Final Year Project Report

Interoperability of Cloud Monitoring Data

University College Cork

Computer Science Department

Author:
Martin J. Bullman

Supervisors:
Prof. John Morison
Vincent Emeakaroha

July 2, 2016

1 Abstract

Cloud computing enables the provision of services in a scalable and on demand fashion. Current practices for efficient cloud service provisioning management are based on monitoring. Monitoring generates the real-time data regarding the state of the cloud infrastructures and deployed services. This data is crucial for gaining insight and making decisions.

Due to the emergence of numerous monitoring tools for managing heterogeneous cloud platforms, the generated monitoring data are formatted using diverse and mostly incompatible data interchange formats. This results in a lack of interoperability and makes the analysis of monitoring data from multi-cloud service deployments difficult to handle.

Numerous human readable data interchange formats and numerous binary serialization formats were investigated and researched for this project. The best human readable and binary formats were then chosen and used. After a rigorous comparison, analysis and testing of all the chosen formats the result was a new more efficient data interchange format for serializing and structuring cloud monitoring data to enable interoperability and easy management of multi-cloud service deployment.

2 Declaration

Declaration of Originality
==========

In signing this declaration, you are conforming, in writing, that the submitted work is entirely your own original work, except where clearly attributed otherwise, and that it has not been submitted partly or wholly for any other educational award.

I hereby declare that:

- This is all my own work, unless clearly indicated otherwise, with full and proper accreditation;
- With respect to my own work: none of it has been submitted at any educational institution contributing in any way to an educational award;
- With respect to another's work: all text, diagrams, code, or ideas, whether verbatim, paraphrased or otherwise modified or adapted, have been duly attributed to the source in a scholarly manner, whether from books, papers, lecture notes or any other student's work, whether published or unpublished, electronically or in print.

Student Signature	Date

3 Acknowledgements

I would like to express my deepest appreciation to all those who provided me the possibility to complete this project and the accompanying report. A special gratitude I give to our final year project mentor Dr Vincent C. Emeakaroha, whose contribution in stimulating suggestions and encouragement, helped me to coordinate my project especially in writing this report.

Furthermore I would also like to acknowledge with much appreciation the crucial role of the Systems Support Staff of the UCC computer science department and my mentor Vincent, who gave the permission to use all required equipment and the necessary materials to complete the Interoperability of Cloud Monitoring Data project.

Last but not least, many thanks go to the supervisor of the project, Prof John Morrison who invested his full effort in guiding me in achieving the goal of completing this project. I have to also appreciate the guidance and help given by other lecturing staff with in the computer science department here in UCC, which was invaluable to the success of my final year project.

Contents

1 Abstract **i**

2 Declaration **ii**

3 Acknowledgements **iii**

4 Introduction **1**
4.1 The Problem 1
4.2 The Aim of the Project 1
4.3 What was Achieved 2
4.4 Main Beneficiaries 2

5 Analysis **3**
5.1 Project Objectives 3
5.2 Background Research 4
5.2.1 Extensible Mark-up Language (XML) 5
5.2.2 JavaScript Object Notation (JSON) 6
5.2.3 YAML Ain't Mark-up Language (YAML) 7
5.2.4 Comma Separated Values (CSV) 8
5.2.5 Relative Expression Based Object Language (REBOL) 9
5.2.6 Portable Object Notation (PON) 10
5.2.7 REadable Notation (REN) 10
5.2.8 Tom's Obvious, Minimal Language (TOML) 11
5.2.9 Java HashMaps 12
5.2.10 Java ArrayList 13
5.2.11 Binary JavaScript Object Notation (BSON) 14
5.2.12 MessagePack 15
5.2.13 Protocol Buffers 16
5.2.14 Apache Thrift 17
5.2.15 Hessian 18
5.2.16 KRYO 18
5.2.17 Concise Binary Object Representation (CBOR) 19
5.3 Data Interchange Formats Chosen 20
5.4 Requirements Analysis 21

6 Design **22**
6.1 Architecture . . . 22
6.1.1 High Level Architecture . . . 22
6.1.2 System Architecture . . . 23
6.2 Software development Methodology . . . 24
6.3 Main Package . . . 25
6.4 Human Readable Package . . . 27
6.5 Binary Serialization Package . . . 28
6.6 Binary Deserialization Package . . . 29
6.7 External Libraries . . . 30

7 Implementation **33**
7.1 Technologies Used . . . 33
7.1.1 Java Programming Language . . . 33
7.1.2 RabbitMQ Message Broker Software . . . 33
7.2 Main Package . . . 34
7.3 Human Readable Formats . . . 36
7.4 Binary Serialization Package . . . 41
7.5 Binary Deserialization Package . . . 43
7.6 Message Sender Package . . . 45
7.7 Message Receiver Package . . . 46
7.8 Software Testing . . . 47
7.8.1 Unit Testing . . . 47
7.8.2 System Testing . . . 47

8 Results & Evaluation **49**
8.1 Overview . . . 49
8.2 Detailed Evaluation . . . 49
8.3 Results for Small Data . . . 51
8.4 Results From Medium Data . . . 53
8.5 Results From Large Data . . . 55
8.6 Results Conclusion . . . 57

9 Conclusion **58**
9.1 Project Summary . . . 58
9.2 Future Work . . . 58

10 Bibliography **59**

4 Introduction

4.1 The Problem

Cloud computing is the practice of using a network of remote servers hosted on the Internet to store, manage, and process data, rather than a local server or a personal computer. The increasing trend of outsourcing services to cloud providers is changing the way computing power is delivered to enterprises and end users. Massive data centers have been built by cloud service providers like Amazon (AWS) and Microsoft (Azure) to provide the infrastructure for their own cloud implementations.

To monitor the state of the cloud, service providers rely on monitoring data to gain an insight into what's actually happening with in the cloud e.g. server performance, user usage and billing etc. As we now have more and more new monitoring agents becoming available they all use different data formats to represent the data. That in a nutshell is the problem this project is trying to solve. How do we make these different data formats interoperable?

4.2 The Aim of the Project

The main aim of this project is to implement a new platform-independent and efficient data interchange format for serializing and structuring cloud monitoring data to enable interoperability and easy management of multi-cloud service deployment. The newly implemented data interchange format will be integrated into an existing monitoring system to evaluate its performance in an OpenStack cloud platform.

This project is also trying solve the problem of vendor lock-in where companies and users of cloud computing resources are tied to a single cloud service provider. Customer using a product or service cannot easily transition to a competitor's product or service as the costs may be to high. Vendor lock-in is usually the result of proprietary technologies that are incompatible with those of competitors.

4.3 What was Achieved

After implementing the majority of the functional requirements of the project, we had a really general application which could handle multiple different human readable data interchange formats. The software could take monitoring data in the format of JSON, CSV, XML, HashMap and ArrayList and convert it to any other of the five formats. e.g. JSON to CSV or HashMap to XML etc. The intelligence of the application is that it can find out what format the input data is and then work out what is the most efficient format to convert it to before serializing the datawith one of the binary serializers used in this project.

There were six different binary serializers implemented to serialize and deserialize the data, these were MessagePack, JAVA, Hessian, CBOR, BSON and KRYO. The goal here is that we can serialize the data with all of the serializers and use the one that decreased the data down to the smallest size in terms of bytes. The monitoring data could then be sent across a network or message bus in the smallest size possible. therefore reducing the load on the network in terms of bandwidth and cost.

4.4 Main Beneficiaries

Beneficiaries of this software would be any of the cloud service providers who may find the application useful. A federation of cloud service providers may use the application to decrease monitoring data and pass it across a network in binary form to another service provider within the federation.

If cloud A is being heavily utilized and does not have the resources to analyze and process the monitoring data then it could decrease the size of the data, serialize it and send it off to cloud B which has resources available. Cloud B could perform the required analysis and sent the results back to cloud A.

5 Analysis

5.1 Project Objectives

Broadly speaking, interoperability can be defined as a measure of the degree to which diverse systems or components can work together successfully. More formally, IEEE and ISO define interoperability as the ability for two or more systems or applications to exchange information and mutually use the information that has been exchanged.

To be more concrete, in the context of cloud computing and this project, interoperability should be viewed as the capability of public clouds, private clouds, and other diverse systems within the enterprise to understand each other's application and service interfaces, configuration, forms of authentication and authorization, data formats etc. in order to cooperate and interoperate with each other.

The main objectives of the project is to implement a new data interchange format which adheres to the detailed requirements analysis listed below. The application had to be able to run on multiple different platforms without any issues. It needed to be an efficient data interchange format that would have better performance compared other data interchange formats that are already available today. The application had to have the capability to serialize the monitoring data into the smallest size possible which would allow it to be passed to different applications to enable interoperability of cloud services.

5.2 Background Research

Before the implementation could begin, a lot of research had to be done on the various data interchange formats available today. Format's were broken into two categories. The human readable formats and the binary serialization formats. There were numerous different types of human readable data interchange formats available e.g JSON, XML, CSV and a whole lot more. There was also numerous different types of binary serilaization formats available CBOR, KRYO, MessagePack and again a whole lot more.

For this project a lot of time was spent researching as many formats as possible from both categories. This was necessary to give a better understanding of how each format structures and represents the data. But not just that, it would allow the analysis of each format in detail and allow me to distinguish if they were suitable for the requirements of this project or not.

Each format was researched extensively and analyzed carefully to determine the key characteristics e.g. was the format human readable or binary, what data types did it support, how efficient was it at representing the data in regards to other formats, If it was a human readable format what was the performance like during serialization and deserialization, if it was a binary format did it require POJO classes to serialize and deserialize the human readable data. All these questions and more had to be answered for each data interchange format.

Every format that was researched during this phase of the project is listed below along with some of the key characteristics associated with that format and why it might be suitable format to be used in this project:

5.2.1 Extensible Mark-up Language (XML)

XML is a mark-up language that defines a set of rules for encoding documents in a format which is both human-readable and machine-readable. It is defined by W3C XML 1.0 specifications.

Some key characteristics of the XML data interchange format:

- XML allows to clearly separate content from form (appearance).
- XML is text-oriented.
- XML is extensible.
- XML is self-describing.
- XML is universal; meaning internationalization is no problem.
- XML is independent from platforms and programming languages.
- XML provides a robust and durable format for information storage.
- XML namespaces allow for sharing of standard structures.
- XML is a future-oriented technology.

Key points on how XML performs during serialization and deserialization:

- XML serialization includes metadata with every object instance and writes them as element tags.
- XML serialization includes less information about data types as binary serialization.
- XML deserialization requires more de-marshalling costs, as the parsers has to parse XML documents and perform a lot of string comparison and matching and it also has to do a lot of data type casting.
- XML serialization only includes public data members and is an example of shallow serialization, therefore it cannot handle cyclic references.

- The consequences of the mentioned facts are that the XML serialized size is considerably larger than binary serialization and that the performance is inferior.

