CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

```
GET /thesis HTTP/1.1
Accept: physical/paper
Accept-Language: en, sv;q=0.5
TE: gzip, deflate;q=0.5
```

$\Downarrow$

```
HTTP/1.1 200 OK
Content-Language: en
Content-Type: physical/paper
Transfer-Encoding: gzip, chunked
```

An Extensible HTTP Client Library for Elixir

Verified with Property Based Random Testing

Master's Thesis in Computer Science - Algorithms, Language and Logic

AXEL JOHNSSON

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2017

Master's Thesis

An Extensible HTTP Client Library for Elixir

Verified with Property Based Random Testing

AXEL JOHNSSON

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2017

An Extensible HTTP Client Library for Elixir
Verified with Property Based Random Testing
AXEL JOHNSSON

Supervisor: Thomas Hallgren, Department of Computer Science and Engineering
Examiner: John Hughes, Department of Computer Science and Engineering

Master's Thesis 2017
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2017

An Extensible HTTP Client Library for Elixir
Verified with Property Based Random Testing
AXEL JOHNSSON
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Use cases of the World Wide Web (the Web) constantly evolve. In order to support present and future applications interacting with the Web, a new Hypertext Transfer Protocol (HTTP) client library is proposed. The present report describes the process of implementing an HTTP client library in the Elixir programming language. The major difference between the current HTTP client library and competing projects in the Elixir landscape is that state and process management are explicit to the user. Property-based and random testing were used during correctness evaluation and have been a success to use in the context of a functional language and an Internet protocol. Four abstractions were built on top of the library where two of them are possible because of the given control over state and process management. Early results tell that the HTTP client library is comparable to existing solutions in terms of robustness and performance.

Keywords: the Web, HTTP, Elixir, Erlang, property-based and random testing.

Acknowledgements

We would like to thank José Valim and the others working with the Elixir programming language for their contributions and for coming up with the initial project idea. John Hughes deserves credit for his suggestion on using property-based and random testing along with his work as the examiner. A special acknowledgement should be made to Thomas Hallgren for his valuable feedback and supervision. Finally, we recognize the support given by Thomas Arts and his colleagues at QuviQ AB.

Axel Johnsson, Gothenburg, September 2017

Contents

List of Code Listings **xiii**

List of Figures **xv**

List of Tables **xvii**

1 Introduction **1**
- 1.1 Background . . . 1
  - 1.1.1 The World Wide Web . . . 1
  - 1.1.2 Elixir . . . 2
  - 1.1.3 QuickCheck . . . 2
- 1.2 Problem Statement . . . 3
- 1.3 Related Work . . . 3
- 1.4 Limitations . . . 3

2 Theory **5**
- 2.1 Formal Language Theory . . . 5
  - 2.1.1 Hypertext Transfer Protocol . . . 6
  - 2.1.2 Parser Combinators . . . 7
  - 2.1.3 Finite Automaton . . . 7
    - 2.1.3.1 Definition . . . 8
    - 2.1.3.2 Motivation . . . 8
- 2.2 Random Testing . . . 9
  - 2.2.1 Strengths . . . 9
  - 2.2.2 Weaknesses . . . 10
- 2.3 Transport Layer Security . . . 10
  - 2.3.1 Protocol Description . . . 10
    - 2.3.1.1 TLS Handshake Protocol . . . 10
    - 2.3.1.2 TLS Record Protocol . . . 11

3 Method **13**
- 3.1 Landscape Survey . . . 13
- 3.2 Implementation . . . 13
  - 3.2.1 Encoder . . . 14
  - 3.2.2 Decoder . . . 15
    - 3.2.2.1 Reference . . . 16
    - 3.2.2.2 FSM . . . 17

3.2.2.3 Default 19
3.2.3 Connection 19
3.2.4 Public API 20
3.2.4.1 Demonstration 22
3.3 Evaluation 23
3.3.1 Correctness 23
3.3.1.1 Decoder Testing 23
3.3.1.2 Public API Testing 25
3.3.2 Extensibility 26
3.3.3 Robustness 26
3.3.3.1 Data Collection 27
3.3.3.2 Data Analysis 27
3.3.4 Performance 27
3.3.4.1 Data Collection 28
3.3.4.2 Data Analysis 28

4 Result **29**
4.1 Correctness 29
4.1.1 Decoder Testing 30
4.1.2 Client Testing 31
4.2 Extensibility 32
4.2.1 Stream 32
4.2.2 GenStage 34
4.2.3 Concurrent Client 36
4.2.4 Pipelining Client 38
4.3 Robustness 40
4.4 Performance 42
4.4.1 Decoder Benchmark 42
4.4.2 Client Benchmark 42

5 Discussion **45**
5.1 Landscape Survey 45
5.1.1 Haskell 45
5.1.2 Clojure 46
5.2 Correctness 46
5.2.1 Decoder Evaluation 47
5.2.1.1 Positive Testing 47
5.2.1.2 Negative Testing 48
5.2.2 Client Evaluation 48
5.3 Extensibility 49
5.4 Robustness 50
5.4.1 Decoder Evaluation 50
5.4.2 Client Evaluation 50
5.5 Performance 51
5.5.1 Decoder Evaluation 51
5.5.2 Client Evaluation 52

6 Conclusion **53**
6.1 Implementation . . . 53
6.2 Evaluation . . . 53
6.3 Future Work . . . 54

References **55**

A Code **I**
A.1 Typespecs . . . I
A.2 Generator Combinators . . . I
A.3 Extensibility Examples . . . III
A.3.1 Stream . . . III
A.3.2 GenStage . . . IV
A.3.3 Concurrent Client . . . VI
A.3.4 Pipelining Client . . . VIII

List of Code Listings

2.1 An example of an HTTP request. 6
2.2 An example of an HTTP response. 6
2.3 Type definition of a parser function as defined in a paper on the subject of combinator parsing [14]. 7
2.4 An example of a parser combinator in which the given function does the *or* operation from Boolean algebra of two parser functions. 7
2.5 ABNF for an HTTP status line as defined by RFC 7230. 9

3.1 Typespecs for the encoder. The types *Chunk.t*, *Message.trailer*, *Respoonse.t* and *Request.t* are given in Section A.1. 15
3.2 Typespecs for the implemented decoders. The types *Chunk.t*, *Message.trailer* and *Response.t* are given in Section A.1. 16
3.3 ABNF for an HTTP message as defined by RFC 7230. 17
3.4 A function for decoding an HTTP message using parser combinators. This example matches the protocol specification in Listing 3.3. 17
3.5 Partial implementation of the *FSM* decoder showing how the first three symbols in an HTTP response are parsed. 18
3.6 The controlling function in the *FSM* decoder. 18
3.7 Partial implementation of the *FSM* decoder showing how a header name in an HTTP response is parsed. 18
3.8 An example of how the Erlang function *split/2* was used when decoding HTTP chunks in the *Default* decoder. 19
3.9 Typespecs for the public API. The types *Chunk.t*, *Connection.t*, *Message.header*, *Stream* and *value* are given in Section A.1. 21
3.10 Example of doing a simple HTTP request with *Net.HTTP*. 22
3.11 Example of doing a simple HTTP request with *hackney*. 22
3.12 Example of doing a simple HTTP request with *httpc*. 23
3.13 A custom HTTP message generator. The helper functions: *sequence*, *many0*, and *optional* are defined in the listing given in Section A.2. . 24
3.14 An example of a *encode-decode-compare* property. 24
3.15 An example of a *stream* property. The functions *encode* and *decode* are locally defined in order to help the process of encoding and decoding respectively. 25

4.1 A sample of an HTTP response head generated with QuickCheck. Non ASCII-values were omitted in the generation for clarity sake. . . 30
4.2 A sample of an HTTP response chunk generated with QuickCheck. . 31

4.3 Implemented functions for use with Elixir's *Stream* module. 33
4.4 Shows how incoming socket data is matched in order for the current consumer to process and dispatch it to downstream consumers. 35
4.5 Exemplifies how the implemented HTTP library can be used in asynchronous mode where data is already available before calling any of the response functions in the *Net.HTTP* module. 35
4.6 The main loop of the *Concurrent Client* handling incoming messages such as user requests and socket data. 37
4.7 The function processing incoming response data in the *Concurrent Client* similarly to the one given in Listing 4.5. 37
4.8 The main loop of the *Pipelining Client* handling user calls. 39
4.9 Function responsible for receiving HTTP responses in the correct order and in a synchronous fashion as part of the *Pipelining Client*. . . 39

List of Figures

2.1 An example of a traffic light implemented as a finite-state machine. . 8

3.1 A dependency tree of the different components in *Net.HTTP*. A filled arrow denotes a fixed dependency while a dotted arrow signals a configurable dependency. 14

4.1 Comparison between *libcurl* and *Net.HTTP* from sending simple requests to 500 different URLs from the Moz Top 500 index. 41

4.2 Performance benchmark of the different decoders when processing various responses. Defining characteristics for each configuration is listed in Table 4.5. The statistics were obtained from 10 000 samples while the standard deviation was used as an error measure. 43

4.3 Performance benchmark of three HTTP clients running on the Erlang virtual machine. The implemented client used the default decoder during this benchmark. The statistics were obtained from 10 000 samples while the standard deviation was used as an error measure. . 44

List of Tables

3.1 Comparison of source code statistics between *Net.HTTP* and two similar projects. 14
3.2 Comparison of source code statistics between the implemented HTTP decoders. 14

4.1 Test coverage of major components of *Net.HTTP*. 29
4.2 A sample obtained when using QuickCheck to randomly generate 1 000 HTTP message heads. Proportions, which are given as percentages, and actual values are given for each class. Note that only the top 15 entries are listed and that *N.* and *V.* are used to denote header name and header value respectively. 30
4.3 A sample obtained when using QuickCheck to randomly generate 1 000 HTTP message chunks. Proportions, which are given as percentages, and actual values are given for each class. Note that only the top 15 entries are listed and that *N.* and *V.* are used to denote extension name and extension value respectively. 31
4.4 A sample obtained when using QuickCheck to randomly generate 1 000 sequences of public API calls. Proportions, which are given as percentages, and actual values are given for each class. Note that only the top 15 entries are listed and that *S.* is used to denote command sequence. 32
4.5 Defining characteristics for each configuration used during decoder benchmark. Size is measured in bytes where a byte is equal to an octet. 42

1
Introduction

The purpose of this chapter is to put the current thesis into context and give ideas of why the work is relevant and interesting. The chapter has been structured into these sections: a short background around the studied area, problem statement, related work and project limitations. The remaining report has been divided into several chapters where: Chapter 2 presents a theoretical background on relevant subjects, Chapter 3 answers how the work has been carried out, Chapter 4 describes the project results, and finally Chapter 5 and 6 give a discussion and a conclusion around the thesis.

1.1 Background

The present section gives a short background to following technologies: the Web, the Elixir and the Erlang programming languages and, a tool called QuickCheck for property-based and random testing.

1.1.1 The World Wide Web

Tim Berners-Lee began to work on the World Wide Web (the Web) while employed at the European Organization for Nuclear Research (CERN) in Geneva, Switzerland in 1989 [3]. In the year of 2000 he wrote: "The vision I have for the Web is about anything being potentially connected with anything.". Today the Web is made up of over 130 trillion individual pages according to Google[1], with respect to their statistic, Tim's prediction turned out right.

The Web is built around three concepts: Universal Resource Identifiers (URIs), Hypertext Transfer Protocol (HTTP) and Hypertext Markup Language (HTML). URIs are used to identify resources on the Web where they serve as edges that interconnect documents and other information. The architecture of the Web is a client-server model where clients and servers communicate via the HTTP protocol. HTML is the standard markup language used on the Web for creating web pages.

A good example of what is possible to do on the Web is Wikipedia[2], a free online

[1]https://www.google.com/insidesearch/howsearchworks/thestory/
[2]https://en.wikipedia.org/

encyclopedia. The website is among the ten most visited sites globally[3] and according to its founder Jimmy Wales, it consists of more than 40 million articles written in over 250 different languages[4].

While most people associate the Web with popular websites such as Wikipedia or Google, it is also a place for machines to communicate with other machines. By using a programmable web client, a program can interact with selected websites. Researchers at Stanford University have used a programmable HTTP client to collect tweets on Twitter for further analysis [8]. Another example is the Ensembl genome database project that gives foreign software access to its database via the HTTP protocol [20].

1.1.2 Elixir

Elixir[5] is a functional, concurrent and general-purpose programming language that first appeared in 2011. The language is developed as an open source project by a number of volunteers on the Internet. To understand Elixir, one must understand the language it builds upon. Erlang[6] is similarly a functional, concurrent and general-purpose programming language created by the Swedish telecommunication company Ericsson in the 80s [1]. The purpose was to design a language and run-time system that would allow the company to program concurrent, distributed, fault-tolerant, scalable and soft real-time software. Elixir is tightly bound to Erlang, examples of things that Elixir do differently are: improved support for collection processing, a new language syntax and focus on developer tooling.

1.1.3 QuickCheck

QuviQ QuickCheck is a novel tool for automatic test case generation built on the ideas of property-based and random testing [2]. It supports test case generation for both functions without side effects and stateful interfaces. The process of randomly generating test cases is controlled with a Domain-Specific Language (DSL) implemented in Erlang. The tool supports testing for race conditions and mocking along with a range of other features. The first version of QuickCheck, which was originally presented at the International Conference on Functional Programming (ICFP) in 2000 by John Hughes and Koen Claessen [4], was implemented in the Haskell[7] programming language. As a clarification, the Haskell version is often referred to as just QuickCheck but in this report, QuickCheck means QuviQ QuickCheck.

[3]http://www.alexa.com/siteinfo/wikipedia.org
[4]http://www.cbsnews.com/news/wikipedia-jimmy-wales-morley-safer-60-minutes/
[5]https://elixir-lang.org/
[6]http://www.erlang.org/
[7]https://www.haskell.org/

1.2 Problem Statement

The purpose of this thesis project is to support needs that existing HTTP client libraries in the Elixir ecosystem have ignored. Current libraries handle state and process management implicitly which reduce the set of possible ways client libraries can be extended. Therefore, a new HTTP client library is proposed that can support a wide range of present and future Internet applications written in the Elixir programming language.

Additionally, answers to the following questions are sought:

1. What are the downsides of giving users control over state and process management?
2. Can Quviq's QuickCheck be used in the context of a functional HTTP client with a satisfying result?
3. What is the best way to implement an HTTP message parser in Elixir? Is a high-level approach using parsing combinators efficient enough, or is it better to use a more low-level approach based on a finite-state machine?
4. How have existing HTTP clients written in a functional programming language solved the problem of streaming requests and responses?

