A performance study of protocols used in a print on demand server

SAM SAM

Master’s Thesis at ICT
Examiner: Fredrik Kilander
Supervisor: Fredrik Kilander
Industrial supervisor: Joe Armstrong

TRITA-ICT-EX-2015:16
Abstract

A performance evaluation of different protocols, both textual and binary based, is done by implementing a simple print-on-demand service as the test bench. The measurements are done with respect to the serialization method, a method defining the procedure to encode structured data to a stream of bytes. The result of this evaluation may play a key in forming a decision regarding which protocol to utilize for mobile networks or networks with high demand. The evaluated protocols are Protocol Buffers, BERT, ASN.1, JSON, and XML. The result shows that Protocol Buffers had best performance for most of the tests and that XML had worst performance for all tests.
Sammanfattning

En prestandautvärdering av olika protokoll, både text och binärbaserade, har gjorts genom att implementera en enkel print-on-demand service som testbänk. Mätningarna är gjorda med avseende på serialiseringsmetoden, en metod som definerar tillvägagångsättet att koda strukurerad data till en ström av bytes. Resultatet av denna evaluering kan spela en roll i bestämmandet av vilken protokoll att använda för mobila nätverk eller nätverk med hög efterfrågan. De evaluerade protokollen är Protocol Buffers, BERT, ASN.1, JSON och XML. Resultatet visar att Protocol Buffers presterade bäst för de flesta testen och XML presterade sämst för alla test.
# Contents

## 1 Introduction
1.1 Motivation ........................................ 1  
1.2 Problem Statement .................................. 2  
1.3 Scope ........................................... 2  
1.4 Purpose .......................................... 3  
1.5 Sustainable development ......................... 3  
1.6 Methodology ...................................... 3  
1.7 Thesis outline ................................... 3

## 2 Background
2.1 Related work ...................................... 5  
2.2 Environment ...................................... 7  
2.2.1 Erlang ........................................ 7  
2.2.2 0MQ .......................................... 9  
2.2.3 LibHaru ....................................... 10  
2.2.4 ZLib .......................................... 10  
2.3 Protocol Description Languages ................. 11  
2.3.1 Protocol Buffers ................................ 12  
2.3.2 BERT ......................................... 14  
2.3.3 ASN.1 ......................................... 15  
2.3.4 JSON .......................................... 17  
2.3.5 XML .......................................... 18  

## 3 Methodology
3.1 Literature study .................................. 21  
3.2 Implementation .................................. 23  
3.3 Measurements and analysis ....................... 24  

## 4 Implementation overview
4.1 The 0MQ connection ............................... 28  
4.2 The client ........................................ 28  
4.3 The server ....................................... 28
5 Code generation
  5.1 Overview ............................................. 31
  5.2 LibHaru API documentation ............................ 31
  5.3 Structured API specification ........................ 32
  5.4 Generated code ...................................... 33
    5.4.1 Port code generation .............................. 34
    5.4.2 Erlang module template ............................ 36
    5.4.3 Protocol buffers code generation ................. 36
    5.4.4 Bert code generation .............................. 37
    5.4.5 ASN.1 code generation .............................. 38
    5.4.6 JSON code generation .............................. 39
    5.4.7 XML code generation .............................. 40

6 Test and measurement programs .......................... 41
  6.1 Overview ............................................. 41
  6.2 Measurements on the client side ....................... 42
  6.3 Measurements on the server side ....................... 43
  6.4 Test data sets ....................................... 43

7 Analysis tools .......................................... 45
  7.1 Overview ............................................. 45
  7.2 From data to charts ................................... 46

8 Empirical evaluation .................................... 49
  8.1 Test design .......................................... 49
  8.2 Test implementation and setup ......................... 49
  8.3 Test results and analysis ............................. 51
    8.3.1 Encoding performance .............................. 51
    8.3.2 Decoding performance .............................. 54
    8.3.3 Size .............................................. 56
    8.3.4 Round trip time ................................... 58
    8.3.5 Overall Performance ............................... 60
    8.3.6 Speed and size performance ......................... 62

9 Conclusions ............................................. 65
  9.1 Summary .............................................. 65
  9.2 Conclusion .......................................... 65
  9.3 Discussion .......................................... 66
  9.4 Future Work ......................................... 67

References ................................................... 69

A All test results ........................................ 73
  A.1 Encoding performance ................................. 73
  A.2 Decoding performance ................................ 75
Chapter 1

Introduction

This chapter introduces the reader to the thesis and presents the motivation.

This work investigates the performance of a network service by sending data (or messages) between a client and a server and measure how the performance varies in respect to changes of the serialization mechanism (used to pack the messages) and the size of the messages. Messages, have specific data, structured according to a specification with predefined syntactic rules. The result of this evaluation may play a key in forming a decision regarding which protocol to utilize for which specific purpose. For instance, to determine which of the evaluated protocols in this study are most fit for mobile networks or networks under load. The protocols that have been evaluated are Protocol Buffers, BERT, ASN.1, JSON and XML. For evaluation of the different protocols, a simple print-on-demand service is implemented for the various protocols. A measurement of performance is made and the results are evaluated.

The network service in this case creates PDF-files from messages sent from a client over a network. The service has two parts, a server and a client, where the former receives messages from network sockets and creates PDF-files, and the latter is the application that sends messages of what to include in the PDF-files. We measure the time to encode and decode messages as well as message density, and also the throughput of the system measured in PDF pages per second.

1.1 Motivation

In wireless communication, the bandwidth is limited. An example of this is LTE 4G (Long-term evolution fourth generation), a standard for wireless communication of high-speed data for mobile phones and data terminals. An LTE 4G cell have a limited capacity of active users with a peak of 300MB/s [20]. This means that a user connected to such a network have a theoretical download speed of 300MB/s, however if several users are sharing this bandwidth the speed per user would de-
crease. Another important aspect to consider is that most mobile network operators sell limited data plans. It is therefore important, when using a wireless network, to be frugal with what you send.

In many wired services it is instead more important to package messages quickly than to send small packets. An example of this is a bank where the load is high with many transactions being made in a short time.

Given the motivations above, which protocol to use is important. Thus the project providers at Ericsson proposed to evaluate protocol description languages for a PDF-generator as a simple practical implementation of this problem.

1.2 Problem Statement

This thesis evaluates packet size and packaging speed of various combinations of protocols and data set types sent over a network. This is to determine the different optimal combinations and answer the following questions for the protocols Protocol Buffer, BERT, ASN.1 JSON and XML:

- Which of the packaging mechanisms are the fastest regardless of data set?
- Which of the packaging mechanisms produce the smallest packet size regardless of data set?

The protocols are used to send messages locally in one computer with and without compression for further evaluation of the protocols compactness. A print on demand service is the test bench and the following performance metrics are studied:

- The number of PDF pages created per second
- The round-trip time of sending a message and receiving an acknowledgment
- The time to serialize a message
- The time to deserialize a message
- The size of a message

1.3 Scope

Given there are many protocols, this thesis focuses on 5 protocols as representatives of most protocols. The measurements that are made focuses on the method of when a message is encoded/decoded and the relation between number of bytes and the actual content of the message. As this thesis is interested in the different serialization mechanisms, it is not relevant to send the messages in a parallel system. Meaning, every next message will only be sent when an acknowledgment is received from the server.
1.4 Purpose

The purpose of this performance study is to answer the questions presented in section 1.2 by evaluating a variety of protocols. The author of this thesis hopes of giving a good understanding to the reader of the difference between packaging mechanisms in a practical sense. The author would also like to convey the importance of choosing the right protocol when developing a service with a limited network or when using a network with great load.

1.5 Sustainable development

The effect of a performance study on protocols could be in many ways beneficial for sustainable development. From an economical aspect, it will most likely be to an advantage for mobile network users to use services with minimal data transfer. It is also ecologically beneficial to apply protocols that use less CPU-power and smaller packets as these two factors have great impact on battery usage. Of course, this does not only apply to mobile phones. The internet of things (IoT) has extended the usage of mobile networks tremendously over the years. Using a more compact and faster protocol for internet devices can have a great influence on both economical and ecological aspects. And naturally, the cheaper the usage of a device becomes, the more people can use it.

1.6 Methodology

A short investigation of different tools and libraries was made. After defining what libraries to use, a print on demand service was implemented together with a specific protocol to create the test bench. The purpose of the experiment is to understand how the protocols perform in a practical sense and the purpose of the PDF-generating service is to have a test bench that can handle a wide variety of data types. This is of course important as we want to stress the performance of the protocols using many different data types in order to answer the problem statement. PDF-manipulating functions are implemented by using a mature library. When the implementation of the test bench core was implemented successfully, more protocols were implemented in an iterative manner. A more detailed methodology of this performance study is described in chapter 3.

1.7 Thesis outline

This thesis has the following structure:

- **Chapter 1:** This chapter introduces the reader to the thesis and presents the problem statement and motivation.
• **Chapter 2:** This chapter describes the techniques used in test bench and the protocols. It also gives a motivation of the used techniques.

• **Chapter 3:** This chapter describe the complete process and methods used for this performance study.

• **Chapter 4:** This chapter starts by presenting an overview of the implementation of the experiment and introduces the reader to the chapters that in detail describes the implementation.

• **Chapter 5:** This chapter describes how the test bench is created for all protocols that are studied, using *code generation*.

• **Chapter 6:** This chapter describes the programs that measures the performance of the protocols in the service.

• **Chapter 7:** This chapter presents the software created to analyze the performance data.

• **Chapter 8:** The first part of this chapter present the chosen test design and the implementation of the performance study. The second part presents the results of the protocol evaluation and a short result analysis.

• **Chapter 9:** This chapter summarizes the work done in this thesis and presents a final conclusion.
Chapter 2

Background

This chapter first presents related work and later describes the environment used for the implementation and the chosen protocol description languages for the performance study.

2.1 Related work

Similar studies have been made, but with different focus. Bruno Gil and Paulo Trezentos [14] analyzed the impact of three protocols in regards to the power consumption and CPU usage on smart phones. The chosen protocols where XML, JSON and Protocol Buffers. The study focused mainly on mobile applications that require data synchronization with a web server. The tests performed where package size measuring, time to encode/decode, time to synchronize of the server and the energy consumption for each test. The protocols were tested under Wifi and 3G network interfaces with two different data sets. They also tested the protocols with and without gzip compression.

The result of their work shows that JSON with compression is the most efficient protocol in synchronization, parsing data on the server side and battery management. Protocol Buffers was less efficient than JSON when the data is compressed. The XML protocol showed weaker results in all tests except for power consumption with data synchronization.

