Application
  - Web Start
- License Headers
- Formatting
- Hints

Configuration: <default config>

Runtime Platform: Project Platform

Main Class: tinyhttpd.TinyHttpd

Arguments:

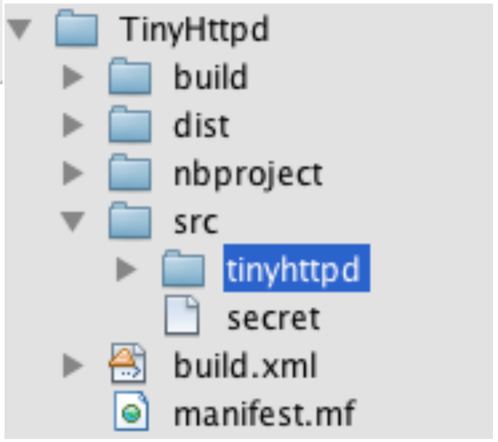
Working Directory: Projects/carlfx/TinyHttpd/src/tinyhttpd

VM Options:

(e.g. -Xms10m)

Run with Java Web Start

(To run and debug the application with Java Web Start, first enable Java Web Start)



```
TinyHttpd
  build
  dist
  nbproject
  src
    tinyhttpd
    secret
  build.xml
  manifest.mf
```

A concurrent HTTP mini-server - usage

Compile TinyHttpd and place it in your class path. Go to a directory with some interesting documents and start the daemon, specifying an unused port number as an argument. For example:

```
% java TinyHttpd 1234
```

You should now be able to use your Web browser to retrieve files from your host. You'll have to specify the nonstandard port number in the URL. For example, if your hostname is foo.bar.com, and you started the server as above, you could reference a file as in:

```
http://foo.bar.com:1234/welcome.html
```

A concurrent HTTP mini-server - Problems

TinyHttpd still has room for improvement. First, it consumes a lot of memory by allocating a huge array to read the entire contents of the file all at once. A more realistic implementation would use a buffer and send large amounts of data in several passes.

TinyHttpd also fails to deal with simple things like directories. It wouldn't be hard to add a few lines of code to read a directory and generate linked HTML listings like most Web servers do.

A concurrent HTTP mini-server - Problems

TinyHttpd suffers from the limitations imposed by the fickleness of filesystem access.

It's important to remember that file pathnames are still architecture dependent--as is the concept of a filesystem to begin with. TinyHttpd should work, as is, on UNIX and DOS-like systems, but may require some customizations to account for differences on other platforms. It's possible to write more elaborate code that uses the environmental information provided by Java to tailor itself to the local system.

A concurrent HTTP mini-server - Problems

The biggest problem with TinyHttpd is that there are no restrictions on the files it can access. With a little trickery, the daemon will happily send any file in your filesystem to the client.

It would be nice if we could restrict TinyHttpd to files that are in the current directory, or a subdirectory.

Assignment

Modify the simple web server so that all the urls that start with the token "process "
(e.g. http://localhost:8000/process)
launch an external process.

For instance,
http://localhost:8000/process/reverse?par1=*string*&par2=*booleanvalue*
should activate an (external) process that takes the par1 string.
If par2 is true, it returns the reversed string (e.g. ROMA -> AMOR).
If par2 is false, it checks if the string is a palindrome, and returns the answer
(true or false). (e.g. ROOR -> true, ROAR –> false)

To see how to start an external process from Java, take a look at
http://www.rgagnon.com/javadetails/java-0014.html

Deadline Sept. 24, 2017, 23:59

SEE WEB SITE: latemar.science.unitn.it

HTTPS Overview

https is a URI scheme which is syntactically identical to the http: scheme normally used for accessing resources using HTTP. Using an https: URL indicates that HTTP is to be used, but with a different default port (443) and an additional encryption/authentication layer between HTTP and TCP.

This system was developed by Netscape Communications Corporation to provide authentication and encrypted communication and is widely used on the World Wide Web for security-sensitive communication, such as payment transactions.

S-HTTP Overview

Secure hypertext transfer protocol' (S-HTTP) is an alternative mechanism to the https URI scheme for encrypting web communications carried over HTTP. S-HTTP is defined in RFC 2660.

Web browsers typically use HTTP to communicate with web servers, sending and receiving information without encrypting it. For sensitive transactions, such as Internet e-commerce or online access to financial accounts, the browser and server must encrypt this information.

The https: URI scheme and S-HTTP were both defined in the mid 1990s to address this need. Netscape and Microsoft supported HTTPS rather than S-HTTP, leading to HTTPS becoming the de facto standard mechanism for securing web communications. S-HTTP is an alternative mechanism that is not widely used.

