Programming Assignment: Development of a Proxy Cache

La prova avrà una valutazione da 0 a 3 punti da aggiungere alla media dell’esito delle due prove di esonero scritte. La gestione dei thread non è da considerarsi obbligatoria.

In this lab you will develop a small web proxy server which is also able to cache web pages. This is a very simple proxy server which only understands simple GET-requests, but is able to handle all kinds of objects, not just HTML pages, but also images.

Code

The code is divided into three classes as follows:

- **ProxyCache** holds the start-up code for the proxy and code for handling the requests.
- **HttpRequest** contains the routines for parsing and processing the incoming requests from clients.
- **HttpResponse** takes care of reading the replies from servers and processing them.

Your work will be to complete the proxy so that it is able to receive requests, forward them, read replies, and return those to the clients. You will need to complete the classes **ProxyCache**, **HttpRequest**, and **HttpResponse**. The places where you need to fill in code are marked with /* Fill in */. Each place may require one or more lines of code.

**NOTE:** As explained below, the proxy uses **DataInputStreams** for processing the replies from servers. This is because the replies are a mixture of textual and binary data and the only input streams in Java which allow treating both at the same time are DataInputStreams. To get the code to compile, you must use the -deprecation argument for the compiler as follows:

```
javac -deprecation *.java
```

If you do not use the -deprecation flag, the compiler will refuse to compile your code!

**Running the Proxy**

Running the proxy is as follows:

```
java ProxyCache port
```

where **port** is the port number on which you want the proxy to listen for incoming connections from clients.

**Configuring Your Browser**
You will also need to configure your web browser to use your proxy. This depends on your browser. In Internet Explorer, you can set the proxy in "Internet Options" in the Connections tab under LAN Settings. In Netscape (and derived browsers, such as Mozilla), you can set the proxy in Edit->Preferences and then select Advanced and Proxies. In both cases you need to give the address of the proxy and the port number which you gave when you started the proxy. You can run the proxy and browser on the same computer without any problems.

**Proxy Functionality**

The proxy works as follows.

1. The proxy listens for requests from clients
2. When there is a request, the proxy spawns a new thread for handling the request and creates an HttpRequest-object which contains the request.
3. The new thread sends the request to the server and reads the server's reply into an HttpResponse-object.
4. The thread sends the response back to the requesting client.

Your task is to complete the code which handles the above process. Most of the error handling in the proxy is very simple and it does not inform the client about errors. When there are errors, the proxy will simply stop processing the request and the client will eventually get a timeout.

Some browsers also send their requests one at a time, without using parallel connections. Especially in pages with lot of inlined images, this may cause the page to load very slowly.

**Caching**

The basic functionality of caching goes as follows.

1. When the proxy gets a request, it checks if the requested object is cached, and if yes, then returns the object from the cache, without contacting the server.
2. If the object is not cached, the proxy retrieves the object from the server, returns it to the client, and caches a copy for future requests.

In practice, the proxy must verify that the cached responses are still valid and that they are the correct response to the client's request. You can read more about caching and how it is handled in HTTP in RFC 2068. For this lab, it is sufficient to implement the above simple policy.

**Programming Hints**

Most of the code you need to write relates to processing HTTP requests and responses as well as handling Java sockets.

One point worth noting is the processing of replies from the server. In an HTTP response, the headers are sent as ASCII lines, separated by CRLF character sequences. The headers are followed by an empty line and the response body, which can be binary data in the case of images, for example.

Java separates the input streams according to whether they are text-based or binary, which presents a small problem in this case. Only DataInputStreams are able to handle both text and binary data
simultaneously; all other streams are either pure text (e.g., BufferedReader), or pure binary (e.g., 
BufferedInputStream), and mixing them on the same socket does not generally work.

The DataInputStream has a small gotcha, because it is not able to guarantee that the data it reads 
can be correctly converted to the correct characters on every platform (DataInputStream.readLine() 
function). In the case of this lab, the conversion usually works, but the compiler will flag the 
DataInputStream.readLine()-method as deprecated and will refuse to compile without the - 
deprecation flag.

