Datanet Assignment 2

Webserver

Casper B. Hansen
University of Copenhagen
2100 Universitetsparken 5
Copenhagen, Denmark
fvx507@alumni.ku.dk

ABSTRACT

Leading into a discussion on the design of the system, I shed light on how my initial thoughts and approach. The design stage is followed in chronological with considerations put forth along the way, so as to allow the reader to understand the rationale behind its conclusions. Lastly, we take a look at the extensibility of the system, and how this translates into scheduled changes to the system implementation.

General Terms

Experimentation, Measurement

Keywords

Web, Server, Protocol

1. OVERVIEW

For the implementation assignments I chose an object-oriented language that is compiled into machine code and thus executed directly by the CPU, making it both fast and extensible for further development on later assignments.

Languages	C / C++
Libraries	BSD Sockets (`<sys/socket.h>`)

By choosing a common language, like C++, and using a standard UNIX library, the server should compile and run on any UNIX system. Considerations were made to add support for Windows systems, but discarded due to time constraints and because I didn't have a Windows environment to test against available. Such cross-platform support could be added using the Winsock library.

2. DESIGN

After gaining some basic knowledge of the API I went on to build a crude program based on what I had learned. Once a working socket program was in place, I turned what was initially pure C-code into classes and went on to design the program logic that would drive the web server.

2.1 Initial Considerations

I began by analyzing various sources of BSD socket programming examples online. Since I had no prior experience with BSD sockets this served to gain an overview of potential class hierarchies. Fortunately, it seems that it didn't need to be as complicated as I had initially thought it to be.

Inspired by a particular[1] example, I decided to follow most of its principles almost to the letter, as I wanted to keep the socket-side programming as simple as possible, and this code provided just that. Slight optimizations were made, but most of the socket classes is creditted to this resource. This makes up the socket layer of the design (see figure 1).

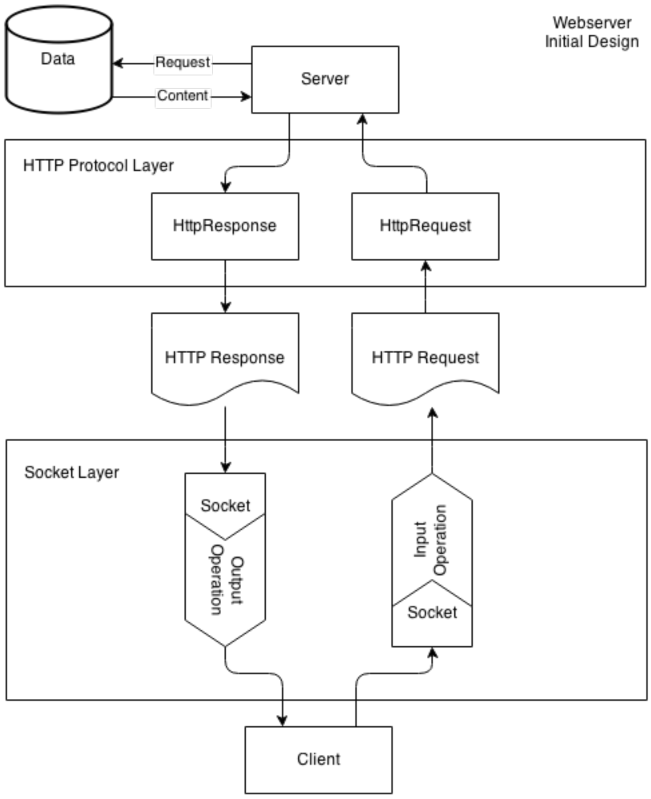

Figure 1: Initial webserver design

[1]BSD http://tldp.org/LDP/LG/issue74/tougher.html

3. PRELIMINARY DESIGN

Having a simple socket back-end in place, I began sketching out a class that would handle the delegation of work and data communication across subsequent class components. The `Server` class is the driving component of the system (see figure 1); it handles initialization, main run-time logic and exception handling.

At this point, the system was able to send and receive socket messages. The system is thus far only meant to support the HTTP protocol, and so the remainder comes down to the two-part task of implementing HTTP messages; 1) parse and validate received requests, and 2) generate and send an appropriate response back.

Since HTTP requests and responses do share some common elements (e.g. the HTTP version), I reason that a superclass (`HttpMessage`) might be useful. Following this, I then subclassed the `HttpMessage` into the two classes `HttpRequest` and `HttpResponse`, which encapsulate the concepts of HTTP requests and responses.

The `HttpRequest` parses the raw socket data received. Should it be invalid class throws an `HttpException` which that are used for error responses — that is, 4xx- and 5xx-HTTP responses (i.e. if the server doesn't support a given request method). If it is valid, the server then tries to process the request, formulating an `HttpResponse` which is then sent back to the socket that received the request (see figure 1).