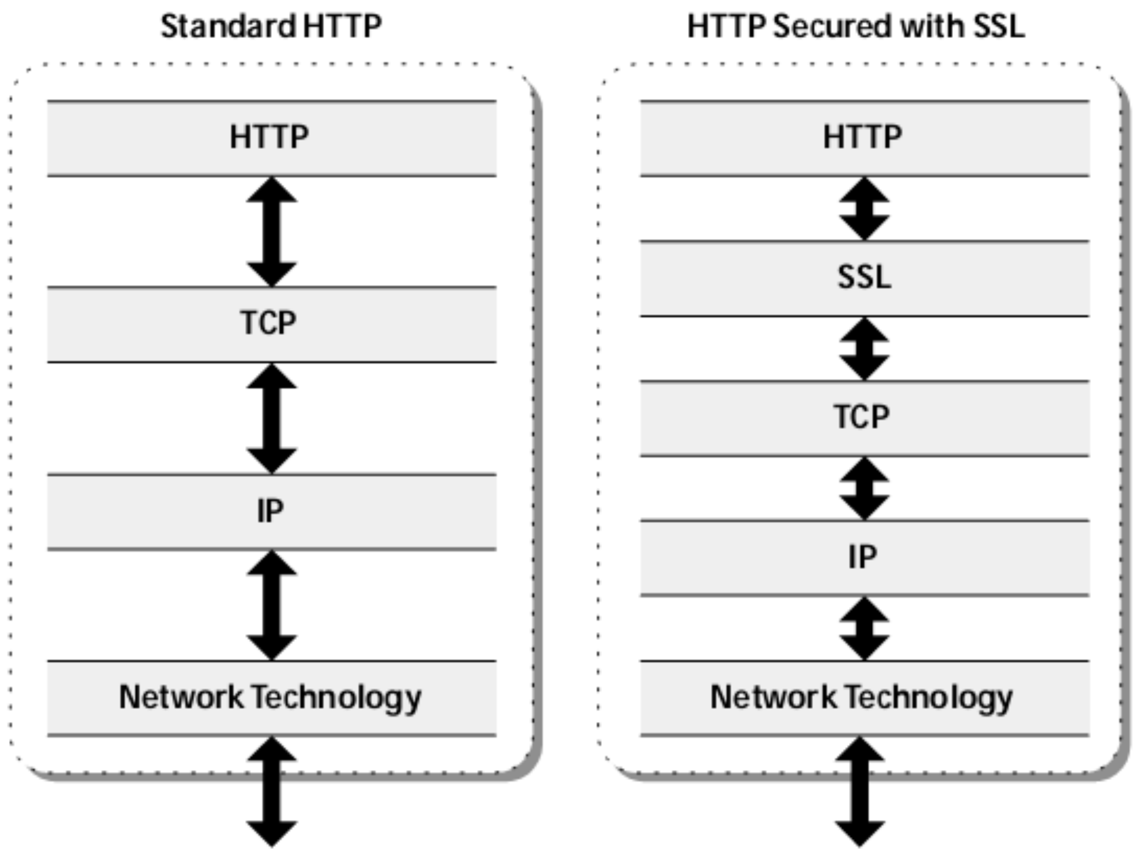
HTTPS

=

HTTP + SSL

HTTPS

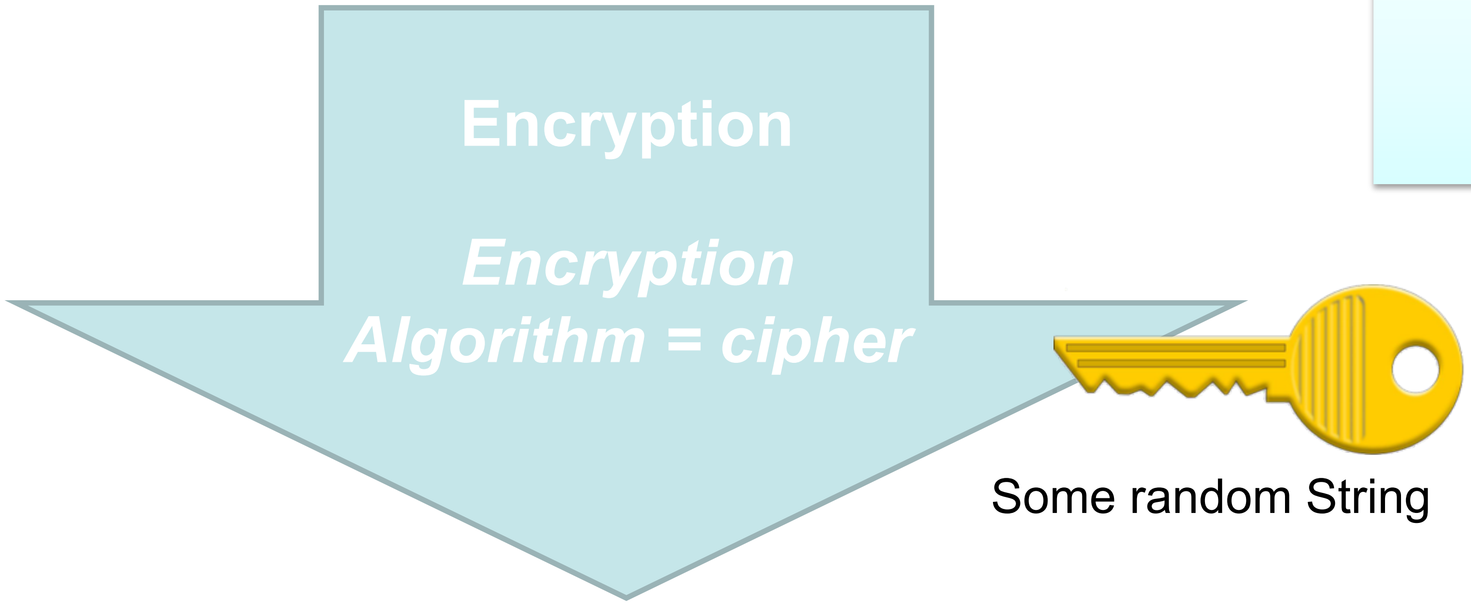
Figure 4.10 ▶
The SSL protocol inserts itself between an application like HTTP and the TCP transport layer. TCP sees SSL as just another application, and HTTP communicates with SSL much the same as it does with TCP.