5.2.2 JavaScript Object Notation (JSON)

JSON is a lightweight data-interchange format. It is easy for humans to read and write. It is easy for machines to parse and generate. It is based on a subset of the JavaScript programming language.

Some key characteristics of the JSON data interchange format:

- JSON's syntax is human readable.
- JSON is lightweight in comparison to XML.
- JSON is estimated to parse up to one hundred times faster than XML.
- JSON specifies how to represent complex data types, there is no single best way to represent a data structure in XML as all data is represented as strings.
- JSON has very good tool support (almost every programming language supports JSON)
- JSON is a better data exchange format compared to XML as it supports different data types and can represent the same data but with a smaller file size.

Key points on how JSON performs during serialization and deserialization:

- When using JSON, we see that the frameworks reduce the serialization time drastically compared to XML.
- JSON serialization appears to give us a gain of 50% to 97% in serialization time.
- JSON serializes a very simple model of key/value pairs, where values can be strings, numbers, Booleans or arrays.
- XML deserialization may sometimes give better performance compared to JSON deserialization depending on the data being deserialized.

5.2.3 YAML Ain't Mark-up Language (YAML)

YAML is a data serialization format that is considered a data serialization format primarily, like JSON. YAML, although not as widespread as XML and JSON, is gaining popularity as an alternative of sorts of JSON. JSON syntax is a subset of YAML 1.2, which means JSON can technically be considered valid YAML. However, YAML cannot be considered valid JSON.

Some of the key characteristics of the YAML data interchange format:

- YAML syntax is human readable.
- YAML supports extensible data types while JSON only supports basic data types.
- YAML includes the ability to self-reference items with in a YAML file using anchors.
- Every JSON file can also be considered a valid YAML file.
- Easy to migrate from JSON to YAML if/when the additional features are required.
- YAML's different syntax options allow data to be presented in different ways
- YAML allows for extremely readable files, but is more complex to generate and parse that JSON.

Key points on how YAML performs during serialization and deserialization:

- YAML is more robust about embedding other serialization formats such as JSON or XML within a YAML file.
- JSON comes out as better for serialization because it is more explicit.
- YAML's supports serializing arbitrary native data structures.

5.2.4 Comma Separated Values (CSV)

A comma-separated values (CSV) file stores tabular data (numbers and text) in plain text. Each line of the file is a data record. Each record consists of one or more fields, separated by commas. The use of the comma as a field separator is the source of the name for this file format.

Some of the key characteristics of the CVS data interchange format:

- This format is the most compact over XML and JSON.
- CSV is human readable and easy to edit manually.
- CSV is simple to implement and parse.
- CSV is processed by almost all existing applications.
- CSV provides a straightforward information schema.
- CSV is faster to handle compared to JSON, YAML and XML.
- CSV can represent the same data as JSON and XML but is smaller in size.
- CSV is considered to be standard format.
- CSV is much more compact compared to XML as you need a start tag and a end tag for every piece of data. In CSV you write the column headers only once.
- CSV is easy to generate.

Key points on how CVS performs during serialization and deserialization:

- CVS has similar performance stats to XML on small data sets but will outperform XML when large data sets are being serialized and deserialized.
- More efficient parsing – This results in a more efficient parsing data compared to XML or JSON.
- Simple base structure to serialize data in by simply using the principles of CSV.

5.2.5 Relative Expression Based Object Language (REBOL)

REBOL is a language that was designed to be human-readable and easy to edit using any standard text editor. To achieve that it uses a simple free-form syntax with minimal punctuation, and a rich set of data types. REBOL data types include URLs, e-mails, date and time values, tuples, strings, tags, etc. REBOL was used as a source of inspiration by the designer of JSON.

Some of the key characteristics of the REBOL data interchange format:

- REBOL strongly influenced the design of JSON.
- REBOL gains its advantage through lightweight domain-specific sub-languages and micro-formats
- REBOL introduces the concept of dialects: small, efficient, domain languages for code, data, and metadata.
- REBOL syntax is human readable.
- REBOL provides a lightweight platform for Internet applications.
- REBOL provides more data types then JSON.

Key points on how REBOL performs during serialization and deserialization:

- Relative expressions, also called "dialects", provide greater efficiency for representing code as well as data, and they are REBOL's greatest strength.
- A simple block of REBOL is simpler and more efficient to send and receive data, since REBOL interpreter can understand many types of data directly.

5.2.6 Portable Object Notation (PON)

PON is a lightweight data-interchange format that builds on the base of the JavaScript Object Notation (JSON) industry standard data-interchange format. A flaw of JSON is that its stream coding is inefficient for certain patterns of data, most notably binary data. PON extends JSON to increase usability and readability while reducing the space cost of encoding binary data without adding complexity. It's a fairly new data interchange format so documentation and examples were non-existent.

Some of the key characteristics of the PON data interchange format:

- The features of PON are a super set of JSON.
- PON syntax is human readable.
- The overall structure of PON is borrowed directly from JSON.
- PON supports some additional data types compared to JSON, being closely coupled with the types of the programming language Java.

Key points on how PON performs during serialization and deserialization:

- PON, supports combinations of text and binary information to be included in the same message.
- PON can handle binary information directly.
- The performance when generating or parsing PON representations is at least comparable to popular, efficient JSON implementations.

5.2.7 REadable Notation (REN)

Ren is a lightweight data-exchange text format. It is programming language independent with familiar conventions. White space separates values, but is not significant beyond that. REN is a very new data interchange format so documentation and examples were non-existent.

Some of the key characteristics of the REN data interchange format:

- Ren is a text-oriented, human readable data format.
- It's much easier to the eyes compared to XML and JSON and at the same time it's at least as powerful.
- Ren provides interoperability between REBOL family (Redbol) and other languages.
- REN has a rich set of data types compared to JSON.

5.2.8 Tom's Obvious, Minimal Language (TOML)

TOML aims to be a minimal configuration file format that's easy to read due to obvious semantics. TOML is designed to map unambiguously to a hash table. TOML should be easy to parse into data structures in a wide variety of languages. Again TOML is a newly released data interchange format which had no documentation or examples of usage available.

Some of the key characteristics of TOML data interchange format:

- It has a fixed set of supported types rather than allowing schemas as YAML does.
- TOML is much closer to YAML in syntax than JSON.
- TOML syntax is human readable
- TOML is designed to map unambiguously to a hash table.
- TOML is much easier to read and edit by hand compared to JSON and YAML.
- TOML should be easy to parse into data structures in a wide variety of languages.

5.2.9 Java HashMaps

Advantages of using Hashmaps as data interchange format:

- The are human readable
- Can store different data types.
- Alows key value access to the data.
- JSON objects used for the conversion of json data too different formats are constructed using java Hash Maps so hashmap performance is better compared to JSON.
- As data size increases the performance of a java hashmap is approaching the performance of CSV which was the best data interchange format.
- Hash maps can be nested so they can represent complex and heavily nested data.
- the get() and put() methods are not synchronized so the have better multitreaded performance during the conversion method.
- since HashMap is a Collection, amongst other things (List etc.) it opens up a wealth of library functions, both native and third party which were helpful when implemting conversion and serialization methods for HashMaps.

Disadvantages of using hashmaps:

- There is the potential for collisions with hashmap keys if data get to large.
- Hashing algorithm does not result in spatial locality. Because hash tables cause access patterns that jump around, this can trigger microprocessor cache misses that cause long delays.
- Occasionally HashMap/Hashtable requires resizing when the original size of HashMap/Hashtable buckets are full. Resizing takes O(n) time as the elements from the previous hashtable/HashMap are transferred to a new bigger Hashtable/HashMap. This is not good for implementation.

5.2.10 Java ArrayList

Advantages of ArrayList:

- They are human readable.
- ArrayList is variable length, which means ArrayList can grow or shrink its size dynamically during runtime.
- ArrayList allow the insertion of primitive types of data like int, float, double, byte, short, long but they must be autoboxed to wrapped type i.e. Integer.
- ArrayList are not strongly typed(Loosely typed) which means they can store different types of data.
- ArrayList directly access an index which improves performance.
- API methods already defind which helped implementing conversion methods to other data formats.
- since ArrayList is a Collection, amongst other things (List etc.) it opens up a wealth of library functions, both native and third party which were helpful when implementing conversion and serialization methods for ArrayList.

Disadvantages of ArrayList as a datainterchange format:

- Data is stored linerly and there is no distiction between where one line of data ends and the next begins e.g when converting CSV to ArrayList it requires extra data to be added to the array to indicate where one line of data ends and the next begins, this impacts data interchange performance.
- If a data entry is added to or removed from an array-based list, data needs to be shifted to update the list can have an impact on preformance.
- In the worst case, for an array-based list with n data entries, an add and a remove takes O(n) time when we are converting from other data interchange formats to ArrayList Format.

5.2.11 Binary JavaScript Object Notation (BSON)

BSON short for Binary JSON, is a binary encoded serialization of JSON-like documents. Like JSON, BSON supports the embedding of documents and arrays within other documents and arrays. BSON also contains extensions that allow representation of data types that are not part of the JSON spec.

Some of the key characteristics of the BSON data interchange format:

- BSON is smaller and faster than JSON.
- BSON stores more metadata in comparison to JSON.
- BSON is faster than JSON when it comes to encoding and decoding.
- BSON adds additional data types unavailable in JSON, notably the BinData and Date data types.
- BSON is lightweight and keeps overhead to a minimum which is important when transferring data over the network.
- BSON is designed to be traversed easily.

Key points on how BSON performs during serialization and deserialization:

- Encoding data to BSON and decoding from BSON can be performed very quickly compared to other data interchange formats like JSON and XML.
- BSON stores binary data directly, avoiding the time overhead and the additional size of base64 encoded text that regular JSON has with binary data.

5.2.12 MessagePack

MessagePack is a computer data interchange format. It is a binary form for representing simple data structures like arrays and associative arrays. MessagePack aims to be as compact and simple as possible.

Some of the key characteristics of the MessagePack data interchange format:

- MessagePack is more compact than JSON.
- MessagPack allows binary data and non UTF-8 encoded strings.
- Compared to BSON, MessagePack is more space efficient.
- Anything that works with JSON will work with MessagePack.
- MessagePack is usually much faster to encode / decode than JSON.

Key points on MessagePack serialization and deserialization performance.

- MessagePack generates bigger payload than Protocol Buffers but depending on the data being serialized you might arrive at a slightly different result.
- MessagePack is designed for efficient transmission over the wire.
- MessagePack provides faster and more robust serialization for agent/-master communications compared to XML and JSON.
- MessagePack-encoded data is about 2/3 the size of JSON.

5.2.13 Protocol Buffers

Protocol buffers are Google's language-neutral, platform-neutral, extensible mechanism for serializing structured data – think XML, but smaller, faster, and simpler. You define how you want your data to be structured once, then you can use special generated source code to easily write and read your structured data to and from a variety of data streams and using a variety of languages.