It is highly recommended that you use the DataInputStream for reading the response.

**Code for proxy cache lab**

This page contains the code for the proxy cache lab. You can copy the code into the correct files.

**ProxyCache.java**

```java
/**
 * ProxyCache.java - Simple caching proxy
 */

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

public class ProxyCache {
    /** Port for the proxy */
    private static int port;
    /** Socket for client connections */
    private static ServerSocket socket;

    /** Create the ProxyCache object and the socket */
    public static void init(int p) {
        port = p;
        try {
            socket = /* Fill in */;
        } catch (IOException e) {
            System.out.println("Error creating socket: "+ e);
            System.exit(-1);
        }
    }

    public static void handle(Socket client) {
        Socket server = null;
        HttpRequest request = null;
        HttpResponse response = null;

        /* Process request. If there are any exceptions, then simply 
         * return and end this request. This unfortunately means the 
         * client will hang for a while, until it timeouts. */

        /* Read request */
        try {
            BufferedReader fromClient = /* Fill in */;
            request = /* Fill in */;
        } catch (IOException e) {
            
```
System.out.println("Error reading request from client: " + e);
return;
}
/* Send request to server */
try {
    /* Open socket and write request to socket */
    server = /* Fill in */;
    DataOutputStream toServer = /* Fill in */;
    /* Fill in */
} catch (UnknownHostException e) {
    System.out.println("Unknown host: " + request.getHost());
    System.out.println(e);
    return;
} catch (IOException e) {
    System.out.println("Error writing request to server: " + e);
    return;
}
/* Read response and forward it to client */
try {
    DataInputStream fromServer = /* Fill in */;
    response = /* Fill in */;
    DataOutputStream toClient = /* Fill in */;
    /* Fill in */
    /* Write response to client. First headers, then body */
    client.close();
    server.close();
    /* Insert object into the cache */
    /* Fill in */
} catch (IOException e) {
    System.out.println("Error writing response to client: " + e);
}

/** Read command line arguments and start proxy */
public static void main(String args[]) {
    int myPort = 0;

    try {
        myPort = Integer.parseInt(args[0]);
    } catch (ArrayIndexOutOfBoundsException e) {
        System.out.println("Need port number as argument");
        System.exit(-1);
    } catch (NumberFormatException e) {
        System.out.println("Please give port number as integer.");
        System.exit(-1);
    }

    init(myPort);

    /** Main loop. Listen for incoming connections and spawn a new *
     * thread for handling them */
    Socket client = null;
    while (true) {
        try {
            client = /* Fill in */;
            handle(client);
        } catch (IOException e) {
            System.out.println("Error reading request from client: " + e);
            /* Definitely cannot continue processing this request, *
             * so skip to next iteration of while loop. */
            continue;
        }
    }
}
HttpRequest.java

/**
  * HttpRequest - HTTP request container and parser
  *
  */
import java.io.*;
import java.net.*;
import java.util.*;
public class HttpRequest {
    /** Help variables */
    final static String CRLF = "\r\n";
    final static int HTTP_PORT = 80;
    /** Store the request parameters */
    String method;
    String URI;
    String version;
    String headers = "";
    /** Server and port */
    private String host;
    private int port;

    /** Create HttpRequest by reading it from the client socket */
    public HttpRequest(BufferedReader from) {
        String firstLine = "";
        try {
            firstLine = from.readLine();
        } catch (IOException e) {
            System.out.println("Error reading request line: " + e);
        }
        String[] tmp = firstLine.split(" ");
        method = /* Fill in */;
        URI = /* Fill in */;
        version = /* Fill in */;
        System.out.println("URI is: " + URI);
        if (!method.equals("GET")) {
            System.out.println("Error: Method not GET");
        }
        try {
            String line = from.readLine();
            while (line.length() != 0) {
                headers += line + CRLF;
                /* We need to find host header to know which server to 
                 * contact in case the request URI is not complete. */
                if (line.startsWith("Host:")) {
                    tmp = line.split(" ");
                    if (tmp[1].indexOf(':') > 0) {
                        String[] tmp2 = tmp[1].split(":");
                    }
                