Cryptography

Important information Data, Data, Data.

Plain Text

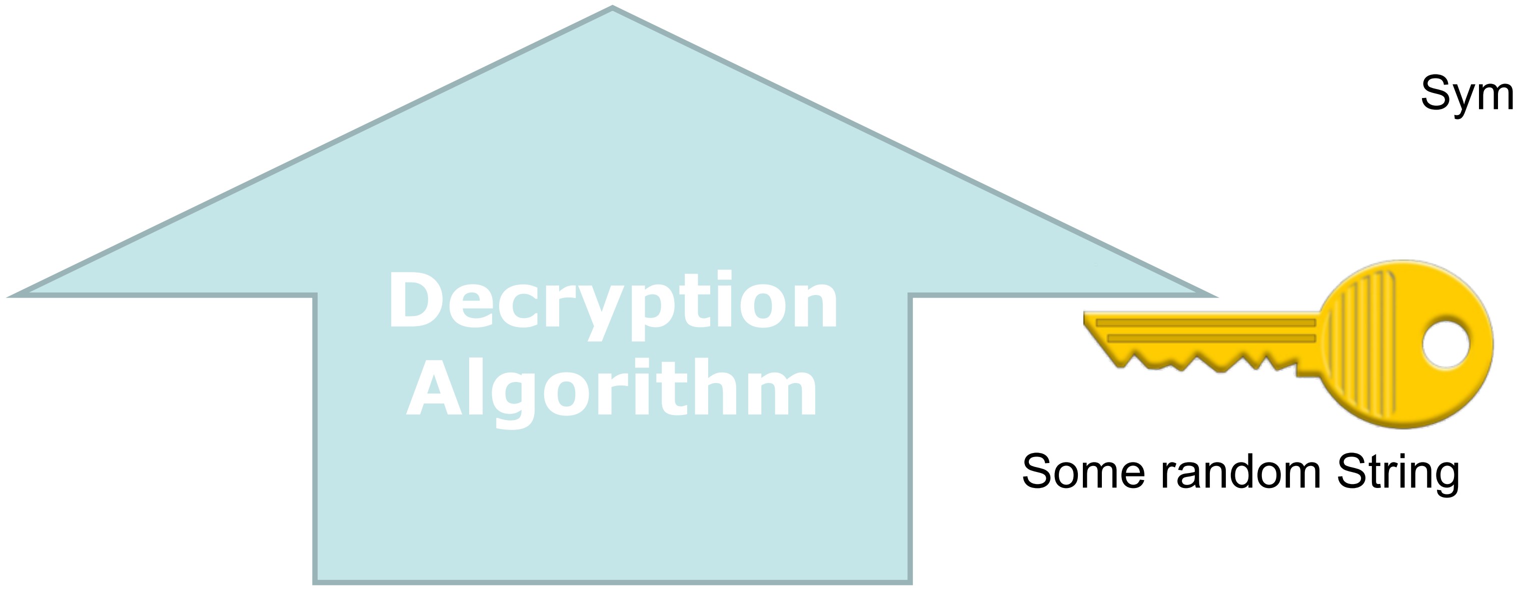


Hh2sh!~hH==E#@ns8676%===sdf

Cipher Text

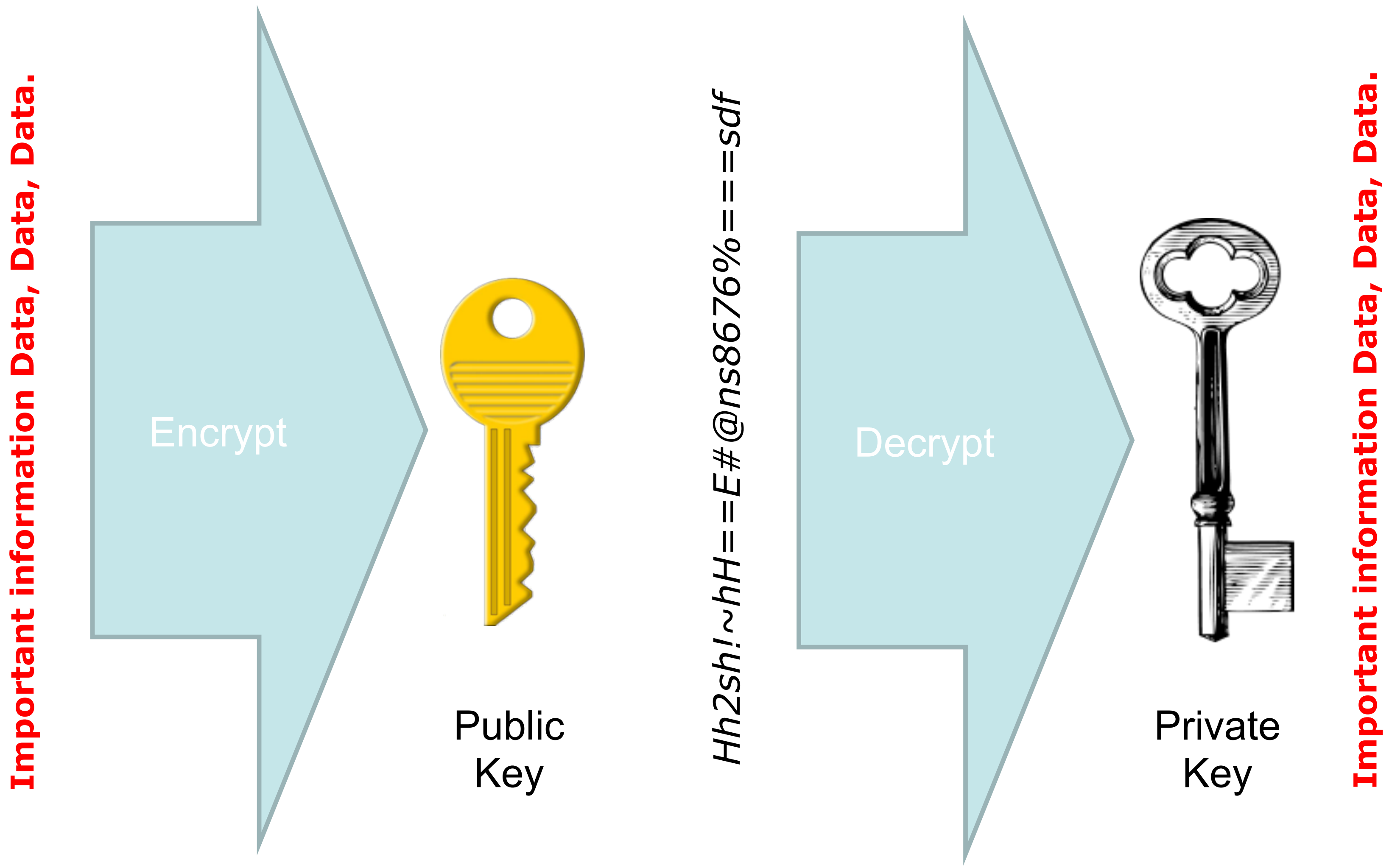
Cryptography cont.

Important information Data, Data, Data.



Hh2sh!~hH==E#@ns8676%===sdf

Asymmetric (public-key) encryption


Important information Data, Data, Data.
Encrypt
Public Key
Hh2sh!~hH==E#@ns8676%===sdf
Decrypt
Private Key
Important information Data, Data, Data.

SSL Session

- Uses asymmetric encryption to privately share the session key
  - Asymmetric has a lot of overhead
- Uses symmetric encryption to encrypt data
  - Symmetric encryption is quicker and uses less resource

SSL Handshake Process

Client requests HTTPS session →

← Certificate sent back (with public key)

Client creates session key (**53**)

Session key encrypted with public key(**X$qp0**)

