Datanet, Spring 2011, DIKU
1st Assignment - HTTP server

Truls Asheim
truls@diku.dk

ABSTRACT

This report describes the implementation of a simple concurrent HTTP server in erlang. All of the basic problems faced when working with HTTP will be exposed because of a restrictive assignment. The server as implemented is able to perform basic file-serving functions using a limited subset the HTTP protocol.

1. INTRODUCTION

In this assignment we are to implement a simple HTTP server in our language of choice. I have implemented my server in erlang. Erlang is a concurrent, functional language designed specifically for implementing concurrent server applications with an emphasis on availability and stability.

As HTTP is a huge and in many ways complex protocol only a subset of it have been implemented in this assignment. It does, however implement enough to be able, with certain limitations, to serve files to HTTP compatible clients.

Finally, a number of tests has been performed on the server to ensure that it works as intended when being queried with both correct and erroneous requests.

2. DESIGN

The design of this server is highly influenced by the chosen implementation language. As it is common in functional languages it is built around many small well defined functions. This creates readable programs which are easy to debug and understand. Furthermore it makes reusing code easier as dependencies within the program are kept to a minimum.

Lets look at the lifecycle of the server. This will serve both as an insight into the ideas behind the design and as a "What does this do when and why?" guide for the code:

1. Upon invocation of `server:start()` a socket is opened and two processes are spawned. `lstn_loop` puts the socket in accepting mode and `mime_lookup` provides a mime lookup service used for providing correct MIME-type for the files served.
2. When `lstn_loop` receives a connection it spawns another instance of itself and calls `rcv_loop`. So basically the process which receives the connection will die when the request has been processed.
3. Once the HTTP request has been received it is processed by the body of `rcv_loop`. Here, the headers is parsed and request type, uri and other information is extracted. If the request type is found to be supported the rest of the request will be processed.
4. `handle_path` its then called and receives the URI, request type and headers as parameters. Based on this it can take one of three paths.
   (a) If the URI refers to a file, the file is opened and returned. The HTTP response is assembled by the function `http_respones`. The process `mime_lookup` is asked to map the extension of the file to a MIME-type.
   (b) If the URI refers to a directory, a directory listing is generated and returned.
   (c) If the URI does not exists or is inaccessible an error is returned. All errors are generated by the `http_error` which generates a complete response based on a HTTP status code.

   All responses which consist of HTML output is passed through the function `html_function` which acts as a template.
5. Finally, the response is sent back to the client and the socket is closed.

The server's ability to handle multiple requests simultaneously is introduced by the actions in step two. Once the listening process receives a request it spawns another instancs into a process which processes the request received. The newly spawned process replaces the listening role the old process once had.

3. HTTP IMPLEMENTED

The purpose of this section is to account for the subset of HTTP implemented. HTTP is a huge protocol so supporting everything is not feasible This section is dedicated to the parts of the HTTP protocol that is implemented.

3.1 Headers

3.1.1 Request

Currently, none of the request headers are taken into account when processing a request. This is in part due to the fact that most of them are irrelevant to the servers prime purpose which is to serve raw files. This section will therefore be dedicated to describing the most important headers which are present in a request and what their purpose is. The decision to avoid taking headers into account was also made because some headers, Accept* especially, is rather complicated to parse.

The infrastructure to take headers into account when generating a response is present in the server so doing this will be easy whenever needed by future expansions.

Accept This header contains the client's preferred MIME-types. Abiding this header is not mandatory according to section 14.1 of the RFC.

Accept-Encoding This is perhaps one of the most important headers in the request. It can be used by the server to ensure that responses are sent in a character set which is understood by the client.

Connection indicates the clients preferred connection type. Its values are keep-alive and close. This server will always send back "Connection: close" to instruct the client to close the connection right after the response have been received.

Host This header is fundamental to most of the internet as it allows servers to use virtual hosting. The Host header contains the hostname used in the request URL and is used by servers to determine which site should be sent.