1.3 Related Work

There are several successful HTTP client libraries in the industry and in the open source community. Two of these projects have been studied as part of the current thesis work. The relevant projects are listed below:

httpc[8] An HTTP client library included in the Erlang distribution which is both configurable and simple to use.

hackney[9] A third-party client written in Erlang developed as an open source project. Due to its large feature set, the library are used directly by users and by other HTTP libraries.

1.4 Limitations

The limitations for this thesis project are listed below:

- Any support for HTTP/2.0 or WebSockets is out of scope. The reason is time, the duration of the project is rather short and therefore it was decided to focus on HTTP/1.1 only.

[8]http://erlang.org/doc/man/httpc.html
[9]https://github.com/benoitc/hackney

- Several features typically present in existing HTTP libraries have been omitted since the goal was to implement a low-level interface which users can extend upon.

2

Theory

The goal of the present section is to introduce theories used in the current thesis project. Section 2.1 briefly describes formal languages and ways of parsing such languages. Section 2.2 gives basic terminology related to random testing and a discussion on the subject. In Section 2.3, the Transport Layer Security (TLS) protocol is presented.

2.1 Formal Language Theory

Formal language theory has its origin in the 1950s [11]. A formal language consists of an alphabet and a set of rules which determines what words or strings that are included in the language. A language's alphabet is a set of symbols that can be combined in order to produce strings of the language. The set of rules defining the structure of a formal language is referred to as a formal grammar. A formal grammar G is defined by a 4-tuple (N, Σ, P, S) where:

- N is a finite set of nonterminal symbols disjoint from Σ.
- Σ is a finite set of terminal symbols, called the alphabet, disjoint from N.
- P is a finite set of production rules where each rule is a combination of elements from N and Σ specifying the syntax of the language.
- S is the start symbol $S \in N$.

The set P containing the production rules of a formal grammar can be restricted in ways in order classify grammars according to their computational requirements. Noam Chomsky's hierarchy consists of the following four types: unrestricted grammar, context-sensitive grammar, context-free grammar and regular grammar where the first grammar is the most powerful meaning that it recognizes the most languages. Each grammar class has one or more computational models corresponding to the class. The software component implementing a formal grammar is often referred to as a decoder or a parser. Various methods exist for describing formal grammars and one of the more popular ones when working with context-free grammars is the Backus-Naur Form (BNF). In the context of Internet protocols, the Augmented Backus-Naur Form (ABNF)[1] that extends BNF with practices and conveniences is often used.

[1]https://tools.ietf.org/html/rfc5234

2.1.1 Hypertext Transfer Protocol

The Hypertext Transfer Protocol (HTTP) is the communication protocol of the Web. It is developed by the World Wide Web Consortium (W3C) where the most recent specification of the HTTP/1.1 protocol is defined in a number of Requests for Comments (RFCs). The primary source when working with the protocol during this project was RFC 7230[2] which defines the message syntax of the protocol.

An HTTP message can be either a request or a response where the former is always sent by a client while the latter is only sent by servers. Both of these messages contain: a start line, a number of headers and an optional body. The start line can be either a request line or a status line depending on if the message is a request or a response respectively. A header is simply a key-value pair. There are two types of bodies: plain and chunked. A plain body is a string of bytes and can be sent when the size of it is known. A chunked body is constructed from a number of chunks and allows for streaming of HTTP messages. When sending a chunked body, a number of trailers, which are identical to headers in terms of syntax, can be included at the end of the message.

An example of an HTTP request is given in Listing 2.1 while an example response is given in Listing 2.2. In the case of the response, lines 1-6 are referred to as the head section throughout this report. Similarly, lines 7-8 are a chunk and line 10 is called the tail.

Listing 2.1: An example of an HTTP request.

```
GET / HTTP/1.1
Host: www.example.com
Connection: close
```

Listing 2.2: An example of an HTTP response.

```
 1  HTTP/1.1 200 OK
 2  Connection: close
 3  Content-Type: text/plain
 4  Transfer-Encoding: chunked
 5  Trailer: Cache-Control
 6
 7  3
 8  ABC
 9  0
10  Cache-Control: max-age=3600
```

[2] https://tools.ietf.org/html/rfc7230

2.1.2 Parser Combinators

A method for implementing context-free grammars is called combinator parsing. When utilizing combinator parsing, parsers are modelled as functions. Larger parsers are built by combining smaller functions using higher order functions [14]. Parsers using this approach often look like the Backus Normal Form (BNF). From here on, the higher order functions used for gluing together parser functions will be referred to as parser combinators.

To illustrate a parser function, its general type is given in Listing 2.3. Elixir's *typespecs*[3] are used in this section when describing types. As can be seen, the input type is a string of characters or symbols while the return type consists of a list of tuples with a generic type a and another string of characters. The returned list can contain: a single element, many elements or none where the latter case denotes an error. The second element in the tuple is the suffix of the input string of characters since sometimes no or just a subset of the input is consumed during parsing.

```elixir
@type parser(a) :: (String.t -> [{a, String.t}])
```

Listing 2.3: Type definition of a parser function as defined in a paper on the subject of combinator parsing [14].

An example of a combinator function is given in Listing 2.4. The function works like the *or* operation in Boolean algebra since it succeeds as along as one of the given parsers is successful. In the case when both parsers succeed, the returned list will contain two tuples.

```elixir
@spec alt(parser(a), parser(a)) :: parser(a)
def alt(f, g) do
  fn input -> do
    f.(input) ++ g.(input)
  end
end
```

Listing 2.4: An example of a parser combinator in which the given function does the *or* operation from Boolean algebra of two parser functions.

2.1.3 Finite Automaton

Finite-State Machines (FSMs) or Finite-State Automata (FSA) have seen a wide range of practical applications such as: image processing, vending machines and automatic test case generation [15, 17, 2]. The current section introduces finite-state machines along with their formal definition and then ends with a motivation for why the model can be used when parsing the HTTP protocol.

[3]https://elixir-lang.org/getting-started/typespecs-and-behaviours.html

2.1.3.1 Definition

A finite-state machine reads input symbols and makes progress through a set of states as the symbols are processed. A function, which is referred to as the transition function, tells how a machine moves from one state to another solely based on the current state and the current input symbol. An example of a traffic light implemented as a finite-state machine is given in Figure 2.1. The set of states, the input alphabet and the transition function can be interpreted from the figure. There are four states and each of them is drawn as a circle, the input alphabet contains two symbols: *stop* and *proceed* while the transition function can be found by looking at the edges.

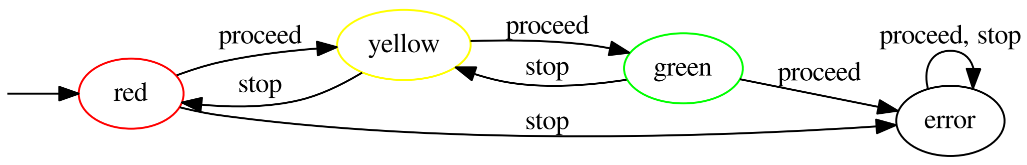


Figure 2.1: An example of a traffic light implemented as a finite-state machine.

The formal definition of a finite automaton can be defined by the 5-tuple (Q, Σ, δ, q_0, F) where [19]:

- Q is a finite set of states.
- Σ is a finite set of input symbols called the alphabet.
- $\delta : Q \times \Sigma \mapsto Q$ is the transition function.
- $q_0 \in Q$ is the start state.
- $F \subseteq Q$ is the set of accepting states.

The start state and the set of accepting states were left out in the introduction above. These states denote where the machine starts and terminates respectively. In Figure 2.1, the start state is pointed to by an arrow while there is no accepting state.

2.1.3.2 Motivation

A major use case of finite-state automata is in implementing grammars for languages referred to as regular languages. The class of languages that can be recognized by a finite-state machine belong to the least powerful grammar type in Chomsky's hierarchy [11]. Even though HTTP is specified with ABNF and thus a context-free grammar, the parts related to HTTP message responses in the protocol can be parsed with a finite-state machine. When working with finite-state automata, several operations can be made on an automaton which produce another finite-state machine. If this is true for a certain operation, then finite-state automata are said to be closed under that operation [19]. An example of operations that finite-state machines are closed under are: union, concatenation and Kleene closure. In Listing 2.5, the ABNF rule for a status line included in an HTTP response is given. Since finite automata are closed under concatenation, it can be proven that there exists a finite-state machine representing the ABNF rule *status-line* if finite-state automata exist for *HTTP-version*, *SP*, *status-code*, *reason-phrase* and *CRLF*. Identical arguments can be made for each of the sub-rules and each of the components in every sub-rule along with all the other rules relevant for an HTTP response.

```
status-line = HTTP-version SP status-code SP reason-phrase CRLF
```

Listing 2.5: ABNF for an HTTP status line as defined by RFC 7230.

A limitation of finite-state machines is their inability to recognize infinite patterns in a grammar due to their finite memory. An example of a language that can not be recognized by a finite-state machine is the language of matching parentheses where the number of opening and closing parenthesis have to be equal. Therefore, it is important to validate the presence of any recursion in the HTTP grammar. Luckily, there is no recursion in the relevant grammar for decoding HTTP messages and a finite-state machine can be implemented for that purpose. In other words, HTTP responses are regular.

2.2 Random Testing

Software testing is an approach for assessing the quality of a computer program. Many techniques exist for doing the testing and depending on what properties are of interest, one should choose methods accordingly. One branch of software testing is functional testing where a program's correctness is evaluated. The components involved in functional testing are: input data, a Software under Test (SUT), output data and a routine for determining the test outcome. This routine is often referred to as an oracle. In random testing, input data is independently generated by using a pseudorandom number generator (PRNG). It may seem naive to trust chance in producing good test cases but it actually competes well in practice with systematic methods [4].

2.2.1 Strengths

- As input data is chosen randomly, human intervention is replaced by chance when doing random testing. Thus, bias towards certain areas in the software is removed. For example, a developer may trust in the correctness of an unknowingly faulty component which a PRNG would not.
- It is sometimes possible to predict the significance of a successful random test. For example, one would could state "It is 90 % certain that some Software under Test will fail no more than once in 1 000 runs" [10]. Unfortunately there are several requirements for such as statement to be made: it is a system level test, a valid operational profile is available, representative input data can be

generated and an effective input oracle is available [10]. An operational profile weight different input values depending on their probability of occurring during operation [9].

- Since test cases are automatically generated by a computer, it is relatively cheap to use random testing. This gives rise to some interesting possibilities such as constructing and running test cases twenty-four hours a day, seven days a week.

2.2.2 Weaknesses

- A requirement for random testing is the availability of an efficient oracle. Oracles must be efficient since a large number of test cases is typically generated.
- Another problem with random testing is the lack of presence of an operational profile for the Software under Test. Without an operational profile, the ability to predict the significance of a random test disappears [10].

2.3 Transport Layer Security

The Transport Layer Security (TLS) protocol is a mean for secure communication over a computer network. The protocol can be layered on top of any existing reliable transport protocol such as the Transmission Control Protocol (TCP). The TLS protocol is designed to prevent: eavesdropping, tampering and message forgery. The TLS protocol is application independent though commonly used on the Web. In fact, around 40 % of the Web traffic was made with TLS in January 2017 [7].

2.3.1 Protocol Description

There are two layers of the TLS protocol: the TLS Handshake protocol and the TLS Record protocol [5]. The combination of the two protocols allows for efficient and private communication where data integrity and authentication are verified.

2.3.1.1 TLS Handshake Protocol

In the initial phase of a communication session via the TLS protocol, the TLS Handshake protocol is used to negotiate security parameters and a shared secret. The negotiation step is possible due to asymmetric cryptography (or public key cryptography) [6, 18] since public key cryptography allows for encryption and authentication without having to transmit any shared secrets beforehand. The shared secret is then used while transmitting application data via the TLS Record protocol.

As part of the handshake, communicating peers exchange certificates containing public keys and other information identifying their selves. In order to prevent message forgery, also referred to as an Man-In-The-Middle Attack (MITM), the public key of a peer has to be verified. Otherwise, a user may be tricked into sending private information to an unintended receiver. A server's public key is verified with a process based on public key cryptography where it is possible for a client to check if the current server is trusted by an authority. It is up to the client to decide which authorities to trust via the use of so called root certificates. Additionally, in the context of TLS and the Web, a client must check the server's domain or IP address against the identity given in the certificate exchanged by the server according to RFC 2818[4].

2.3.1.2 TLS Record Protocol

The TLS Record protocol prevents eavesdropping and tampering through the use of symmetric cryptography and Keyed-Hashing for Message Authentication (HMAC). When sending a message over the TLS Record protocol, the message is made private by encrypting it using symmetric cryptography while the data integrity and authentication can be verified using HMAC [16]. Both methods require the use of a shared secret. The shared secret is negotiated in the early phase of a communication session using the TLS Handshake protocol.

[4]https://tools.ietf.org/html/rfc2818

3 Method

The current chapter aims to describe how the proposed Hypertext Transfer Protocol (HTTP) client library was implemented along with criteria and methods for evaluating the resulting software. It also introduces the landscape survey that was done during the project, see Section 3.1. Section 3.2 presents the implementation while Section 3.3 gives the evaluation.

3.1 Landscape Survey

As a way of gaining knowledge and ideas from HTTP client libraries implemented in other functional programming languages, the following clients were studied: *http-client*[1] and *clj-http*[2] written in Haskell and Clojure respectively. The former library is developed as an open source project and provides a low-level interface along with several extensions while the latter library has a more high-level feature set. The reasons for choosing an HTTP client library written in Haskell are the language's way of lazily evaluating expressions and that it is a statically typed language. On the other hand, the reason for studying an HTTP library built with Clojure is that it is a new language running on a virtual machine originally intended for another host language. The approach was to study the documentation and source code for both of these libraries. Since different libraries support different feature sets, most of the effort was spent on the basics such as how requests and responses are sent and received.

3.2 Implementation

Since the main goal of the current thesis was to produce an extensible HTTP client library, much time was spent working on the implementation. The library is called *Net.HTTP* while the architecture of the implementation can be divided into the following components: *Encoder*, *Decoder*, *Connection* and *Public API*. See Figure 3.1 for a dependency tree of the different components. Every component is described in more detail in this section.

[1]https://github.com/snoyberg/http-client
[2]https://github.com/dakrone/clj-http

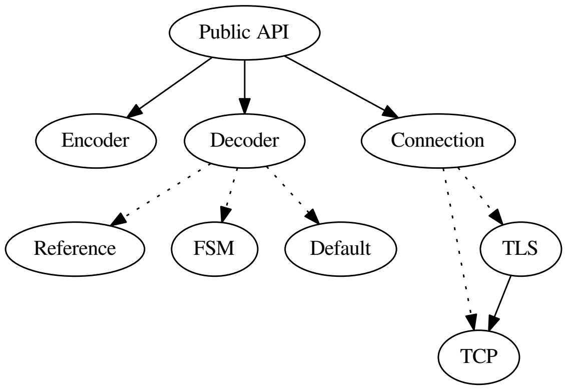


Figure 3.1: A dependency tree of the different components in *Net.HTTP*. A filled arrow denotes a fixed dependency while a dotted arrow signals a configurable dependency.