This work is of course relevant as it investigates the protocols in limited environment and tries to find the most optimal for specific metrics such as time and energy consumption.

Gligorić et al. [15] describes a performance evaluation of XML and Protocol Buffers. The reason for the evaluation was to analyze which protocol to use for the EcoBus project to minimize cost and latency. EcoBus is a system that monitors public transport vehicles in a large urban area. On top of every vehicle there is a device that measures weather parameters, current location and gas measure-
ments (CO, CO₂, NO₂). The devices sends the information via the mobile network interface GPRS.

The test were not made on these devices but rather on an Android mobile device, as all the EcoBus devices ran old software and could not use Protocol Buffers. Four tests were done on both XML and Protocol Buffers measuring parsing time without any network connectivity, parsing time with network as well as memory and battery tests. The result of the tests showed a significant difference between the two with Protocol Buffers in favor. The authors conclude that when using Protocol Buffers instead of XML, the system will reduce transfer time, cost of the running system and increase battery time.

This work is related as it highlights the importance of choosing an optimal protocol when a service is using limited resources.

Maeda [28] presents a performance evaluation made over twelve Java object serialization libraries in XML, JSON, Thrift, Avro, Protocol Buffers and Java object serialization via RMI (remote method invocation). The Java object that was used for the evaluation is an object for ordering coffee with an arbitrary number of options. The evaluation was designed to test the libraries by serializing and deserializing the objects 500 times with different set of options varying from 0 to 900. Measurements where made on the time to serialize and deserialize as well as the size of the serialized objects.

Libraries using XML and JSON protocols on average were least preferable. The author states that there is no best solution and each library is good in the context it was developed.

This work measures both time of encoding/decoding messages and the size of the packets for many well known protocols. It is therefore related to this study.

Nurseitov et al. [31] conducted a case study where a comparison between JSON and XML was made. The test bench created was a client and server that sends Java object encoded in the two protocols. The server and client had separate hardware interconnected by a switch. The measurements made where: the number of objects sent, total time to send all objects, average time per object transmission, user CPU utilization, system CPU utilization and memory utilization. The test cases consists of two scenarios. In the first scenario, a single time-consuming transmission of large quantity of objects is run. This is to achieve accurate average measurements. For the seconds scenario, multiple test cases are run with an increasing amount of objects. This is to determine if the two protocols differ statistically as the number of encoded objects increases. The results show that JSON is significantly faster and uses fewer resources than XML.

As the case study tested JSON and XML on many aspects such as time to send packets, it is related to this thesis.

Müller et al. [30] evaluated a change of the underlying communication protocol of the EPC (Electronic Product Code) network. The EPC network is used to
share physical product information between companies. The identification codes are stored in RFID tags for each product. This kind of network is used in pharmaceuticals with the great load of billions of packages sent a year. The communication protocol used in this network technology is XML and SOAP and the conducted experiment was to change this to Protocol Buffers.

Two factors were measured for the test, processing time (time construct a message) and serialization (size of the serialized message). The results showed a great improvement of the service. The message size was reduced by about 75%, the processing time by at least 11% for XML and at least 59% for SOAP communication.

This related work is a good example of how only changing the underlying protocol, a noticeable improvement is made on the service. It is therefore highly related to this thesis.

2.2 Environment

The evaluation is constructed, as earlier mentioned, by sending messages to a print on demand service and measuring different metrics. The service was created with the programming languages Erlang and C. The messaging library used for transport is 0MQ and for PDF generation, the C library LibHaru is used. The following section describe the essential environment parts of the service and a motivation of why these techniques are used in this study.

2.2.1 Erlang

Erlang is a functional programming language developed by Ericsson as a result of a research project during the 1980s. It was intentionally designed to create distributed, concurrent, fault-tolerant, soft real time telephony applications. Soft real time systems require a response in milliseconds and a missed deadline will only result in a degrade of application quality. The language was released under an open source license in 1998 [11] mainly to spread the language outside Ericsson [39]. Now, the language is widely used in many areas such as Amazon’s scalable database services, Yahoo’s social bookmarking service and the messaging application WhatsApp with over 500 million users [8] [34] [22].

Erlang is categorized by the following features [5]:

- Concurrency

Erlang code is run on the Erlang virtual machine that, among many things, handles the concurrency of extremely lightweight processes. As the scheduling is handled in the virtual machine, there are no requirements on host operating system regarding this matter. These processes have no shared memory and their inter-process communication is handled via asynchronous messages.
• Distribution

As mentioned above, Erlang was designed to create distributed applications, it is therefore designed to run in distributed environments. A distributed Erlang application can be created by connecting many Erlang virtual machines (also called Erlang nodes) over a network. As the nodes are Erlang virtual machines, the host operating systems may differ between them. Erlang nodes communicate just as Erlang processes do, making the code and the architecture very flexible.

• Robustness

As Erlang is designed to create fault-tolerant applications, it has many primitives for error detection. Processes and Erlang nodes can even monitor each other and have exchangeable nodes in case of an error.

• Soft real-time

Soft real-time systems require response time in milliseconds. Erlang supports programming soft real-time and have therefore a core that take this in account. An example of this is the fast garbage collection techniques used in Erlang.

• Hot code upgrade

Erlang code can be loaded to an application during run time. An Erlang application can for instance during operation start a process with the new code, redirect from the old to the new code and then kill the old process. Hot code upgrade introduces high up time of the systems as maintenance, bug fixes, upgrades and more can be made without having to stop the application.

• Incremental code loading

As Erlang has a dynamic code loading, the user can control how the code is loaded. This means that the code can be loaded when needed. This could be useful when for instance developing an application, were constant testing is needed.

• External interfaces

Erlang gives the option to make Erlang programs communicate with the processes of the host operating system making Erlang really flexible. The communication protocol is the same as when communicating with Erlang processes and nodes. Meaning, Erlang can communicate with programs written in any
programming language by using the Erlang communication protocol. Erlang also enables the user to integrate C code in an Erlang function and load it to the Erlang core. Using this, the code would be treated just as any other Erlang function.

As the language is in such a high level, programs written in Erlang often become shorter than if written in traditional programming languages such as C and C++ [32]. Shorter code means faster implementation time and easier maintenance thus making the development cheaper.

Given the motivation above, the flexibility of the language and the key features of Erlang, it becomes a natural choice to use Erlang for this service.

2.2.2 0MQ

0MQ [21], Zero Messaging Queue, is an open source asynchronous messaging library. The aim for the library is to increase simplicity and be used in scalable distributed applications or concurrent applications. The library is widely used in many different applications and is even used as the primary messaging system for CERN controls middleware [13]. The official programming language for the library is C, although third party implementations exist in more than 40 programming languages.

The idea behind 0MQ is to create networks functioning like human brains with trillions of neurons firing of messages to each other, a massive parallel network with no central control. The architectural structures of such networks are not limited to one-to-one connections as the building blocks of traditional networking. 0MQ uses a highly generalized socket that accepts many-to-many connections. The sockets can be set to a connection pattern and by using these patterns, highly complex network circuits can be created. If specified in the circuit, messages can be sent asynchronously without any messaging queue, hence the name.

The basic connection patterns available are the following:

- **Request-reply**

This pattern provides remote procedure call and task distribution. It connects a set of clients to a set of services.

- **Publish-subscribe**

This pattern provides data distribution by connecting a set of publishers to a set of subscribers.

- **Push-pull**

This pattern provides message distribution and collection through pipelines that can be arranged with multiple steps, including loops.

- **Exclusive pair**
This pattern provides one-to-one connection between two nodes with bi-directional communication.

These building blocks offer a paradigm that simplifies the implementations, as it raises the abstraction level and covers the underlying low-level network-related codes for the developer. Given the motivations of usage simplicity and the broad usage of this library, it becomes a natural choice for the print on demand service of this study.

2.2.3 LibHaru

LibHaru [25], or the Haru library, is an open source and cross-platform library for generating PDF-files. It was written by Takeshi Kanno in ANSI-C, meaning it can be compiled with any compliant C compiler. It is the only widely used open source C or C++ library for PDF generation that is under active development. Some implementations exist to use LibHaru in other programming languages, although none for Erlang. The library includes the following PDF manipulation features:

- Adding text and geometrical figures
- Adding Outline, text annotation and link annotation
- Embedding PNG and Jpeg images
- Embedding Type1 font and TrueType font
- Compressing document
- Encrypting document
- Using various character sets (ISO8859-1 16, MSCP1250 8, KOI8-R)
- Adding characters using Chinese/Japanese/Korean fonts and encodings

By using the functions provided by the library, a PDF document can be created without the prior knowledge of PDF-file structures. This means that the functions provided have intuitive names and the library is well documented. LibHaru is considered to be a very mature PDF-file generating library and it is therefore used in the print on demand service of this study.

2.2.4 ZLib

ZLib is a library for lossless compression data created by Jean-loup Gailly and Mark Adler in 1995 [19]. It was originally written in the programming language C but implementations for other languages exist, including Erlang. The compression algorithm used by this library is deflate that, among many things, tries to reduce byte redundancies and shorten bit representation for common symbols. This algorithm
has been used in many programs and formats such as PNG and ZIP. The ZLib library wraps the compressed data with a header and checksum. It can either compress with or without a header for format representation. The available formats are *zlib* and *gzip*, were the former is more commonly used in data streams and the latter is more commonly used for compressed files (with .gz as the file name extension).

Using this compression library, evaluation of the protocol compactness is possible by compressing the packets encoded by each protocol.

### 2.3 Protocol Description Languages

A protocol description language defines syntactic rules of how to represent data.

It is important to distinguish between *text-based protocols* and *binary-based protocols*, where the former represent encoded values for human readable text and the latter represent the values in binary form. If a message is to be sent with the number 123, it would require three text characters to store in a text-based protocol. If the message was sent with a binary-based protocol the number would be stored as a specific data type.

Common text-based protocols are:

- JSON
- XML
- YAML
- Comma Separated Values

Common binary-based protocols are:

- ASN.1
- Apache Avro
- Thrift
- Protocol Buffers

The following protocols description languages are chosen for this study, which are representatives of the broad categories of protocols:

- Protocol Buffers
- ASN.1
- BERT
- JSON
- XML
Both Protocol Buffers and ASN.1 are chosen as they are similar to many binary-based protocols, such as Thrift, Avro and others. BERT is chosen to represent self-descriptive binary-based protocols. Both XML and JSON are chosen as they are nearly ubiquitous in applications that uses text-based protocols.

