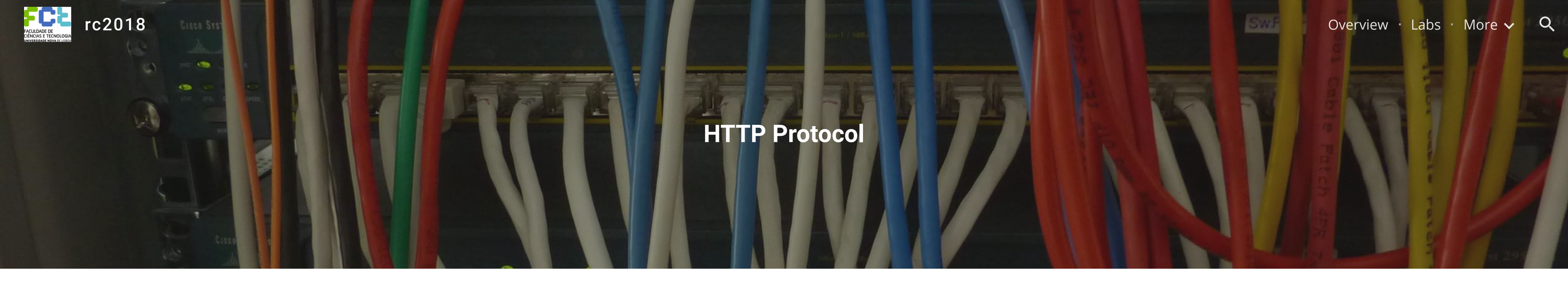


HTTP Protocol

Computer Networks

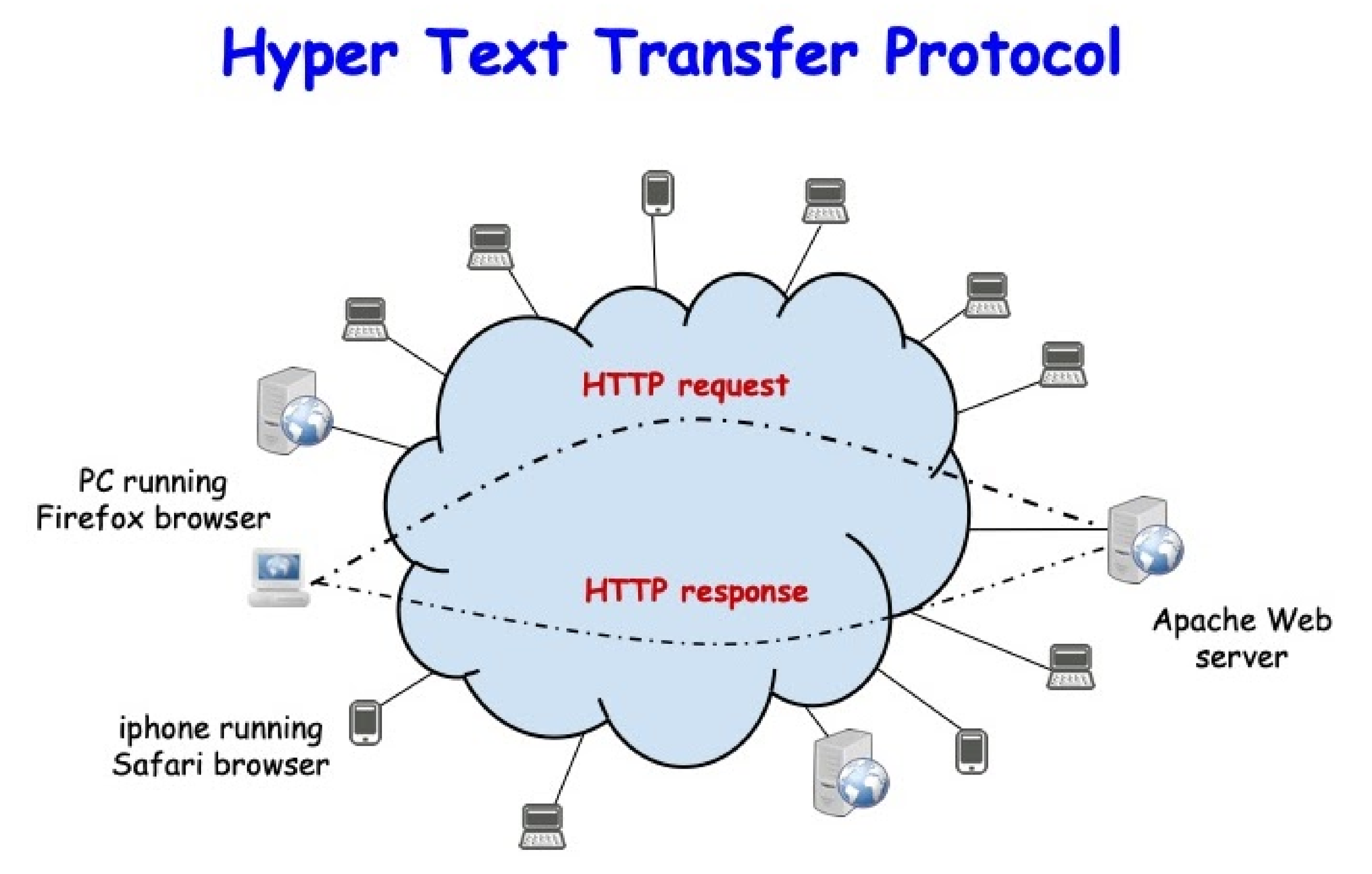
Lab #4 - Using the HTTP Protocol to Download Digital Objects from a Server