Some of the key characteristics of the Protocol Buffers data interchange format:

- Protocol Buffers allow you to define simple data structures in a special definition language.
- Protocol Buffers was designed to be smaller and faster than XML.
- Protocol Buffers use very dense data (small output).
- Protocol Buffers are very fast at processing.
- Protocol buffers currently supports generated code in Java, Python, and C++.
- Protocol Buffers have much better documentation/examples over Apache Thrift.

Key points on Protocol Buffers serialization and deserialization performance:

- Protocol Buffers are much simpler that XML for serializing structured data.
- Protocol Buffers are 20 to 100 times faster than XML during serialization.
- Protocol Buffers requires POJO classes to represent the data before serialization.
- Protocol Buffers generate data access classes that are easier to use programmatically.

5.2.14 Apache Thrift

Apache Thrift is a software framework for scalable cross-language services development. In simpler words, it allows for easy exchange of data (variables, objects) between applications written in different languages.

Some of the key characteristics of the Apache Thrift data interchange format:

- Simplicity, Thrift code is simple and approachable, free of unnecessary dependencies.
- Transparency, Thrift conforms to the most common idioms in all languages.
- Consistency Niche, language-specific features belong in extensions, not the core library.
- Performance, Thrift Strives for performance first, elegance second.
- Thrift can be compiled in more languages compared to Protocol Buffers.

Key points on Apache Thrift serialization and deserialization performance:

- Cross-language serialization with lower overhead than alternatives such as XML and JSON due to use of binary data.
- Requires POJO classes to represent the data before serialization
- Very similar performance to Google's Protocol Buffers.
- Thrift only takes half the time to serialize objects compared to JSON.
- Thrift is not as adaptable as JSON and XML since data sent in a binary format cannot be parsed unless the receiver has the ".thrift" file and other necessary include files to deserialize and reconstruct the data.

5.2.15 Hessian

Hessian is a dynamically-typed, binary serialization and Web Services protocol designed for object-oriented transmission. Hessian is dynamically-typed, compact, and portable across languages.

Some of the key characteristics of the Hessian data interchange format:

- It can self-describe the serialized types, i.e. not require external schema or interface definitions.
- Language-independent, including supporting scripting languages.
- Hessian is readable or writable in a single pass.
- Hessian is as compact as possible.
- Hessian was implemented to be as fast as possible.
- Hessian supports Unicode strings.
- Supports 8-bit binary data without escaping or using attachments.
- Supports encryption, compression, signature, and transaction context envelopes.

5.2.16 KRYO

KRYO is a fast and efficient object graph serialization framework for Java. The goals of the project are speed, efficiency, and an easy to use API. The project is useful any time objects need to be persisted, whether to a file, database, or over the network.

Some of the key characteristics of the KRYO data interchange format:

- Kryo and CBOR are clearly more performant than JSON in terms of time spent serializing and size of payload.
- Kryo and CBOR are close: Kryo performs better but CBOR can represent the same data slightly smaller.

- Kryo has the fastest raw serialization/deserialization performance by a significant amount over both CBOR and Hessian.

Key points on KRYO serialization and deserialization performance:

- No Schema needed to serialize or deserialize any of the Java types.
- No POJO class required to represent the structure of the human readable data before being serialized or deserialized.

5.2.17 Concise Binary Object Representation (CBOR)

The Concise Binary Object Representation (CBOR) is a data format whose design goals include the possibility of extremely small code size, fairly small message size, and extensibility without the need for version negotiation. CBOR is very similar to the previous KRYO format and has many of the same characteristics.

Some of the key characteristics of the CBOR data interchange format:

- CBOR is based on the wildly successful JSON data model.
- Concise encoding as some applications also benefit from CBOR itself being encoded in binary.
- Extensible, so CBOR is able grow with its applications and to incorporate future developments
- Stable format, CBOR is defined in an Internet Standards Document, RFC 7049. The format has been designed to be stable for decades.

Key points on CBOR serialization and deserialization performance:

- No Schema needed to serialize or deserialize any of the Java types.
- No POJO class required to represent the structure of the human readable data before being serialized of deserialized.

5.3 Data Interchange Formats Chosen

After completing extensive research on over fifteen different data interchange formats including some of the industry standard and most utilized formats like XML, JSON and Protocol Buffers and some of the new formats like TOML, PON and REN a decision needed to be made to choose some of the best human readable formats and some of the best binary formats.

For the human readable formats the main goal was to use all the best attributes of the chosen formats to implement a new data interchange format. After careful analysis the human readable format that were picked are, JSON, XML, CSV, ArrayList and HashMap. The main reasons for choosing these formats were, they are the most widely used, all are industry standards and they all had a good number of libraries available online to help when writing the code. These libraries would be invaluable to simplify the process of converting from one format to another.

The binary formats that where picked are, BSON, KRYO, CBOR, JAVA, MessagePack and Hessian. The main reason these formats were chosen are, they do not require Java POJO classes to represent the data, this was a key requirement as monitoring data that's provided as input to the application may be structured in multiple different ways. Binary formats like Protocol Buffers and Apache Thrift required Java POJO classes to represent the data. That meant these formats could not be used for this project. All the chosen binary formats have the functionality to serialize and deserialize input data structured in any way.

5.4 Requirements Analysis

After careful analysis of the human readable formats and the binary formats that were picked to implement this new format, a list of functional requirements was created. The list below shows all the best attributes of all the chosen formats. The newly created data interchange format needed to implement the majority of these requirements.

Requirements of the data interchange format:

1. Be human readable.
2. Be as compact and efficient as possible.
3. Provide decent performance compared to other data interchange formats.
4. Support many different data types.
5. Must be free of unnecessary dependencies.
6. Provide support for binary serialization.
7. Beneficial to have libraries for serialization and deserialization.
8. Provide Cross-language serialization.
9. Provide a robust and durable format for information interchange.
10. Be independent from platforms and programming languages.
11. Must be easy for machines to parse and generate.
12. Must keep overhead to a minimum which is important when transferring data over the network to keep the bandwidth usage low.
13. Reuse: Information can be created once and reused many times.
14. Self-describing and easy to understand.

6 Design

6.1 Architecture

6.1.1 High Level Architecture

The interoperability of cloud monitoring data system is designed to sit at the edge of a cloud. Cloud service providers can connect the system to a monitoring agent or database that they are utilizing to export the monitoring data. Monitoring data is passed from the cloud into our application software where it can then be processed.

Figure 1 shows the high level architecture. A federation of cloud service providers may have the application connected to their cloud. Now the federation can use the application to convert the monitoring data to a different format, serialize the data and use the network or message bus to transfer the binary data to another cloud in the federation. The main point here is any cloud within the federation can send or receive monitoring data. This is just an example of how the application may be used in a real world example.

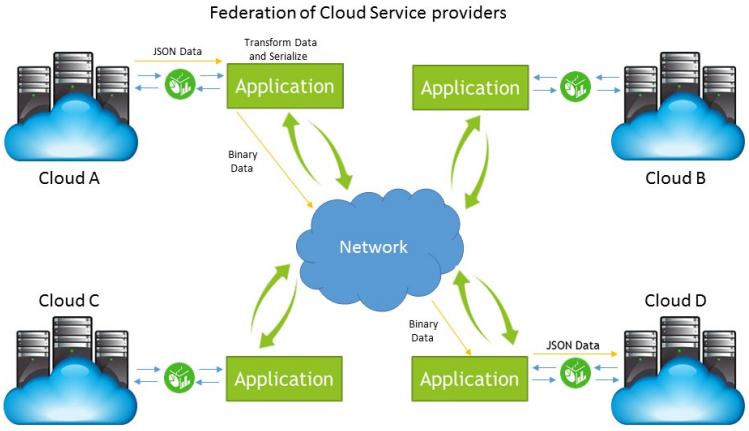

Figure 1: High Level Architecture Overview.

6.1.2 System Architecture

Digging a little deeper into the architecture of the system you can see from Figure 2 how monitoring data is passed as input to the system as one of the five Human Readable formats. The system should be smart enough to figure out if the format of the input data can be converted to a more efficient format. If so then the system will perform a conversion operation. Once the data has been converted the system will pass the data onto the binary serializers. Again the system should be smart enough to choose the best serializer to serialize the given data format into binary form.

The system must also be able to reverse the process described above. If the input data is in binary form it should be able to choose the right deserializer to deserialize the data back into the human readable format that was serialized. This enables cloud services providers to have interoperable monitoring data where the data can be sent and received between multiple different clouds within the federation.

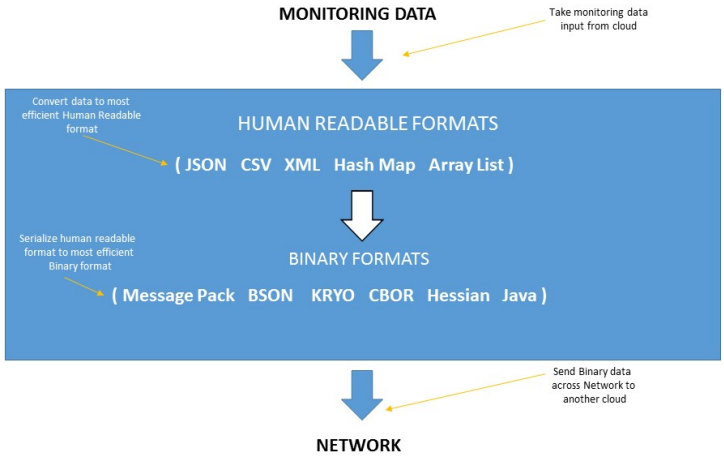

Figure 2: The System Architecture Overview

6.2 Software development Methodology

After looking at number of different software engineering methodologies like spiral, incremental, V-model etc. The one that best fitted the design of this project was the tried and tested waterfall methodology. A couple of reasons lead me to choose this methodology:

- This model is simple and easy to understand and use.
- It is easy to manage due to the rigidity of the model.
- In this model phases are processed and completed one at a time. This suited my project as it followed a very logical design process were each phase had to be complete before the next could begin.
- Waterfall model works well for smaller projects where requirements are very well understood.