The chosen protocols are used as RPC (remote procedure call) protocols. An RPC enables a request of a service from another process or program connected through a network. It could for instance cause a subroutine to run in another computer. There are many well known and used RPC-protocols such as CORBA, Sun RPC, JSON-RPC, SOAP and many more. As this thesis mainly focuses on the serialization mechanism of the chosen protocols and not the procedure of requesting a service through a network, it will not discuss any RPC-protocols.

All the chosen protocol description languages are described in detail in the sections below.

### 2.3.1 Protocol Buffers

Protocol Buffers [16] are Google’s mechanism for serializing structured data. The structure of the data is predefined in an IDL (Interface Definition Language) specification file with `.proto` as the file name extension. These files contain a language-neutral description of the protocol buffer data i.e. specification of the messages that are sent between the communicating nodes. Both sides of the communication must have the same description in order for the interpretation of the messages to work. The description is then used to generate a programming language specific data type and encoding/decoding functions for each side of the server/client communication.

**An example:**

Imagine a JAVA-client connected to a C++ server. Both the client and the server use library code for encoding and decoding messages which is generated from a single `.proto`-file. In this example, the message specification file is a `.proto`-file that contains a description of a data-type representing a person with both name and age, called `person.proto`. From the specification one can generate C++ and Java code which packs and unpacks people data. This means that the server has a C++ class, specifically generated from `person.proto`, that can encode/decode messages and represent a `person`-type in normal C++ syntax. Likewise, the client has a JAVA class with the same functionality but in normal JAVA syntax. If the server now wants to send data of a person it can use the generated C++ class, as follows:

```c++
Person person;
string data; //A container for the serialized version of the message
person.set_name("John Smith");
person.set_age(23);
person.SerializeToString(&data);
```
The message that is in the C++ string data can now be sent to the client that can parse the message by using its generated JAVA class, as follows:

```java
byte[] data; //This contains the encoded message from the server.
Person person = Person.parseFrom(data);
System.out.println(person.getName());
System.out.println(person.getAge());
```

How the data is transported between the client and server is not discussed here.

**Structure**

The interface description language of Protocol Buffers makes it possible to create highly complex message structures. The previous example can be expanded to include more fields as the example below shows, written in the language of Protocol Buffers:

```java
message Person {
  required string name = 1;
  required int32 id = 2;
  optional string email = 3;

  enum PhoneType {
    MOBILE = 0;
    HOME = 1;
    WORK = 2;
  }

  message PhoneNumber {
    required string number = 1;
    optional PhoneType type = 2 [default = HOME];
  }

  repeated PhoneNumber phone = 4;
}
```

The specification above could be meant to represent information for a person in an address book.

The allowed fields are divided into two categories; scalar types and composite types. Scalar types include fields for representing integers, floating point numbers, booleans and a sequence of bytes in form as text (UTF-8 or 7-bit ASCII encoding) or as a byte array. Composite field types in the structure are `enum` (enumerator) and `message`. Messages introduces nesting in the data structure as they can contain any other field type.
A field can be assigned with one of the following tags: required, optional or repeated. The message structure above specifies, among other things, that a message:

- must contain a name value
- can be serialized without an email value
- can have one or more phone numbers fields

The numbers that are assigned to the field specifications are the identifications numbers of the fields in the serialized data.

**Encoding**

When looking at the serialized data of the messages there are other data types used, called wire types which are a sequence of bits. There are six wire types, each with specified rules of how to interpret the bits. Data is encoded as a 3 bit tag value followed by the data itself. An example of a wire type is the VarInt type which specifies an integer that does not have a predefined byte length, it varies in relation to the actual value of the integer. A VarInt uses the MSB (Most significant bit) to indicate the end of the integer value’s byte sequence. VarInts are used to store the message fields identification numbers. This is why the id-number is manually inputed, because VarInts varies in bytes depending on the integer value. Therefore; the frequently used fields should have lower id-numbers to minimize the overall sent data.

### 2.3.2 BERT

BERT(Binary ERlang Term) [33] is a binary serialization format that is based on Erlang’s External Term Format(ETF). It does not require any Interface Description Language(IDL) specification or programming language specific code generation like Protocol Buffers, i.e the protocol is self descriptive. As there are no predefined specifications of what data types to include in a message, BERT becomes highly flexible in structure of the messages and is very suitable for dynamic and agile work flow as the messages can be expanded under operation. It is fully compatible with Erlang’s binary serialization format mechanism which is used when calling the built-in Erlang function `term_to_binary`. The data types (or terms) that are supported within a message are similar to Erlang terms, here are some examples:
2.3. PROTOCOL DESCRIPTION LANGUAGES

The terms list and tuple can contain other terms. Using these simple terms, BERT defines standard formats for complex data types. The complex data types have a general structure with a tuple that has a the atom bert as the first element. An example of a complex type is the dictionary data type that is similar to a hash table with key/value pairs. Its structure specification is {bert, dict, KeyAndValues} where the last element is a list of 2-tuples representing the pairs, for instance {bert, dict, [{name, "John"}, {age, 23}]}.

Encoding

The serialized data, BERP(Binary ERlang Packets), consists of a 4 byte header followed by the serialized message according to Erlang binary to term standard. The value of the 4 byte header is the length of the actual message in bytes, which give the opportunity to send a series of BERT messages in different length. As the length indicator is 4 bytes, the maximum length of a BERT message is 4GB.

2.3.3 ASN.1

ASN.1 or Abstract Syntax Notation One is a protocol description language that was first standardized in 1984 [12]. The latest version was standardized in late 2008 [37]. The language has a wide range of application usage such as email protocols, car and truck monitoring, streaming media on the Internet and telephony [36]. Similarly to Protocol Buffers, the messages are defined in a file written in an ASN.1 interface description language that is used to generate language specific code for encoding and decoding the data. Implantations of ASN.1 code generation exists to more than 150 programming languages, including Erlang [23] [1].

Structure

The specification file, also called ASN.1 data description, is written in the standardized language. An example of such a file is as follows:

```
People DEFINITIONS AUTOMATIC TAGS ::= BEGIN
PhoneType ::= ENUMERATED{
    mobile,
    home,
    work
}
```
ASN.1 has many syntax and data types. The example above is enough to explain the most common usage. As seen above the complete description is wrapped around the definition of a module called People. The types are tagged automatically with the AUTOMATIC TAGS setting, although this can be changed to manually set the tags similarly to Protocol Buffers. The data types that ASN.1 define are in two categories, basic types and constructed types, where the latter are for nesting. For instance, the type SEQUENCE is a constructed type of a numeric string for storing a phone number, and a value of an enumerator for storing the type of phone number.

Encoding

ASN.1 has several encoding rules of taking data and serialize it, they are the following [12]:

- Basic encoding rules (BER)
- Canonical encoding rules (CER)
- Distinguished encoding rules (DER)
- Packed encoding rules (PER)
- Light weight encoding rules (LWER)
- BACnet encoding rules
- Octet encoding rules (OER)
- Signaling specific Encoding Rules (SER)
- XML encoding rules (XER)

The most common encoding rules are BER and PER [23] and the chosen encoding rule for this study is the basic encoding rule (BER).

The encoding principle of the BER rule is to always use the triplet format TLV (Type, Length, Value), sometimes called (Tag, Length, Value). With this format,
the transferable data can represent any message from any ASN.1 description. The
tag, or the type is referring to the identification of a description from the ASN.1
description file. The length specify the byte length of the value. Finally, the last
part of this format contain the value of the data. If the data type is a constructed,
as discussed above, the value will contain other TLV triplets.

2.3.4 JSON

JSON, or JavaScript Object Notation, is a text-based notation for denoting values
that is based on a subset of the JavaScript programming language [24].

JSON messages must be written in Unicode characters and when forming such
a message, two structures can be used. They are objects and arrays. The object
structure uses key/value pairs as the following shows:

{key : value , key : value}

The keys are text wrapped by quotation marks, i.e strings. A colon must exist
after every key for separation of the key and the value. The pairs are separated by
comma characters. There can either be no, one or many key/value pairs (empty
object).

The array structure is an ordered list that can either consist of no elements
(empty array), or many elements. The syntax for this is as the following shows:

[value, value, value]

The complete array is wrapped by square brackets and comma characters are used
to separate the elements.

The types of values that these two structures can include are the following:

- null (empty value)
- Boolean
- Number
- String
- Array
- Object

A boolean value can either be true or false. A number can be a negative
or positive integer or floating point number. Numbers can also include exponents.
Examples of numbers are the following:

-34  23.0  45.8e+12  1.9E-16
Strings are, like in most programming languages, a series of text characters wrapped by quotation marks.

JSON is a format that accepts nested structures, as both arrays and objects are considered as values. Here is an example of a JSON structure:

```json
{
  "firstName": "John",
  "lastName": "Smith",
  "age": 25,
  "employed" : true,
  "carModel" : null,
  "address": {
    "streetAddress": "21 2nd Street",
    "city": "New York",
    "state": "NY",
    "postalCode": "10021"
  },
  "phoneNumber": [
    {
      "type": "home",
      "number": "212 555-1239"
    },
    {
      "type": "fax",
      "number": "646 555-4567"
    }
  ],
  "gender": {
    "type": "male"
  }
}
```

### 2.3.5 XML

XML, or Extensible Markup Language, is a markup language that is designed and standardized by World Wide Web Consortium (W3C) [38]. The language is extended from SGML (Standard Generalized Markup Language). XML is widely used in many applications and has been extended to other protocols such as Simple Object Access protocol (SOAP). As it is a markup language, it is completely text based meaning there are no defined data types. It rather defines textual syntactic rules of the encoding in unicode characters called markup. Text in an XML-structure is often called an XML document.

XML define many markup rules, the following list contain the most common ones:
2.3. PROTOCOL DESCRIPTION LANGUAGES

- **Element**

An XML document contains one or more elements. An element is a logical component that is denoted with a tag name. The following shows the most basic version of an element component:

```xml
<person />
```

The tag name of the element in the example above is `person`. The example also presents an empty element, meaning it does not have any content. Content of an element can either be more elements, called child elements or non-markup content. The following are expansions of the previous example in both forms:

```xml
<person>
  <child />
  <child />
</person>

<person>John Smith</person>
```

If the element is not an empty element, an end tag is needed to indicate where the element ends as the examples above shows. Child elements can have the same tag names. This useful for example when creating a list of elements with the same logical structure and grouping them with the same tag name.