To get an understanding of the size of the implemented library and to put it into relation with similar projects, see Table 3.1. There, the number of files, number of blank lines, number of comments and number of source code lines are given for each project. The numbers were collected with a tool called *cloc*[3]. Note that all of the implemented decoders are included in the comparison as well as a number of files used by *httpc* that is shared with a web server included in the Erlang distribution. Furthermore, similar statistics are given for each of the decoders used in this project in Table 3.2.

Table 3.1: Comparison of source code statistics between *Net.HTTP* and two similar projects.

Project	**Files**	**Blanks**	**Comments**	**Source Lines of Code**
Net.HTTP	16	339	114	1 666
hackney	29	956	1 058	6 631
httpc	17	909	1 551	5 193

Table 3.2: Comparison of source code statistics between the implemented HTTP decoders.

Decoder	**Files**	**Blanks**	**Comments**	**Source Lines of Code**
Reference	1	56	35	283
FSM	1	85	18	278
Default	1	39	2	185

3.2.1 Encoder

An encoder converts a piece of information from one format to another. In the context of building an HTTP client library, an encoder has to be implemented

[3]https://github.com/AlDanial/cloc

that converts an in-memory HTTP request into the in-transfer format defined in the HTTP specification. The encoder produced during the project was designed in two parts: a set of functions that encode certain HTTP values and a stateful interface gluing these functions to form a complete encoder. Typespecs for the encoder functions are given in Listing 3.1.

Listing 3.1: Typespecs for the encoder. The types *Chunk.t*, *Message.trailer*, *Respoonse.t* and *Request.t* are given in Section A.1.

```elixir
@doc "Encodes the start line and headers of an HTTP message."
@spec encode_head(Response.t | Request.t)
  :: {:ok, binary} | {:error, any}

@doc "Encodes a chunk of an HTTP message."
@spec encode_chunk(Chunk.t | binary)
  :: {:ok, binary} | {:error, any}

@doc """
Encodes the body of an HTTP message.

The second argument should match the length given in the
"Content-Length" header.
"""
@spec encode_body(binary, integer)
  :: {:ok, binary} | {:error, any}

@doc "Encodes trailers of an HTTP message."
@spec encode_tail([Message.trailer])
  :: {:ok, binary} | {:error, any}
```

Each of these functions return a tuple with a status flag as the first element and a value as the second element. Depending on the status flag, encoded data or an error explanation is returned. The stateful interface guides the user through the process of encoding a full message by deciding what encoder functions to use according the message in process and it also deals with errors.

An IO list is an efficient data structure for producing sequences of binary data in Erlang and Elixir [12]. An IO list can contain: bytes, binaries and other IO lists. The typical use case of an IO list is when sending binary data over the wire. Instead of combining and copying binaries when encoding an HTTP message, IO lists are used to efficiently construct encoder outputs.

3.2.2 Decoder

A decoder does the opposite of an encoder and in the case of an HTTP library, the purpose of a decoder is to convert an in-transfer HTTP message in to an in-memory format. Similarly to as with the encoder built during the project, a set of decoder functions were implemented to do the actual decoding while a stateful interface was provided to glue the decoding functions together. Typespecs for each of these functions are given in Listing 3.2.

Listing 3.2: Typespecs for the implemented decoders. The types *Chunk.t*, *Message.trailer* and *Response.t* are given in Section A.1.

```
@doc "Decodes the status line and headers of an HTTP response."
@spec decode_head(binary) :: {:ok, Response.t, binary}
                           | {:ok, :more, binary}
                           | {:error, any, binary}

@doc "Decodes a chunk of an HTTP response."
@spec decode_chunk(binary) :: {:ok, Chunk.t, binary}
                            | {:ok, :more, binary}
                            | {:error, any, binary}

@doc "Decodes the body of an HTTP response."
@spec decode_body(binary, integer) :: {:ok, Chunk.t, binary}
                                    | {:ok, :more, binary}
                                    | {:error, any, binary}

@doc "Decodes trailers of an HTTP response."
@spec decode_tail(binary) :: {:ok, [Message.trailer], binary}
                           | {:ok, :more, binary}
                           | {:error, any, binary}
```

Three different decoders were implemented during the project: *Reference*, *FSM*, *Default*. A common layer was put in front of these decoders making it possible to share the *decode_body* function along with the stateful interface. Each of the decoders are described in the remaining of this section.

3.2.2.1 Reference

The *Reference* decoder was built with many small parsers that were combined using parser combinators based on the theory introduced in Section 2.1.2. Several libraries exist for working with parser combinators in Elixir but instead of using one of those, a custom library was built during the project. The reason was to get a better understanding of the relevant theory and mechanics. In the context of the implemented library for working with combinator parsing, a parser is just a function that takes a state and a potential number of arguments which are then used to compute a new state. The state contains a list of parsed results, the current status and the remaining input. Also, several combinators were built in order to tag and process results produced by prior parsers in the call chain.

The *Reference* decoder served as a reference implementation since the way the parser was written is strikingly similar to the way the HTTP protocol is defined by RFC

7230. Thus, the number of bugs introduced was expected to be small and motivates the choice of implementing one of the decoders using parser combinators. As an example of the similarities between the HTTP protocol definition and the corresponding parser built using parser combinators, the Augmented Backus-Naur Form (ABNF) definition of an HTTP message is given in Listing 3.3 while the parser can be seen in Listing 3.4.

Listing 3.3: ABNF for an HTTP message as defined by RFC 7230.

```
message = start-line *( header-field CRLF ) CRLF [ message-body ]
```

Listing 3.4: A function for decoding an HTTP message using parser combinators. This example matches the protocol specification in Listing 3.3.

```
def message() do
  Combinators.sequence([
    start_line(),
    Combinators.many0(Combinators.sequence([
          header_field(),
          crlf(),
        ])),
    crlf(),
    optional(message_body()),
  ])
end
```

3.2.2.2 FSM

The *FSM* decoder was implemented as a Finite-State Machine (FSM). The theory behind a finite-state machine is given in Section 2.1.3. The motivation for implementing one of the decoders as a finite-state machine was the success seen in the NGINX[4] and Node.js[5] projects. Two major benefits with the finite-state machine approach are simplicity and efficiency.

The implementation depends highly on Elixir's binary pattern matching[6]. An example of this is given in Listing 3.5 where the first three symbols of an HTTP response are parsed. As can be seen, the functions takes three arguments: input data, a list of parsed values and the current state. Input data are matched against values that start with either "H" or "T" in combination with the current state. The return value is a 4-tuple containing the status, the next state, a list of parsed values and the remaining input. The function given in Listing 3.6 controls the process of decoding by making the decoder progress or return. Another example of how the *FSM* decoder is implemented is given in 3.7 where a header name is parsed. This

[4]https://nginx.org/
[5]https://nodejs.org/
[6]https://elixir-lang.org/getting-started/binaries-strings-and-char-lists.html

time a single function is used to match every possible octet value with the exception of “:” which terminates the name according to the protocol specification.

Listing 3.5: Partial implementation of the *FSM* decoder showing how the first three symbols in an HTTP response are parsed.

```
defp decode_status_line("H" <> rest, acc, :start) do
  {:ok, :version_1, rest, acc}
end

defp decode_status_line("T" <> rest, acc, :version_1) do
  {:ok, :version_2, rest, acc}
end

defp decode_status_line("T" <> rest, acc, :version_2) do
  {:ok, :version_3, rest, acc}
end
```

Listing 3.6: The controlling function in the *FSM* decoder.

```
defp do_decode(data, acc, state, f) do
  case f.(data, acc, state) do
    {:ok, :done, rest, acc} ->
      {:ok, acc, rest}
    {:ok, state, rest, acc} ->
      do_decode(rest, acc, state, f)
    {:more, _state, _rest, _acc} ->
      {:ok, :more, data}
    {:error, state, _rest, _acc} ->
      {:error, state, data}
  end
end
```

Listing 3.7: Partial implementation of the *FSM* decoder showing how a header name in an HTTP response is parsed.

```
defp decode_header(":" <> rest, acc, :name) do
  {octets, acc} = Enum.split_with(acc, &is_binary/1)
  name = octets |> Enum.reverse |> Enum.join

  {:ok, :whitespace_left, rest, [{:name, name} | acc]}
end

defp decode_header(<<octet::bytes-size(1)>> <> rest, acc, :name) do
  {:ok, :name, rest, [octet | acc]}
end
```

3.2.2.3 Default

The main goal of the *Default* decoder was to reduce the amount of maintenance required during the lifetime of the decoder component. This is interesting from a more practical standpoint. The Erlang distribution provides a function called *decode_packet/3* in the *erlang* module which is able to parse the start line and headers of an HTTP message. The function was wrapped in order to fit the interface expected by decoders in the implemented library. The *decode_packet/3* also works with HTTP trailers since those are identical to headers in terms of syntax. Hence, the only function that was built from scratch was the one for decoding HTTP chunks. Since HTTP protocol uses newlines heavily, another Erlang function named *split/2* located in the *binary* module was used to split incoming data based on newlines. This allowed for a simple and efficient solution when parsing HTTP chunks. To illustrate the use of *binary:split/2*, a routine for partitioning data into lines is given in Listing 3.8.

Listing 3.8: An example of how the Erlang function *split/2* was used when decoding HTTP chunks in the *Default* decoder.

```
def decode_line(data) do
  case :binary.split(data, ["\n", "\r\n"]) do
    [_] ->
      {:ok, :more}
    [line, rest] ->
      {:ok, line, rest}
  end
end
```

3.2.3 Connection

In order to abstract the differences between the Transmission Control Protocol (TCP) and the Transport Layer Security (TLS) protocol, a component was implemented that layers the two protocols. The Erlang modules: *gen_tcp*[7] and *ssl*[8] allow for working with TCP and TLS respectively.

The *ssl* module leaves the process of peer validation to its users. The concept is introduced in Section 2.3.1.1 when describing the TLS Handshake Protocol. The plan was to work out a solution based on the Certifi[9] project which provides a collection of root certificates. Unfortunately, this was harder than expected, thus the implemented HTTP client library still lacks support for peer validation.

[7]http://erlang.org/doc/man/gen_tcp.html
[8]http://erlang.org/doc/man/ssl.html
[9]https://certifiio.readthedocs.io/en/latest/

3.2.4 Public API

Much effort was spent working on the public Application Programming Interface (API) of the implemented HTTP client library. In case the library gets inducted into the Elixir standard library[10] and people start using the implementation, any change is costly to make. Especially since it is aimed to be a low-level library that other projects will depend on.

In increasing the reliability of the public API, Elixir typespecs[11] was used which allow functions to be annotated with types. This is useful since Elixir is a dynamically typed language and by adding type annotations, static analysis can be applied. Guard clauses[12] were also used in order to increase reliability of the public API. In Elixir, guard clauses allow for more complex pattern matching.

Typespecs for the functions included in the public API is given in Listing 3.9.

[10]https://hexdocs.pm/elixir
[11]https://elixir-lang.org/getting-started/typespecs-and-behaviours.html
[12]https://elixir-lang.org/getting-started/case-cond-and-if.html

Listing 3.9: Typespecs for the public API. The types *Chunk.t*, *Connection.t*, *Message.header*, *Stream* and *value* are given in Section A.1.

```
@doc "Opens a new connection."
@spec connect(String.t, integer, transport, keyword)
  :: {:ok, Connection.t} | {:error, any}

@doc "Encodes and sends an HTTP request head."
@spec request(Connection.t, atom, String.t, [Message.header])
  :: {:ok, Stream.t} | {:error, any}

@doc """
Encodes and sends HTTP request body data.

Data can be a plain body, a chunk or nil. In the latter case, an
HTTP last chunk is sent over a connection to end the current
request. The acceptance of a given value depends on how a stream was
configured when calling ‘request/4‘.
"""
@spec request(Stream.t, binary | Chunk.t | nil)
  :: {:ok, Stream.t} | {:error, any}

@doc """
Receives and decodes values via a stream.

The option ‘block: false‘ can be applied in order to only call for
data once.
"""
@spec response(Stream.t, keyword)
  :: {:ok, [value], Stream.t} | {:error, any, Stream.t}

@doc "Decodes potential values in ‘data‘."
@spec decode(Stream.t, binary)
  :: {:ok, [value], Stream.t} | {:error, any, Stream.t}

@doc "Closes a connection."
@spec close(Connection.t) :: :ok | {:error, any}

@doc "Configures a connection’s socket for more data."
@spec next_packet(Connection.t) :: :ok | {:error, any}

@doc "Returns current encoder and decoder statuses."
@spec status(Stream.t) :: {atom, atom}
```

3.2.4.1 Demonstration

The purpose of this section is to show how a simple request is made with: *Net.HTTP*, *hackney* and *httpc*. See Listing 3.10 for how it works with *Net.HTTP*.

Listing 3.10: Example of doing a simple HTTP request with *Net.HTTP*.

```
host = "www.example.com"
port = 80
transport = :http
options = []

{:ok, connection} = Net.HTTP.connect(host, port, transport, options)

method = :get
path = "/some-path"
headers = []

{:ok, stream} = Net.HTTP.request(connection, method, path, headers)

options = []

{:ok, values, _stream} = Net.HTTP.response(stream, options)
```

Similarly, Listing 3.12 shows how it can be done with *hackney*. Note that the library has also support for a more high-level interface to do simple requests. It was chosen to do the low-level version to make it more comparable with Listing 3.10.

Listing 3.11: Example of doing a simple HTTP request with *hackney*.

```
Host = <<"www.example.com">>
Port = 80
Transport = hackney_tcp
Options = []

{ok, ConnRef} = hackney:connect(Transport, Host, Port, Options)

Method = get
Path = "/some-path"
Headers = []
Body = <<>>

{ok, _Status, _ResponseHeaders, ConnRef} =
  hackney:send_request(ConnRef, {Method, Path, Headers, Body})

{ok, ResponseBody} = hackney:body(ConnRef)
```

Listing 3.12 demonstrates how to do a simple request with *httpc*. Since connections and requests cannot be separated when using *httpc*, the example looks differently compared with the other two HTTP client libraries.

Listing 3.12: Example of doing a simple HTTP request with *httpc*.

```
Method = get
URL = 'http://www.example.com/'
Headers = []
HTTPOptions = []
Options = []

{ok, {_Status, _ResponseHeaders, Body}} =
  httpc:request(Method, {URL, Headers}, HTTPOptions, Options)
```

3.3 Evaluation

In determining the success of the implementation, the following evaluation criteria were used: correctness, extensibility, robustness and efficiency. The criteria are given in decreasing order of importance.