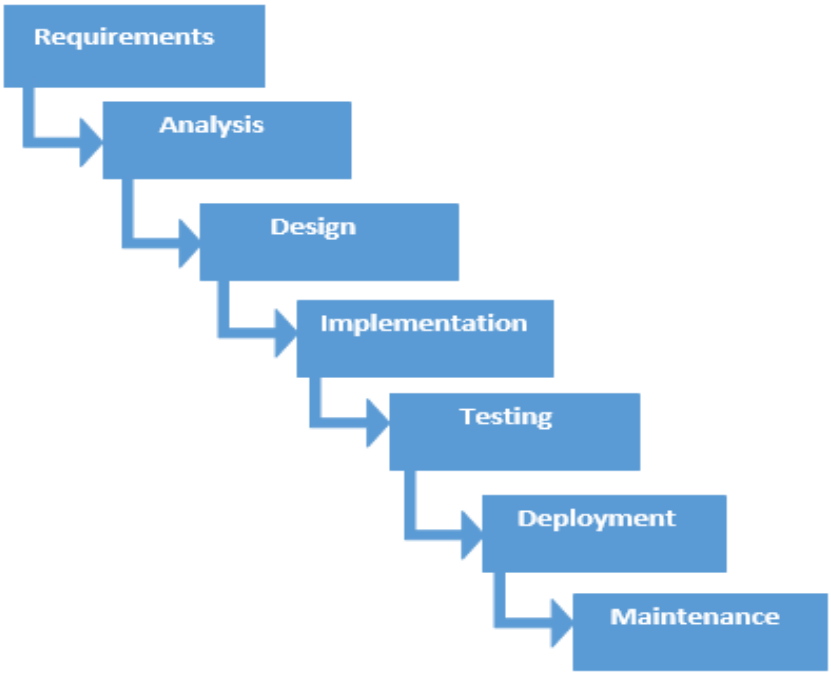

Figure 3: Waterfall Methodology

6.3 Main Package

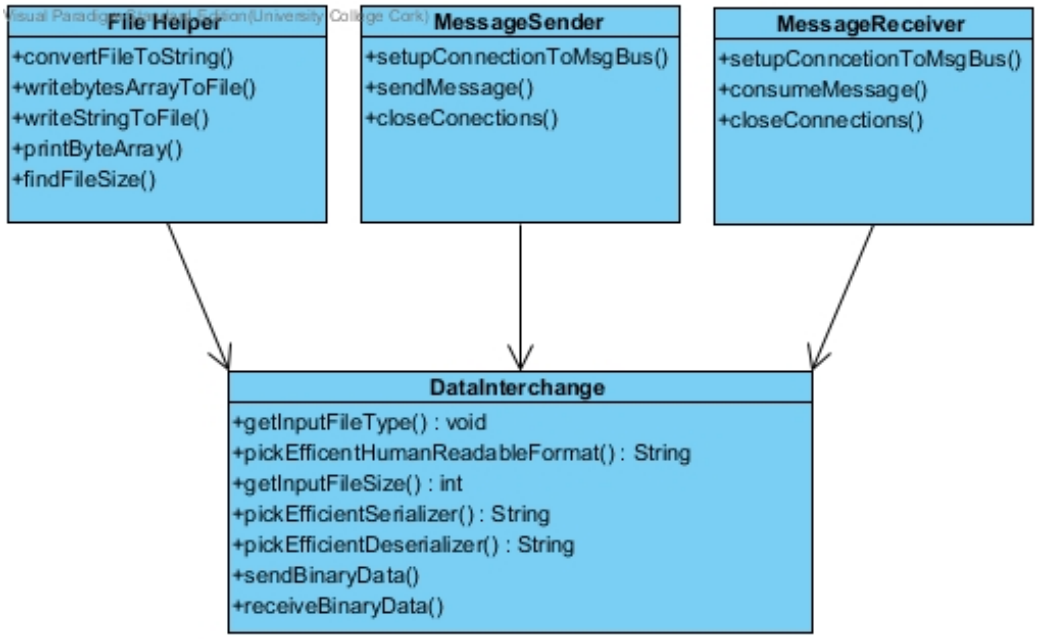

Figure 4: Class diagrams for the main, Helper and Messaging Classes.

The design process begin with the main data interchange class. This class would be the brains of the application as it would decide how the input data was processed, converted and serialized within the application. It will contain methods to do the following, figure out what format the input data is, pick the most efficient format to convert the input data to, pick the most efficient binary serializer to serialize the converted data or pick the right deserializer to deserialize the data if its in binary form.

To demonstrate the project in action a message bus would need to be set up so the serialized data could be sent to another cloud. A sender and receiver class would need to be implemented to handle this communication. This will be necessary to enable separate instances of the application to send and receive monitoring data in binary form.

A file helper class would also be needed which would contain methods to perform operations on file objects. As three of the human readable formats (JSON, XML, CSV) would be provided as input to the application as files this class would be crucial to manipulate these file objects. The file helper class would contain methods to convert a file to string, find the size of the file in terms of bytes, convert a string to file object, write string data to a file etc.

To show how these classes will integrate with the other classes within the application they have been added to the rest of the class diagrams as well. These classes were used by classes from the human readable formats package and the binary formats package.

6.4 Human Readable Package

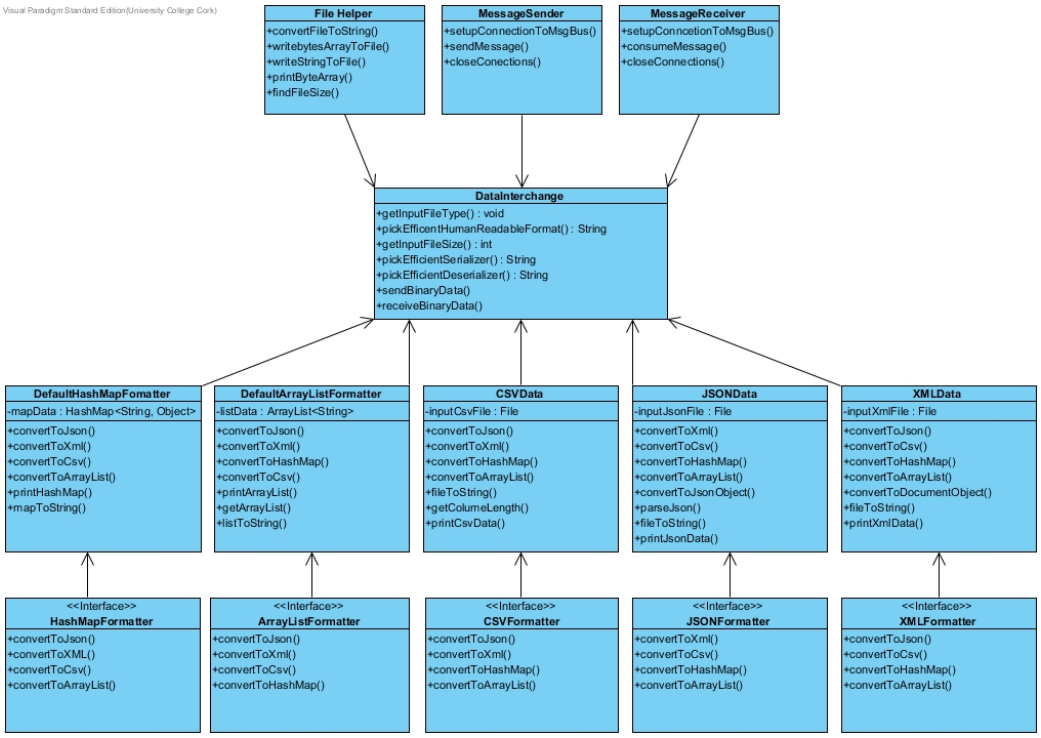

Figure 5: Class diagrams for the Human Readable formats.

The human readable package will contain all the classes for the human readable formats. Five classes would need to be implemented in this package which are the DefaultJsonFormatter, DefaultXmlFormatter, DefaultCsvFormatter, DefaultHashMapFormatter, DefaultArrayListFormatter. Each of these classes needed methods to convert the data into the other four human readable formats e.g. The JSON class would need methods to convert JSON into XML, CSV, ArrayList and HashMap.

All the human readable classes will also need to implement different interfaces. These interfaces are necessary to ensure that each human readable class has implementations for the correct conversion methods.

6.5 Binary Serialization Package

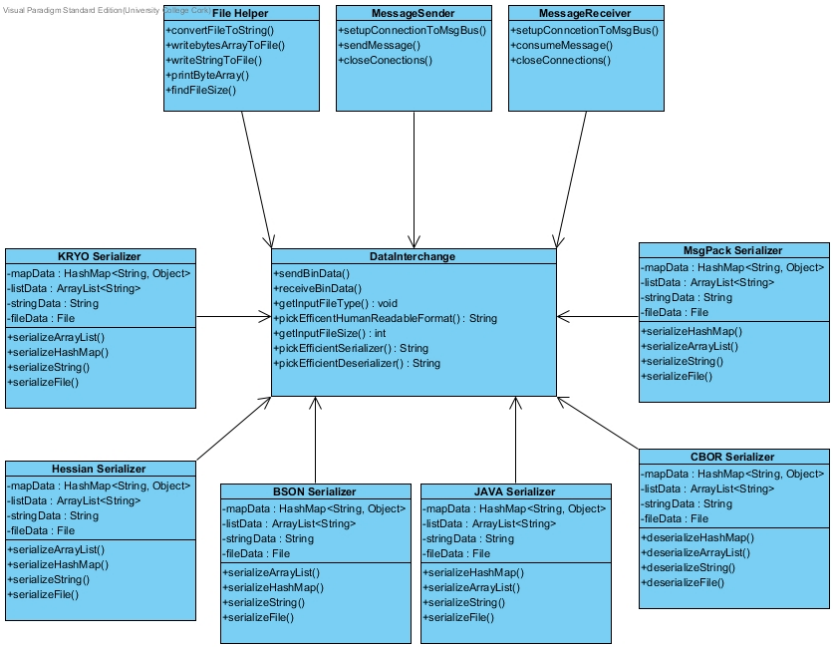

Figure 6: System Architecture Overview.

The binary serialization package will contain all the classes for the binary serialization formats. This package will be used to serialize data with the six serialization formats chosen for this project. Six new classes would need to be implemented in this package which are the BSONSerializer, KRYOSerializer, HessianSerializer, JavaSerializer, MessagePackSerializer and CBORSerializer classes.

Each of the serialization classes would need to implement methods to serialize different Java data types e.g. the BSONSerializer class would need methods to serialize human readable data that was stored in a HashMap, ArrayList, String and File into binary data.

The methods were necessary to enable the application serialize different types of monitoring data into binary form. The messaging classes described above could then be used to send the binary data to another cloud.