- **Attribute**

An element can include many or no attributes. An attribute is a key/value pair providing more information to the element structure outside the element content. The following presents how the previous example can be expanded with attributes:

```xml
<person name="John Smith" age="45">
  <child age="12" />
  <child age="5" />
</person>
```

The attributes presented above for the first element are `name` and `age` with the values `John Smith` and `45`. The names of the attributes are unique identifiers meaning, in the example above, the element `person` can not have more than one attribute with the name `age`.

- **Character Data**

There are some characters such as `<` and `>`, that are not allowed to use as content. This is because they are used as a part of the markup for the XML document. To solve this, the content can be inserted in a section called character data. The section
is created by adding `<!CDATA[ at the beginning of the desired content and `]]>` at the end. Anything in between of these character sequences is treated as non-markup content.

Another way of escaping markup, is to use a character reference. A character reference treats a sequence of characters to a single character without. The reference is wrapped with the characters `&` and `;`. For example, the character sequence `&lt;` refer to the character `<`, were `lt` stands for “less than”. There are many predefined character references and the complete list of unicode characters can be accessed with numeric character references. A numeric character reference is constructed by wrapping a unicode number reference with `&` and `;`. For instance, a division sign can be inserted by the references `&divide;`, `&#247;` or `&#x00F7;`.
Chapter 3

Methodology

This chapter describe the complete process and methods used for this performance study.

The complete project can be illustrated in three phases as depicted below:

![Methodology overview](image)

**Figure 3.1:** Methodology overview

As seen above, this project started with a literature study were important search of information was made. This was in order to implement the test bench, where the protocol measurements were made. The following sections describe the methods used during these phases.

3.1 Literature study

The literature study was initialized by first searching for related work of this type of evaluation. This was to understand the current scientific progress of this problem and other researchers conclusions. This is of course important as there is an interest in appending and avoiding the repetition of previous work. The search for related work was done in both KTH Primo [7] and Google Scholar [17]. The search phrases used in the search were different protocol names in a combination of the words *performance, evaluation* and similar.

Some of the articles found were less preferable even though they were related as they were merely pointing out the syntactic differences, how easy it is to implement the protocol or comparing the performance without any real life application. An
example of this is the evaluation made by Kaur and Fuad [26]. They evaluated Protocol Buffers and XML without any real life application and the evaluation included more than just performance such as readability, prerequisite knowledge, implementation and other. The evaluation conducted by Gligorić et al. [15] on the other hand was also an evaluation of changing XML to Protocol Buffers but was instead for an existing application.

Using this information, the project began to take shape. This thesis mainly focuses on how the protocols perform in a practical sense. The chosen approach was therefore to create a fully functional service. The purpose of the PDF-generating service was to have a fully functional service with a wide variety of data types. This is to compare the protocols both in practical operation and fairly between the different protocols.

A pre-study was initiated and decisions were made on key parts of the implementation based on the gathered results. The purpose of the pre-study was to understand which programming language to use, which messaging library would serve the experiment best and of course all the various protocol libraries to analyze before truly engaging in the implementation, which would require writing code. Part of this study was to truly understand the feasibility of this project, how complex the job would be and what the limitations were. In a sense it helped shape all the following work as it was on this basis all decision were implemented.

The key parts (or requirements) of the implementation that emerged during the pre-study are the following:

- The print on demand service consists of a server and a client
- The service is written in Erlang
- The messaging library of the service is 0MQ
- The PDF library of the service is LibHaru
- The client sends messages to the server
- The server receives messages from the client
- The messages are sent according to a protocol
- The protocol is exchangeable without changing the service
- The tested protocols are Protocol Buffers, BERT, ASN.1, JSON and XML
- The server maps the messages to function calls of the PDF-library
- The client is be able to read messages from a file and send them sequentially to the server
- Three test cases are written that resembles realistic usage of a print on demand service
• The complete service measures all the performance metrics as mentioned in the problem statement (section 1.2), they are:
  – The number of PDF pages created per second
  – The round-trip time of sending a message and receiving an acknowledgment
  – The time to serialize a message
  – The time to deserialize a message
  – The size of a message

• The performance of all the metrics are saved in files

3.2 Implementation

The implementation process of the test bench is according to the figure below:

![Implementation process](image)

Figure 3.2: Implementation process

The test bench was designed to include the core features and serve the purpose of the performance study. It is therefore designed to be an MVP (minimum viable product) [6]. This was created by agile software development until all the core features were implemented. The test bench implementation was considered done when the application were at the stage of sending messages from a client to a server, create a PDF file accordingly and measure all the interesting metrics of the system.

The test bench was first implemented with only one protocol for message packaging. After a successful prototype of the service, more protocols were implemented iteratively. For each protocol to implement, a simple test of the library were performed, library integration was implemented in the test bench and finally the complete service was tested. The system was tested by using a small test data set with expected results such as size and time differences between the different data sets.

The library for generating PDF files (LibHaru) has many functions and not all can be implemented by hand to be encoded and decoded for each protocol. The test bench is therefore created to only include a select few. When the test bench included all the metric measurements using all protocols, an application were
written to automatically generate code that uses the rest of the LibHaru functions in the test bench. The method of partly writing code by hand and partly generate code is to be as time efficient as possible while still create a service that is close to a real life application. This is described in the next chapters.

3.3 Measurements and analysis

The created test bench were chosen to save the performance data as raw numbers of all the measured metrics. A separate tool was created to visualize the data for analysis. The reason of creating an application instead of finding a third party application is because the functionality is very easy to implement and it creates a much greater freedom when using the data.

Before the measurements were performed, data sets were written that are realistic to use in such a service. The data sets were also constructed to represent a variety of data types as it is relevant to the problem statement. This is to give a fair performance study of the protocols while still try to answer the questions stated in section 1.2.

After creating the data sets, the test procedure was specified. It was designed to be stressful for the system by sending the data sets many times separately and combined for each protocol. Different components of the system were tested. As specified in section 1.2, the measurements were the following:

- The number of PDF pages created per second
- The round-trip time of sending a message and receiving an acknowledgment
- The time to serialize a message
- The time to deserialize a message
- The size of a message

As the mentioned measurements above both include size and speed of the protocols, they are fit to answer the problem statement introduced in section 1.2. The test design and procedure is described in chapter 8.

The test was performed after all preparation according to the test design and the measurements were saved. The mean and the standard deviation were calculated from the measurements data files. This is not only to get the average performance of the protocols in this service but also to get an understanding of how great the variation of the performance is for each protocol. Using these values and the created visualization tool, graphs were generated and analyzed. As the problem definition was to understand which of the protocols perform best in size and in speed, the analysis was merely a comparison between the protocols' performance. As it is a comparison, the graphs are not presented in the measured units but rather a percentage difference with a base line. The chosen base line is the average measurement of XML as XML is a widely used and known protocol.
3.3. MEASUREMENTS AND ANALYSIS

The measurements for all the components is an important factor in this experiment. It creates an understanding of the service’s behavior for the different protocols. It is also important as it shows the consistency of the performance between the protocols throughout the service.
Chapter 4

Implementation overview

This chapter gives an overview of the implemented service and explains the chosen approach of how messages are sent between client and server.

The complete test bench system is illustrated in figure 4.1.

![Diagram](image)

**Figure 4.1:** Test bench system overview

The code generator creates the complete server/client and measurement programs for each protocol by using template files and the LibHaru documentation. This is described in chapter 5.

The generated measurement programs executes the service with the provided test cases. After execution, it creates performance data files. This is described in chapter 6.

The analysis tool uses the performance data to create graphs and charts. This is described in chapter 7.

The server/client templates consists of a client written in Erlang and a server
written in both Erlang and C. They are connected with 0MQ and this is described in the sections below.

4.1 The 0MQ connection

The 0MQ library used for this service is the official native Erlang 0MQ implementation. This means there are no bindings between the original C implementation of 0MQ and the Erlang library, thus making any latency minuscule. Bindings provide a middleware to bridge libraries written in a language different from the program.

The messaging pattern chosen for 0MQ is Request and Reply as it implements RPC(remote procedure call). This is crucial in determining the time it takes to send and receive a message. RPC calls functions with a mandatory return response, ensuring data reliability.

The complete flow, of how a remote call is sent from a client process to the server and finally to LibHaru for PDF-creation, is described in the sections below.

4.2 The client

![Figure 4.2: Client message flow](image)

Figure 4.2 shows how the message is converted through the different steps from the initial call of the client node. First, the message is in normal Erlang terms (namely X in the figure) that is sent to the encoder module for serialization. The encoder module is automatically generated for each protocol, this is described in chapter 5 below. Second, the serialized message (Y) is sent to the 0MQ module and third, by using the 0MQ REQ-socket the message is sent through the network in a form of a 0MQ-message (Z).

4.3 The server

As there are no Erlang implementation for the LibHaru library, a binding was written between the C library and the Erlang program. This was done by creating an Erlang port [4]. An Erlang port is a binding using communication to and from a separate process initiated by the Erlang program and is in a persistent state until the port is terminated.
4.3. THE SERVER

The port is used to read messages from the Erlang program and convert it to C data types. The LibHaru specific data types are contained within the C side and are not sent through the port.

![Figure 4.3: Server message flow](image)

Figure 4.3 shows the server side of the message conversion, although this flow is opposite of the client version as it needs to restore the message back to normal Erlang terms for processing. The server uses a REP-socket to receive the message (Z) and extract the serialized message (Y). By then using the same automatically generated module it can this time decode the stream of bytes in to Erlang terms that can later be sent to the PDF-processing module. After the PDF-function is called and the PDF has been edited, an acknowledgement is then sent back to the client insuring it that the message has been fully processed.
Chapter 5

Code generation

This chapter explains in detail how code for this service is generated using the LibHaru API documentation.

5.1 Overview

To use the LibHaru library, a binding for each function needs to be written. An Erlang module, that can both encode and decode messages, for each protocol is also in need. To write this manually is very time-consuming and should therefore be automated. The purpose of using the complete LibHaru library is to have a wide variety of data types. As mentioned in previous chapters, this is important to have a fair test for all the protocols in both time and size. It is also crucial to have an unbiased code that is completely generated. Another purpose of this is to have a dynamical system in case of future changes in the experiment that requires changes in the code. Figure 5.1 illustrates the chosen approach.

A structured API specification file is created by using the LibHaru documentation. This is then used to generate the POD-service code for all protocols. These parts are described in detail in the sections below.