3.3.1 Correctness

Correctness of the implementation was tested with Quviq AB's QuickCheck which is a product for automatic test case generation. The tool is based on functional and random testing as described in Section 2.2. In QuickCheck, input data is generated by using a pseudorandom number generator (PRNG) controlled with a Domain-Specific Language (DSL) while oracles are defined by properties provided by the user. Both pure functions as well as stateful programs can be tested with QuickCheck. The implemented encoder and decoders were tested with stateless QuickCheck properties since the encode and decode functions operate without side effects. The method was insufficient when evaluating the public API of the implemented HTTP client library as the result of a function call depends on previous operations. Thus, the QuickCheck features related to stateful testing was used instead.

3.3.1.1 Decoder Testing

When using QuickCheck, it must be defined how test data should be generated. Since test data is produced randomly, the stochastic process has to be controlled. A large number of generators for fundamental data types are provided by the QuickCheck library, the common approach is to use these generators and then extend them when necessary. In order to test each of the decoders with QuickCheck, a number of generator functions were implemented that can generate values as described in the HTTP specification. The specification defines the syntax of the protocol using

ABNF as shown in Listing 3.3 in Section 3.2.2.1. A slightly simplified QuickCheck generator for HTTP messages can be seen in Listing 3.13.

Listing 3.13: A custom HTTP message generator. The helper functions: *sequence*, *many0*, and *optional* are defined in the listing given in Section A.2.

```
def http_message_gen() do
  sequence([
    start_line_gen(),
    many0(sequence([
          header_field_gen(),
          newline_gen(),
        ])),
    newline_gen(),
    optional(message_body_gen()),
  ])
end
```

QuickCheck also requires a property, or a test oracle as it is called in the context of random testing, to be defined before test cases can be generated. During evaluation, two types of properties were used: *encode-decode-compare* and *stream*. In the former case, a section of an in-memory HTTP message was generated, then encoded and decoded, and finally compared. By using this approach, both the encoder and a decoder were tested simultaneously. With the exception of request lines in the encoder since it is HTTP responses that are generated. In the full test suite, there were one *encode-deocde-compare* property per decoder and message part where a part was either: *head*, *chunk* or *tail*. An example of such a property is given in Listing 3.14, as can be seen, the *FSM* decoder is used to decode a randomly generated head part of an HTTP message.

Listing 3.14: An example of a *encode-decode-compare* property.

```
property "FSMDecoder: Message head" do
  forall head <- Random.Generators.head do
    {:ok, encoded} = encode(:head, head)
    {:ok, decoded, ""} = FSMDecoder.decode_head(encoded)

    ensure decoded == head
  end
end
```

The other type of property is *stream* which was used to test that the decoders can handle streaming properly. This includes both insufficient input and the gluing layers described in Section 3.2.1 and Section 3.2.2. The approach was to generate a full in-memory HTTP response, encode it, split the encoded output, decode the two parts and then compare the resulting response with the original response. The position where a split occur was selected randomly. Hence, a *stream* property is like an *encode-decode-compare* property but with a complete HTTP response and a split. One property per decoder was used during evaluation. The property that was used for the *FSM* decoder is given in Listing 3.15.

Listing 3.15: An example of a *stream* property. The functions *encode* and *decode* are locally defined in order to help the process of encoding and decoding respectively.

```
property "FSMDecoder: Stream parsing" do
  decoder = Protocol.decoder(:fsm)

  forall values <- Random.Generators.response do
    {:ok, message} = encode(Net.HTTP.Protocol.encoder, values, "")

    position = :rand.uniform(byte_size(message) + 1) - 1
    <<left::bytes-size(position)>> <> right = message

    {:ok, decoded_left, decoder} = decode(decoder, left)
    {:ok, decoded_right, _decoder} = decode(decoder, right)

    decoded = decoded_left ++ decoded_right

    ensure decoded == values
  end
end
```

3.3.1.2 Public API Testing

As stateless properties do not suffice when testing software with side effects in QuickCheck, an extended paradigm is required. Instead of putting an oracle, which is the component that determines the outcome of a test, in a property as shown in Listing 3.14, a finite-state model is used. A stateful property then becomes a matter of generating sequences of commands and determining the outcome of those sequences. A QuickCheck model has to know the success or failure of every step in a command sequence. Hence, the model serve as the real oracle when testing programs with side effects.

When testing the public API of the implemented HTTP client library, a finite-state model was designed that was used to test functions included in the API. With QuickCheck generating sequences of API function calls, a myriad of different sequences were tested. To make the testing environment more realistic, a popular HTTP server written in Erlang called *cowboy*[13] was setup to handle incoming requests. Connections are opened both with and without encryption when running the generated test cases. Along with every request made when testing the public API, a configuration structure was generated that described a certain set of features or options. The options used were: request body type, response body type, request method, persistent connections and a set of randomly generated headers. A body

[13]https://github.com/ninenines/cowboy

type can be either plain or chunked. This allowed the interactions between features to be tested easily. For example, one configuration could be: chunked request body, plain response body, the PUT method and a keep-alive connection.

3.3.2 Extensibility

A qualitative approach was taken when evaluating the extensibility of the implemented public API. Throughout the present report, the term extensibility refers to the ability to support many applications. When seeking extensibility, convenience is often sacrificed. This is an acceptable trade off since extensibility is preferred in the case of the implemented library.

Multiple abstractions were built on top of *Net.HTTP* as a way of ensuring extensibility which are all listed below:

- Processing HTTP messages using the Elixir Stream[14] module which operates on composable and lazy enumerables. This allows users to process HTTP messages using typical functional functions such as: *map*, *filter*, *zip* and *take*.
- A GenStage[15] producer. The GenStage library is an open source project that allow distributed processes in Elixir programs to exchange events with backpressure. By providing a producer on top of *Net.HTTP*, GenStage consumers can process incoming data in a distributed streaming fashion.
- Concurrent HTTP connections in a single Erlang process when using asynchronous receive. The authors were not able to reproduce this abstraction on top of either *hackney* or *httpc*.
- An extension of the library that allows for pipelining of HTTP requests. Even though *httpc* supports HTTP pipelining, a custom solution cannot be implemented on top of it. When trying to do the same thing on *hackney*, the outcome was unsuccessful.

3.3.3 Robustness

The term robustness was used as a measure on interoperability between the implemented HTTP client library and the Web. This is an important measure for the long-term success of the library. The way robustness was evaluated, was through interaction between *Net.HTTP* and a set of popular websites. More than that, *libcurl* was used as a reference implementation so that the behavior of *Net.HTTP* could be verified. When doing the robustness evaluation, the *Default* decoder was in use.

Roughly 20 years ago, Daniel Stenberg[16] started to work on the *cURL* project. The project consists of *curl* and *libcurl* which is a command line interface and a software

[14]https://hexdocs.pm/elixir/Stream.html
[15]http://elixir-lang.org/blog/2016/07/14/announcing-genstage/
[16]http://bookcurl.haxx.se/

library respectively for communicating over many different protocols via a computer network. With companies using the *cURL* project such as: Apple, Google, IBM, Intel, Spotify, Yahoo and VMware, it should be fair to say that *libcurl* can be used as a reference implementation when evaluating the robustness of *Net.HTTP*.

Unfortunately, the combination of time and priority did not allow for a well configured use of peer verification when communicating over TLS. Therefore, both of the HTTP client libraries were run without peer validation.

3.3.3.1 Data Collection

The Moz Top 500[17] is a list of the 500 most popular domains on the Web. Popularity is measured by the number of linking root domains from an index of 19 billion domains and 189 billion pages. Links were formed from these domains pointing to the root path of each domain. Each link was given as input to *Net.HTTP* and *libcurl*. The response of a request was stored in *PostgreSQL* where status code, headers, body and a flag denoting completion were recorded. For each client library, a driver had to be built. The purpose of such a driver was to control respective clients and writing information to the database.

3.3.3.2 Data Analysis

Each response or response pair was analyzed using the following measures:

- The term *completion* of a request was used to denote whether the associated response was received and decoded without any errors.
- In determining the *success* of a request, the returned status code was expected to be 200 OK.
- Response headers associated with each link were validated by a case insensitive comparison between the set of header names obtained with *Net.HTTP* and *libcurl* respectively. Only header names were compared since a header values may change between requests.
- When validating the response body for each link, the length of the bodies given by the two clients were expected to be within a 10 % difference. As with headers, there may be unique content in each response received for a single URL.

3.3.4 Performance

When evaluating the performance of implemented HTTP client library, the following two questions were asked:

- How do the different decoders perform? The relevant decoder implementations are: *Reference*, *FSM* and *Default*. The first decoder was built using parser

[17]https://moz.com/top500/

combinators and has served as a reference implementation. The second decoder was implemented as a finite state machine. The third and final decoder was based on functionality included in the Erlang distribution.

- What is the performance of the implemented client in comparison with other HTTP clients running on the Erlang platform? The chosen clients to compare with were: *httpc* and *hackney*. The former client is bundled with the Erlang distribution while the later one is a popular open source project.

3.3.4.1 Data Collection

When evaluating the performance of the implemented decoders, generated responses with different characteristics were used as inputs to the decoders. The idea was to vary the header value size, the number of chunks and the total response size. The number of headers and the header name size was chosen empirically from data obtained when doing the robustness evaluation described in Section 3.3.3. There was only one step required for each decoder and website:

- Decode given response with the current decoder 10 000 times.

In the case of evaluating performance of the HTTP clients, a local web server was used to serve responses similarly to those used when evaluating the performance of the decoders. The reason for having a local web server was to minimize the amount of time spent outside of the running client. In order to make the comparison more even, it was ensured that the clients sent identical sets of request headers. This time the *Default* decoder was used when evaluating the performance of *Net.HTTP*. Each client were evaluated as listed below:

- Start web server.
- Make 10 000 requests.
- Stop web server.

3.3.4.2 Data Analysis

Simple statistics was used to analyze collected data. The arithmetic mean was used to average obtained samples as a measure of the central tendency in the current distribution. As a measure of error, the standard deviation was utilized.

4

Result

The purpose of the present chapter is to give results collected when evaluating the implemented Hypertext Transfer Protocol (HTTP) client library during the course of the current thesis project. Evaluation regarding: correctness, extensibility, robustness and performance are given in Section 4.1, Section 4.2, Section 4.3 and Section 4.4 respectively.

4.1 Correctness

When evaluating the correctness of the HTTP client library, the problem was divided into two parts: testing of protocol encoding and decoding along with testing of the public Application Programming Interface (API) of the library. As in Section 3.3.1, the former is all about the syntax of the HTTP protocol while the latter handles behavior and protocol semantics.

Test coverage of the implemented HTTP client library obtained while doing the correctness evaluation is given in Table 4.1. The term relevant lines is used to denote lines that were monitored during execution by the coverage tool *ExCoveralls*[1].

Table 4.1: Test coverage of major components of *Net.HTTP*.

Component	**Coverage (%)**	**Relevant Lines**
Public API	60.8	51
Encoder	96.2	52
Decoder	96.3	27
Reference	98.5	65
FSM	89.9	109
Default	92.3	65
Connection	88.5	26
TCP	80.0	5
TLS	88.9	9

[1]https://github.com/parroty/excoveralls

4.1.1 Decoder Testing

When evaluating the correctness of the encoder and the decoders, QuickCheck was used to produce test cases by combining randomly generated input data and the properties described in Section 3.3.1.1. For example, when testing the *FSM* decoder with HTTP heads, roughly 20 000 test cases were generated and tested in 60 seconds.

A sample of a generated HTTP head is given in Listing 4.1. In order to get a better idea of what kind of input data that was generated, an extract with statistics is given in Table 4.2. The selected statistics are: HTTP status codes, number of headers, along with sizes in octets of header names and header values, and their hexadecimal octet values.

Listing 4.1: A sample of an HTTP response head generated with QuickCheck. Non ASCII-values were omitted in the generation for clarity sake.

```
HTTP/1.1 505 HTTP Version Not Supported\r\n
_: IS\t  \tX]\r\n
r: \r\n\t\t  ~+K  \t&\r\n
L6D: \r\n\t\tS\tVx\r\n
nD~8: F 5W\t U!:   x-\r\n
5|#L: \r\n\t  N\r\n
\r\n
```

Table 4.2: A sample obtained when using QuickCheck to randomly generate 1 000 HTTP message heads. Proportions, which are given as percentages, and actual values are given for each class. Note that only the top 15 entries are listed and that *N.* and *V.* are used to denote header name and header value respectively.

Status		**Headers**		**N. Size**		**N. Char**		**V. Size**		**V. Char**	
2.68	307	16.77	0	11.22	1	3.36	39	9.96	0	35.96	20
2.67	403	14.59	1	10.73	2	3.36	30	3.87	1	35.93	9
2.56	410	12.04	2	10.06	3	3.35	36	1.86	11	3.52	D
2.55	300	9.88	3	9.48	4	3.35	34	1.83	8	3.52	A
2.54	503	8.06	4	8.60	5	3.34	37	1.82	6	0.12	5F
2.54	409	6.96	5	8.10	6	3.33	32	1.81	5	0.12	72
2.54	303	6.06	6	7.31	7	3.32	35	1.80	7	0.12	32
2.52	501	5.09	7	6.67	8	3.32	38	1.79	9	0.12	3E
2.52	204	4.51	8	5.98	9	3.30	33	1.78	10	0.12	2D
2.50	505	3.85	9	5.28	10	3.28	31	1.75	12	0.12	31
2.50	408	3.31	10	4.55	11	2.26	2A	1.74	13	0.12	63
2.49	504	2.48	11	3.75	12	2.25	2D	1.72	4	0.11	56
2.49	402	2.14	12	3.08	13	2.25	21	1.66	15	0.11	3B
2.49	414	1.56	13	2.36	14	2.23	7C	1.65	16	0.11	52
2.48	502	1.28	14	1.61	15	2.23	24	1.64	14	0.11	22

Similarly, a sample of an HTTP chunk is given in Listing 4.2 with statistics in Table

4.3. This time the statistics are: chunk data size in octets, number of extensions along with extension name and value sizes also in octets and as well as their octet values.

Listing 4.2: A sample of an HTTP response chunk generated with QuickCheck.

```
3;M|g;z4;0hE|W="\§";py="";hs4K9+=_1|;~4j\r\n
tCf\r\n
```

Table 4.3: A sample obtained when using QuickCheck to randomly generate 1 000 HTTP message chunks. Proportions, which are given as percentages, and actual values are given for each class. Note that only the top 15 entries are listed and that *N.* and *V.* are used to denote extension name and extension value respectively.