6.6 Binary Deserialization Package

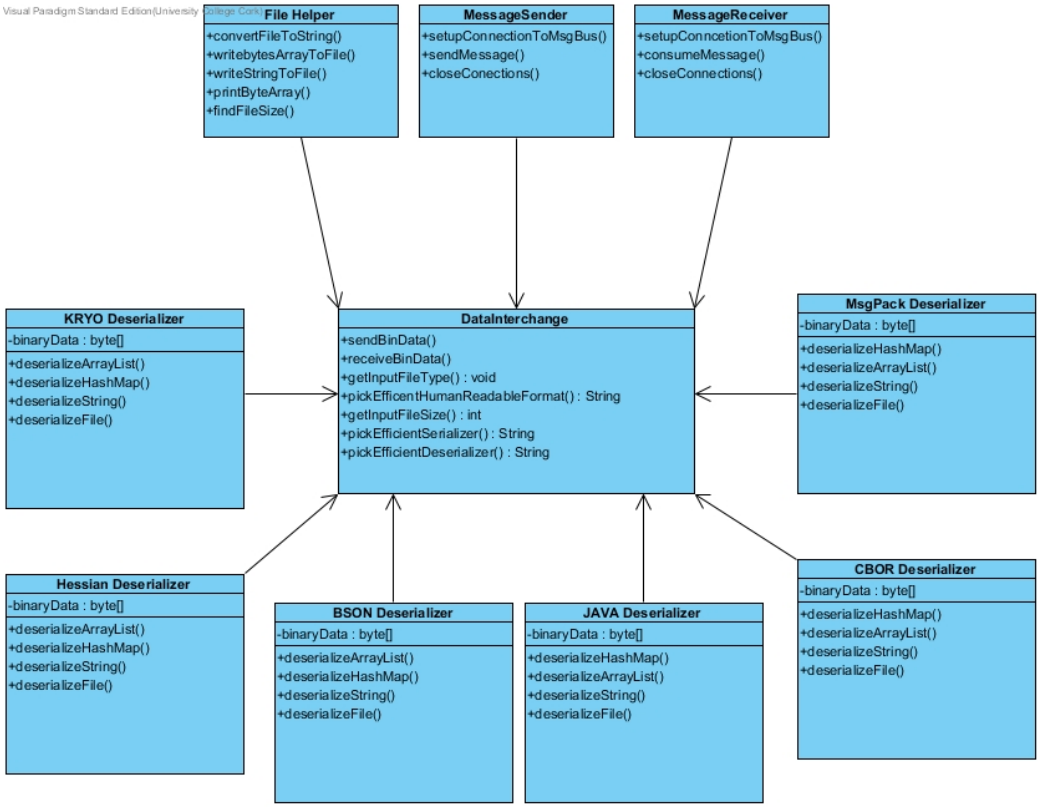

Figure 7: System Architecture Overview.

The binary deserialization package will be nearly identical to the binary serialization package. This package will contain all the class for binary data deserialization. Similar to the binary serialization package there would need to be six new classes implemented which are the BSONDeserializer, KRYODeserializer, HessianDeserializer, JavaDeserializer, MessagePackDeserializer and CBORDeserializer classes.

Each of the deserialization classes would need to implement methods to deserialize binary data back into the original human readable format e.g. the BSON deserializer class would need methods to convert binary data back into a Java HashMap, ArrayList, String or File object.

These methods were necessary to enable the application deserialize binary data into different Java data types. The application would now be able to receive binary data from the messaging class which was passed from another cloud and deserialize it.

6.7 External Libraries

To enable monitoring data to be converted from one of the five human readable formats chosen for this project to another format we had to make use of numerous free and open source libraries that were available online. Again for the six binary serialization formats, we made use of these open source libraries to serialize and deserialize the monitoring data.

The use of these libraries was in valuable in saving time. So there was no need to start reinventing the wheel and program all the conversion functions from scratch. The contributors to these libraries are far more experienced programmers compared to myself and done a far better job of writing the code to convert from one format to another. It made sense to use as many of the libraries as possible again to save time.

A small number of functions to convert one format to another e.g. XML to CSV had to be coded by myself as there was no libraries available for this. It did however take a considerable amount of time to learn how to use these libraries properly. All the external libraries used to convert from one format to another and to serialize and deserialize the data are listed below with the justification of why they were chosen over different libraries which had the same functionality, the bibliography section contains all the references to the libraries.

- **JSON Simple Library:** JSON.simple, is a simple Java library for processing, reading and writing JSON data. The library will be used along with the Jackson libraries to convert JSON data to CSV, XML, HashMap and ArrayList. The reason for using this library is that it was small, compact and really efficient compared to the Jackson library but it did not have some of the features of Jackson library.

- **Jackson Core Library:** Jackson is a very popular and efficient Java based library to serialize or map Java objects to JSON and vice versa. The main reason Jackson was picked over other JSON libraries such as Google's GSON library (which was also considered) is that Jackson has much better performance compared to GSON. Also Jackson had much more features compared to GSON which could be used in the serialization and deserialization of JSON data.

- **Jackson Databind Library:** A library which provided general data-binding functionality for Jackson. The only reason this library was used is that the Jackson core library required it to function properly and also enable some special features. Therefore it had to be included.

- **Jackson Annotations Library:** The Jackson JSON toolkit which contains a set of Java annotations which you can use to influence how JSON is read into objects, or what JSON is generated from the objects. Again the Jackson core library will not function properly without the inclusion of the Jackson Annotations library. This is the main reason the library was needed.

- **BSON4Jackson Library:** This library adds support for BSON to the Jackson JSON library. To enable BSON serialization and deserialization of monitoring data this library was needed. The main reasons for choosing this library is that there are no other BSON libraries available with the same features as this library and it made use of the Jackson library that was already chosen. The library also had a large user base and plenty of documentation which would come in handy if I had any issues using the library.

- **Jackson Dataformat CSV Library:** Jackson extension for adding support for reading and writing CSV formatted data into other human readable formats. Some of the reason this library was picked are similar to the BSON4Jackson library in that it made use of the already chosen Jackson Library. The performance was slightly better compared to other CSV libraries like CSV Commons or openCSV and there was plenty of documentation online to help me if I ran into any problems using the library. These are just another few reasons why the Jackson CSV library was picked over the other libraries.

- **CBOR for Java Library:** The Concise Binary Object Representation (CBOR) is a data format whose design goals include the possibility of extremely small code size, fairly small message size, and extensibility without the need for version negotiations. This library will be needed to aid in the process of CBOR serialization and deserialization of monitoring data. As CBOR was chosen as one of the binary formats to be used in this project this library had to be included.

- **Hessian Library:** Hessian 2.0 provides cross-language binary object serialization with efficiencies better than java.io serialization. Again as Hessian was chosen as one of the six binary formats for this project this library was required to enable Hessian serialization and deserialization of monitoring data.

- **KRYO Library:** KRYO is a fast and efficient object graph serialization framework for Java. For the same reasons the BSON and CBOR libraries were needed the KRYO library was also needed. As KRYO was chosen as one if the six binary formats this library was required to enable KRYO serialization and deserialization of monitoring data.

- **MessagePack Library:** MessagePack is an efficient binary serialization format. It lets you exchange data among multiple languages like JSON. But it's faster and smaller. For the same reasons as the rest of the binary serialization libraries where chosen can be applied to the MessagePack library. As MessagePack was picked as one of the binary formats this library enabled MessagePack serialization and deserialization of monitoring data.

- **Objenesis Library:** Objenesis is a library dedicated to bypass the constructor when creating an object. On any JVM there is. This library was require to enable the KRYO library function properly and to enable some special features for KRYO serialization and deserialization of monitoring data.

- **RabbitMQ Library:** The RabbitMQ library was needed to setup the communication bus between two VM's so they could send and receive binary data. So the RabbitMQ library was required to implement these features.

7 Implementation

7.1 Technologies Used

7.1.1 Java Programming Language

Java was the programming language of choice for this project. A couple of reasons lead to this choice. The first is that Java is a platform independent language which is crucial for this application as it must be able to run on different platforms. Java was also a prerequisite for this project and has excellent performance. Other advantages of using Java are listed below:

1. Excellent tooling: IDE, CI, etc.
2. Vast array of 3rd party libraries.
3. Huge amount of documentation available.
4. Large pool of developers available to provide help if needed.
5. Excellent specification.

7.1.2 RabbitMQ Message Broker Software

Implementation of this application required a messaging bus to enable the testing of the application. The messaging bus was needed to send binary data from one cloud to another cloud. As this project was going to be deployed on a OpenStack cloud environment it made sense to choose RabbitMQ as the message broker software.

OpenStack already uses RabbitMQ to handle communication between the different OpenStack components e.g Neutron, Nova, Horizon etc. So to make the integration process as easy as possible we choose RabbbitMQ as the messaging broker software for this project.

7.2 Main Package

Data Interchange Class
The implementation began with the main class which was the data interchange class. This class provided the intelligence of the application. It decides the flow of control through the entire application.

All the other classes linked back to this class. The data interchange class takes monitoring data as input in any of the given formats e.g. XML, JSON, CSV, Java HashMaps or ArrayLists. The class contained implementations of the following methods to process the input data:

- **pickEfficientHumanReadableFormat()**
- **pickEfficentSerializer()**
- **pickEfficentDeserializer()**
- **getInputFileType()**
- **getInputFileSize()**
- **sendBinaryData()**
- **receiveBinaryData()**

The most important methods in this class and probably the entire project were pickEfficientHumanReadableFormat(), pickEfficentSerializer(), pickEfficentDeserializer(). These three methods encapsulated the whole idea of this project. They could not be fully implemented until some evaluation was done on the different human readable and binary formats.

First I needed to find out which human readable formats would be more efficient if they were converted to another format e.g If the input data was in JSON format what would be the best format to convert the JSON data into in order to improve the efficiency or size in terms of bytes. The same evaluation needed to be done for the binary formats.

We needed to figure out which one of the six serializers was the best at serializing the different Java types (HashMap, ArrayList, String and File). Once this knowledge was obtained for all the human readable and binary formats the implementation of the class began. Java switch statements enable me to implement this functionality and I didn't run into any serious issues when writing the code.

File Helper Class
The file helper class contained implementations of the following methods:

- **convertFileToString()**
- **writeBytesToFile()**
- **writeStringToFile**
- **findFileSize**

These helper methods provided the functionality to perform basic operations on files.

7.3 Human Readable Formats

JSON Formatter Class
The JSONFormatter class contains all the methods that were needed to convert monitoring data in JSON form to any of the other four human readable formats (XML, CSV, HashMap, ArrayList). The JSON.simple and the Jackson libraries was used in implementing these conversion methods. The class contained the following methods:

- **convertToXML()**
- **convertToCSV()**
- **convertToHashMap()**
- **convertToArrayList()**
- **convertToJsonObject()**
- **convertToJsonObject()**
- **parseJson()**
- **printJsonData()**

The convertCSV() method was quite difficult to implement, if the JSON data was heavily nested the method won't be able to convert the data to the CSV format. The documentation for both libraries was excellent which help during the coding of the conversion methods. The rest of the methods were implemented without any issues.