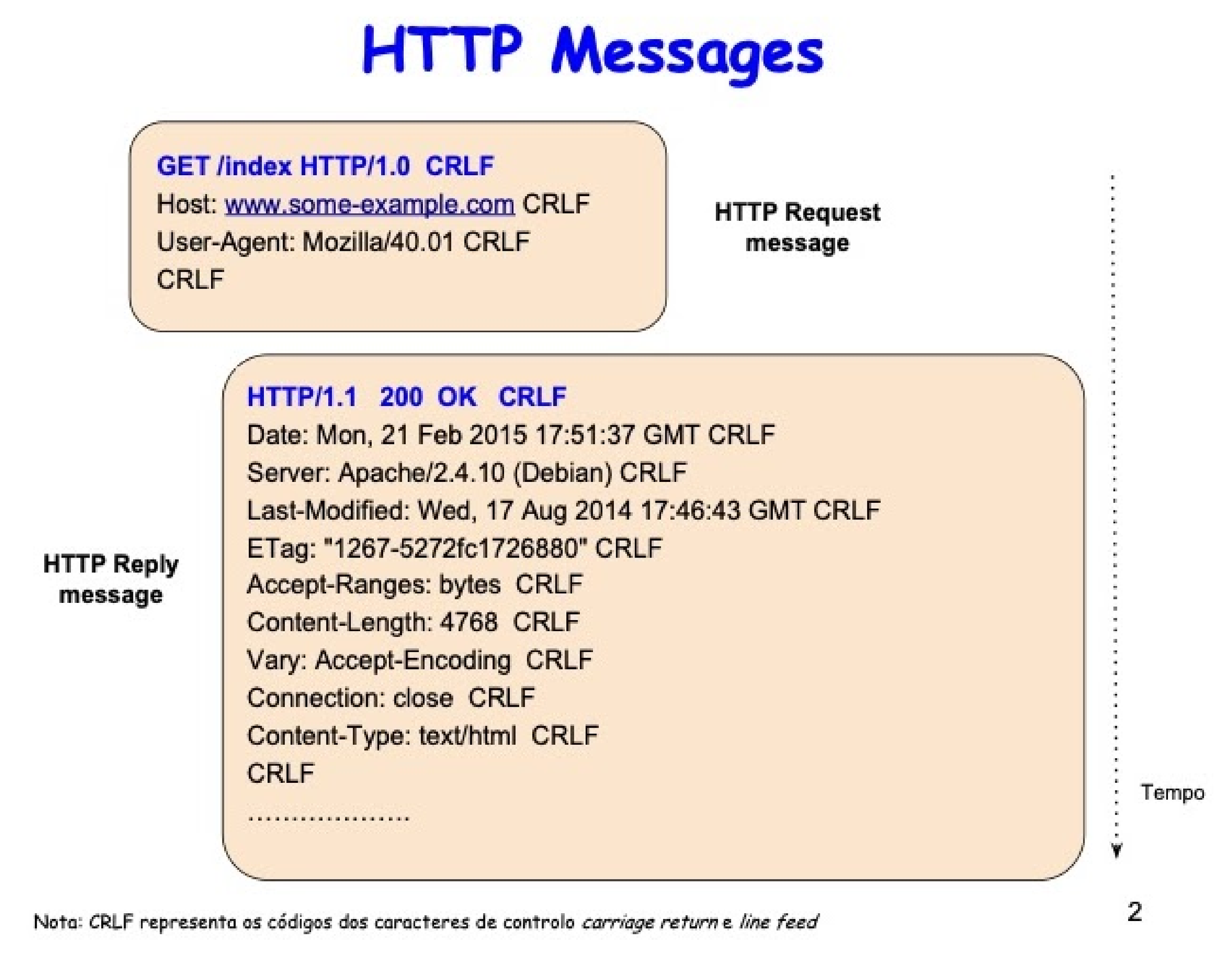
Summary

- HTTP Protocol
- Examples and utilities
- HTTP gets using ranges
- File transfer over HTTP with and without ranges

HTTP Protocol

- HTTP is a client / server protocol
- The server expects the client to open a TCP connection to its port (by default the port 80)
- In version 1.0 of the protocol, each request / reply HTTP transaction uses a different TCP connection
- In version 1.1 several HTTP transactions can share the same TCP connection





HTTP request header-fields

There are many different request header-fields that the client can send to the sever into the request message header. Below you will find some of a very long list.

Request Header-fields

Header-fields	Exemplo
User-Agent	User-Agent: Mozilla/40.0
Accept-Charset	Accept-Charset: utf-8
Accept-Encoding	Accept-Encoding: gzip
Accept-Language	Accept-Language: en-UK
If-Modified-Since	If-Modified-Since: Tue, 02 Feb 2016 14:25:41 GMT
If-Match	If-Match: "756154ad8d .... 3102f1349317f"
Range	Range: bytes=500-999
....	

3

HTTP requests made "by hand"

By using the following commands, you can access an HTTP server to see its replies. Try the following one:

```
telnet asc.di.fct.unl.pt 80
GET / HTTP/1.0 <return>
<return>
.... analyze the result.
```

Or the following one:

```
telnet www.google.com 80
GET / HTTP/1.0 <return>
<return>
.... analyze the result.
```

If your system doesn't have the **telnet** command, you can use instead the **nc** command:

```
nc -c asc.di.fct.unl.pt 80
GET / HTTP/1.0 <return>
<return>
.... analyze the result.
```

Or the following one:

```
nc -c www.google.com 80
GET / HTTP/1.0 <return>
<return>
.... analyze the result.
```

HTTP reply header-fields

The server sends information to the client by also using the header of the reply message, which is composed of different reply header-fields. These header-fields contains several informations, namely the reply object meta data, that are usefull to the client.

Reply Header-fields

Header-fields	Exemplo
Server	Server: Apache
Last-Modified	Last-Modified: Tue, 02 Feb 2016 14:25:41 GMT
Content-Type	Content-Type: text/html; charset=utf-8
Content-Length	Content-Length: 348
Content-Encoding	Content-Encoding: gzip
ETag	ETag: "3d2-52aca46b79fd9"
Accept-Ranges: bytes	Accept-Ranges: bytes
....	....