3.1.2 Response

This section contains a description of the headers included in a response from the server and their relation to the RFC.

Connection Connection: closed is sent with every response. This is required by RFC2616 section x.x. The value closed requires the client to close the connection immediately after the request have been received.

Date The date field is populated with a timestamp in RFC1123 format during header compilation in the server. The presence of this header and usage of RFC1123 date format is required by RFC2616 section 14.18

Last-Modified is sent only if the request is for a file. Its value is taken from the file system. Its presence is required when we get to implementing cache functionality.

Content-Length Holds the number of octets in the response body. This is taken from the filesize when sending a file or by counting the number of bytes in the response when returning a filelisting.

Content-Type is sent with every response. If the response is a directory index or a error message this is set to `text/html`. If the request if for a file its extension is used to look up a MIME-type in `/etc/mime.types`

3.2 Limitations

The server only accepts requests from HTTP/1.1 clients. Requests made in other versions of the HTTP protocols will be rejected with a 505 error code. Despite of this being in violation with RFC2616 this decision was made to avoid dealing with the complexities introduced by backwards compatibility. For instance, HTTP/1.1 compatible servers and clients are required by RFC2616 section 3.3 to handle three date formats in order to maintain backwards compatibility with HTTP/1.0 clients.

It doesn't implement chunked coding. In order to conform to RFC2616 section 3.6.1 a server must be able receive requests transmitted in chunked encoding even if it is unable to send them. Implementing this was deemed to be beyond the scope of this assignment.

The server is also somewhat lacking in the area of error handling. If the server crashes while handling a request the TCP socket is closed immediately without notice. If this was handled correctly, the client would receive a 500 status code.

4. EXTENSIBILITY

One of the requirements of this assignment was that the design of the implemented server should support future expansion without requiring major changes. This server does indeed fulfill this requirement given its modular design. All of the basic functions for handling the HTTP protocol are present and can be reused immediately. Furthermore, file handling is limited to a single function which means that the server can be made to serve any content simply by changing this one function. The functions for parsing HTTP request are also flexible enough to support currently unsupported HTTP methods and non-standard headers.

In the next assignment we are to implement a caching HTTP proxy server. One obvious way to transform this server into a proxy would be to simply replace the file handling code with a datastore and a HTTP client. Since we are working in erlang an obvious choice for at datastore would be ETS. ETS is erlang's native key/value store capable of handling large amounts and can which can exist both in memory or on disk.

Of course not all functionality required by a proxy is present. For instance, cache-related headers are currently not taken into consideration when processing a request. This will need to be added along with code for doing date processing and conversion. The HTTP request header parser also makes no distinctions based on the semantics of the request URL. This is also required by a working proxy server. All of this can however, be implemented within the limits of the design.

5. TESTING

Below is a list of all the tests performed together with an indication of whether or not the operation succeeded with expected results. The tests have been executed manually using a combination of `curl(1)` and Firefox 4 with Firebug. "White Box" testing has been used, meaning that only results and error conditions which are either known to the author or deliberately emitted by the code are exposed. As noted earlier, should an unexpected critical condition arise the socket will be closed immediately without any response being sent.

- GET requests for existing files: As expected, The file is returned

- GET requests for existing folders: As expected: A list of files in the folder is returned.

- GET requests for non-existing resource: As expected, A 404 code is returned together with a HTML error page

- GET requests for files/folders without read access: As expected, 403 is returned

- HEAD successful requests: As expected. The headers returned by a HEAD request are identical to those returned by a GET request.
- HEAD failed request: **FAIL** (In violation), A full reply with a body containing an error message. According to RFC2616 section 9.4 a HEAD request must **never** include a body.
- Request with invalid request line (for instance, “GET/ FOOTTP/a.b”): As expected, 400 is returned
- Request method other than GET or HEAD: As expected: 501 is returned
- Request contains invalid header (for instance, header contains a line which does not match the format “Header-Field: header value”: As expected, 400 is returned