The `ServerSocket` was then extended to be able to transmit HTTP messages via the stream operators — using `>>` to receive socket data and `<<` to send socket data.

3.1 Limitations

The system is limited to support only a few of the requests documented in the HTTP RFC 2616[2] specification. A table of the level of support on each is given below.

GET	Partially Supported, minimal use of fields
HEAD	Fully Supported, needs optimization
DELETE	Planned Support, due for 3rd revision
POST	Planned Support, due for 2nd revision
PUT	Planned Support, due for 3rd revision
OPTIONS	Planned Support, no due date set
CONNECT	Not Supported
TRACE	Not Supported

The `GET` request is the most essential, as it provides the server with information on what the client wishes to access on the server. No server can function without it, and therefore support for it was implemented.

Since the system is to be extended into a peer-to-peer (or P2P) system, the `HEAD` request was inevitable, as in a P2P system a client may not wish to download a file, but rather simply want information about it. This is what the `HEAD` request is meant for, and so in preparation of the coming system extension support for this was added.

3.2 Extensibility

Many considerations went into the initial design of the system, but many were discarded due to the time constraint. Thus, many design considerations were jotted down and therefore not a part of this version, but rather subject to change in the following assignments.

The following discusses each part of the implementation that ought be changed.

3.2.1 HTTP Messages

The abstraction of a generic HTTP message is considered good, but as responses to different requests vary much in their structure and required data, it may prove useful to subclass the `HttpResponse` into each supported response (i.e. `HttpGetResponse`).

3.2.2 Generated HTML

Some HTTP responses require the generation of HTML content. Such functionality should be encapsulated in a class. This will probably follow from the HTTP response subclassing, as discussed (see section 3.2.1). Having such a class would make it easier to extend the system in any aspect that may require server generated content.

4. TESTS

Most tests were carried out by accessing the server via the `telnet`. Unfortunately, I did not have time to show test cases of the exception handling mechanism, although it does work, and can be verified by compiling and running the server on the readers own system.

4.1 GET

To test the `GET` method, we must first ensure that the server can serve files. In the test below I queried the server for the `/sample.txt` file. It is expected to return an HTTP response with the 200 code, meaning everything went well, followed by the server name, date, correct MIME-type, the length of the content and that the server closes the connection.

```
HTTP/1.0 200 SERVER MESSAGE
Server: datanet/1.0a
Date: Thu, 15 May 2014 23:12:52 GMT
Content-Type: text/plain
Content-Length: 60
Connection: close

This is a sample text file.
The content type is plain/text.
```

Figure 2: Response for `GET /sample.txt HTTP/1.0`

When the server is queried for a directory we want it to serve the `index.html` file, if it exists. The test below shows the server handling this request.

[2]HTTP (RFC 2616) — `http://www.w3.org/Protocols/rfc2616/rfc2616.html`

```
HTTP/1.0 200 SERVER MESSAGE
Server: datanet/1.0a
Date: Thu, 15 May 2014 23:18:39 GMT
Content-Type: text/html
Content-Length: 172
Connection: close

<html>
    <head>
        <title>index.html</title>
    </head>

    <body>
        <h1>index.html</h1>
        <p>This is the root .html-file.</p>
    </body>
</html>
```

Figure 3: Response for `GET / HTTP/1.0`

Following up on the previous test, if no such file exists we wish to generate and serve a list of files in the queried directory. For purposes of testing this, I renamed the `index.html` to `_index.html` temporarily.

```
HTTP/1.0 404 SERVER MESSAGE
Server: datanet/1.0a
Date: Thu, 15 May 2014 23:23:41 GMT
Content-Type: text/html
Content-Length: 260
Connection: close

<html><head><title>404 - Not Found</title></head><body><ul><li><a href="../">../</a></li><li><a href="test/">test/</a></li><li><a href="./">./</a></li><li><a href="sample.txt">sample.txt</a></li><li><a href="_index.html">_index.html</a></li></ul></body></html>
```

Figure 4: Response for `GET / HTTP/1.0`, where no `index.html` can be found

4.2 HEAD

In testing the `HEAD` request there aren't many cases, we simply expect the header of a `GET` response to be returned. Which is indeed the case, as shown below.

```
HTTP/1.0 200 SERVER MESSAGE
Server: datanet/1.0a
Date: Thu, 15 May 2014 22:59:28 GMT
Content-Type: text/plain
Content-Length: 60
Connection: close
```

Figure 5: Response for `HEAD /sample.txt HTTP/1.0`

5. REFERENCES

[1] James F. Kurose, Keith W. Ross, *Computer Networking, A Top-Down Approach*, Pearson Education, Sixth Edition, 2013