5.2 LibHaru API documentation

The documentation of the LibHaru library can be cloned locally via Git [9]. This is done by the following bash command:
The documentation consists partly of API documentation files that all have the same pattern of the function specification, making it easy for a program to extract the specifications from these files. An example of this is as follows:

```c
HPDF_New (HPDF_Error_Handler user_error_fn, void *user_data);
```

```c
HPDF_Page_TextOut (HPDF_Page page, HPDF_REAL xpos, HPDF_REAL ypos, const char *text);
```

The examples above are used in the following sections to show how they get transformed to C code for the LibHaru binding and to all the encoding and decoding functions for all the chosen protocols.

All functions in the LibHaru documentation are specified between the delimiters as seen above. To extract these snippets, the text editor Sublime Text [27] was used, as the editor provides a search function for regular expression with an option of selecting all the search results. By using the search expression `\s\S*?<`, the editor was able to select all the function specification which where copied into a new file with the text `<?xml version="1.0" encoding="utf-8"?><wrapNode>` before and `</wrapNode>` after, making it into an XML-file for further processing. The tool that processes this file is described in the next section.

### 5.3 Structured API specification

The XML documentation of LibHaru is processed in a tool that was written in C#. The program reads the XML-file and extracts all the data within the XML-elements with the tag name of `pre` that contains the function specifications. The content within these elements are processed further by splitting the text with white space delimiters to retrieve the return type, function name and arguments. Given the example in the previous section, the return type `HPDF_STATUS` and the function name `HPDF_SaveToFile` is extracted from the text by splitting it with white space delimiters. The arguments are then extracted by searching for everything between parentheses and splitting it with a comma delimiter.
The information of the functions are then saved in yet another XML-file, making XML the language natural specifications of the messages i.e. the input of the code generating program. The reason for this extra step is to give the freedom to manually change the API specification in an easy way before generating the code. The file is stored under the name of api-spec.xml and given the function `TextOut`, the XML-file is constructed according to this structure:

```
<api>
  <function name="New" returnType="HPDF_Doc" handle="Error_Handler"/>
  <function name="TextOut" returnType="HPDF_STATUS" handle="Page">
    <arg name="xpos" type="HPDF_REAL" />
    <arg name="ypos" type="HPDF_REAL" />
    <arg name="text" type="const char*" />
  </function>
  ...
</api>
```

As the first argument of each LibHaru function is a LibHaru specific handle, it is removed from the argument list and added as an attribute to the XML-element as no handle is sent by the service but is instead declared, assigned to and used within the Erlang/C port. The XML-file `api-spec.xml` is used to generate complete client/server code for each protocol, which is described in the next section.

### 5.4 Generated code

The tool for generating code from the structured API specification was written in C#. The input of this tool is a folder with the structure as follows:

```
|input
|--common
|----client
|----server
|----protocol-specific
|----asn1
|------erl-client-asn1
|------erl-server-asn1
|----bert
|------erl-client-bert
|------erl-server-bert
|----json
|------erl-client-json
|------erl-server-json
|----protobuff
|------erl-client-protobuff
```
Within the folder input the XML-file api-spec.xml is placed and is the first file the tool reads. It extracts all the XML-elements with the tag name function and stores all the information as a list of C# objects.

After this, the tool creates a folder named output and copy all the content of the input folder protocol-specific, thus creating a base for the folders of the generated code. The input folders with the prefix erl-client and erl-server all contain scripts for downloading related libraries and building code specifically for client/server/protocol combination, i.e. specific Makefiles.

The tool then distributes all the content from the input folders common/client and common/server to the client/server protocol specific output folders. These folders contain the common source code for the client and server framework, described in section 4.2 and 4.3 above.

After folder setup, the tool starts generating code in respect to the API specification. The files that are generated from the program are:

- port.c
- pdf_writer.c
- pdf_writer.h
- Multiple versions of protocol_handler.erl for each protocol
- commands.proto
- AsnOnePDFMessages.asn

The generated content of these files are described in the sections below.

5.4.1 Port code generation

The generated code of the Erlang/C port is on the C-side and is the code that interprets the messages and redirects the function calls to the specific LibHaru functions depending on the message from the Erlang program. The port code contained in the file port.c reads the messages from the standard input as stream of bytes and by using the C library Erlang interface [2], the port is able to decode the bytes to C data types. All Erlang messages share the same structure as the decoding requires the knowledge of the message structure. All messages are tuples with an atom as the first element representing the LibHaru function to call and the rest of the elements are the arguments for the function. An example of such a message is {textOut, 100.1, 200.2, "Hello World!"} which prints the text “Hello World!” at the position (100.1,200.2).
The port first extracts the name of which function to call and then compares the name to different if-statements for further decoding of the message, as follows:

```c
... if (!strcmp("new", command)) {
    PDFWRITER_New();
} else //--- HPDF_Page_TextOut ---/
if (!strcmp("textOut", command)){
    double xpos;
    ei_decode_double(buf, &index, &xpos);
    double ypos;
    ei_decode_double(buf, &index, &ypos);
    char text[60000];
    ei_decode_string(buf,&index, text);
    PDFWRITER_TextOut(xpos,ypos,text);
} else //--- HPDF_Page_Stroke ---/
...```

The variable `command` is a `char[]` and contains the name of the atom in first element of the tuple. This code is automatically generated based upon the structured API specification (`api-spec.xml`). The values are decoded according to the data type of the arguments specified in the XML-file.

After fully decoding all the arguments from the Erlang message, the port calls functions stored in the automatically generated file `pdf_writer.c` and `pdf_writer.h` where the former contains the content of the functions and latter contains the function prototypes.

In the example above the functions `PDFWRITER_TextOut(xpos,ypos,text)` and `PDFWRITER_New()` are called, which have the following content:

```c
... void PDFWRITER_New()
{
    current_Pdf = HPDF_New(error_handler, NULL);
}
void PDFWRITER_TextOut(float xpos, float ypos, const char* text)
{
    HPDF_Page_TextOut(current_Page,xpos,ypos,text);
}
...```

The variable `current_Page` is the handle for the current page and is only used in the file `pdf_writer.c` as this is a LibHaru specific data type and is not in the scope of this thesis to use within Erlang messages. The available LibHaru handles stored in the port are `error_handler`, `current_Pdf`, `current_Page` and `current_Font`. If a function takes the handle `HPDF_Font`, as in the function
HPDF_Page_SetFontAndSize(HPDF_Page, HPDF_Font, HPDF_Real), it will be treated as a string until the final step as it will assign the handle current_Font with the desired font and call the LibHaru function as follows:

```c
void PDFWRITER_SetFontAndSize(const char* font, float size)
{
    current_Font = HPDF_GetFont (current_Pdf, font, NULL);
    HPDF_Page_SetFontAndSize(current_Page, current_Font, size);
}
```

### 5.4.2 Erlang module template

A generated Erlang module for encoding and decoding have the same API, regardless of which protocol the module uses. This is to make it exchangeable without affecting the external code that calls the encoding and decoding functions.

After automatic generation, the module is saved under the name of `protocol_handler.erl` and contain the following functions:

- **name**, this returns the name of the protocol
- **init**, if needed, this function initializes the protocol implementation
- **encode_message**, this function encodes from Erlang messages to a stream of bytes ready to be sent through the network
- **decode_message**, this function decodes serialized messages to Erlang messages

### 5.4.3 Protocol buffers code generation

To decode a Protocol Buffers message from a stream of bytes, the decoder must know which message the bytes correspond to in the `.proto` message. As this service is in need of a message that represent one of several different functions, this creates a problem. One solution, as described in Google’s official Protocol Buffers developer website [18], is to create a union type with one `.proto` message type that can include one of many other `.proto` message types. This means the generated `.proto` file has the following, structure:

```proto
message Functions
{
    optional New new = 1;
    optional TextOut textout = 2;
    ...
}
message New
{
}
```
message TextOut
{
    required float xpos = 1;
    required float ypos = 2;
    required string text = 3;
}

As seen above, the .proto message type that should be sent between the client and server is the Functions message type. The message names in this case correspond to the function names and the fields within them are the arguments. This description is automatically generated and stored with file name commands.proto. The official implementation library of Google Protocol Buffers is written in C++, JAVA and Python. As a result, the chosen encoder and decoder library is a third party Erlang library. The library include functions for scanning .proto files and Erlang code generation for encoding and decoding specifically for the given message specifications. It also generates an Erlang header file for representing the data types in Erlang syntax, they are represented as records and the header file include the records specifications. The generated Protocol Buffers version of protocol_handler.erl uses these records to encode and decode messages to and from Erlang messages as the generated code below shows:

... encode_message({new})->
    commands_pb:encode(#functions {new = #new{}});

encode_message({textOut, Xpos, Ypos, Text})->
    commands_pb:encode(#functions {
        textout = #textout
            {xpos = Xpos,ypos = Ypos,text = Text}});
...

5.4.4 Bert code generation

For encoding and decoding BERT-messages, the official Erlang implementation was used. As BERT extends the Erlang term format with more complex data types, it is fully compatible with the Erlang messages used in this service. This means that a generated Erlang module of the BERT implementation is not needed, although it is still created to give a fair pattern matching latency accross all protocols. The following is the generated code of protocol_handler.erl for BERT:

... encode_message({new})->
    bert:encode({new});
encode_message({textOut, Xpos, Ypos, Text})->
bert:encode({textOut,Xpos,Ypos,Text});

... decode_message(SerializedData)->
  Message = bert:decode(SerializedData),
  case Message of
    {new}->
      {new};
    {textOut, Xpos, Ypos, Text}->
      {textOut, Xpos, Ypos, Text};
  end;

5.4.5 ASN.1 code generation

Similarly to the Protocol Buffers code generation described in section 5.4.3, a union type has been used within the IDL description file of ASN.1. The structure of the description file is generated in the following way:

AsnOnePDFMessages DEFINITIONS AUTOMATIC TAGS ::= BEGIN
  Functions ::= SEQUENCE {
    new New OPTIONAL,
    textOut TextOut OPTIONAL,
    ...
  }

  New ::= SEQUENCE {
  }

  TextOut ::= SEQUENCE {
    xpos REAL,
    ypos REAL,
    text PrintableString
  }

Erlang built-in functions were used as the library for implementing ASN.1 [1]. Using the built-in library, the description is loaded and specific encoding and decoding functions are generated. The generated files are an Erlang module and a header file. The module contains functions for encoding and decoding and the header file include record specification to represent the messages.