XML Formatter Class
The XMLFormatter class contains all the methods that were again needed to convert monitoring data in XML form to any of the other four formats (JSON, CSV, ArrayList, HashMap). This class was one of the most difficult to code as there was no libraries available to help with writing the code for the conversion method. These needed to coded up from scratch. The class contained the implementation of the following methods:

- **convertToJson()**
- **convertToCSV()**
- **convertToHashMap()**
- **convertToArrayList()**
- **convertToDocumentObject()**
- **fileToString()**
- **printXmlData()**

The convertToCsv() method was really difficult to implement as it was nearly impossible to write code to handle different XML file structures. To get around this problem the convertToJson() method was used to convert the XML to JSON first. The JSON string is passed to an instance of the JSON formatter class where the convertToCSV() method from this class is used to convert the JSON to CSV. Two conversions operations are performed which almost doubles the time for the application to convert XML to CSV. Not an ideal solution but it solved the issue

The convertToHashMap() method was another method which had to first convert the XML data to JSON and use a method from the JSON formatter class to convert the JSON data to a Java HashMap containing the same data. This solution as the same issues as the convertToCsv() method in that Two conversions operations are performed which almost doubles the time.

The convertToArrayList() method again is another method which made use of methods from the JSON formatter class. It has the same issues as the two previous methods in that the conversion time is almost doubled due to the two conversion operations that are needed to convert XML to a Java ArrayList. The rest of the methods were implemented without any issues.

CSV Formatter Class
The CSVFormatter class contains all the methods that were again needed to convert monitoring data in CSV form to the other four human readable formats (JSON, XML, ArrayList, HashMap). The jackson-dataformat-csv-2.7.0 and the Jackson Core libraries were used to help writing the code to implement the conversion methods. The class contained the following methods:

- **convertToJSON()**
- **convertToXML()**
- **convertToHashMap()**
- **convertToArrayList()**
- **fileToString()**
- **getColumeLength**
- **printCsvData()**

The convertToXml() method was a fairly complex method to implement which made extensive use of the inbuilt Java XML libraries. Took a large amount of time to implement the method as a lot of learning was involved in reading the XML library API documentation. The rest of the methods were implemented without any issues.

ArrayList Formatter Class
The ArrayListFormatter class contains all the methods that were again needed to convert monitoring data in an ArrayList form to any of the other of the four human readable formats (JSON, XML, CSV, HashMap). The class required no external libraries and contained implementations for the following libraries:

- **convertToJSON()**
- **convertToXML()**
- **convertToCSV()**
- **convertToHashMap()**
- **printArrayList()**
- **listToString()**

The convertToJson() method was another difficult method to implement as data within an ArrayList has no structure. Solving the issue was achieved by using the convertToCsv() method to first convert the ArrayList to a CSV string. A method from the CSV formatter class was then used to convert the CSV string into JSON data. Increases the time for this method to complete the conversion of an ArrayList of data into JSON data.

The convertToXml() method was also a difficult method to implement for the same reasons, the data stored in an ArrayList has no structure so there is now way of knowing how to construct the data in XML format. The solution was similar to the method used in the method above. First converting the ArrayList to a CSV string and then using a method from the CSV formatter class to convert CSV to XML. The rest of the methods were implemented without any issues.

HashMap Formatter Class
The HashMap formatter class contains all the methods that were again needed to convert monitoring data in a Java HashMap form to any of the other four human readable formats (JSON, CSV, ArrayList, HashMap). The class required no external libraries and contained implementations for the following libraries:

- **convertToJSON()**
- **convertToXML()**
- **convertToCSV()**
- **convertToArrayList()**
- **printHashMap()**
- **mapToString()**

The convertToJson() and the convertToXml() methods were also difficult to implement for exactly the same reasons mentioned in the previous class, No real structure to data so building JSON or XML structured data was extremely difficult. The same method the solve the issues in the previous class was used here, the first step was to convert the HashMap data to CSV data using convertToCSV() method in this class, next was to use a method from the CSV formatter class to convert CSV to XML or JSON data. The rest of the methods were implemented without any issues.

7.4 Binary Serialization Package

BSON Serializer Class
The BSON serializer class required the use of the BSON4Jackson and the Jackson Databind libraries to enable BSON serialization of different Java types. The BSON serializer class contained implementations for the following methods (All the serializer classes contain the same methods):

- **serializeArrayList()**
- **serialzeHashMap()**
- **serialzeString()**
- **serializeFile()**

These methods provided the functionality to serialize Java ArrayList, HashMap, String and file objects using BSON. Coding these methods was fairly difficult but once one method was done the code could be reused for the other methods. The Jackson ObjectMapper object was used to serialize the different Java types into binary data.

One issue arose with the serializeFile() method, for some reason the BSON serializer would not serialize the file object and would cause the program to fail. After trying various changes I could not fix the problem and eventually found out that BSON cannot serialize Java file objects from the BSON online documentation.

KRYO Serializer Class
Kryo serializer class required the use of the KRYO-2.24.0 library to enable KRYO serialization of different Java data types. The KRYO serialization class contained the same methods as the BSON serializer class but they have different implementations.

These methods provided the functionality to serialize Java ArrayList, HashMap, String and file objects using KRYO. The use of KRYO and Output objects helped in the coding process to serialize the different data types. Implementing the methods was not straight forward as learning how to use the library took some time.

CBOR Serializer Class
CBOR serializer class required the use of the cbor-java-master library to enable CBOR serialization of different Java data types. The CBOR class contained the same methods as the BSON serializer class but again they have different implementations.

These methods provided the functionality to serialize Java ArrayList, HashMap, String and file objects using CBOR. Using the CBOR library when implementing the methods helped with the coding. By using CBOR-Writer and Output objects from the CBOR library it simplified the process of serializing the different data types. Although two of the methods throw up issues when they where being unit tested. The serialzeHashMap() and serializeFile() method did not work as expected and caused the program to fail. After trying numerous solutions I could not fix the issues. It was now becoming clear to me that none of the serializers would be able to serialize file objects.

Java Serializer Class
Java Serializer class used the inbuilt Java libraries to serialize the different data types. This class was implemented to provide a benchmark to compare how the other serializers performed against the Java serializer and contained the same methods as the BSON serializer class.

These methods provided the functionality to serialize Java ArrayList, HashMap, String and file objects using Java serialization. Using the ObjectOutputStream object to convert the data into binary data helped with coding the methods.

Hessian Serializer Class
Hessian serializer class required the use of the hessian-4.0.33 library to enable Hessian serialization and deserialization of the different Java types. This class contained the same methods as the BSON serializer class but again they have different implementations.

These methods provided the functionality to serialize Java ArrayList, HashMap, String and file objects using Hessian. With plenty of documentation the implementation of the methods was not to difficult. By using Hessian2Output objects to serialize the different data types and convert it into binary data was a huge help in the coding process. Encountered no issues when unit testing these methods.

MessagePack Serializer Class
MessagePack serialization class required the use of the msgpack-0.6.12 library to enable MessagePack serialization all the data types. Again the class contained the same methods as previous classes but with different implementations.

These methods provided the functionality to serialize all the Java data types using MessagePack. The library took a bit of time to get to grips with but using the MessagePack and Packer objects to serialize the data helped with the implementation of these methods.

7.5 Binary Deserialization Package

The Binary deserialization package is almost identical to the serialization package. It contains the same number of classes and the same number of methods in each class. The only difference is that instead of serializing the different Java data types we are deserializing binary data back to the original data. Every library that was required for each serializer class was also needed for the corresponding deserializer class. The same objects used in each serializer class to serialize data were used to deserialize data in the corresponding deserializer class only the method used was different

As the deserialization package is so similar to the serialization package rather than repeating and explaining the entire implementation of the deserialization package along with the classes and their methods, I'll just give a quick overview of the package and explain some of the main issues I had implementing the classes in this package.

BSON Deserialzer Class
Very similar implementation to the BSON Serializer class, but the methods in this class take binary data as input instead of Java data types. The Binary serialization class diagram in the design section shows this. The class contained method implementations to convert this binary data back to the original data type.

- **deserializeArrayList()**
- **deserialzeHashMap()**
- **deserialzeString()**
- **deserializeFile()**

Encounter issues with the deserializeString() method in this class, for some reason the method would not deserialize the binary data back into a string object. Spent a lot of time trying to solve this issue but had no success. Therefore the method was not functioning as it was intended to.

KRYO Deserializer Class
Exact same implementation as the corresponding KRYO serializer class. Again the only difference is the methods in this class take binary data as input instead of different Java data types. Depending on the binary data one of the four methods listed in the BSON Deserializer class will be used to deserialize the data.

CBOR Deserializer Class
Exact same implementation as the corresponding KRYO serializer class. Again has the same methods implemented as the BSON deserializer class. Even though the serializeHashMap() method was not functioning properly in the corresponding serializer class, the deserializeHashMap() in this class was still implemented just in case the serializeHashMap() method was fixed in the future.

Java Deserializer Class
Exact same implementation as the corresponding Java serialization class. Again has the same methods implemented as the BSON deserializer class above. No real issues implementing this class.

Hessian Deserializer Class
Exact same implementation as the corresponding Hessian serialization class. Again has the same methods implemented as the BSON deserializer class above. No real issues implementing this class.

MessagePack Deserializer Class
Exact same implementation as the corresponding MessagePack serialization class. Again has the same methods implementented as the BSON deserializer class above. No real issues implementing this class.

7.6 Message Sender Package

In The design phase the message sender class was originally meant to be included in the main package but after getting into the implementation of the code it was decided it should be placed in its own package to keep both the MessageSender and MessageReceiver classes separate. Both classes needed to be independent of each other as they will be running on two different machines, one machine to send the data and the other machine to receive the data.

The MessageSender class require the use of the rabbitMQ library to enable message bus communication between two separate machines. It contained the implementation of the following methods:

- **setupConnectionToMsgBus()**
- **sendData()**
- **closeConncetion()**

The code for this class was provide by my mentor Dr, Vincent C. emeakaroha. As configuring the messaging bus was not part of the project. However code did need to be written to enable a working demonstration of the application for the project demo day. This involved adding code to the class which would run through all five human readable formats, serialize the data with all six binary formats and then send the data across the messaging bus to the Message Receiver class running on a different machine.

7.7 Message Receiver Package

Just as with the MessageSender class, the MessageReceiver class was originally meant to be be included in the main package but was also placed in its own package for the same reasons mentioned above. The class contained implementations for the following functions:

- **setupConnectionToMsgBus()**
- **ReceiveData()**
- **closeConncetion()**

The code for this class was provided by my mentor Vincent. Again however code needed to written to enable a working demonstration of the application for the project demo day. This time the code written needed to receive the binary data from the RabbitMQ message bus which was sent from the MessageSender class. The class needed to deserialize all the binary data with all six binary deserializers and reproduce the human readable data back to it original from. Writing the code to implement these features took a considerable amount of time to get right.

7.8 Software Testing

7.8.1 Unit Testing

Every human readable conversion method, serialization method and deserialization method implemented in this project had a corresponding JUnit test method. JUnit is a simple framework to write repeatable tests which was necessary in this project.