5

Java code examples

The java.net package has many classes to speedup the development of programs using network protocols. In what concerns the protocol HTTP, there is one, among many, that can be used to parse and access URLs:

- Class **URL** - allows parsing an url to, for example, get its different components. See file `URLparse.java` in the source code repository

There are many other classes available in the same package to develop programs based in the HTTP protocol. However, due to pedagogical reasons, you can only use the class URL to parse an url.

Any other requierements of your programs should be implemented by yourself or using the class ***Http***, available in the source code repository, which provides some extra methods to facilitate the development of Java programs build directly on top of the HTTP Protocol.

Below you will find several Java source code examples

Parsing an URL and opening a TCP connection to the server

```
String url = args[0]; // for example "http://google.com"
URL u = new URL(url);
// Assuming URL of the form http://server-name/path ....
int port = u.getPort() == -1 ? 80 : u.getPort();
String path = u.getPath() == "" ? "/" : u.getPath();
Socket sock = new Socket( u.getHost(), port );
OutputStream out = sock.getOutputStream();
```

Composing and sending a request to the server

```
String request = String.format(
"GET %s HTTP/1.0\r\n"+
"Host: %s\r\n"+
"User-Agent: X-RC2018\r\n\r\n", path, u.getHost());
out.write(request.getBytes());
```

Parsing the request message header in the server

```
line = Http.readLine(in);
String[] request = Http.parseHttpRequest(line);
line = Http.readLine(in);
while ( ! line.equals("") ) {
line = Http.readLine(in);
}
if( request[0].equalsIgnoreCase("GET") && request[1] != "") {
sendFile(request[1], out);
} else {
sendsNotSupportedPage(out);
```

Example - sending a reply message to the client

```
/**
* Sends an error message "Not Implemented"
*/
private static void sendsNotSupportedPage(OutputStream out)
throws IOException {
String page =
"<HTML><BODY>Demo server: request Not Implemented</BODY></HTML>";
int length = page.length();
String header = "HTTP/1.0 501 Not Implemented\r\n";
header += "Date: "+new Date().toString()+"\r\n";
header += "Content-type: text/html\r\n";
header += "Server: "+"X-Server-RC2018"+"\r\n";
header += "Content-Length: "+String.valueOf(length)+"\r\n\r\n";
header += page;
out.write(header.getBytes());
}
```

Sending the reply header and payload

```
File f = new File(name);
long size = f.length();
FileInputStream file = new FileInputStream(f);
StringBuilder header = new StringBuilder("HTTP/1.0 200 OK\r\n");
header.append("Server: "+"X-Server-RC2018"+"\r\n");
header.append("Content-Length: "+String.valueOf(size)+"\r\n\r\n");
out.write(header.toString().getBytes());
// send payload
byte[] buffer = new byte[1024];
for(;;) {
int n = file.read(buffer);
if( n == -1) break;
out.write(buffer, 0, n);
}
```

Program demos - HTTP client and server

In the source code repository you will find a naif HTTP client (***HttpClientDemo.java***) that is able to request an object denoted by the url passed as argument. The reply of the server is shown to the user (but it is not parsed, nor interpreted).

Study its code and try to access some urls like for example:

- http://google.com
- http://www.google.com
- http://asc.di.fct.unl.pt
- http://asc.di.fct.unl.pt/rc

Explain the output and understand how it works.

In the source code repository you will also find a naif HTTP server (***HttpServerDemo.java***) that is able to serve the requested files from its local file system. For example, if the server is running in the same machine as your browser, you can interact with it using: **`http://localhost:8080`**

You can use the browser of your choice and try to access URL **`http://localhost:8080`**. The browser will also show the answer of the server.

For example, if the server is executing in your localhost, and in its current directory there is a file called "***Http.java***", using the URL **`http://localhost:8080/Http.java`**, allows one to see the contents of that file.