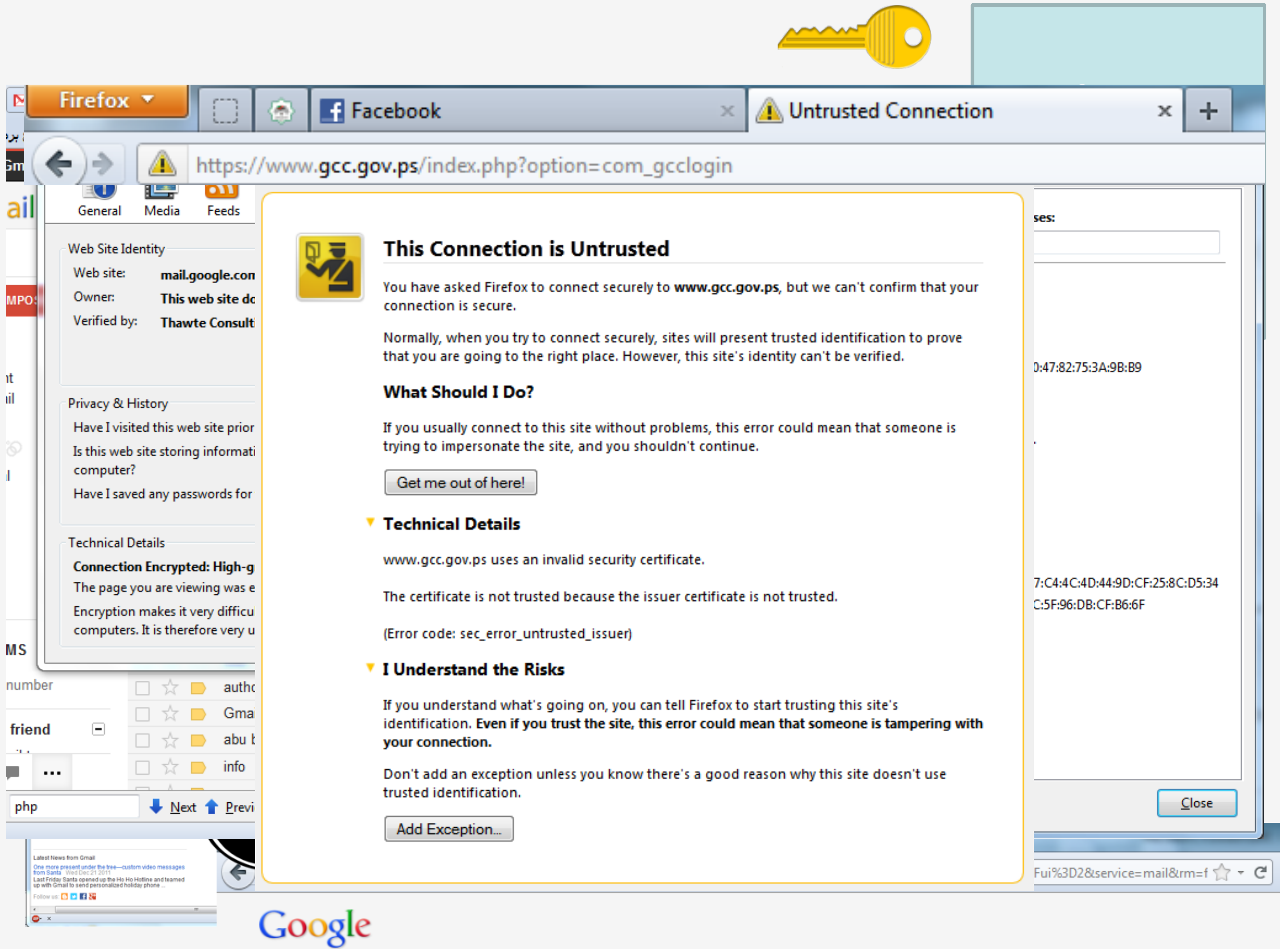
At this point only client knows session key

Encrypted session key sent to server →

session key decrypted with private key

At this point both client and server knows session key

Session encrypted with ↔ symmetric session key (53)



Web Site Identity
Web site: mail.google.com
Owner: This web site do
Verified by: Thawte Consulti

Privacy & History
Have I visited this web site prior
Is this web site storing informati computer?
Have I saved any passwords for

Technical Details
Connection Encrypted: High-g
The page you are viewing was e
Encryption makes it very difficul computers. It is therefore very u

This Connection is Untrusted

You have asked Firefox to connect securely to **www.gcc.gov.ps**, but we can't confirm that your connection is secure.

Normally, when you try to connect securely, sites will present trusted identification to prove that you are going to the right place. However, this site's identity can't be verified.

What Should I Do?

If you usually connect to this site without problems, this error could mean that someone is trying to impersonate the site, and you shouldn't continue.

Get me out of here!

Technical Details

www.gcc.gov.ps uses an invalid security certificate.

The certificate is not trusted because the issuer certificate is not trusted.

(Error code: sec_error_untrusted_issuer)

I Understand the Risks

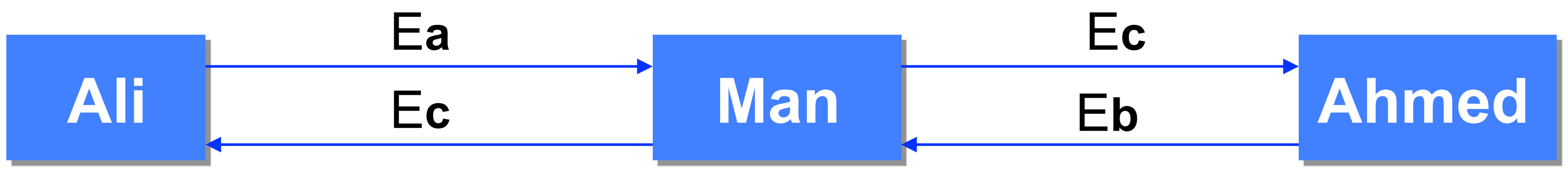
If you understand what's going on, you can tell Firefox to start trusting this site's identification. **Even if you trust the site, this error could mean that someone is tampering with your connection.**

Don't add an exception unless you know there's a good reason why this site doesn't use trusted identification.

Add Exception...

Man-in-the-Middle (MITM) Attack Concept

- There were away to get around the encryption instead o0f trying to break it



E{a,b,c} = Ali's, Ahmed's, and Man's public keys, respectively

- Ali wants to send secure messages to Ahmed.
- Man intercepts Ali's messages.
- Man talks to Ali and pretends to be Ahmed.
- Man talks to Ahmed and pretends to be Ali.

MITM Attack Concept

- Ali uses the *public key* she thinks she received from Ahmed (Man’s)
- Ahmed uses the key he thinks is Ali’s (also Man’s)
- As a result, Man not only gains *access* to secure information but also can *modify* it (e.g. *transfer money to a different account* etc.)