The main reasons are, To make sure all the conversion, serialization and deserialization methods do what they are meant to do and to also make sure I keep the number of bugs low. As already explained in the implementation process a small number of the conversion method did not function correctly and is was due to these JUnit test methods that these issues were found.

The list of JUnit tester methods is to long to list here but can be seen in the code base. Implementation of the methods was done in a single class. The need for this class was not considered during the design phase and as it became apparent during the implementation phase that testing individual methods was necessary it was then added.

7.8.2 System Testing

To test the whole system functions as expected we set up two VM's to act as two different cloud service providers. The RabbitMQ messaging broker software was used as the messaging bus to provide communication between the two VM's. The application was run on both the VM's. VM1 was sending the serialized binary data and VM2 was receiving the binary data. The MessageSender and the MessageReceiver classes were used to send and receive this data.

Methods were coded to serialize all five human readable formats with all six of the binary serialization formats. Using the sendMessage() from the message sender class to then send the binary data to VM2. VM2 then used the consumeMessage() method from the message receiver class to get the binary data. More methods where then coded to used all six binary deserialization formats to deserialize the binary data back into all Five human readable formats.

By configuring the test setup it enable me to really test out all parts of the software system. It threw up some bugs in my code and allowed me to fix them making the whole application even more robust. The main reasons for system testing is verification which ensures that We have built the system correctly. That is, that the software correctly implements the solution. Validation ensures that we have built the correct system. That is, the solution realized in the software does actually solve the original problem.

8 Results & Evaluation

8.1 Overview

To evaluate the performance of the human readable and binary data interchange formats we wanted to comprehensively test the application by running it with data of different sizes. The idea here was to gather results from testing the application with different data sizes and then compare and analyze the results.

From here we could then conclude which human readable format (JSON, CSV, XML, Array List and Hash Map) was the most efficient at representing the data. Depending on which one of those types was the most efficient for the given human readable format we could finally serialize that data with all six of the binary formats.

The binary format that was most efficient or serialized the data to the smallest size in terms of bytes would obviously be the best to use. After the data was serialized to the smallest size possible the application could then for e.g. transfer the data in binary form across a network or pass it to a message bus.

8.2 Detailed Evaluation

The size of the data used in testing and evaluating the application was grouped into three different categories, small, medium and large. Each category has the same data represented in the five different human readable formats JSON, CSV, XML, ArrayList and HashMap. For example the data used in the small data size are shown below in all five human readable formats and notice all formats have the same data.

Listing 1: Example of the small JSON File.

```
1 {"employees":[
2     {"firstName":"John", "lastName":"Doe"},
3     {"firstName":"Anna", "lastName":"Smith"},
4     {"firstName":"Peter", "lastName":"Jones"}
5 ]}
```

Listing 2: Example of the small CSV file.

```
firstName    lastName
John   Doe
Anna   Smith
Peter  Jones
```

Listing 3: Example of the small XML file.

```
<root>
    <employees>
        <firstName>John</firstName>
        <lastName>Doe</lastName>
    </employees>
    <employees>
        <firstName>Anna</firstName>
        <lastName>Smith</lastName>
    </employees>
    <employees>
        <firstName>Peter</firstName>
        <lastName>Jones</lastName>
    </employees>
</root>
```

Listing 4: Example of a small Java ArrayList.

```
[firstName,lastName, John,Doe, Anna,Smith, Peter,Jones]
```

Listing 5: Example of a small Java HashMap.

```
Key = 0, Value = firstName,lastName,
Key = 1, Value = John,Doe,
Key = 2, Value = Anna,Smith,
Key = 3, Value = Peter,Jones,
```

By just looking at all five human readable formats above you can see XML and JSON are the largest in size whereas CSV, ArrayList and HashMap are the smallest. So to back this up with some evidence we needed to go through all five of the human readable formats one at a time, serialize the data in that format with all six binary serialization formats and record the results and then repeat this for all the different data size categories.

So for example if we take the JSON format, we first convert the JSON data into a Java ArrayList and serialize this list with all six binary serialization formatters, then record the results. Next would be to convert the JSON data to a Java Hash Map and serialize this map with all six serialization formatters and again record the results. These steps were repeated for XML and CSV and the original JSON data where they are all serialized with all six binary formatters and the results are recorded. The tables below show the results of this evaluation for all five human readable formats in all data size categories.

8.3 Results for Small Data

Results serializing small data that contain the same data in different formats.

Small JSON File			File Size: 157 Bytes		
	ArrayList	hashMap	JSON	XML	CSV
MassagePack	137	115	160	343	54
BSON	146	164	162	345	56
KRYO	160	157	160	343	52
CBOR	136	—	159	343	53
Hessian	140	138	163	346	57
JAVA	194	365	164	347	58

Small CSV File			File Size: 53 Bytes		
	ArrayList	hashMap	JSON	XML	CSV
MassagePack	60	125	173	363	56
BSON	150	144	175	365	58
KRYO	92	123	173	363	54
CBOR	60	—	172	363	55
Hessian	65	80	176	366	59
JAVA	144	235	177	367	60

Small XML File			File Size: 338 Bytes		
	ArrayList	hashMap	JSON	XML	CSV
MassagePack	146	122	239	341	54
BSON	155	175	241	343	56
KRYO	169	167	239	341	52
CBOR	145	—	238	341	53
Hessian	149	146	242	344	57
JAVA	203	398	243	345	58

Small Java HashMap					
	ArrayList	hashMap	JSON	XML	CSV
MassagePack	60	125	176	363	57
BSON	150	144	178	365	59
KRYO	92	123	176	363	55
CBOR	60	—	175	363	56
Hessian	65	80	179	366	60
JAVA	144	235	180	367	61

Small Java ArrayList					
	ArrayList	hashMap	JSON	XML	CSV
MassagePack	60	123	233	363	57
BSON	150	144	235	365	59
KRYO	92	123	233	363	55
CBOR	60	—	232	363	56
Hessian	65	80	236	366	60
JAVA	144	235	237	367	61

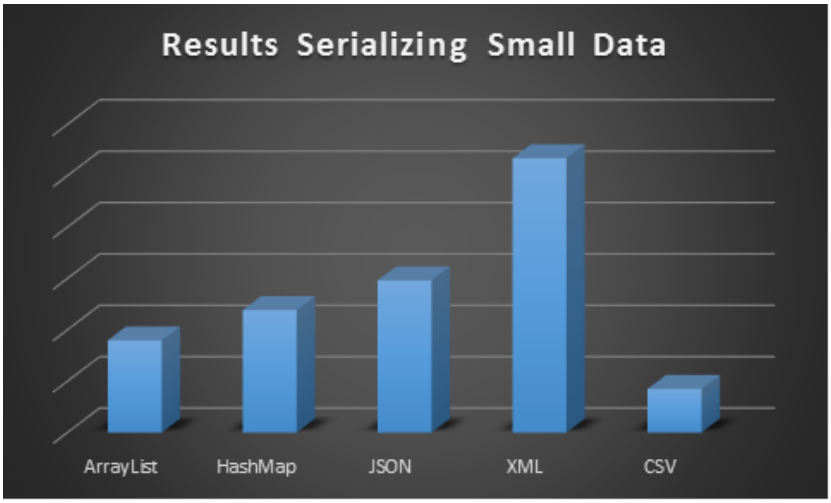

Figure 8: Graph showing result from small data sets.

As can be clearly seen from the graph representation of the result relating to the small data size Category, the CSV format can represent the same data as the other formats but in a fraction of the size compared to XML and JSON.

8.4 Results From Medium Data

Results from serializing medium data with all binary serializers.

Medium JSON File			File Size: 266,917 Bytes		
	ArrayList	hashMap	JSON	XML	CSV
MassagePack	196,702	170.859	266,922	373,928	59,796
BSON	196,709	207,294	266,922	373,928	59,798
KRYO	196,724	87,271	266,921	373,927	59,797
CBOR	196,702	—	266,922	373,928	59,796
Hessian	196,721	162,533	266,948	373,963	59,803
JAVA	196,763	126,966	266,930	373,936	59,800

Medium CSV File			File Size: 60,314 Bytes		
	ArrayList	hashMap	JSON	XML	CSV
MassagePack	60,320	8500	240,294	373,462	60,317
BSON	148,292	128,856	240,294	373,462	60,319
KRYO	69,543	62,646	240,293	373,461	60,318
CBOR	60,320	—	240,294	373,462	60,317
Hessian	60,325	61,530	240,317	373,497	60,324
JAVA	78,452	81,279	240,302	373,470	60,321

Medium XML File			File Size: 373,459 Bytes		
	ArrayList	hashMap	JSON	XML	CSV
MassagePack	175,456	165,371	264,199	373,464	54,202
BSON	175,463	168,557	264,199	373,464	54,204
KRYO	175,478	73,345	264,198	373,463	54,203
CBOR	175,456	—	264,199	373,464	54,202
Hessian	175,473	143,457	264,225	373,499	54,209
JAVA	175,517	142,627	264,207	373,472	54,206

Medium Java HashMap					
	ArrayList	hashMap	JSON	XML	CSV
MassagePack	60,320	8,509	248,646	373,462	68,181
BSON	148,292	128,856	248,646	373,462	68,181
KRYO	69,543	62,646	248,645	373,461	68,180
CBOR	60,320	—	248,646	373,462	68,181
Hessian	60,325	61,530	248,669	373,497	68,189
JAVA	78,452	81,279	248,654	373,470	68,189

Medium Java ArrayList					
	ArrayList	hashMap	JSON	XML	CSV
MassagePack	60,320	8,500	264,199	373,462	60,057
BSON	148,292	128,856	264,199	373,462	60,059
KRYO	69,543	62,646	264,198	373,461	60,058
CBOR	60,320	—	264,199	373,462	60,057
Hessian	60,325	61,530	264,225	373,497	60,064
JAVA	78,452	81,279	264,207	373,470	60,061

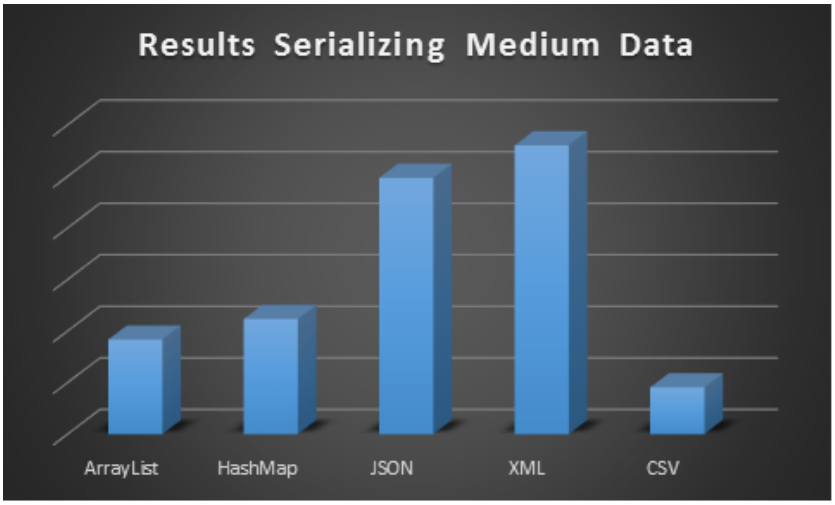

Figure 9: Graph showing result from medium data sets.

