TCP Networking in Java

Some reminders

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

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 is currently in draft form (evolution of SPDY).

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.

HTTP Overview

HTTP Responses

An HTTP response contains a result code, header fields, and a body.
The HTTP protocol expects the result code and all header fields to be returned before any body content.

Some commonly used status codes include:

- 100: Continue
- 200: OK

- 404: the requested resource is not available
- 401: the request requires HTTP authentication
- 500: an error occurred inside the HTTP server that prevented it from fulfilling the request
- 503: the HTTP server is temporarily overloaded and unable to handle the request

For detailed information on this protocol, see the Internet RFCs: HTTP/1.0 (RFC 1945), HTTP/1.1 (RFC 2616). (http://www.rfc-editor.org/rfc.html)

See also http://en.wikipedia.org/wiki/Http

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.

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.

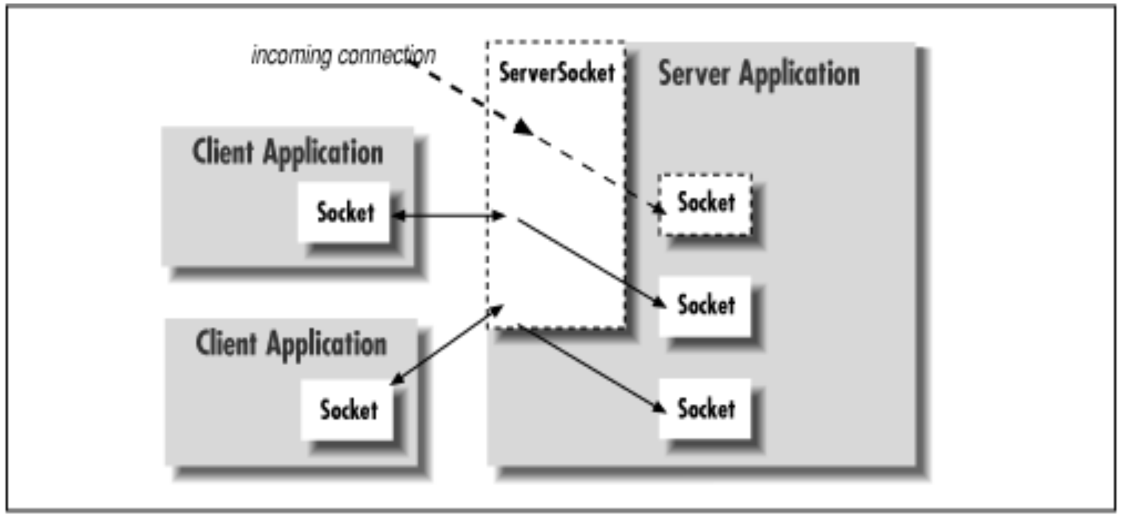
Sockets

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

```java
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.

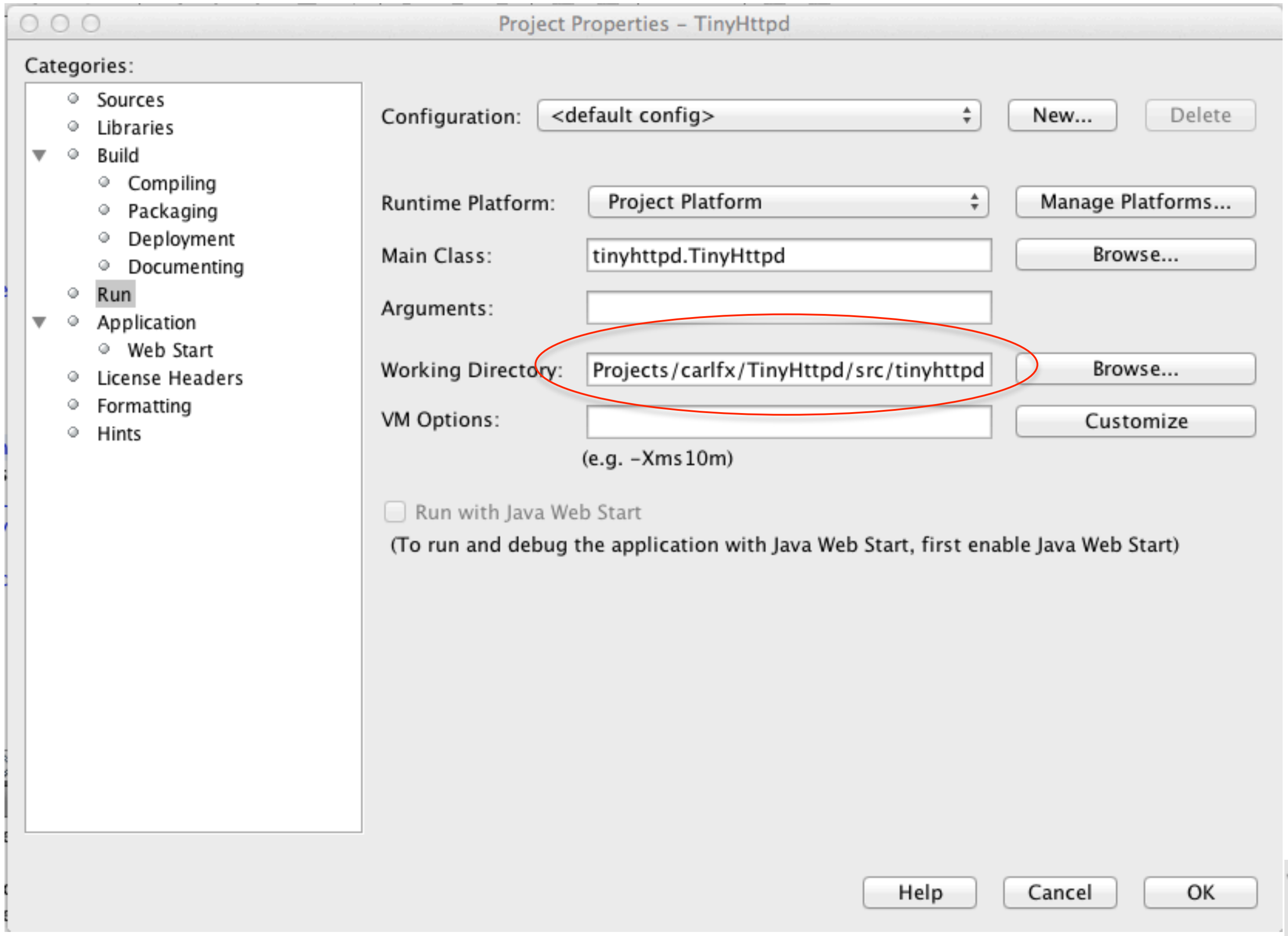
(On a time-slicing system, this is less important.)

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:

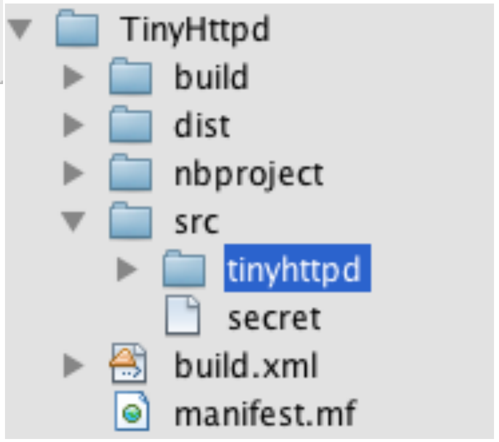
Working Directory: Projects/carlfx/TinyHttpd/src/tinyhttpd

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.