The generated version of protocol_handler.erl for ASN.1 uses the records provided and has the following structure:

encode_message({new})->
  {ok,Bin} = 'AsnOnePDFMessages':encode('Functions',#{

...
5.4. GENERATED CODE

new = #'New'{}
Bin;

encode_message({textOut, Xpos, Ypos, Text})->
{ok,Bin} = 'AsnOnePDFMessages':encode(
'Functions',#'Functions' {
  textOut =#'TextOut'{
    'xpos' = erlang:float_to_list(Xpos,[{decimals,2}]),
    'ypos' = erlang:float_to_list(Ypos,[{decimals,2}]),
    'text' = Text}}),
Bin;
...

5.4.6 JSON code generation

The chosen JSON encoder and decoder is a third party library called Mochiweb [29]. The function TextOut is encoded in the following JSON message structure:

{"funName":"textOut", "xpos":"120", "xpos":"300", "text":"Hello!"}

As seen above, the key funName denotes the desired function and the rest of the key/value pairs denote the arguments. The code generator generates the following code for creating the JSON messages:

... encode_message({new})->
erlang:iolist_to_binary( mochijson2:encode('{{funName,new}}'));
encode_message({textOut, Xpos, Ypos, Text})->
erlang:iolist_to_binary(
  mochijson2:encode('{{funName,textOut},
    {xpos,Xpos},
    {ypos,Ypos},
    {text,erlang:list_to_binary(Text)}}')));
...

When decoding the serialized message, the conversion to Erlang messages is as follows:

Message = mochijson2:decode(SerializedData),
case Message of
  {struct,[[<<"funName">>,<<"new">>>]}->
    {new};
  {struct,[[<<"funName">>,<<"textOut">>>],
    {<<"xpos">>,Xpos},{{"ypos">>,Ypos},
    {<<"text">>,Text}}}->
    ...

5.4.7 XML code generation

The chosen XML structure is the following:

```xml
<fun name = "textOut">
  <arg name = "xpos">200.0</arg>
  <arg name = "ypos">400.0</arg>
  <arg name = "text">Hello World!</arg>
</fun>
```

The chosen XML encoder and decoder is XMERL [3] and is an extension to the built in Erlang library. The library expresses the XML structure in normal Erlang terms and can encode as the example code below:

```erlang
encode_message({new})->
  xmerl:export_simple([{\{'fun',[{name,'new'}], []}\}], xmerl_xml);
encode_message({textOut, Xpos, Ypos, Text})->
  xmerl:export_simple([{\{'fun',[{name,'textOut'}], [\]
    {arg,[\{name,xpos\}], [io_lib:format("-p","{Xpos}\])}],
    {arg,[\{name,ypos\}], [io_lib:format("-p","{Ypos}\])}],
    {arg,[\{name,text\}], [Text]]}\}], xmerl_xml);
```

The XML can later be decoded by using Erlang records as follows:

```erlang
case FunName of
  "new"->
    {new};
  "textOut"->
    [Xpos_xmlElement,Ypos_xmlElement,Text_xmlElement] = Children,
    [Xpos_xmlText] = Xpos_xmlElement#xmlElement.content,
    [Ypos_xmlText] = Ypos_xmlElement#xmlElement.content,
    [Text_xmlText] = Text_xmlElement#xmlElement.content,
    {textOut,erlang:list_to_float(Xpos_xmlText#xmlText.value),
     erlang:list_to_float(Ypos_xmlText#xmlText.value),
     Text_xmlText#xmlText.value};
```

...
Chapter 6

Test and measurement programs

This chapter explains how test are done and what is measured, for both the server and the client.

6.1 Overview

To study the performance of the protocol description languages, stress test and measurements functions were implemented. These functions are included in both the server and client code for different measurements. This is illustrated in figure 6.1.

![Figure 6.1: Test overview](image)

The test cases are text files with Erlang formatted messages that correspond to the LibHaru functions. As seen in figure 6.1, the test cases are used in the client code to transmit the instructions over the network to the server. Both the client and the server outputs performance data from the measurements.
Time measurements in Erlang can be made in two ways by either using the \textit{runtime} (CPU-time) or the \textit{wall clock} (operating system time). The former measures the calculation time of the Erlang virtual machine while the latter measures the actual time using the operating system time.

What measurements are done and how the data is stored, is described in the sections below.

### 6.2 Measurements on the client side

To start the stress test and measure the service, the function \texttt{client:test/2} is called. This function takes two arguments. The first argument is a list of filenames that directs to the test cases and the second argument is an integer of how many times the test cases will run.

The function first loads the test cases and stores all the Erlang messages, from each test case, in a list. This list is then sent sequentially to the server as many times as provided. For each message sent, the client measures the following two factors:

- The time to encode a message
- The time to send a message and receive an acknowledgement (round trip time)

The measurements of the encoding time is done by calling the built-in function \texttt{timer:tc/3} which returns the number of microseconds for a complete function call using runtime measurements. Using runtime measurements is of course important here as we want to measure the calculation time of the encoding procedure. The function that is measured is \texttt{protocol_handler:encode/1}. This function is, as described in section 5.4, automatically generated for each protocol.

The measurements of the round trip time is done by using the built-in function \texttt{os:timestamp/0}. A time stamp is stored before a message is sent to the server and after an acknowledgment is received. The time difference between the time stamps is measured by the function \texttt{timer:now_diff/2}. This measurement uses wall clock. Using runtime measurement is not applicable as this client side measurement will mainly wait for the server to respond.

After the instruction list has been successfully sent to the server, the client saves all the measurements in two files. Each file include one type of performance data. The performance data files all share the same structure, the following shows an example of this:

\begin{verbatim}
Protocol name: bert
Data content: encoding time
Date: 29/7
Time: 17:01:15
Data:
[6,14,5,7,3,8,6,4,7,3,6,4,6]
\end{verbatim}
The example file above have the file name 729_170115_bert_encoding.pd. The name consists of the time the test was finished and the type of measurement, followed by the file name extension .pd. For the same test that produced the example above, another file with the name 729_170115_bert_roundTrip.pd is created. This file has the following content:

Protocol name: bert
Data content: round trip time
Date: 29/7
Time: 17:01:15
Data:
[283,333,199,221,231,615,347,263,341,232,240,229,415]

6.3 Measurements on the server side

The server is started by calling the function podserver:start which makes the server wait for any incoming messages. When a message is received, the server application measures the following factors:

• The size of the serialized message

• The time to decode a message

If the message is an instruction to add a page, the program will also measure the time between add page messages.

The size of the serialized messages is calculated by the built in function erlang:byte_size. The time to decode is measured in the same way as the client measures the encoding time, by the function timer:tc/3. The time between add page messages are calculated by calling the built in function erlang:statistics/1, with the argument wall_clock. This measures the time in milliseconds between each call to the function. This is called each time the server get an add page message.

The server will, after a test, produce three performance data files. They will contain data for the messages size, time to decode messages and the time between each created PDF page.

6.4 Test data sets

Three data sets were created, they represent:

• A complete novel (contains 441 messages)

• 10 pages of geometrical figures and text snippets (contains 207 messages)

• An invoice with text and a bar-code (contains 218 messages)
These data sets were chosen to represent the typical use cases and have a variety of data types to send from client to server. The focus of chosen data sets were the context and not the number of messages. As the first data set is a novel, it mostly consists of text. The second data set consists mostly of messages with numeric values. The last data set is a mixture of the different data types as the invoice both include text and graphical line segments. The invoice data set also include many function calls without parameters for switching modes between drawing and writing text.

An example of such a file is the following:

```
{new}.
{addPage}.
{setFontAndSize,"Helvetica-Bold",20.0}.
{beginText}.
{textOut,400.0,770.0,"Hello World!"}.
{endText}.
{saveToFile,"Hello_world.pdf"}.
```
Chapter 7

Analysis tools

This chapter describes how performance data is used to create charts.

7.1 Overview

Figure 7.1 illustrates an overview of the procedure for visualizing performance data.

The performance data files, as described in chapter 6, are files from the measurements of the test. The analysis tool uses the data files to produce an HTML-file that, when run in a web browser, renders the chart. This is described in the section below by presenting an example of the whole processes.
7.2 From data to charts

The tool for reading the performance data files and generating charts is written in visual C#. It starts by listing all the performance data files and gives the user the ability to chose the desired files to include in the visualization. It also gives the option add labels to the visualization such as axes name and title. The program can then read the data and create an HTML file with the given configuration by the user.

An example

Imagine we want to visualize the following two performance data files:

Protocol name: json
Data content: packet size
Date: 30/8
Time: 21:29:19
Data:
[17,21,64,40,50,43,71,20,23,66,64,64,40,62,65,68,59,67,60]

The test bench will save the file depending on the name of the protocol, the performance metric as well as the time and date of creation. This performance data will be saved under the name of 830_212919_json_size.pd.

Protocol name: xml
Data content: packet size
Date: 30/8
Time: 21:30:20
Data:
[38,42,118,81,121,84,157,41,44,135,133,118,81,131,134,137,113,136,129]

Similarly, this file will be saved under the name of 830_213020_xml_size.pd.

If these files are located next to or within a sub directory of the data reader tool, they will be listed when the program starts as the tool will search for the file name extension .pd. The user can then select these files to add in the visualization and command the tool to read the data. The final step is to save the data as an HTML file that, when launching it in a web browser, will generate the chart. The HTML files that are generated uses a JavaScript library called D3.js [10] to render charts. The charts are generated as vectorized SVG images and rendered in the web browser.

Depending on the configuration set by the user, the chart is generated in different ways. The tool offers a visualization of the data as a line chart, a bar chart, or an error bar chart with both average value and standard deviation. Line charts and bar charts are not used in this thesis and therefore only the error chart is shown below. Using the example data above, the figure below is generated:
The figure above is a visualization of the average and the standard deviation of all the measurements for each protocol. Notice how XML average is on 100%. This is because the tool uses XML average as reference point for all values. The charts are saved as an SVG-file and are directly inserted in this thesis without any post modification.

The user has also the option of grouping the performance files into two groups and presenting the result as a scatter plot. This is useful when analyzing the correlation between two performance metrics. The figure below is an example of a scatter plot:

![Figure 7.2: Error chart example, XML average as reference point](image)

**Figure 7.2:** Error chart example, XML average as reference point

![Figure 7.3: Scatter plot example, XML average as reference point](image)

**Figure 7.3:** Scatter plot example, XML average as reference point
As seen in the example figure above, the chart combines two metrics and we can easily see correlations.