As can be seen from the graph above the results form the medium data size are interesting. The JSON format is getting worse as the data size is increased and it is almost as bad as XML. Again the CSV format is the clear winner here with ArrayList in second with the HashMap not far behind.

8.5 Results From Large Data

Results serializing large data with all binary serializers.

Large JSON File			File Size: 27,888,867 Bytes		
	ArrayList	hashMap	JSON	XML	CSV
MassagePack	22,060,863	19,190,354	27,888,867	41,748,475	6,809,875
BSON	22,060,870	23,296,539	27,888,867	41,748,475	6,809,875
KRYO	22,060,886	9,868,007	27,888,867	41,748,475	6,809,875
CBOR	22,060,863	—	27,888,867	41,748,475	6,809,875
Hessian	22,062,884	18,262,541	27,891,422	41,752,299	6,810,498
JAVA	22,060,924	14,261,681	27,888,875	41,748,483	6,809,883

Large CSV File			File Size: 6,838,864 Bytes		
	ArrayList	hashMap	JSON	XML	CSV
MassagePack	6,867,867	999,795	26,903,033	41,690,541	6,838,869
BSON	18,587,125	14,511,701	26,903,033	41,690,541	6,838,869
KRYO	7,896,367	7,189,378	26,903,033	41,690,541	6,838,869
CBOR	6,867,867	—	26,903,033	41,690,541	6,838,869
Hessian	6,872,867	7,058,737	26,905,497	41,694,359	6,839,495
JAVA	8,868,643	9,234,536	26,903,041	41,690,549	6,838,879

Large XML File			File Size: 6,838,864 Bytes		
	ArrayList	hashMap	JSON	XML	CSV
MassagePack	19,766,909	18,636,098	29,625,212	41,690,543	6,255,613
BSON	19,766,916	18,857,947	29,625,212	41,690,543	6,255,613
KRYO	19,766,932	8,271,632	29,625,212	41,690,543	6,255,613
CBOR	19,766,909	—	29,625,212	41,690,543	6,255,613
Hessian	19,768,720	16,081,524	29,627,926	41,694,361	6,256,185
JAVA	19,766,970	15,983,147	29,625,220	41,690,551	6,255,621

Large Java HashMap					
	ArrayList	HashMap	JSON	XML	CSV
MassagePack	6,867,867	999,795	27,830,873	41,690,541	7,737,746
BSON	18,587,125	14,511,701	27,830,873	41,690,541	7,737,746
KRYO	7,896,367	7,189,378	27,830,873	41,690,541	7,737,746
CBOR	6,867,867	—	27,830,873	41,690,541	7,737,746
Hessian	6,867,872	7,058,737	27,833,422	41,694,359	7,738,456
JAVA	8,868,643	9,234,536	27,830,881	41,690,549	7,737,754

Large Java ArrayList					
	ArrayList	hashMap	JSON	XML	CSV
MassagePack	6,867,867	999,795	29,625,212	41,690,541	6,838,870
BSON	18,587,125	14,511,701	29,625,212	41,690,541	6,838,870
KRYO	7,896,367	7,189,378	29,625,212	41,690,541	6,838,870
CBOR	6,867,867	—	29,625,212	41,690,541	6,838,870
Hessian	6,872,867	7,058,737	29,627,926	41,694,359	6,839,498
JAVA	8,868,643	9,234,536	29,625,220	41,690,549	6,838,878

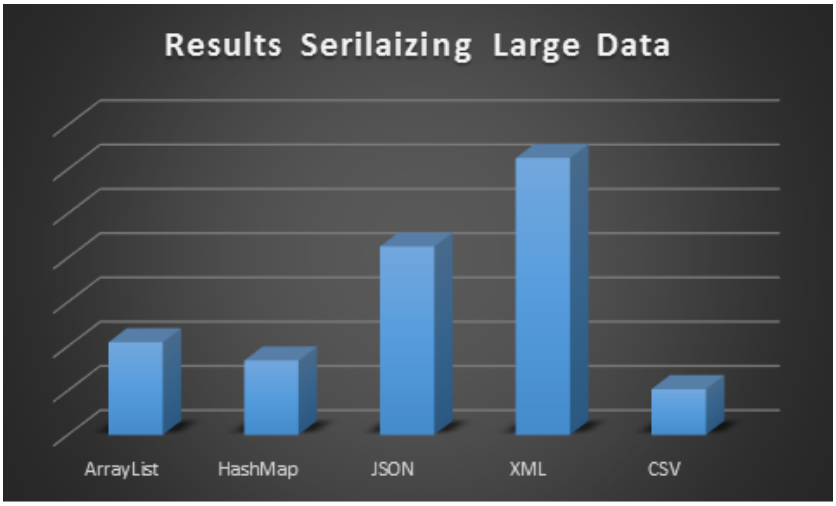

Figure 10: Graph showing result from large data sets.

As can be seen from the graph, the results from the large data are interesting. By comparing the three graphs from the different data category size we can clearly see as the monitoring data increases in size the HashMap format gets closer and closer to the performance of CSV. If the application was tested with even larger data my guess is the performance of the HashMap format would get even closer and might even surpass the performance of CSV.

8.6 Results Conclusion

Based on the results contained in the tables and graphs above we have concluded that the most efficient, compact, and best performing human readable format is the CSV format. Again it can represent the same data as the other formats but in a fraction of the size. This result can clearly be verified from the tables and graphs shown above. If we look at the results for the CSV format in all three different data size categories we can see that it consistently produces the smallest data size across all data size categories.

After confirming that CSV was the best human readable format in terms of efficiency and data size in terms of bytes we needed to choose which of the binary serialization formats could actually serialize the CSV data to the smallest size in terms bytes. After looking at the serialization results for the binary formats the one that consistently serialized the data to the smallest size in terms of bytes was MessagePack. By looking at the table rows for the Message Pack serializer we can again see that in the three different data size categories Message Pack produced the best results.

9 Conclusion

9.1 Project Summary

The project required a lot of research of data interchange formats at the beginning. The research was required to see what kind of formats were already available and being used in industry. Some of the best human readable formats and binary formats where then picked to be used in the project. The main goal was to use the best characteristics of the chosen formats to implement a new data serialization format.

The application required the implementation of Java classes to perform the conversion operations to convert from one human readable format to another. All the methods for the conversion operations were coded and working as expected.

The application also required the implementation of Java classes to perform the serialization and deserialization of data. Again the majority of the methods to perform these operations were coded and working except for two methods which are the serializeHashMap() method in the CBOR serializer class and the deserializeString() method in the BSON deserializer class. A messaging communication bus was setup to pass the binary data between two VM's to demonstrate the application working for the project demo day.

Overall for me, the project was a success. Even though some aspects of the application were not completely implemented the project still achieved the majority of the goals specified in the analysis section. If I was to start the project again I would of course make some changes but that is all part of the leaning process. The project was a pleasure to work on as I had a real interest in this topic before the project began.

9.2 Future Work

Any future work on this project would include adding even more data interchange format to the application to incorporate the best feature of these new formats to make this application even better. Also fixing the non functioning methods mentioned in the previous section could also be done.

10 Bibliography

References

[1] G. Imre, M. Kasz, T. Levendovszky and H. Charaf "A Novel Cost Model of XML Serialization" Department of Automation and Applied Informatics, Budapest University of Technology and Economics, Budapest, Hungary

[2] Marjan Hericko, Matjaz B. Juric, Ivan Rozman, Simon Beloglavec, Ales Zivkovic "Object Serialization Analysis and Comparison in Java and .NET" Newsletter ACM SIGPLAN Notices Homepage archive Volume 38 Issue 8, August 2003 Pages 44-54.

[3] Mattias Nordahla , Boris Magnussona "A lightweight data interchange format for Internet of Things in the PalCom middleware framework." The 10th International Conference on Future Networks and Communications (FNC 2015).

[4] Audie Sumaray Christopher Virginia, USA & S. Kami Makki Lamar, University Beaumont, Texas, USA "A Comparison of Data Serialization Formats For Optimal Efficiency" Newport University Newport USA.

[5] Bruno Gil DCTI, ISCTE-IUL & Paulo Trezentos "Impacts of data interchange formats on energy consumption and performance in" DCTI, ISCTE-IUL, Lisboa, Portugal.

[6] Nurzhan Nurseitov, Michael Paulson, Randall Reynolds, Clemente Izurieta "Comparison of JSON and XML Data Interchange Formats" Department of Computer Science, Montana State University – Bozeman, Montana, 59715, USA

[7] Vincent C. Emeakaroha, Philip Healy, Kaniz Fatema and John P. Morrison "Analysis of Data Interchange Formats for Interoperable and Efficient Data Communication in Clouds" Irish Centre for Cloud Computing and Commerce University College Cork, Ireland.

[8] G. Imrea,1, M. Kasz´oa,2, T. Levendovszkyb,3 and H. Charafa,4 "A Novel Cost Model of XML Serialization" Department of Automation and Applied Informatics, Budapest University of Technology and Economics, Budapest, Hungary b Institute for Software Integrated Systems, Vanderbilt University, Nashville, TN USA

[9] Zijing Tan ⇑, Liyong Zhang, Wei Wang, Baile Shi "XML data exchange with target constraints" School of Computer Science, Fudan University, Shanghai, China

[10] Kazuaki Maeda, "Performance Evaluation of Object Serialization Libraries in XML, JSON and Binary Formats" Department of Business Administration and Information Science, Chubu University

[11] "https://github.com/FasterXML/jackson-dataformat-csv" Jackson CSV Formatter Apr 10, 2011

[12] "http://hessian.caucho.com/doc/hessian-serialization.html" Hessian Serializer Home Page

[13] "http://bsonspec.org/" Bson Serializer Home Page

[14] "http://cbor.io/" CBOR Serializer Home Page

[15] "https://github.com/EsotericSoftware/kryo" KRYO Serializer Home Page

[16] "https://github.com/FasterXML/jackson" Jackson Core Library Home.

[17] "http://mvnrepository.com/artifact/com.caucho/hessian" Hessian Library Home.

[18] "https://github.com/naphaso/cbor-java" CBOR Library Home.

[19] "https://github.com/michel-kraemer/bson4jackson" BSON Library Home.

[20] "http://www.java2s.com/Code/Jar/j/Downloadjsonsimplejar.html" JSON Simple Library Home.