Network Programming [CACS355]

Er.Sital Prasad Mandal

BCA- VI
Mechi Campus
Bhadrapur, Jhapa, Nepal

(Email : info.sitalmandal@gmail.com)

Text Book

- **Elliotte Rusty Harold, “Java Network Programming” O’Reilly, 2014**
- **David Reilly, Michael Reilly, “ Java Netwoking Programming and DistributedComputing”**

Books

Unit - 4

- 4 HTTP
- 4.1 The Protocol
  - Keep-Alive
- 4.2 HTTP Methods
  - The Request Body
  - Cookies
  - CookieManager
  - CookieStore

Unit - 4

HTTP

The Hypertext Transfer Protocol (HTTP) is a standard that defines how a web client talks to a server and how data is transferred from the server back to the client.

HTTP is usually thought of as a means of transferring HTML files and the pictures embedded in them.

It can be used to transfer pictures, Microsoft Word documents, Windows *.exe files, or anything else that can be* represented in bytes.

Unit - 4

The Protocol

HTTP is the standard protocol for communication between web browsers and web servers.

- HTTP specifies how a client and server establish a connection,
- how the client requests data from the server,
- how the server responds to that request, and
- finally, how the connection is closed.

HTTP connections use the TCP/IP protocol for data transfer.

The Protocol

For each request from client to server, there is a sequence of four steps:

1. Client opens a TCP connection to the server on port 80, by default; other ports may be specified in the URL.
2. Client sends a message to the server requesting the resource at a specified path.
3. Server sends a response to the client.
4. The server closes the connection.

The Protocol

Each request and response has the same basic form:

- ✓ A header line
- ✓ An HTTP header
- ✓ Containing metadata
- ✓ A blank line
- ✓ A message body

The Protocol

A typical client request :

```
GET /index.html HTTP/1.1
User-Agent:   Mozilla/5.0 (Macintosh; Intel Mac OS X 10.8; rv:20.0)
              Gecko/20100101 Firefox/20.0
Host: en.wikipedia.org
Connection: keep-alive
Accept-Language: en-US,en;
Accept-Encoding: gzip, deflate
Accept: text/html,application/xhtml+xml,application/xml
```

The Protocol

A typical successful response :

```
HTTP/1.1 200 OK
Date: Sun, 21 Apr 2013 15:12:46 GMT
Server: Apache
Connection: close
Content-Type: text/html; charset=ISO-8859-1
Content-length: 115
```

The Protocol

A typical successful response :

```
HTTP/1.1 200 OK
Date: Sun, 21 Apr 2013 15:12:46 GMT
Server: Apache
Connection: close
Content-Type: text/html; charset=ISO-8859-1
Content-length: 115
```

A response code :

- ✓ 100 to 199 always indicates an informational response
- ✓ 200 to 299 always indicates success
- ✓ 300 to 399 always indicates redirection
- ✓ 400 to 499 always indicates a client error
- ✓ 500 to 599 indicates a server error

Keep-Alive

HTTP 1.0 opens a new connection for each request.
The time taken to open and close all the connections in a typical web session can outweigh the time taken to transmit the data, especially for sessions with many small documents.

This is a problematic for encrypted HTTPS connections using Secure Sockets Layer & Transport Layer Security.

In HTTP 1.1 and later, the server doesn’t have to close the socket after it sends its response.

A client indicates that it’s willing to reuse a socket by including a *Connection field in the* HTTP request header with the value *Keep-Alive:*
Connection: Keep-Alive

Keep-Alive

We can control Java’s use of HTTP Keep-Alive with several system properties:

- http.keepAlive to “true or false” to enable/disable. Default enable.
- http.maxConnections to the number of sockets. The default is 5.
- http.keepAlive.remainingData to true, It is false by default.
- sun.net.http.errorstream.enableBuffering to true, It is false by default.
- sun.net.http.errorstream.bufferSize, The default is 4,096 bytes.
- sun.net.http.errorstream.timeout , It is 300 milliseconds by default.

HTTP Methods

Communication with an HTTP server follows a request-response pattern: one stateless request followed by one stateless response.