Data Size		**Extensions**		**N. Size**		**N. Char**		**V. Size**		**V. Char**	
50.12	0	16.91	0	11.27	1	3.34	38	50.06	0	16.96	5C
10.73	1	14.80	1	10.79	2	3.34	36	5.17	2	12.14	22
7.52	2	11.70	2	10.11	3	3.34	34	4.16	4	7.07	9
5.71	3	9.82	3	9.37	4	3.33	39	3.76	3	7.06	20
4.77	4	8.13	4	8.68	5	3.33	30	3.68	5	3.66	21
3.91	5	7.23	5	8.13	6	3.33	37	3.58	6	1.35	31
3.29	6	5.89	6	7.39	7	3.33	31	3.27	7	1.35	39
2.74	7	5.14	7	6.58	8	3.33	35	3.12	8	1.34	32
2.45	8	4.33	8	5.87	9	3.32	32	2.76	9	1.34	35
2.08	9	3.80	9	5.21	10	3.31	33	2.76	1	1.33	34
1.71	10	3.14	10	4.52	11	2.24	26	2.56	10	1.33	38
1.48	11	2.66	11	3.80	12	2.24	25	2.31	11	1.32	37
1.06	12	2.13	12	3.07	13	2.24	7E	2.10	12	1.32	36
1.01	13	1.66	13	2.35	14	2.23	7C	1.80	13	1.32	30
0.69	14	1.23	14	1.69	15	2.23	5E	1.58	14	1.32	33

4.1.2 Client Testing

In the case of the public API of the implemented HTTP client library, sequences of commands were generated with QuickCheck and instead of having properties serving as oracles, a state model was used. See Table 4.4 for a set of statistics collected when running QuickCheck for 60 seconds. The statistics are: the length of a command sequence, command signatures and features. The last term is used in QuickCheck when testing stateful programs when one wants to record a certain state. For example, one of the entries in Table 4.4 and under *Feature* is called “Connect (TCP) - OK” which represents a successful call to the connect function with the intent of operating over TCP. On the other hand, the features ending with “Error” denote bad calls meaning that both positive and negative testing were done.

Table 4.4: A sample obtained when using QuickCheck to randomly generate 1 000 sequences of public API calls. Proportions, which are given as percentages, and actual values are given for each class. Note that only the top 15 entries are listed and that *S.* is used to denote command sequence.

S. Length		**Command**		**Feature**	
6.60	3	21.82	Request Chunk	13.52	Request Body - Error
6.40	2	20.75	Connect	13.51	Request Chunk - OK
5.20	7	20.28	Request Body	12.62	Request Head - OK
5.10	5	12.62	Request Head	10.60	Connect (TCP) - OK
4.70	1	10.98	Response	8.32	Request Chunk - Error
4.50	6	10.89	Request Last Chunk	8.27	Request Last Chunk - Error
4.50	4	2.65	Close	8.26	Connect (TLS) - OK
4.50	0			6.77	Request Body - OK
4.20	8			6.04	Response - OK
3.60	9			4.95	Response - Error
3.40	14			2.65	Close - OK
3.30	13			2.62	Request Last Chunk - OK
3.10	11			1.89	Connect - Error
3.00	10				
2.80	16				

4.2 Extensibility

As described in Section 3.3.2, a qualitative approach was taken when evaluating the extensibility of the public API of the implemented HTTP client library. The resulting example abstractions built on top of the API are shown in this section. The examples given are the following: *Stream*, *GenStage*, *Concurrent Client* and *Pipelining Client* with full implementations listed in: Section A.3.1, Section A.3.2, Section A.3.3 and Section A.3.4 respectively.

4.2.1 Stream

In the first example, *Net.HTTP* is combined with the Elixir *Stream* module. This can be achieved by using the function *resource/3*, which is included in the *Stream* module, in order to create a usable stream. The function takes three higher-order functions where the first one is for setting up a new stream, the second function is expected to generate elements while the third function does potential tear down of obtained resources. The implementations of these three functions are given in Listing 4.3. The first function makes a connection to the URL represented by the given URI[2] structure and then sends a request over the connection. The second function calls the implemented HTTP library in a way that makes the program synchronously wait for data and then emits received values as soon as there are any

[2]https://hexdocs.pm/elixir/URI.html

available. When a complete response has been received or in the case of an error, the stream is halted. The third and final function closes the connection. Since the first function expects no arguments, a so called closure is applied as a method for handling parameters. This technique is also used in the remaining functions for consistency.

Listing 4.3: Implemented functions for use with Elixir's *Stream* module.

```
defp start_fun(url) when is_map(url) do
  host = url.host
  port = url.port
  transport = String.to_atom(url.scheme)

  path = url.path

  fn ->
    with {:ok, conn} <- Net.HTTP.connect(host, port, transport),
         {:ok, stream} <- Net.HTTP.request(conn, :get, path) do
      stream
    else
      {:error, reason} ->
        raise StreamError, reason: reason
    end
  end
end

defp next_fun() do
  fn stream ->
    case Net.HTTP.response(stream, block: false) do
      {:ok, values, stream} when is_list(values) ->
        {values, stream}
      _otherwise ->
        {:halt, stream}
    end
  end
end

defp after_fun() do
  fn stream ->
    Net.HTTP.close(stream.connection)
  end
end
```

4.2.2 GenStage

In the second example, a library called *GenStage* is used which allow Elixir processes to exchange events with back-pressure which prevents any process from being overloaded. This is useful when processing large amount of data at a high throughput. The purpose of the present example is to show how the implemented HTTP library can be combined with *GenStage* to process incoming tweets from Twitter. In *GenStage* there are these concepts of producers and consumers where the former actor provide data or events and consumers connect to a producer and then processes received events. Producers and consumers run in their own Elixir process and thus they can run concurrently on a single machine but also distributed over multiple machines.

Since data may arrive from Twitter at any point during an indefinite time period, the implemented HTTP client library was configured in asynchronous mode meaning that socket data is delivered as Elixir messages. Handling of socket messages is shown in Listing 4.4 where both TCP and TLS data are matched. When data is available, the function *on_data/2*, which is shown in Listing 4.5, is called. There, the received data is decoded and sent to the *dispatch/3* function. As can be seen in the definition of the *on_data/2* function, *next_packet/1* from the implemented library is called. The purpose is to tell the socket that the user is ready for more data, this is another measure to prevent overloading. When received data is dispatched, it is delivered to a downstream consumer.

Listing 4.4: Shows how incoming socket data is matched in order for the current consumer to process and dispatch it to downstream consumers.

```elixir
def handle_info({:tcp, _socket, data}, state) do
  on_data(data, state)
end

def handle_info({:ssl, _socket, data}, state) do
  on_data(data, state)
end
```

Listing 4.5: Exemplifies how the implemented HTTP library can be used in asynchronous mode where data is already available before calling any of the response functions in the *Net.HTTP* module.

```elixir
defp on_data(data, state = {stream, demand, buffer}) do
  case Net.HTTP.decode(stream, data) do
    {:ok, values, stream} ->
      Net.HTTP.next_packet(stream.connection)
      dispatch(stream, demand, buffer ++ values)
    {:error, reason, _stream} ->
      {:stop, reason, state}
  end
end
```

4.2.3 Concurrent Client

The purpose of the *Concurrent Client* abstraction is to give a working example on how the implemented library can be used to concurrently handle multiple responses asynchronously in a single Elixir process. The heart of the *Concurrent Client* is its loop function which waits for incoming messages. See Listing 4.6 for the function definition. All but the first message are related to TCP or TLS communication. When data comes in on an open socket, it is sent as an Elixir message to the current instance of the *Concurrent Client*. For simplicity sake, error or socket closed events lead to termination of the client process. To make an HTTP request with the *Concurrent Client*, a message has to be sent with a signature that matches the first clause listed in the receive block of the loop function.

The function *on_process/3*, which is called when a *process* event is sent to a *Concurrent Client*, does a simple HTTP request to the given URL and modifies the state to include the newly acquired stream structure. When *on_process/3* is called, a request is sent the same way as it is done in the *Stream* example as seen in Listing 4.3 and therefore that functionality is omitted here. The function *on_data/3* is given in Listing 4.7 which decodes received response data. On line 52, the matching socket is retrieved from the client state. Another interesting line is number 60 where the socket is made prepared for a new round of data.

Listing 4.6: The main loop of the *Concurrent Client* handling incoming messages such as user requests and socket data.

```elixir
def loop(state) do
  receive do
    {:process, url, sender} ->
      loop(on_process(state, url, sender))
    {:tcp, socket, data} ->
      loop(on_data(state, socket, data))
    {:tcp_closed, _socket} ->
      {:error, :closed}
    {:tcp_error, _socket, reason} ->
      {:error, reason}
    {:ssl, socket, data} ->
      loop(on_data(state, socket, data))
    {:ssl_closed, _socket} ->
      {:error, :closed}
    {:ssl_error, _socket, reason} ->
      {:error, reason}
  end
end
```

Listing 4.7: The function processing incoming response data in the *Concurrent Client* similarly to the one given in Listing 4.5.

```elixir
def on_data(state, socket, data) do
  stream = state[socket]

  case Net.HTTP.decode(stream, data) do
    {:ok, _values, stream} ->
      case Net.HTTP.status(stream) do
        {_encoder_status, :done} ->
          clear_stream(state, stream)
        _otherwise ->
          :ok = Net.HTTP.next_packet(stream.connection)
          store_stream(state, stream)
      end
    {:error, _reason, _stream} ->
      state
  end
end
```

4.2.4 Pipelining Client

In the *Pipelining Client* example, it is shown that the implemented HTTP library can be extended in order to support pipelining of requests. The idea is to drive the client library from a separate process via Elixir message passing. When a *request* message comes into the process, an HTTP request is sent to the current HTTP server. The main loop of the *Pipelining Client* is given in Listing 4.8 where two types of messages are handled: *request* and *response*. The user is expected to send a series of *request* messages in order to pipeline a set of HTTP requests. The latter message type *response* should be sent to the *Pipelining Client* as soon as the user has no more requests to send and is ready to receive the associated responses.

The logic responsible for receiving is given in Listing 4.9. As can be seen, each *stream* structure, which controls a HTTP request-response pair, is processed one by one. The reason for why it is possible to pipeline requests without much effort using the implemented client library lies in the separation between a connection and a *stream*.

Listing 4.8: The main loop of the *Pipelining Client* handling user calls.

```
def loop({conn, queue}) do
  receive do
    {:request, path, sender} ->
      case Net.HTTP.request(conn, :get, path, []) do
        {:ok, stream} ->
          send(sender, {:ok, self()})
          loop({conn, [stream | queue]})
        {:error, reason} ->
          send(sender, {:error, reason, self()})
      end
    {:response, sender} ->
      case on_response(Enum.reverse(queue), []) do
        {:ok, responses} ->
          send(sender, {:ok, responses, self()})
          loop({conn, []})
        {:error, reason} ->
          send(sender, {:error, reason, self()})
      end
  end
end
```

Listing 4.9: Function responsible for receiving HTTP responses in the correct order and in a synchronous fashion as part of the *Pipelining Client*.

```
defp on_response([], responses) do
  {:ok, Enum.reverse(responses)}
end

defp on_response([stream | streams], responses) do
  case Net.HTTP.response(stream) do
    {:ok, values, _stream} ->
      on_response(streams, [values | responses])
    {:error, reason, _stream} ->
      {:error, reason}
  end
end
```

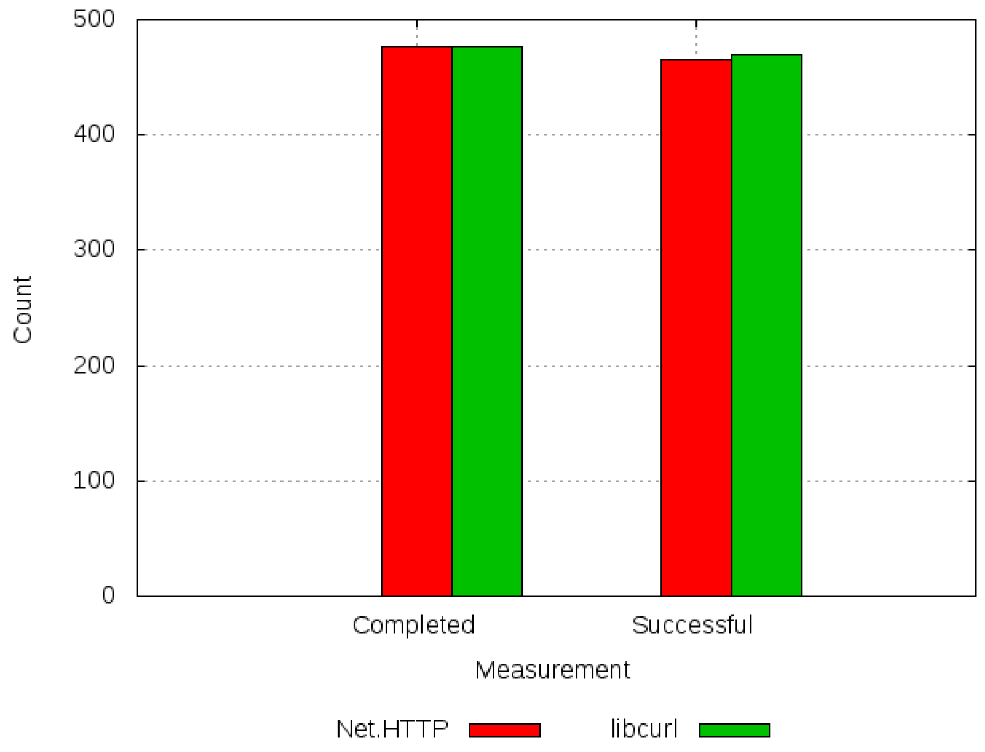
4.3 Robustness

In the process of evaluating the robustness of the implemented HTTP client library, data was collected from running the library as well as *libcurl* against 500 different URLs on the the Web. Four different measures were used when interpreting the data as described in Section 3.3.3. Results from the first two could be computed directly from a single request while in the other group, request-pairs were compared. One request from *Net.HTTP* and one request from *libcurl* to the same URL is said to form a request-pair. A request-pair was only considered if both responses returned status code 200.