            }
        } catch (IOException e) {
            System.out.println("Error reading request line: " + e);
        }
    }
}
host = tmp2[0];
port = Integer.parseInt(tmp2[1]);
}
else {
host = tmp[1];
port = HTTP_PORT;
}

line = from.readLine();
}
)
catch (IOException e) {
System.out.println("Error reading from socket: "+ e);
return;
}
System.out.println("Host to contact is: " + host + " at port " + port);

/** Return host for which this request is intended */
public String getHost() {
return host;
}

/** Return port for server */
public int getPort() {
return port;
}

/**
 * Convert request into a string for easy re-sending.
 */
public String toString() {
String req = "";

req = method + " " + URI + " " + version + CRLF;
req += headers;
/* This proxy does not support persistent connections */
req += "Connection: close" + CRLF;
req += CRLF;
return req;
}

HttpResponse.java

/**
 * HttpResponse - Handle HTTP replies
 *
 *
 */
import java.io.*;
import java.net.*;
import java.util.*;

public class HttpResponse {
final static String CRLF = "\r\n";
/** How big is the buffer used for reading the object */
final static int BUF_SIZE = 8192;
/** Maximum size of objects that this proxy can handle. For the
 * moment set to 100 KB. You can adjust this as needed. */
final static int MAX_OBJECT_SIZE = 100000;
/** Reply status and headers */
String version;
int status;
String statusLine = "";
String headers = "";
/* Body of reply */
byte[] body = new byte[MAX_OBJECT_SIZE];

/** Read response from server. */
public HttpResponse(DataInputStream fromServer) {
    /* Length of the object */
    int length = -1;
    boolean gotStatusLine = false;

    /* First read status line and response headers */
    try {
        String line = /* Fill in */;
        while (line.length() != 0) {
            if (!gotStatusLine) {
                statusLine = line;
                gotStatusLine = true;
            } else {
                headers += line + CRLF;
            }

            /* Get length of content as indicated by
             * Content-Length header. Unfortunately this is not
             * present in every response. Some servers return the
             * header "Content-Length", others return
             * "Content-length". You need to check for both
             * here. */
            if (line.startsWith(/* Fill in */) ||
                line.startsWith(/* Fill in */)) {
                String[] tmp = line.split(" ");
                length = Integer.parseInt(tmp[1]);
            }
            line = fromServer.readLine();
        }
    } catch (IOException e) {
        System.out.println("Error reading headers from server: " + e);
        return;
    }

    try {
        int bytesRead = 0;
        byte buf[] = new byte[BUF_SIZE];
        boolean loop = false;

        /* If we didn't get Content-Length header, just loop until
         * the connection is closed. */
        if (length == -1) {
            loop = true;
        }

        /* Read the body in chunks of BUF_SIZE and copy the chunk
         * into body. Usually replies come back in smaller chunks
         * than BUF_SIZE. The while-loop ends when either we have
         * read Content-Length bytes or when the connection is
         * closed (when there is no Connection-Length in the
         * response. */
        while (bytesRead < length || loop) {
            /* Read it in as binary data */
        }
    } catch (IOException e) {
        System.out.println("Error reading body from server: " + e);
        return;
    }
}
int res = /* Fill in */;
if (res == -1) {
    break;
}
/* Copy the bytes into body. Make sure we don't exceed
 * the maximum object size. */
for (int i = 0;
    i < res && (i + bytesRead) < MAX_OBJECT_SIZE;
    i++) {
    /* Fill in */
}
bytesRead += res;
}
) catch (IOException e) {
    System.out.println("Error reading response body: " + e);
    return;
}

/**
 * Convert response into a string for easy re-sending. Only
 * converts the response headers, body is not converted to a
 * string.
 */
public String toString() {
    String res = ""
    res = statusLine + CRLF;
    res += headers;
    res += CRLF;
    return res;
}