Using the different visualization techniques, one can analyze the data in different ways. It is also important to remember that when using a tool for automatic graph generation, mistakes such as combining the wrong protocol name and average performance miscalculations are omitted.
Chapter 8

Empirical evaluation

The first part of this chapter describes the chosen test design and the methods for the evaluation. The second part presents the results of the tests.

8.1 Test design

The test is to empirically evaluate all the chosen protocols. By alternating the different protocols running the same test design and given the same data, one can measure slight differences in performance. The three data sets are described in section 6.4. The chosen test design is the following:

1. For each protocol send each data set 10 times
2. For each protocol send each data set, with compressed messages, 10 times
3. For each protocol send all data sets sequentially 10 times
4. For each protocol send all data sets sequentially, with compressed messages, 10 times

The reason for sending the data 10 times is to stress the system for a better benchmarking.

How this is done is described in the section below.

8.2 Test implementation and setup

The server is started directly after compilation with the bash command `make` that initiates the specific `Makefile`. By calling the command `make zipped` the server that uses zlib compression is instead run.

A test is initiated by calling the function `client:test/2` or `client_zipped:test/2` for the client that uses zlib. Given the test design in section 8.1, the specific function calls were according to the following list:
1. For each protocol, start the server with the bash command `make`. On the client side, do the following:
   a) Run `client:test(["doc1"],10).` This will send the first data set (a novel) 10 times.
   b) Run `client:test(["doc2"],10).` This will send the second data set (geometrical figures) 10 times.
   c) Run `client:test(["doc3"],10).` This will send the third data set (an invoice) 10 times.

2. For each protocol, start the server with the bash command `make zipped`. On the client side, do the following:
   a) Run `client_zipped:test(["doc1"],10).` This will send the first data set (a novel) 10 times with compressed messages.
   b) Run `client_zipped:test(["doc2"],10).` This will send the second data set (geometrical figures) 10 times with compressed messages.
   c) Run `client_zipped:test(["doc3"],10).` This will send the third data set (an invoice) 10 times with compressed messages.

3. For each protocol, start the server with the bash command `make` and on the client side run `client:test(["doc1","doc2","doc3"],10).` This will send all three data sets sequentially, according to the given list, 10 times.

4. For each protocol, start the server with the bash command `make zipped` and on the client side run `client_zipped:test(["doc1","doc2","doc3"],10).` This will send all three data sets sequentially, according to the given list, 10 times with compressed messages.

The computer and environment that ran all the tests have the following specifications:

- Erlang OTP version: 17 RC2
- Operating system: Ubuntu 12.04 64bit
- Memory: 8GB
- CPU: Intel Core i5-3427U CPU @ 1.80GHz x 4

The performance data were combined in the chart generator, described in chapter 7, and the resulting charts are presented in the section below.
8.3 Test results and analysis

The test results are categorized under the specific measurements in the following five sub sections. For each measurement, four charts are presented; the results for running each data set separately and for running all the tests sequentially. The data presented is the mean value and the standard deviation of the performance data in form of error bar charts. All values have XML mean as reference point and are presented in percent instead of the metric unit. Due to the difference in performance between the three data sets, the results of the test run for the combined data sets (test 3 and 4 in Test design, section 8.1), are not presented in this section but can be viewed in appendix A. As described in 6.4, the data sets contain 441, 207 and 218 messages respectively. The number of messages are sent 10 times thus creating measurements of 4410, 2070 and 2180 data points. Keep in mind that the following graphs are generated from the raw data of the measurements and are not modified but instead directly imported to this report.

8.3.1 Encoding performance

The encoding procedure is executed before compressing the message with zlib. It is therefore not relevant to show encoding performance of the zlib version of the test. The following charts presents the results:

![Graph](image)

Figure 8.1: Time to encode, novel data set, 4410 data points, XML average as reference point
Figure 8.2: Time to encode, graphics data set, 2070 data points, XML average as reference point

Figure 8.3: Time to encode, invoice data set, 2180 data points, XML average as reference point

In figure 8.1, the time to encode the novel data set is faster for the binary-based protocols (Protocol Buffers, BERT and ASN.1) than the text-based protocols (JSON and XML). It may seem strange that BERT has a negative lower bound value. This is a result of few relatively high values in the RAW data that will lead to high standard deviation and when applied to the lower bound it becomes extremely low. As these large amount of data points are just a multitude of the same data set, this phenomenon still occurs.
Protocol Buffers performed better than all other protocols, with both lower mean value and standard deviation. Although the standard deviation of the protocols overlap and the lower bound for BERT, ASN.1 and JSON are lower than the lower bound of Protocol Buffers.

This pattern changes when encoding the graphics data set, see figure 8.2. BERT is instead the best performer with both lowest mean and standard deviation. Approximately 20 percent units faster than Protocol Buffers.

The pattern changes once again when looking at figure 8.3, encoding performance for the invoice data set. In this chart, we can see that ASN.1 is slower than the text-based protocol JSON. A natural thought would be why the chart of encoding the invoice data set is not a combination of the former two chart patterns. It is important to remember that this data set is not only a combination of text and graphics, but includes also many function calls with no parameters. This is described in section 6.4. There are many factors contributing to the order of performance and the change of the performance between the data sets. It is therefore important to remember that the specific implementation of the protocol encoding and decoding procedure play a role in these measurements. The reason ASN.1 is slower than JSON when encoding the invoice data set could be a result of the implementations of the two encoders leading to a slower performance of ASN.1 for these kind of messages. BERT is the best performer in this measurement as well. For this data set, the standard deviation of BERT is not overlapping the standard deviation of Protocol Buffers.
8.3.2 Decoding performance

Similarly to the encoding performance results, it is not relevant to present the results from the tests using zlib. The following charts presents the results:

**Figure 8.4:** Time to decode, novel data set, 4410 data points, XML average as reference point

**Figure 8.5:** Time to decode, graphics data set, 2070 data points, XML average as reference point
Figure 8.6: Time to decode, invoice data set, 2180 data points, XML average as reference point

We can observe from the figures above that XML have a dire performance change compared to the encoding performance and the other protocols. There are many contributing factors to these XML decoding results. It could for instance be due to the implementation of the XML decoder and the usage of XML for this service (see section 5.4.7). It takes more steps to extract the desired value, after the XML document have been converted to an Erlang data structure, than other protocols.

We can observe from figure 8.4 that decoding the novel data set follows the same performance order pattern as when encoding the same data set, see figure 8.1. The binary-based protocols performs on average better than the text-based protocols.

In both figure 8.5 and 8.6, the performance order changes as ASN.1 on average decodes slower than JSON for the graphics and invoice data set. As stated in section 8.3.1, the performance is affected by the used protocol as well as the used implementation.
8.3.3 Size

The following charts show the average packet size for each protocol and data set:

**Figure 8.7:** Byte size, novel data set, 4410 data points, XML average as reference point

**Figure 8.8:** Byte size, graphics data set, 2070 data points, XML average as reference point
### 8.3. TEST RESULTS AND ANALYSIS

#### Table 8.3: Size comparison in percent of data sent with different protocols. Percentage is calculated based on XML average size as reference point.

<table>
<thead>
<tr>
<th>Protocol/Zip</th>
<th>Percent</th>
</tr>
</thead>
<tbody>
<tr>
<td>protobuf</td>
<td>33.07</td>
</tr>
<tr>
<td>protobuf_zip</td>
<td>22.67</td>
</tr>
<tr>
<td>bert</td>
<td>32.72</td>
</tr>
<tr>
<td>bert_zip</td>
<td>22.39</td>
</tr>
<tr>
<td>asn1</td>
<td>34.82</td>
</tr>
<tr>
<td>asn1_zip</td>
<td>22.07</td>
</tr>
<tr>
<td>json</td>
<td>38.80</td>
</tr>
<tr>
<td>json_zip</td>
<td>27.18</td>
</tr>
<tr>
<td>xml</td>
<td>69.30</td>
</tr>
<tr>
<td>xml_zip</td>
<td>50.77</td>
</tr>
</tbody>
</table>

#### Figure 8.8: Byte size, invoice data set, 2180 data points, XML average as reference point

In figure 8.7 we can see that regardless of protocol, there is a benefit of compressing the novel data set packets. This is of course normal as the data sent using this data set is textual and most likely have repeated occurrences of text strings. As strings are saved similarly for of the tested protocols, the difference is instead in the syntax. We can also see that binary-based protocols on average produced smaller packets than the text-based protocols. Protocol Buffers produced the smallest packets and the compressed version of these were the most optimal for this test.

Figure 8.8 shows the serialized message size of the graphics data set. Further compression of these packets makes a better difference for text-based protocols. This is normal as the text-based protocols uses textual tags with repeated occurrences. As the numbers in the packets are also represented in text, repeated occurrences of symbols in the messages will most likely happen.

When compressing the binary-based protocols with ZLib, the results are either no significant difference (as with BERT) or a difference to the worse (as with ASN.1 and Protocol Buffers). This is because the messages of Protocol Buffers and ASN.1 are so small that when using compression, the packets becomes bigger in size from the ZLib format header and checksum.

In figure 8.9, we can observe that further compressing the invoice data set follows the same pattern discussed above. If comparing figure 8.9 and figure 8.8 we can see that BERT benefits more from ZLib compression. This is because the invoice data set also include textual messages instead of mostly numbers as in the graphics data set.

[Figure 8.9: Byte size, invoice data set, 2180 data points, XML average as reference point]
8.3.4 Round trip time

The round trip time is the time between sending the serialized message from the client side and receiving an acknowledgement from the server as described in section 6.2. The charts below present the results:

**Figure 8.10:** Round trip time, novel data set, 4410 data points, XML average as reference point

**Figure 8.11:** Round trip time, graphics data set, 2070 data points, XML average as reference point
8.3. TEST RESULTS AND ANALYSIS

The round trip time in this test is deeply affected by the decoding time and the size of the messages. This is because the time measurements of this metric include the decoding time and the time to send a message from the client to the server.

We can from figure 8.10 see that for the novel data set, only ASN.1 benefits from the ZLib compression. In figure 8.7, all the messages on average became smaller in size when further compressed regardless of used protocol. The somewhat unexpected results in figure 8.10 could be due to the latency of decompressing the messages with ZLib, making the further compressed versions less preferable. Most likely, these results would change if the service were instead tested in a non local network as the size would matter more than the decompression time. As Protocol Buffers decoded the fastest and produced the smallest packets for the novel data set, it is natural that Protocol Buffers have the fastest round trip time.