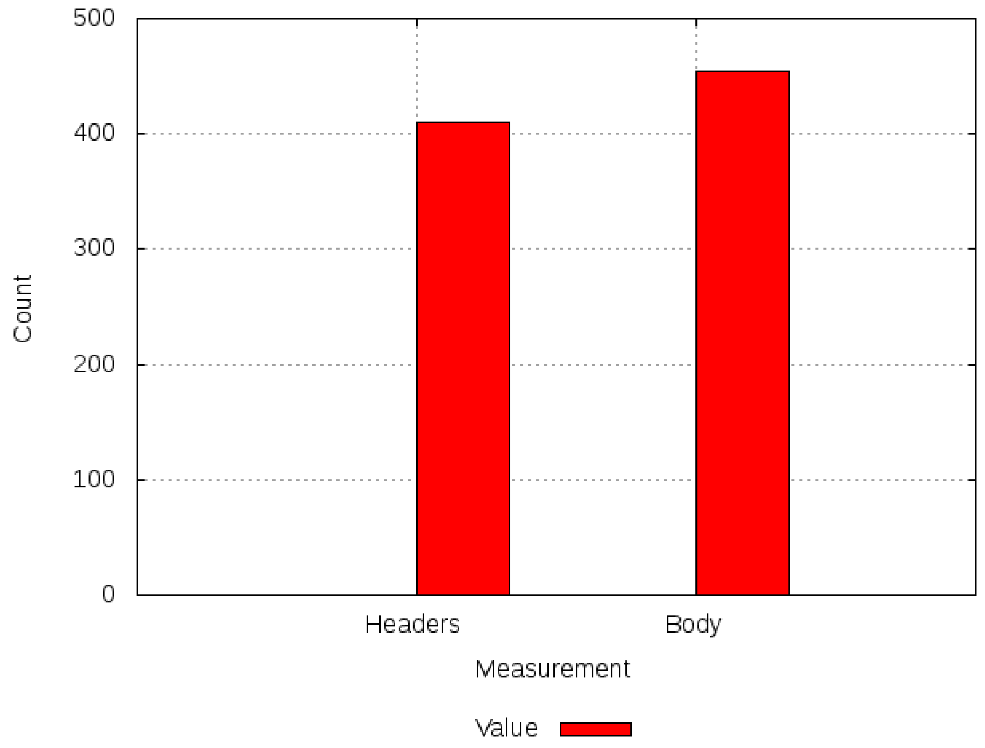
In the first group, data was obtained on the basis of a single request. A request is referred to as *completed* in case the calling procedure terminated without any errors while a request is said to be *successful* if the HTTP status code 200 is returned. The fraction of *completed* and *successful* requests per client library is given in Figure 4.1a.

In the second group, the metrics: *headers* and *body* were computed by comparing responses received from a request-pair. Since HTTP header field names are case-insensitive and because header field values may differ from response to response, lowercase versions of the two header name sets from each request-pair were compared. Hence, whenever the set of lowercase header field names from one response is identical with those from another response for the same URL, the header section is assumed to be successful. When comparing response bodies, the lengths of received bodies were compared and expected to be within a 10 % difference. See Figure 4.1b for the results from the *headers* and *body* comparisons.

From the set of URLs that were processed unsuccessfully, only two of them were due to problems with the protocol syntax when using the implemented HTTP client library with the *Default* decoder. The first problematic response was missing the newline terminating the head section while the other response had an invalid chunk.



(a) Comparison between 500 requests per HTTP client library.



(b) Comparison between 461 successful request-pairs.

Figure 4.1: Comparison between *libcurl* and *Net.HTTP* from sending simple requests to 500 different URLs from the Moz Top 500 index.

4.4 Performance

As part of evaluating the performance of the implemented HTTP client, two different benchmarks were made with focus on HTTP decoding and doing simple requests against a local web server respectively.

4.4.1 Decoder Benchmark

The performance of the implemented decoders was evaluated by feeding the decoders with several generated responses. The characteristics for each response that were used are outlined in Table 4.5. The header value size, number of chunks and total size were varied while the status line, number of headers, header name size and a header for signaling chunked encoding were kept constant. The number of headers was 15 and the header name size was 10 bytes for each response. Each website was decoded 10 000 times per decoder and configuration during the benchmark.

Table 4.5: Defining characteristics for each configuration used during decoder benchmark. Size is measured in bytes where a byte is equal to an octet.

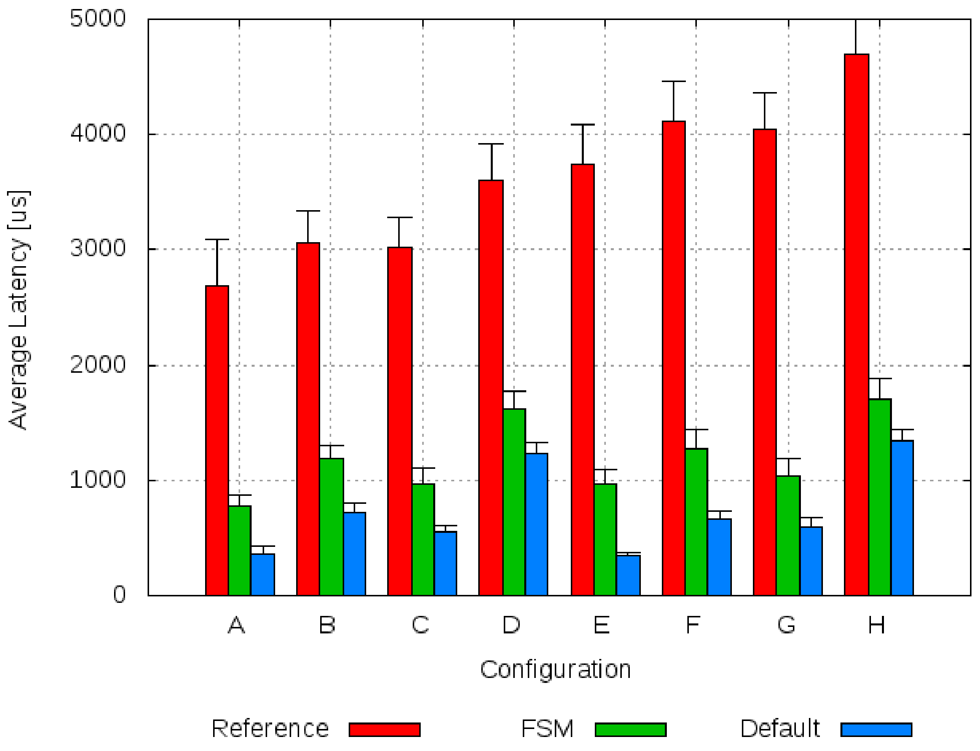
Configuration	**Header Value Size**	**Chunks**	**Chunk Size**	**Total Size**
A	30	4	125 000	500 743
B	30	4	250 000	1 000 743
C	30	8	62 500	500 771
D	30	8	125 000	1 000 779
E	60	4	125 000	501 193
F	60	4	250 000	1 001 193
G	60	8	62 500	501 221
H	60	8	125 000	1 001 229

Average latency and average throughput obtained from the benchmark are given in Figure 4.2a and Figure 4.2b respectively.

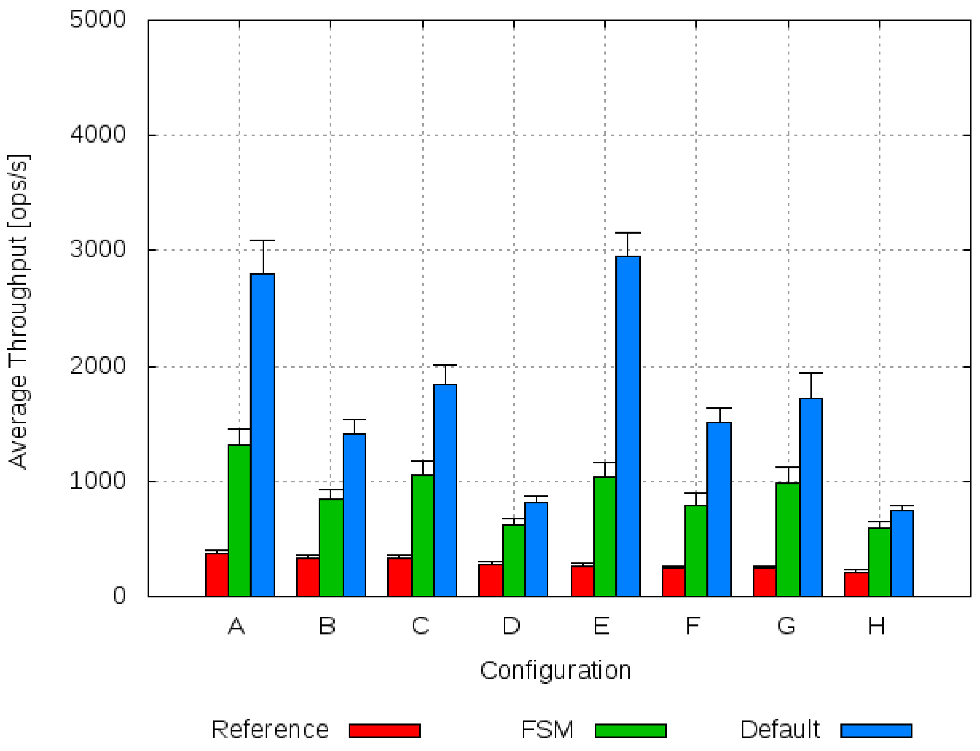
4.4.2 Client Benchmark

Three clients that run on the Erlang virtual machine were evaluated with respect to performance in the second benchmark. The relevant clients are: *Net.HTTP*, *hackney*, *httpc* where the first client is the implemented during the project, the second client a popular open source client and the third client is included in the Erlang distribution. As with the decoder benchmark, the configurations given in Table 4.5 were used. For each configuration and decoder, 10 000 requests were sent.

Average latency and average throughput obtained from the benchmark are given in Figure 4.3a and Figure 4.3b respectively.

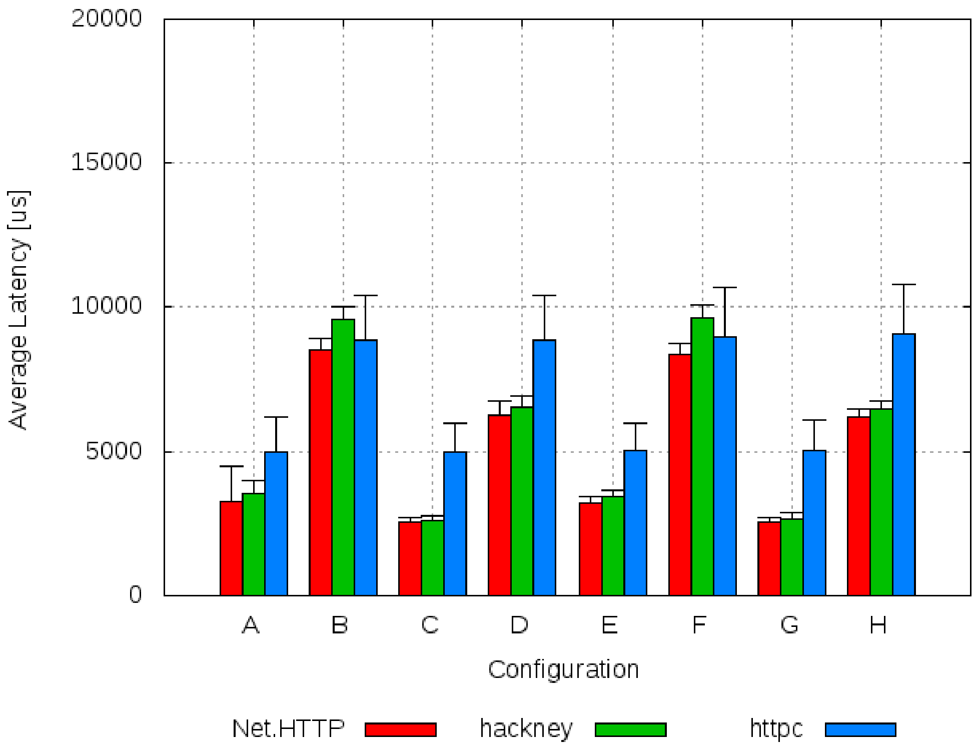


(a) Latency

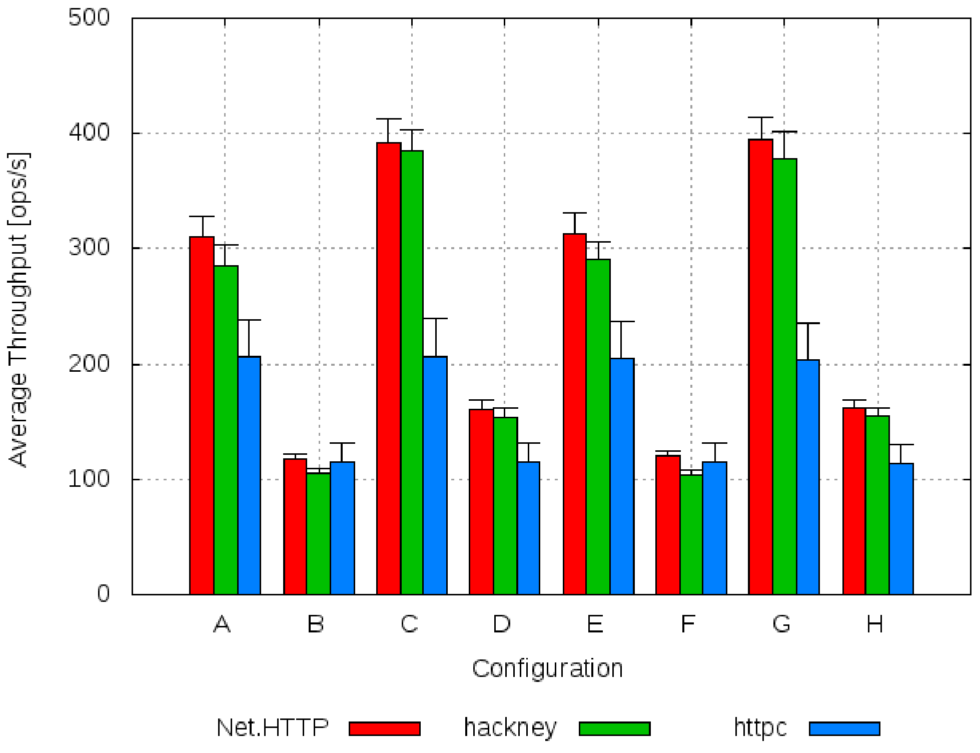


(b) Throughput

Figure 4.2: Performance benchmark of the different decoders when processing various responses. Defining characteristics for each configuration is listed in Table 4.5. The statistics were obtained from 10 000 samples while the standard deviation was used as an error measure.



(a) Latency



(b) Throughput

Figure 4.3: Performance benchmark of three HTTP clients running on the Erlang virtual machine. The implemented client used the default decoder during this benchmark. The statistics were obtained from 10 000 samples while the standard deviation was used as an error measure.

5
Discussion

The current chapter discusses the results presented in Chapter 4 and tries to connect those results and lessons learned during the project with the questions given in the problem statement but first off is a discussion on the conducted landscape survey.

5.1 Landscape Survey

The purpose of the present section is to discuss the survey that was done during the project regarding the *http-client* and *clj-http* Hypertext Transfer Protocol (HTTP) client libraries. More specifically, to answer Question 4 given in Section 1.2.