Each HTTP request has two or three parts:

- Start line containing the HTTP method and a path to the resource.
- Header of name-value fields that provide meta-information.
- Request body containing a representation of a resource (POST and PUT only)

There are four main HTTP methods, operations that can be performed:

- GET
- POST
- PUT
- DELETE

The Request Body

The GET method retrieves a representation of a resource identified by a URL.
The exact location of the resource we want to GET from a server is specified by the various parts of the path and query string.

POST and PUT are more complex. In these cases, the client supplies the representation of the resource, in addition to the path and the query string.

The representation of the resource is sent in the body of the request, after the header. That is, it sends these four items in order:

1. A starter line including the method, path and query string, and HTTP version
2. An HTTP header
3. A blank line
4. The body

For example, this POST request sends form data to a server:

```
POST /cgi-bin/register.pl HTTP 1.0
Date: Sun, 27 Apr 2013 12:32:36
Host: www.cafeaulait.org
Content-type: application/x-www-form-urlencoded
Content-length: 54
username=Elliotte+Harold&email=elharo%40ibiblio.org
```

The Request Body

However, the HTTP header should include two fields that specify the nature of the body:

- A Content-length field that specifies how many bytes are in the body.
- A Content-type field that specifies the MIME media type of the bytes

For example, here’s a PUT request that uploads an Atom document:

```
PUT /blog/software-development/the-power-of-pomodoros/ HTTP/1.1
Host: elharo.com
User-Agent: AtomMaker/1.0
Authorization: Basic ZGFmZnk6c2VjZXJldA==
Content-Type: application/atom+xml;type=entry
Content-Length: 322


<?xml version="1.0"?>
<entry xmlns="http://www.w3.org/2005/Atom">
<title>The Power of Pomodoros</title>
<id>urn:uuid:101a41a6-722b-4d9b-8afb-ccfb01d77499</id>
<updated>2013-02-22T19:40:52Z</updated>
<author><name>Elliotte Harold</name></author>
<content>I hadn’t paid much attention to Pomodoro...</content>
</entry>
```

Cookies

Many websites use small strings of text known as *cookies to store persistent client-side* state between connections.

Cookies are passed from server to client and back again in the HTTP headers of requests and responses.

Cookies are limited to nonwhitespace ASCII text, and may not contain commas or semicolons.
To set a cookie in a browser, the server includes a Set-Cookie header line in the HTTP header.

For example, this HTTP header sets the cookie “cart” to the value “ATVPDKIKX0DER”:

```
HTTP/1.1 200 OK
Content-type: text/html
Set-Cookie: cart=ATVPDKIKX0DER
```

If a browser makes a second request to the same server, it will send the cookie back in a Cookie line in the HTTP request header like so:

```
GET /index.html HTTP/1.1
Host: www.example.org
Cookie: cart=ATVPDKIKX0DER
Accept: text/html
```

Cookies

As long as the server doesn’t reuse cookies, this enables it to track individual users and sessions across multiple, otherwise stateless, HTTP connections. Servers can set more than one cookie.

For example, a request made to Amazon fed my browser five cookies:

```
Set-Cookie:skin=noskin
Set-Cookie:ubid-main=176-5578236-9590213
Set-Cookie:session-token=Zg6afPNqbaMv2WmYFOv57zCU1O6Ktr
Set-Cookie:session-id-time=2082787201l
Set-Cookie:session-id=187-4969589-3049309
```

In addition to a simple name=value pair, cookies can have several attributes that control their scope including:

- expiration date,
- path,
- domain,
- port,
- version, and
- security options

Cookies

For example, this request sets a user cookie for the entire *foo.example.com domain:*

```
Set-Cookie: user=elharo;Domain=.foo.example.com

Set-Cookie: user=elharo; Path=/restricted

Set-Cookie: user=elharo;Path=/restricted;Domain=.example.com

Cookie: user=elharo; Path=/restricted;Domain=.foo.example.com

Set-Cookie: user=elharo; expires=Wed, 21-Dec-2015 15:23:00 GMT

Set-Cookie: user="elharo"; Max-Age=3600

Set-Cookie: key=etrogl7*;Domain=.foo.example.com; secure

Set-Cookie: key=etrogl7*;Domain=.foo.example.com; secure; httponly
```

Cookies

Here’s a complete set of cookies sent by Amazon:

Set-Cookie: skin=noskin; path=/; domain=.amazon.com; expires=Fri, 03-May-2013 21:46:43 GMT

Set-Cookie: ubid-main=176-5578236-9590213; path=/; domain=.amazon.com; expires=Tue, 01-Jan-2036 08:00:01 GMT

Set-Cookie: session-token=Zg6afPNqbaMv2WmYFOv57zCU1O6KtrMMdskcmllbZ
cY4q6t0PrMywqO82PR6AgtfIJhtBABhomNUW2dITwuLfOZuhXILp7Toya+
AvWaYJxpfY1lj4ci4cnJxiuUZTev1WV31p5bcwzRM1Cmn3QOCezNNqenhzZD8TZUnOL/9Ya;
path=/; domain=.amazon.com; expires=Thu, 28-Apr-2033 21:46:43 GMT

Set-Cookie: session-id-time=2082787201l; path=/; domain=.amazon.com; expires=Tue, 01-Jan-2036 08:00:01 GMT

Set-Cookie: session-id=187-4969589-3049309; path=/; domain=.amazon.com; expires=Tue, 01-Jan-2036 08:00:01 GMT

CookieManager

Java 5 includes an abstract java.net.CookieHandler class that defines an API for storing and retrieving cookies.

Java 6 fleshes this out by adding a concrete java.net.CookieManager subclass of CookieHandler can be use.

Before Java will store and return cookies, you need to enable it:

```
CookieManager manager = new CookieManager();
CookieHandler.setDefault(manager);
```

Three policies are predefined:

- CookiePolicy.ACCEPT_ALL All: cookies allowed
- CookiePolicy.ACCEPT_NONE: No cookies allowed
- CookiePolicy.ACCEPT_ORIGINAL_SERVER: Only first party cookies allowed

CookieManager

For example, this code fragment tells Java to block third-party cookies but accept firstparty cookies:

```
CookieManager manager = new CookieManager();
manager.setCookiePolicy(CookiePolicy.ACCEPT_ORIGINAL_SERVER);
CookieHandler.setDefault(manager);
```

you can implement the CookiePolicy interface yourself and override the shouldAccept() method:

```
public boolean shouldAccept(URI uri, HttpCookie cookie)
```

Example 6-1. A cookie policy that blocks all .gov cookies but allows others

```
import java.net.*;
public class NoGovernmentCookies implements CookiePolicy {
@Override
public boolean shouldAccept(URI uri, HttpCookie cookie) {
if (uri.getAuthority().toLowerCase().endsWith(".gov")
|| cookie.getDomain().toLowerCase().endsWith(".gov")) {
return false;
}
return true;
}
}
```

CookieStore

We can retrieve the store in which the CookieManager saves its cookies with the getCookieStore() method:

```
CookieStore store = manager.getCookieStore();
```

The CookieStore class allows you to add, remove, and list cookies so you can control the cookies that are sent outside the normal flow of HTTP requests and responses:

```
public void add(URI uri, HttpCookie cookie)
public List<HttpCookie> get(URI uri)
public List<HttpCookie> getCookies()
public List<URI> getURIs()
public boolean remove(URI uri, HttpCookie cookie)
public boolean removeAll()
```

Example 6-2. The HTTPCookie class

```
package java.net;
public class HttpCookie implements Cloneable {
public HttpCookie(String name, String value)

public boolean hasExpired()
public void setComment(String comment)
public String getComment()
public void setCommentURL(String url)
public String getCommentURL()
public void setDiscard(boolean discard)
public boolean getDiscard()
public void setPortlist(String ports)
public String getPortlist()
public void setDomain(String domain)
public String getDomain()
public void setMaxAge(long expiry)
public long getMaxAge()

public void setPath(String path)
public String getPath()
public void setSecure(boolean flag)
public boolean getSecure()
public String getName()
public void setValue(String value)
public String getValue()
public int getVersion()
public void setVersion(int v)
```

```
public static boolean domainMatches(String domain,
String host)
public static List<HttpCookie> parse(String header)

public String toString()
public boolean equals(Object obj)
public int hashCode()
public Object clone()
}
```

Unit - 4

4 HTTP

4.1 The Protocol

Keep-Alive

4.2 HTTP Methods

The Request Body

Cookies

CookieManager

CookieStore

Review unit -4
The End