MITM and Certificates

- Digital Certificates designed to solve the problem but do they always help ?
- The MITM would have to create his own certificate with a private/public key.
- He still sit between client and server, acting as server to the client and client to the server, listening in on everything sent between the two.

The solution “chain of trust”

- To verify the authenticity and identity of the certificates themselves.
- linked back to a trustworthy source of certificates.
- Web browsers and operating systems will only trust certificates that directly or indirectly link back to one of a handful of CAs, the "root CAs.‟
- Any certificate that doesn't link back to a root CA such as a self-signed certificate will generate a big scary warning in the browser.
    - **How to create a self-signed SSL Certificate ...**
    - http://www.akadia.com/services/ssh_test_certificate.html

Conclusion

- HTTPS only slightly slower than HTTP.

  - Cost Of Security

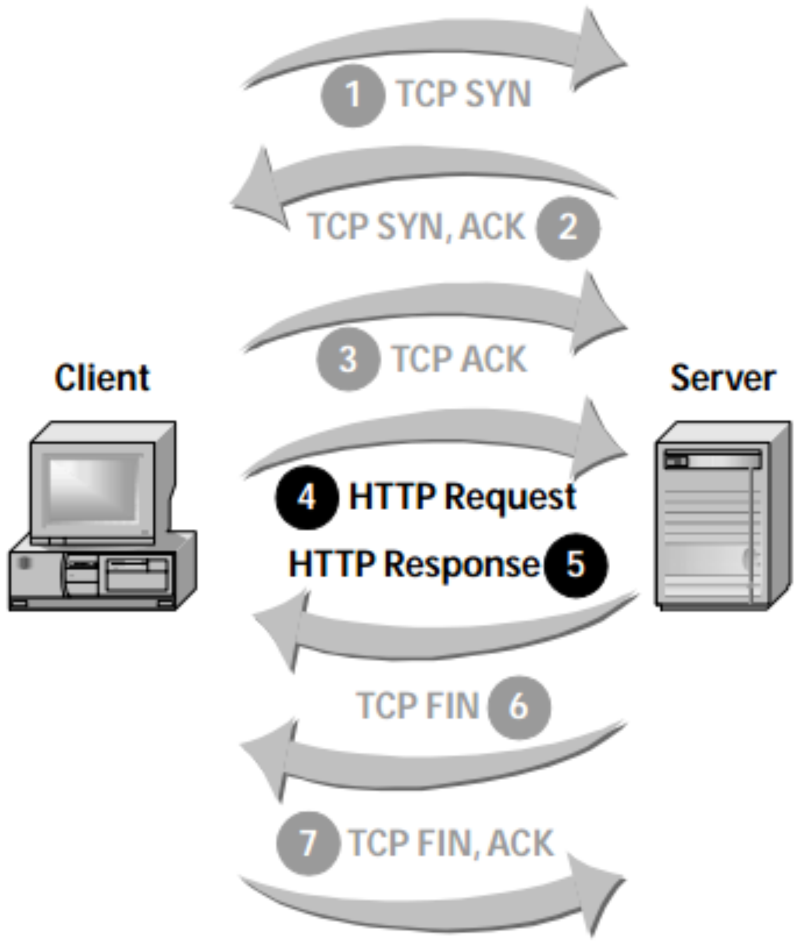
Reference

- HTTP Essentials Protocols for Secure, Scaleable Web Sites by Stephen Thomas .
- HTTP The Definitive Guide.
- View HTTP Request and Response Header < http://web-sniffer.net/ >