5.1.1 Haskell

In order to understand how *http-client* can be used when sending and receiving an HTTP message body, a key prerequisite is an understanding of Haskell's lazy evaluation. The evaluation strategy means that no expression is computed more than necessary. Lazy evaluation in a functional setting allows for greater modularity and the concept of lazy IO [13]. The first example on what can be done with *http-client* is about lazy IO. Imagine a file that needs to be uploaded to an HTTP server but is too large to fit in memory, in Haskell it is possible to implement the required functionality as if the whole file contents is to be read at once while in reality, it is streamed. Note, that for this method to work seamlessly, the size of the file must be known beforehand since otherwise Haskell would need to read the whole file right away. Doing something like this is not possible in Elixir since the language is strictly evaluated. When using *Net.HTTP*, the expected approach is to read the file piece by piece and send them one at a time. Even though both of these libraries are meant to be low-level, Haskell's lazy evaluation brings high-level composition.

A similarity between *http-client* and *Net.HTTP* is how both of their respective languages support multiple behaviors with a single function identifier. For example, when sending a request body with *http-client*, the library adapts to what kind of request body is given. In the case of Elixir, the *request* function behaves differently depending on the number of arguments and their values. With this approach compact interfaces can be constructed. Though we find it even more interesting in the

Haskell case due its static types.

Another similarity is how the two libraries handle response bodies. Both of their public Application Programming Interfaces (API) provide functions for reading and decoding data synchronously. It is possible to read one chunk at a time or the whole response body at once. In the case of *Net.HTTP*, a response structure containing status code, protocol version and headers is given as the first value in a list of items when calling for data, the remainder of the list may be chunks and trailers. The way it is done in *http-client* is compelling since there, a response data structure contains everything including one of the fields that represents the body. Since Haskell supports parameterized data types, this field can be either data in transit or a complete body. It might be possible to imitate this approach in Elixir with the help of pattern matching though we assume it will be more complex without a static type system.

5.1.2 Clojure

It seems that *clj-http* is much more high-level than *Net.HTTP* when looking at the features provided. Even though the feature sets differ between the HTTP client libraries, both of these projects support multiple methods of dealing with requests and responses. The Clojure library achieves this through polymorphism via the use of multimethods[1]. The way a request body or a response body is streamed is by passing an instance of the *InputStream* or *OutputStream* classes respectively. Note that the classes come from Java and is an example of how Clojure developers can reuse much from the Java world which we think is comparable to what can be done in Elixir with Erlang functionality. If it is more relevant to use another method for sending or receiving an HTTP message body with *clj-http*, it can be easily configured.

As we have seen, all three HTTP client libraries try to reduce required API surface by using either function overloading or a algebraic data type when it comes to sending or receiving requests and responses respectively. This is a compelling choice since it makes each API more compact.

5.2 Correctness

The purpose of this section is to carry out a discussion on the correctness evaluation that was done as part of the project. The discussion starts with the syntax aspect and then ends with protocol behavior and the public API. The answer to Question 2 in Section 1.2 is yes, we believe the work that has been carried out during this project testifies about the suitability of QuviQ QuickCheck in the context of building a client library for a popular Internet protocol.

[1]https://clojure.org/reference/multimethods

5.2.1 Decoder Evaluation

The present section discusses positive and negative testing related to the encoder and decoder components as part of the implemented HTTP client library.

5.2.1.1 Positive Testing

When thinking about building an HTTP decoder, one can see that the set of valid HTTP messages is a large multidimensional space. A common approach when testing computer programs is to split the input space into partitions where each value in a partition is assumed to be equally treated by the software under test. When writing test cases manually, it is hard to cover each partition even once. Instead, with well implemented QuickCheck generators, the tool can automatically cover much of the input space. Fortunately, the present project can exemplify this, as can be seen in Table 4.1, almost all lines in the encoder and decoders are covered when testing with QuickCheck. The lines that miss coverage are due to the absence of negative testing.

With the *Reference* decoder, it would probably suffice with traditional unit tests since it was easily implemented with a custom Domain-Specific Language (DSL). It is another matter when it comes to the *FSM* and *Default* decoders since they are more complex. In our opinion, QuickCheck has been an invaluable tool in the context of testing a functional HTTP client library.

We should also give credit to the problem at hand. The properties that were used when testing the syntax, all boils down to the following steps: encode, decode and compare. This is sometimes said to be a strong property. We think it is true, even though it only verifies that the an encoder and a decoder work well together and not that they follow the actual specification. We experienced this while working on the syntax components where the test suite reported OK and a bug appeared during live testing. A solution could be to have two different programmers implementing the QuickCheck generators and another working on the software under test.

As can be seen in Table 4.2, the distributions of status codes and header name values are fairly uniform while the distributions for the number of headers and header name size are skewed towards small values. The reason for keeping the latter two distributions low is due to efficiency. Still, more and larger headers are generated during testing. The reason for why header value sizes are not uniform is the more complex grammar behind header values. An example of the complexity of header values can be seen in the final column where generated octet values are listed. An HTTP header value can be folded into separate lines which makes the decoders more complex and as can be seen in the distribution, such values are generated.

Similarly, in Table 4.3, column 1, 2, 3 and 5 have skewed distributions towards small values and the distribution for chunk names is uniform. Decoding HTTP chunk extension values can be tricky since there are two mechanisms for escaping special symbols. Multiple characters can be escaped with quotes while a single character can be escaped with backslash. As can be seen in the table, both quotes and backslash are the most commonly seen octets when generating chunk extension values which tells that quoting has actually been tested.

5.2.1.2 Negative Testing

One thing that might have come to mind is the lack of negative testing when evaluating the correctness of the implemented decoders. The reason is that we think it reduces robustness when any slightly invalid HTTP message is rejected. Hence, between the two, we choose robustness. Note that it is not black and white, from the ways the decoders are implemented, a message with major flaws will be discarded. Also, we see two methods for doing negative testing on the decoders with QuickCheck, which are described below, but unfortunately, both of them have their drawbacks. In both cases, a valid HTTP message is assumed to be generated and then a single byte is modified at a random position in the message.

- Feed the generated input message to each of the implemented decoders and then fail in case any of the *FSM* or *Default* decoders do not match with the result of the *Reference* decoder.
- Make the choice of a replacement byte intelligent and select only a value from a set that is disjoint with the set of allowed values found at the chosen position in the given message.

The problem with the first method is that it is expected to be time consuming during testing since there is a great chance of modifying the selected byte in a valid way. On the other hand, the second approach should be time consuming when implementing.

5.2.2 Client Evaluation

It is well known that the number of tests required for a piece of software grows exponentially with the number of features provided. QuickCheck solves this in terms of developer time with its ability to generate both input data and commands randomly. For example, when adding a function to the public interface, one can just extend the QuickCheck model with support for the newly added function and it will be included in command sequences randomly generated in the future. Hence, if QuickCheck is run long enough, the new function will be combined with existing functions in the public API. Another example is the many different behaviors a request-response pair can take. With traditional unit tests, a test case involving: plain request body, chunked response body, closed connection and method has to be manually specified. Then, if an extra variable is added to signal compression, several additional test cases must be written. With QuickCheck, it is instead a matter of extending the model with support for the new variable. Thus, the exponential time required by a programmer may be reduced to something that is linear.

When working with QuickCheck state machine models, a model must know exactly everything that is going on. This limitation appeared when deciding to use the *cowboy* HTTP server. We believe it is too fragile if not impossible to know and control how the server sends data over a socket. By not having control over data transmission from *cowboy*, we were not able to test asynchronous receive or streaming synchronous receive with QuickCheck. This explains why the test coverage shown in Table 4.1 is fairly low.

Table 4.4 shows distributions for command sequence lengths, commands and features. As with most sequences in QuickCheck, command sequences tend to skew against short lengths. The commands distribution can easily be adjusted in QuickCheck and as can be seen in the table, the most frequent commands are about establishing connections or sending requests. In the final column, features distribution is shown. Features is a concept in QuickCheck which allow users to signal interesting calls or command sequences. Furthermore, it is a useful way to do both positive and negative testing in the same model. It is desirable to have the feature “Request Chunk - OK” high since then multiple chunks are more probable to be sent in the same request.

5.3 Extensibility

Ideally, every possible use case of the implemented HTTP client library should have been evaluated but this approach is not really feasible due to time and the unknown of potential applications of the library. Instead four examples were selected that we think covers many use cases. It might seem counter intuitive to go the distance of building a new HTTP client library with the goal of giving full control over state and process management for a run-time system where cheap concurrency is a main selling point. We argue that the produced library has its place in the Elixir ecosystem because of our obliviousness around what users want to do with the HTTP protocol.

Two of the abstractions given during extensibility evaluation cannot be implemented with *hackney*. Implementing *Concurrent Client* is impossible with the HTTP client library since it uses an implicit process when working with asynchronous receive. Also, we have not been able to pipeline requests with *hackney* due to it using the same reference ID for a connection and its current request. The *httpc* library does support pipelining but it does not give control over process creation and therefore prevents direct pipelining by a user.

An obvious drawback with the chosen approach is the lack of convenience but we still think it is the right choice when aiming to build a low-level library. An example of this drawback can be noticed in Listing 4.6 where much redundancy because of the messages sent by the *gen_tcp* and *ssl* modules. By allowing implicit process creation, a proxy process could serve as a proxy and only send a single message per type. Another drawback is the ability to control the semantics aspect of the protocol since so much space is left over to the users of the library. From the same function in Listing 4.6 as before, the user must figure out if a closing of the connection by the server is expected or not depending on headers exchanged when operating asynchronously. The discussion of these drawbacks serve as an answer to Question

1 in Section 1.2.

5.4 Robustness

The current section discusses the method and results from the robustness evaluation that was made during the project. The implemented HTTP client library shows promising results when running against websites on the Internet, though it should be further emphasized that this is with certificate validation disabled.

5.4.1 Decoder Evaluation

Unfortunately, it is not enough to strictly adhere the protocol specification. From our experiences when running the implemented decoders against popular websites on the Web, it has become apparent that a 100 % success rate during correctness evaluation is not always sufficient. For example, a response from a popular website could not be parsed with the *Reference* decoder since it is too strict.

Regarding the cases given in Section 4.3, one received HTTP response was incomplete while the other had unexpected white space in a chunk. The fact that these two websites seem to work in a web browser, makes it possible for the website administrators to continue with their invalid HTTP responses. Therefore, it is not obvious what a new HTTP decoder should do, one alternative is to reject any HTTP response that is even slightly invalid with respect to the protocol specification while the other way is to be more forgiving. The first alternative is the most tempting for us who are developing the decoder and because choosing the second alternative strengthens the incentives for not communicating with valid HTTP syntax.

All in all, we are happy with the results from the robustness evaluation in terms of protocol syntax and we are glad to see that the *Default* decoder works on the real Internet.

5.4.2 Client Evaluation

Since URLs constructed from domains listed in The Moz Top 500 index were used, HTTP redirects were crucial in finding the root page of a website. Because the implemented HTTP client library does not support redirects naively, it was extended with support for the feature. From our experience, it seems relatively complicated to get the logic to work in every corner case. Still, as can be seen in Figure 4.1, almost all of the completed responses returned the status code 200. The completed but not successful requests may depend on non-existing resource at the server-side or on the redirection routine.

We have also experienced variance meaning that things change at the requested websites. The more time that was spent on doing the robustness evaluation, the more variance seemed to affect the results. For example, the presence of response header names tend to vary more than expected.

It is a sorrow that we were not able to properly configure TLS certificate verification in the implemented HTTP client library meaning that the robustness evaluation could not be executed with certificate verification enabled for either of the HTTP client libraries.

5.5 Performance

The purpose of the present section is to interpret and discussed the observed results when doing the performance evaluation of the implemented HTTP client library. To answer Question 3 in Section 1.2, we can look at the results presented in Chapter 4 and conclude that the *FSM* is many times faster than the *Reference* decoder for various workloads. When weighting in other aspects such as maintainability and robustness, it is no longer crystal clear, even though we name the *FSM* decoder the best choice.

5.5.1 Decoder Evaluation

It is clear from Figure 4.2 that the *Reference* decoder lags behind the other implemented decoders. Even in case *A* where the total response size was the least, the average latency of the *Reference* decoder was almost 110 % higher than the average latency measured when evaluating the *FSM* decoder. In the case of the *Default* decoder, the difference is roughly 152 % in terms of average latency. Hence, from a performance perspective, the *FSM* and *Default* decoders win.

The difference in performance between the *FSM* and *Default* decoders is not as significant. Again in case *A*, the *Default* decoder is close to 72 % faster than the *FSM* decoder in terms of average latency. Even though state machines are known to be efficient, the *Default* decoder is even more efficient because the two major functions that the decoder relies on: *erlang:decode_packet/3* and *binary:split/2* are implemented in the C programming language. As of today, the *FSM* makes two function calls in the Erlang virtual machine (BEAM) per octet when parsing an HTTP response. By sacrificing readability, the number of calls may be reduced to one per octet. Even then, we see that the *Default* decoder will be faster than the *FSM* decoder.

From the performance observed, we can learn that bigger header values do not make much of a difference when using the *FSM* and *Default* decoders while this is not true when using the *Reference* decoder. Minor differences can be found between the number of chunks when using the former two mentioned decoders. Since every other configuration have the double total response size, it was expected to see those bars high when looking at the observed response time. This also true for all of the implemented decoders.

5.5.2 Client Evaluation

The purpose of the clients benchmark was to put the performance of the implemented HTTP client library in comparison with *hackney* and *httpc*. The former library is developed as an open source project while the latter is included in the Erlang distribution.

From the different configurations listed in Table 4.5 and results observed during the decoder benchmark, the larger HTTP responses were expected to require more time during transmission and decoding. As seen in Figure 4.2, the bars follow a saw shape were odd ones have better performance than even ones due to total response size.

Another assumption was that a greater number of chunks while keeping the total response size constant would lead to more processing time since there is more decoder work to do. This turned out to be wrong, apparently B and F are the slowest configurations and those have four chunks each. After consideration, we believe this is due to the way *Net.HTTP* and *hackeny* receive data. Both of these libraries receive data synchronously directly from a socket interface and they process data as soon as there is something available meaning that their decoders may get called wastefully. On the other hand, *httpc* seems unaffected by the number of chunks. We believe the reason is that it uses a different strategy when reading socket data. Even though *httpc* seems to be more efficient in the cases of B and F, the variability of the observed results are almost always greater than for *Net.HTTP* and *hackney*.

6
Conclusion

This section summarizes the project at hand and gives conclusions made from our experience around the implemented Hypertext Transfer Protocol (HTTP) client library and chosen evaluation criteria. In the end, we try to describe the most probable future for the resulting implementation.