In figure 8.11, the results of the graphics data set is presented. We can see that the text-based protocols have a shorter round trip time when using ZLib compression. This is of course expected as both JSON and XML messages could be further compressed (see figure 8.8). As for the binary-based protocols, the ZLib compression is not preferable because the messages size is larger in this data set and the time to decompress messages is also included here. BERT and Protocol Buffers have the shortest round trip time. This is because BERT have the fastest decoding time and Protocol Buffers produce the smallest packets for the graphics data set.

In figure 8.12 we can observe that measuring the round trip time of the messages in the invoice data set, ZLib compressed messages have a better impact on the text-based protocols and a negative impact on the binary-based protocols. This is, as stated above, both because the packet size of the binary-based protocols will increase when further compressing these kind of messages and the time to decompress the packets is taken into account. We can also see that messages encoded with ASN.1
have a longer round trip time than JSON. This is probably a result of slow decoding of these kind of messages when using ASN.1.

8.3.5 Overall Performance

The measurements of how much time it takes to generate a PDF page is converted to the unit of pages per second and presented in the charts below. The number of pages of the data sets are 30, 10 and 1 respectively. As they are sent 10 times each, the number of data points becomes 300, 100 and 10. Keep in mind that these chart differ from previous charts as the performance is better when the numbers are higher.

Figure 8.13: Pages per second, novel data set, 300 data points, XML average as reference point
The number of pages generated per second in the test bench is affected by all parts of the service. It is therefore normal that the figures above are a combination of the figures in the previous sections.

In figure 8.13 we can observe that Protocol Buffers on average have the best performance when sending messages from the Novel data set compared to the other protocols used, including the ZLib compressed Protocol Buffers messages. As discussed in section 8.3.3, the performance will likely to change order if the test would
have been made in a network were size would matter more.

In both figure 8.14 and 8.15 the overall performance of the graphics and invoice data set show that BERT is on average the best performer. This is of course natural as BERT was the fastest protocol to encode and decode those type of messages. Again, we see that XML consistently has the worst performance.

8.3.6 Speed and size performance

As mentioned in the introduction, this thesis is mainly interested in the “speed” and “size” of the protocol description languages. These two attributes are in the test bench represented by the byte size of the messages (see section 8.3.3), and the number of pages produced per second (see section 8.3.5). When combining the results of the two attributes in the same visualization, we can get a better overview of the “speed” and “size” performance for each protocol. As we are handling two variables, the chosen visualization of this is to use scatter plots.

The following scatter plots are a combination of the two performance metric results presented in section 8.3.3 and section 8.3.5 for all data sets. Post modification of the graphs has been made. The arrows are combining the protocols from normal performance to performance with ZLib compression.

![Figure 8.16: Size and speed relation, novel data set, XML as reference point](image)

The preferred position of a performance in the scatter plot is the top left corner for fastest service performance and smallest packets. As we can see in figure 8.16, there is no clear winner in both speed and size when taking the ZLib compressed messages into account. An obvious observation is the clustering of two groups in
the scatter plot. The circles left of 60% in size are the ZLib compressed results of the messages and the circles right of 60% in size are the messages without ZLib compression. If we compare these two groups of results we can see that when compressing the messages, the distance in both axises between the protocols becomes shorter. As discussed in section 8.3.3, the size of the packets becomes closer in size when compressing the novel data set as we are compressing text. As the speed attribute is affected by the size, it is natural that speed performance becomes closer between the protocols when further compressing the messages. In both groups, Protocol Buffers has the best performance in both metrics i.e. it has the left and top most position.

Figure 8.17: Size and speed relation, graphics data set, XML as reference point

In figure 8.17, we can see that compressing the messages with ZLib for the graphics data set have a different impact on the performance compared to when sending the novel data set. As the data is encoded in different ways for the graphics data set and the messages include small amount of data, further compression of the messages does not always benefit. As a result, the ZLib versions of the messages are closer in size to the messages for the same protocol rather than closer to the other ZLib compressed messages as seen in figure 8.16.

The impact of further compression varies for the different protocols. For the binary-based protocols, speed decreases when compressed further. In addition to this, Protocol Buffers and ASN.1 encoded messages becomes larger. The text-based protocols will benefit in size when further compressed with ZLib.

Whether further compressed or not, the binary-based protocols have better per-
formance in both speed and size than the text-based protocols. This is visible in the figure as a left and top most clustering.

There is no clear winner for both metrics as Protocol Buffers produce on average the smallest packets and BERT on average performs the fastest. BERT is not necessarily the fastest performer as pages per second is not a reliable measurement as time measurements depend on the environment. Although we with great certainty conclude that Protocol Buffers is the winner in size as this a reliable measurement (size depends on the protocol specification and the data set).

As seen in figure 8.18, the invoice data set is similar in performance order to figure 8.17. This is as mentioned before due to the similarity of the two data sets. However, what differentiates the two results is the poor performance of ASN.1 when dealing with these kind of messages.
Chapter 9

Conclusions

This chapter presents a summary of the work done in this thesis, a conclusion and potential future work.

9.1 Summary

This thesis presented a performance evaluation of various protocol description languages (Protocol Buffers, BERT, ASN.1, JSON and XML) including ZLib compression of the serialized data. Three test data sets were used and the test bench was a print on demand service created for this purpose. The evaluation measured various metrics under the categorization of speed and size. Results were presented and compared first between the protocols for each performance metric and then in relation of pages per second and size in bytes of the serialized messages.

9.2 Conclusion

The questions this work is interested in were presented in section 1.2:

- Which of the packaging mechanisms are the fastest regardless of data set?
- Which of the packaging mechanisms produce the smallest packet size regardless of data set?

The results show that by changing the type of data set the performance outcome changes. When measuring the speed metrics the fastest protocol for producing PDF-pages changes for the different data sets. When measuring the size metric, the messages could be compressed with ZLib for some protocol/data set combinations while others produced larger packets.

As the results were varying in performance one can not conclude any winning protocol for any of the questions asked. Even if the size measurements are reliable
and Protocol Buffers were on average the best performer there is no absolute guarantee it is the protocol that produces the smallest packets for any data set given. The data can be represented in many different ways and when sending long text with Protocol Buffers one could compress it even more with ZLib compression.

Although there are no absolute certainty, one can still reason that the choice of protocol for best performance of either speed or size, highly depends on the specific application at hand. This is due to the affect transmitted data, environment and chosen implementation has on the protocol performance.

Instead of concluding the winner protocol, the results shows which of the protocols are least preferable for the measured metrics. On average, the text based protocols produce bigger messages and are therefore not optimal when size is important. XML was the least preferable protocol description language of all tests.

9.3 Discussion

Given the results of this thesis, a normal question is if the communication services we use daily can be optimized. If speed is desired, the best combination of the environment and protocol is difficult to find for a specific application. One can safely assume that there are services with not the most optimal protocol for speed performance. Of course, some of the protocols have been chosen for other purposes such as being self descriptive or human readable friendly. If these services would change to a faster serialization mechanism, it may not affect the performance of the application as it may not be the bottleneck.

When services are used in a limited network, whether it is in bandwidth or a limited amount of traffic, surely there are benefits in making the packets smaller. If a service changes its communication protocol from XML to Protocol Buffer the average packet size would become smaller, as the results of this thesis suggests. If the transmitted data of this service resembles the graphics data set, the average data transmitted would decrease with around 80%. If the transmitted data instead resembles the novel data set, the average transmitted data would decrease with around 20%. What would this mean for the users with limited network? It would be fair to say it would provide users with low bandwidth a faster experience of the service. For the users with a limited amount of traffic data per month this would allow them to use more of their limited data. For the users with a data plan where cost is calculated on the amount of data being transmitted, one can with great certainty say it will be cheaper for them. If the service is used on mobile phones, a protocol change could affect the phone battery usage in a positive way as discussed in section 2.1. This is of course a great selling point for a service provider. Assume there was a chat service for mobile phones that uses highly compressed messages and another using the common XML protocol without any compression as communication protocol. Will mobile phone users with very limited network choose the former chat service over the latter?
9.4 Future Work

The study completed in this project suggests that considering protocol is vital for a network service, especially when under constraints. For future work, one can study a bigger variety of protocols and investigate the difference between them. As a result of this project, one might also consider studying protocols under different data types, types of services, environments and implementations of protocols. With the right resources, one could create a more intricate statistical model of the results. Thus enabling developers the ability to more accurately choose the protocol most suited for their needs.
References


REFERENCES


Appendix A

All test results

The following sections presents all test results.

A.1 Encoding performance

Figure A.1: Time to encode, novel data set, 4410 data points, XML average as reference point
Figure A.2: Time to encode, graphics data set, 2070 data points, XML average as reference point

Figure A.3: Time to encode, invoice data set, 2180 data points, XML average as reference point
A.2. DECODING PERFORMANCE

Figure A.4: Time to encode, all data sets, 8660 data points, XML average as reference point

A.2 Decoding performance

Figure A.5: Time to decode, novel data set, 4410 data points, XML average as reference point
Figure A.6: Time to decode, graphics data set, 2070 data points, XML average as reference point

Figure A.7: Time to decode, invoice data set, 2180 data points, XML average as reference point
A.3 Size

Figure A.8: Time to decode, all data sets, 8660 data points, XML average as reference point

Figure A.9: Byte size, novel data set, 4410 data points, XML average as reference point
Figure A.10: Byte size, graphics data set, 2070 data points, XML average as reference point

Figure A.11: Byte size, invoice data set, 2180 data points, XML average as reference point
A.4. ROUND TRIP TIME

Figure A.12: Byte size, all data sets, 8660 data points, XML average as reference point

A.4 Round trip time

Figure A.13: Round trip time, novel data set, 4410 data points, XML average as reference point
Figure A.14: Round trip time, graphics data set, 2070 data points, XML average as reference point

Figure A.15: Round trip time, invoice data set, 2180 data points, XML average as reference point
Figure A.16: The average RTT, all data sets, 8660 data points, XML average as reference point

Figure A.17: Pages per second, novel data set, 300 data points, XML average as reference point
Figure A.18: Pages per second, graphics data set, 100 data points, XML average as reference point

Figure A.19: Pages per second, invoice data set, 10 data points, XML average as reference point
A.6 Speed and size performance

Figure A.20: Overall performance, all data sets, 410 data points, XML average as reference point

A.6 Speed and size performance

Figure A.21: Size and speed relation, novel data set, XML as reference point
Figure A.22: Size and speed relation, graphics data set, XML as reference point

Figure A.23: Size and speed relation, invoice data set, XML as reference point
Figure A.24: Size and speed relation, all data sets, XML as reference point