Exercise A

The provided client is able to use the HTTP request / reply protocol to obtain a file and show its content. Modify class **`HttpClientDemo`** in a way that it may be used to download files from a HTTP server. Call your class **`getFile`** for example.

In the source code repository there is another class (URLget) which uses the class URL to download an object from an HTTP server. It is shown just for study purposes. Due to pedagogical reasons, you also cann't use it to complete any of your exercises.

Range requests

During the (already very long) life of the HTTP protocol, many extensions have been introduced. Most of those extensions introduced new request and reply headers-fields.

To facilitate the access to very big or huge objects, HTTP supports partial requests. This feature is inspired from reading direct access files in slices.

To make a partial request, one needs at least two resources:

- An HTTP server that understands range requests and replies to them
- A client using the range request header-field. Examples:
  - **Range: bytes=100-199** // requesting the 100 bytes starting with byte 100
  - **Range: bytes=100-** // requesting from byte 100 to the end of the object

HTTP request made "by hand"

By using the following commands, you can access an HTTP server to see its replies. Try the following one:

```
telnet asc.di.fct.unl.pt 80
GET / HTTP/1.0 <return>
Range: bytes=10-20 <return>
<return>
.... analyze the result.
```

Source code example:

```
OutputStream out = sock.getOutputStream();
String request = String.format("GET %s HTTP/1.0\r\n"+
"Host: %s\r\n"+
"Range: bytes=100-199\r\n"+
"User-Agent: X-RC2018\r\n\r\n", path, u.getHost());
out.write(request.getBytes());
```

You can learn about this feature of the HTTP protocol by starting here or looking at RFC 7233.

Range requests and replies are specially useful to deal with multimedia information as you will see in the next week's work.

In the source code repository you will also find a lazy HTTP server ***(HttpLazyServer.java)*** that is able to serve the requested files and partially supports ranges. In fact, the full support of RFC 7233 ranges is quite complex and extensive. The provided server only supports ranges of the forms shown above. This server is **lazy** since it only sends at most **MAX_BYTES** bytes in each reply. You can find this constant in its source file. Therefore, if you want to know it, you should use the **RTFC** method (Read The F. Code), which is the only accessible method when no manual is available.

To test the actions of the lazy server, launch it in a directory where you also put the file **`Earth.jpg`** (which size is around 13 Mbytes). You can access it by using the url **`http://localhost:8080/Earth.jpg`** with the browser of your choice.

Exercise - downloading a file using successive range downloads

Now, assuming that you already have extended the demo HTTP client to transfer a file to the local file system, modify your `getFile` class in order to be able to fully transfer a file of any size from the ***HttpLazyServer.java.***

Since in your future work, you will be directed to use docker hosted web servers, you can also test your client with a different HTTP server. This one is available in a docker container and can be loaded using the command:

```
docker run -it --rm -p 8080:8080 -v $(PWD):/public danjellz/http-server
```

This server fully supports HTTP ranges with no limitation related to the size of the requested object.

You must run this docker in the current directory of the content you want to serve.

Java Tips

Class: FileOutputStream

A file can be open in **Append mode**

```
File f = new File (fileName);
FileOutputStream fos = new FileOutputStream(f,true); // append mode
```

Class: File

the method `length()` can be used to know the **length of a file**.

```
File f = new File (fileName);
long size = f.length();
```

Class: RandomAccessFile

Methods **`seek(), length() and skipBytes()`** can be used to read or write slices of a file. Examples:

```
File f = new File (fileName);
RandomAccessFile file = new RandomAccessFile ( f, "rw" );
file.skipBytes(n); // Attempts to skip over n bytes of input discarding the skipped bytes.
file.seek(k); // Sets the file-pointer offset, measured from the beginning of this file, at which the next read or write occurs.
n = file.read( buffer, 0, len ); // Reads up to len (or buffer.length) bytes of data from this file into an array of bytes
```