6.1 Implementation

The act of implementing an HTTP client library in the Elixir programming language has been rewarding. Features such as pattern matching, immutability and higher order functions have been both convenient and practical. All in all, we are satisfied with the internal design of the library meaning how logic has been structured and divided into components or modules as they are called in Elixir. It has been great to gain experience with the Mix[1] tool which provides tasks for building, testing and managing dependencies of Elixir projects. The experience is similar with ExDoc[2] which is a tool for generating Elixir source code documentation.

6.2 Evaluation

QuickCheck has been a blast working with during the present project. It has worked really well in the settings of protocol syntax and semantics. The combination of strong oracles and the DSL provided by QuickCheck for writing test data generators has allowed for a successful use of random testing. We are confident that QuickCheck suits the task of evaluating correctness of an HTTP client library written in a functional programming language.

With the set out focus on extensiblity, a major decision from the Elixir's core team was to make state and process management explicit and by doing so we had to sacrifice one of the Elixir run-time system's key abstractions which is cheap processes. Nonetheless, we think that the proposed design is better suited for other people to build upon. Despite numerous other challenges and obstacles, we are confident that the goal of building an extensible HTTP client library has been achieved with

[1]https://hexdocs.pm/mix/Mix.html

[2]https://github.com/elixir-lang/ex_doc

respect to the various examples given.

We recall one time when we were playing with the client library against a popular site on the Web when it was configured to use the *Reference* decoder and found out that the decoder failed as its compliance with the HTTP specification was too strong. Yet again, we have been reminded how hard it is to build software at widespread use. Furthermore, by testing the client systematically against a large set of websites has been a good way of evaluating robustness. We judge it as a realistic way of gaining experience from the real Web.

Another aspect of the *Reference* decoder is its performance, it was found when doing the performance evaluation that the decoder is way slower than the *FSM* and *Default* decoders even in the case where the difference was the smallest. Hence, we can conclude with respect to the robustness and performance properties that the *Reference* decoder is suboptimal even though it is the most convenient to implement. The difference between the latter decoders is not as significant, thus other properties such as maintainability have a greater impact when deciding what decoder to use.

6.3 Future Work

The project idea originates from the people working with the Elixir programming language. A possible future for *Net.HTTP* is consequently in the language's standard library but in order to get there, we have to further adapt the implementation to match standards and expectations after the end of the present project. More work is required, the program needs time with developers and it needs time operating on the Web. We see especially that more work can be done with the extensibility and robustness properties of the library. We think the best way to continue the job would be to release the software out in the wild as a standalone project for the time being.

References

[1] Joe Armstrong. *Making reliable distributed systems in the presence of sodware errors.* PhD thesis, The Royal Institute of Technology Stockholm, Sweden, 2003.

[2] Thomas Arts, John Hughes, Joakim Johansson, and Ulf Wiger. Testing telecoms software with quviq quickcheck. In *Proceedings of the 2006 ACM SIGPLAN workshop on Erlang*, pages 2–10. ACM, 2006.

[3] Tim Berners-Lee, Mark Fischetti, and Michael L Foreword By-Dertouzos. *Weaving the Web: The original design and ultimate destiny of the World Wide Web by its inventor.* HarperInformation, 2000.

[4] Koen Claessen and John Hughes. Quickcheck: A lightweight tool for random testing of haskell programs. *ICFP*, 2000.

[5] Tim Dierks. The transport layer security (tls) protocol version 1.2. 2008.

[6] Whitfield Diffie and Martin Hellman. New directions in cryptography. *IEEE transactions on Information Theory*, 22(6):644–654, 1976.

[7] Adrienne Porter Felt, Richard Barnes, April King, Chris Palmer, Chris Bentzel, and Parisa Tabriz. Measuring https adoption on the web. 2017.

[8] Alec Go, Richa Bhayani, and Lei Huang. Twitter sentiment classification using distant supervision. *CS224N Project Report, Stanford*, 1(12), 2009.

[9] Dick Hamlet. When only random testing will do. In *Proceedings of the 1st international workshop on Random testing*, pages 1–9. ACM, 2006.

[10] Richard Hamlet. Random testing. *Encyclopedia of software Engineering*, 1994.

[11] Michael A Harrison. *Introduction to formal language theory.* Addison-Wesley Longman Publishing Co., Inc., 1978.

[12] Fred Hébert. *Learn you some Erlang for great good!: a beginner's guide.* No Starch Press, 2013.

[13] John Hughes. Why functional programming matters. *The computer journal*, 32(2):98–107, 1989.

[14] Graham Hutton. Higher-order functions for parsing. *Journal of functional programming*, 2(03):323–343, 1992.

[15] Juhani Karhumäki. Applications of finite automata. *Mathematical foundations*

of computer science 2002, pages 40–58, 2002.

[16] Hugo Krawczyk, Ran Canetti, and Mihir Bellare. Hmac: Keyed-hashing for message authentication. 1997.

[17] Ana Monga and Balwinder Singh. Finite state machine based vending machine controller with auto-billing features. *arXiv preprint arXiv:1205.3642*, 2012.

[18] Ronald L Rivest, Adi Shamir, and Leonard Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[19] Michael Sipser. *Introduction to the Theory of Computation*, volume 2. Thomson Course Technology Boston, 2006.

[20] Andrew Yates, Kathryn Beal, Stephen Keenan, William McLaren, Miguel Pignatelli, Graham RS Ritchie, Magali Ruffier, Kieron Taylor, Alessandro Vullo, and Paul Flicek. The ensembl rest api: ensembl data for any language. *Bioinformatics*, 31(1):143–145, 2015.

A
Code

A.1 Typespecs

```
# From Net.HTTP.Protocol.Message
@type header :: {String.t, binary}
@type trailer :: header

# From Net.HTTP.Protocol.Request
@type t :: %__MODULE__{
  headers: nil | [Message.header],
  method: nil | atom,
  target: nil | binary,
  version: nil | {integer, integer},
}

# From Net.HTTP.Protocol.Response
@type t :: %__MODULE__{
  headers: nil | [Message.header],
  status: nil | integer,
  version: nil | {integer, integer},
}

# From Net.HTTP.Protocol.Chunk
@type t :: %__MODULE__{
  data: nil | binary,
  extensions: nil | [{String.t, nil | binary}],
}
```

A.2 Generator Combinators

```
defmodule Random.Generators.Combinators do
  use EQC.ExUnit

  def many0(generator) do
    sequence(list(generator))
  end

  def many1(generator) do
    sequence(non_empty(list(generator)))
  end

  def times(generator, n) do
    Enum.map(1..n, fn _ -> generator end)
  end

  def multiple(generator, 0, max) do
    optional(multiple(generator, 1, max))
  end

  def multiple(generator, min, max) do
    oneof(Enum.map(min..max, fn n ->
          times(generator, n)
        end))
  end

  def optional(generator) do
    oneof([generator, ""])
  end

  def sequence(generator) do
    let value <- generator do
      Enum.join(value, "")
    end
  end

  def range(from, to) do
    as_binary(choose(from, to))
  end

  def as_binary(generator) do
    let value <- generator do
      <<value>>
    end
  end
end
```

A.3 Extensibility Examples

All of the extensibility examples are given in this section.

A.3.1 Stream

```
defmodule Support.Extensibility.StreamError do
  defexception [:reason]

  def message(error) do
    to_string(error.reason)
  end
end

defmodule Support.Extensibility.Stream do
  alias Support.Extensibility.StreamError

  def new!(url) do
    Stream.resource(start_fun(url), next_fun(), after_fun())
  end

  defp start_fun(url) when is_binary(url) do
    start_fun(URI.parse(url))
  end

  defp start_fun(url) when is_map(url) do
    host = url.host
    port = url.port
    transport = String.to_atom(url.scheme)

    path = url.path

    fn ->
      with {:ok, conn} <- Net.HTTP.connect(host, port, transport),
           {:ok, stream} <- Net.HTTP.request(conn, :get, path) do
        stream
      else
        {:error, reason} ->
          raise StreamError, reason: reason
      end
    end
  end

  defp next_fun() do
```

```
    fn stream ->
      case Net.HTTP.response(stream, block: false) do
        {:ok, values, stream} when is_list(values) ->
          {values, stream}
        _otherwise ->
          {:halt, stream}
      end
    end
  end

  defp after_fun() do
    fn stream ->
      Net.HTTP.close(stream.connection)
    end
  end
end
```

A.3.2 GenStage

In the present section, a full and working example of how the implemented HTTP client library can be combined with the GenStage project and Twitter's streaming API. In order to communicate with Twitter, every request has to be authorized by passing a certain request header. This is what the *Twitter* module is for and was built during this thesis project.

```
defmodule Support.Extensibility.GenProducer do
  use GenStage

  alias Support.Extensibility.Twitter

  def init(params) do
    case request(params) do
      {:ok, stream} ->
        {:producer, {stream, 0, []}}
      {:error, reason} ->
        {:error, reason}
    end
  end

  def handle_demand(new_demand, {stream, demand, buffer}) do
    dispatch(stream, new_demand + demand, buffer)
  end

  def handle_info({:tcp, _socket, data}, state) do
    on_data(data, state)
```

```elixir
  end

  def handle_info({:ssl, _socket, data}, state) do
    on_data(data, state)
  end

  defp on_data(data, state = {stream, demand, buffer}) do
    case Net.HTTP.decode(stream, data) do
      {:ok, values, stream} ->
        Net.HTTP.next_packet(stream.connection)
        dispatch(stream, demand, buffer ++ values)
      {:error, reason, _stream} ->
        {:stop, reason, state}
    end
  end

  defp dispatch(stream, 0, buffer) do
    {:noreply, [], {stream, 0, buffer}}
  end

  defp dispatch(stream, demand, buffer) do
    {to_dispatch, remaining} = Enum.split(buffer, demand)
    {:noreply, to_dispatch, {stream, demand - length(to_dispatch), remaining}}
  end

  defp request(params) do
    url = URI.parse(Twitter.base_url)

    host = url.host
    port = url.port
    transport = String.to_atom(url.scheme)

    method = :get
    path = url.path <> "?" <> URI.encode_query(params)
    authorization = params |> Map.to_list |> Twitter.authorization
    headers = [{"Authorization", authorization}]

    with {:ok, conn} <- Net.HTTP.connect(host, port, transport),
         :ok <- Net.HTTP.next_packet(conn),
         {:ok, stream} <- Net.HTTP.request(conn, method, path, headers) do
      {:ok, stream}
    else
      {:error, reason} ->
        {:error, reason}
    end
  end
```

```
end
```

A.3.3 Concurrent Client

```
defmodule Support.Extensibility.ConcurrentClient do
  def start() do
    spawn(fn -> loop(%{}) end)
  end

  def stop(pid) do
    Process.exit(pid, :normal)
  end

  def process(pid, url) do
    send(pid, {:process, url, self()})

    receive do
      {:ok, ^pid} ->
        :ok
      {:error, reason, ^pid} ->
        {:error, reason}
    end
  end

  def loop(state) do
    receive do
      {:process, url, sender} ->
        loop(on_process(state, url, sender))
      {:tcp, socket, data} ->
        loop(on_data(state, socket, data))
      {:tcp_closed, _socket} ->
        {:error, :closed}
      {:tcp_error, _socket, reason} ->
        {:error, reason}
      {:ssl, socket, data} ->
        loop(on_data(state, socket, data))
      {:ssl_closed, _socket} ->
        {:error, :closed}
      {:ssl_error, _socket, reason} ->
        {:error, reason}
    end
  end

  def on_process(state, url, sender) do
    case get(url) do
```

```
      {:ok, stream} ->
        send(sender, {:ok, self()})
        store_stream(state, stream)
      {:error, reason} ->
        send(sender, {:error, reason, self()})
        state
    end
  end

  def on_data(state, socket, data) do
    stream = state[socket]

    case Net.HTTP.decode(stream, data) do
      {:ok, _values, stream} ->
        case Net.HTTP.status(stream) do
          {_encoder_status, :done} ->
            clear_stream(state, stream)
          _otherwise ->
            :ok = Net.HTTP.next_packet(stream.connection)
            store_stream(state, stream)
        end
      {:error, _reason, _stream} ->
        state
    end
  end

  def get(url) when is_binary(url) do
    get(URI.parse(url))
  end

  def get(url) when is_map(url) do
    host = url.host
    port = url.port
    transport = String.to_atom(url.scheme)

    path = URI.parse(url).path

    with {:ok, conn} <- Net.HTTP.connect(host, port, transport),
         :ok <- Net.HTTP.next_packet(conn),
         {:ok, stream} <- Net.HTTP.request(conn, :get, path) do
      {:ok, stream}
    else
      {:error, reason} ->
        {:error, reason}
    end
  end

  defp store_stream(state, stream) do
    Map.put(state, stream.connection.socket, stream)
  end

  defp clear_stream(state, stream) do
    Net.HTTP.close(stream.connection)
    Map.delete(state, stream.connection.socket)
  end
end
```

A.3.4 Pipelining Client

```
defmodule Support.Extensibility.PipeliningClient do
  def start(host, port, transport) do
    case Net.HTTP.connect(host, port, transport) do
      {:ok, conn} ->
        spawn(fn -> loop({conn, []}) end)
      {:error, reason} ->
        {:error, reason}
    end
  end

  def request(pid, path) do
    send(pid, {:request, path, self()})

    receive do
      {:ok, ^pid} ->
        :ok
      {:error, reason, ^pid} ->
        {:error, reason}
    end
  end

  def response(pid) do
    send(pid, {:response, self()})

    receive do
      {:ok, responses, ^pid} ->
        {:ok, responses}
      {:error, reason, ^pid} ->
        {:error, reason}
    end
  end

  def loop({conn, queue}) do
    receive do
      {:request, path, sender} ->
        case Net.HTTP.request(conn, :get, path, []) do
          {:ok, stream} ->
            send(sender, {:ok, self()})
            loop({conn, [stream | queue]})
          {:error, reason} ->
            send(sender, {:error, reason, self()})
        end
      {:response, sender} ->
        case on_response(Enum.reverse(queue), []) do
          {:ok, responses} ->
            send(sender, {:ok, responses, self()})
            loop({conn, []})
          {:error, reason} ->
            send(sender, {:error, reason, self()})
        end
    end
  end

  defp on_request(conn, queue, path, sender) do
    case Net.HTTP.request(conn, :get, path) do
      {:ok, stream} ->
        send(sender, {:ok, self()})
        loop({conn, [stream | queue]})
      {:error, reason} ->
        send(sender, {:error, reason, self()})
    end
  end

  defp on_response([], responses) do
    {:ok, Enum.reverse(responses)}
  end

  defp on_response([stream | streams], responses) do
    case Net.HTTP.response(stream) do
      {:ok, values, _stream} ->
        on_response(streams, [values | responses])
      {:error, reason, _stream} ->
        {:error, reason}
    end
  end
end
```