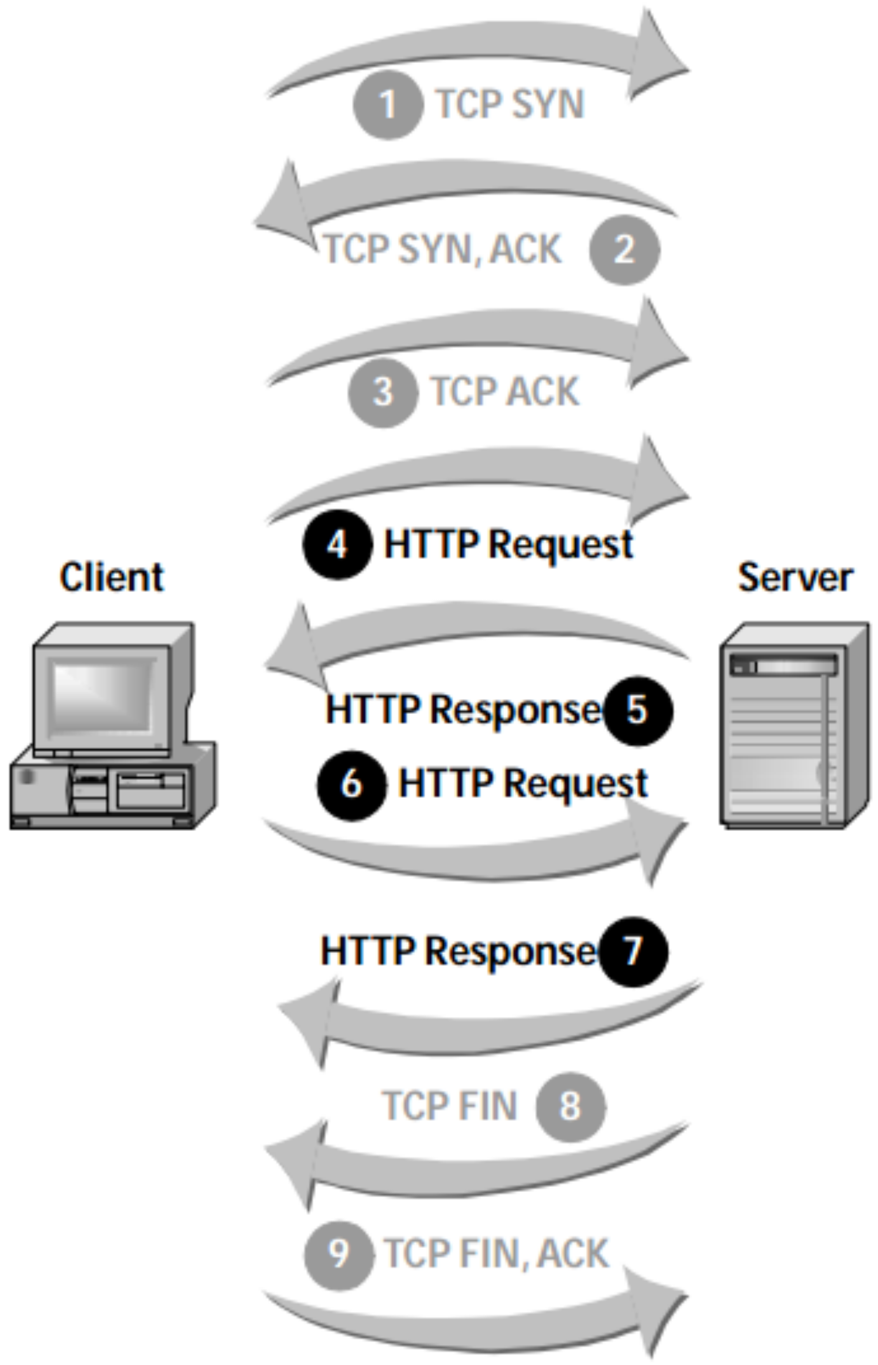
USFUL MATERIAL

- TO BE SORTED OUT

HTTP requires a TCP connection

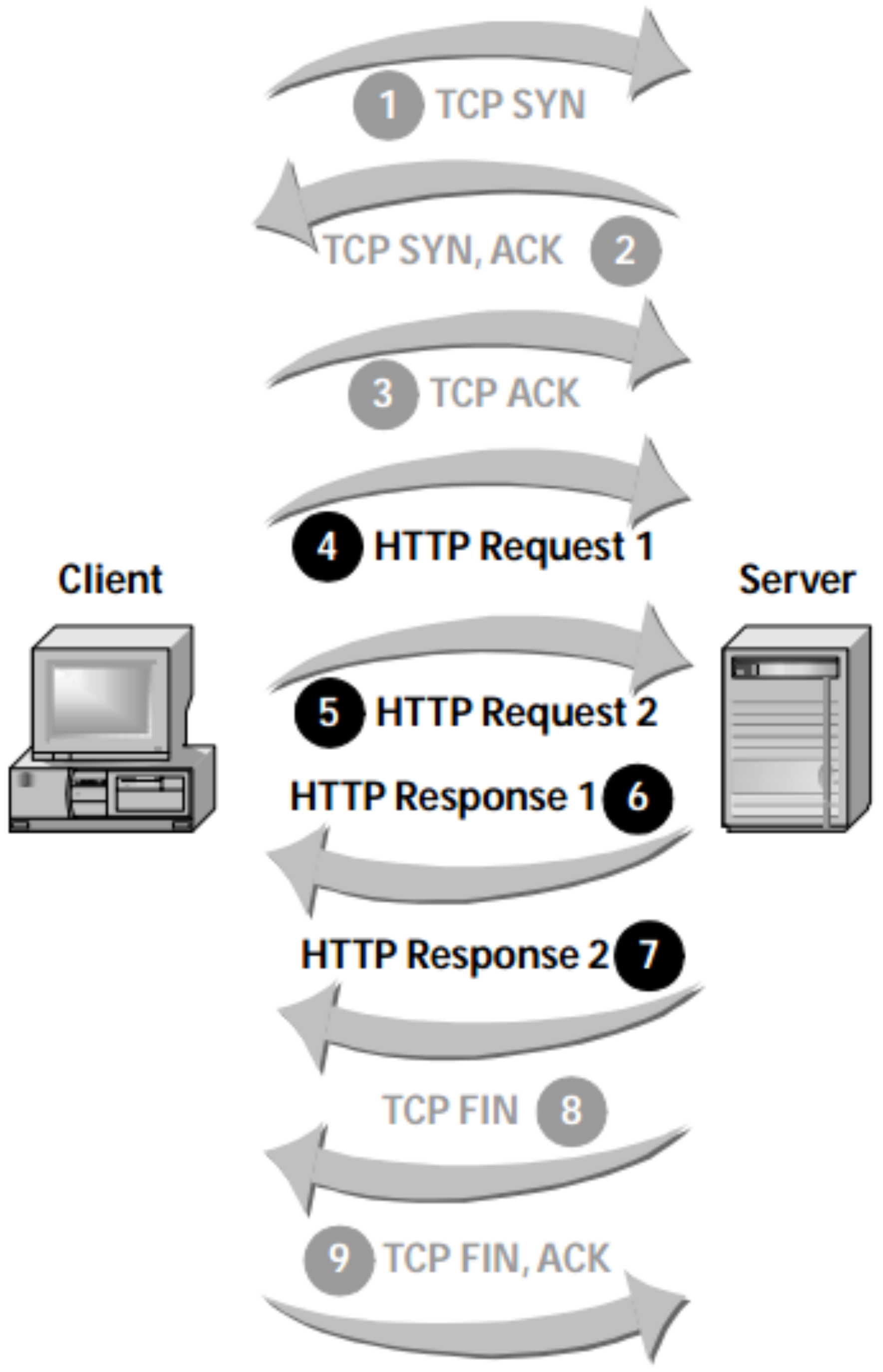


◀ **Figure**
Before systems can exchange HTTP messages, they must establish a TCP connection. Steps 1, 2, and 3 in this example show the connection establishment. Once the TCP connection is available, the client sends the server an HTTP request. The final two steps, 6 and 7, show the closing of the TCP connection.



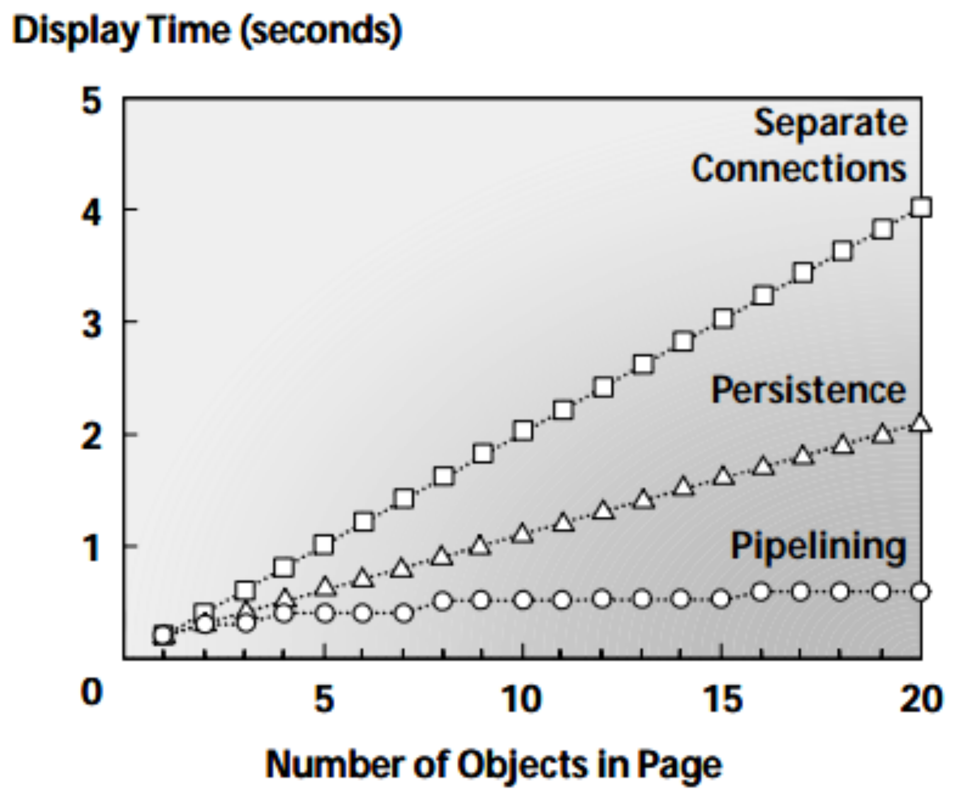
◀ **Figure 2.4**
With persistent connections, a client can issue many HTTP requests over a single TCP connection. The first request is in step 4, which the server answers in step 5. In step 6 the client continues by sending the server another request on the same TCP connection. The server responds to this request in step 7 and then closes the TCP connection.

Figure 2.5 ▶
Pipelining lets an HTTP client issue new requests without waiting for responses from its previous messages. In the figure, the client sends its first request in step 4. It immediately follows that with a second request in step 5. The client does not wait for the server's response, which arrives in step 6.



Compares the performance of pipelining, persistence, and single, serial connections

Figure 2.6 ▶
Both persistence and pipelining can offer significant improvements in HTTP performance, especially for complex Web pages with many objects. As the graph shows, a Web page with 20 objects (not atypical) can take about 4 seconds when the client uses serial connections. Persistence and pipelining together can reduce this time to less than 1 second.


Display Time (seconds)
Number of Objects in Page
Separate Connections
Persistence
Pipelining