PURDUE UNIVERSITY
GRADUATE SCHOOL
Thesis/Dissertation Acceptance

This is to certify that the thesis/dissertation prepared

By GEETHA R SATYANARAYANA

Entitled
A QUANTITATIVE COMPARISON & EVALUATION OF PROMINENT MARSHALLING/UN-MARSHALLING FORMATS IN DISTRIBUTED REAL-TIME & EMBEDDED SYSTEMS

For the degree of Master of Science

Is approved by the final examining committee:

JAMES H HILL
Chair

RAJEEV RAJE

MIHRAN TUCERYAN

To the best of my knowledge and as understood by the student in the Thesis/Dissertation Agreement, Publication Delay, and Certification Disclaimer (Graduate School Form 32), this thesis/dissertation adheres to the provisions of Purdue University's "Policy of Integrity in Research" and the use of copyright material.

Approved by Major Professor(s): JAMES H HILL

Approved by: SHIAOFEN FANG 7/12/2016

Head of the Departmental Graduate Program Date

A QUANTITATIVE COMPARISON & EVALUATION OF PROMINENT MARSHALLING/UN-MARSHALLING FORMATS IN DISTRIBUTED REAL-TIME & EMBEDDED SYSTEMS

A Thesis

Submitted to the Faculty

of

Purdue University

by

Geetha R. Satyanarayana

In Partial Fulfillment of the

Requirements for the Degree

of

Master of Science

August 2016

Purdue University

Indianapolis, Indiana

To Amma & Appa

ACKNOWLEDGMENTS

"All that I am, or hope to be, I owe it to my angel mother." I cannot begin to imagine how life would have been if it weren't for all the hardships my mother had to go through, and all the sacrifices she had to make as a single mother to bring my brother and me to where we are today. I cannot thank her enough for her love, her support, her belief in me and her constant motivation to give my best in everything I do. She has always been my biggest strength and inspiration, and will always be. I am very grateful for having such a wonderful mother and I truly owe everything I am to her. I am also eternally thankful to my late father, who has showered his blessings on me, at all times.

I would like to thank my advisor, and *Guru*, Dr. James H. Hill for making me a part of his team, and teaching me good software design and principles. His ideas, support and motivation while doing my research project during the past two years has not only made me a better software engineer, but also a stronger person. I would like to thank Dr. Rajeev Raje, for being on my committee and also for teaching me the concepts of distributed systems, which interested me so much that I want to pursue it as a future career path. I also thank Dr. Mihran Tuceryan for willingly accepting to serve as my committee member.

I would like to thank the Department of Computer Science, IUPUI, for funding the most part of my tuition expenses for my courses and research work at IUPUI, and also giving me the opportunity to be a teaching assistant for the last two semesters, which has been an invaluable experience. I would like to thank Nicole Wittlief for her support in all administrative matters. I would like to thank Meagan Senesac, my supervisor at the Office of Undergraduate Admissions, IUPUI, for giving me the opportunity to work for her, and her support to pursue my academics along with the part-time work. I would like to thank Manjula Peiris and Dennis Feiock, for assisting

me whenever I had trouble moving forward with my project, and helping me achieve my research goals.

I would like to thank the two men in my life, my brother Vaidyanathan, and my fiancé Hayawardh, for their belief in me, that I could achieve everything I want. I truly couldn't have achieved this without their love and support. I would also like to thank all my family, especially my uncle Mr. G Neelakantan and aunt Mrs. Chitra Neelakantan, for their love and moral support. I would like to thank all my friends for helping me during tough times, and being there for me when needed.

Finally, I would like to thank the Almighty, for bringing all these wonderful people in my life, and also for giving me the strength to pursue all my personal and career goals.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABBREVIATIONS

ACE	Adaptive Communication Environment
BSON	Binary JSON
CDR	Common Data Representation
CORBA	Common Object Request Broker Architecture
DAC	Data and Acquisition Controller
DDS	Data Distribution Services
DRE	Distributed Real-time and Embedded
EINode	Embedded Instrumentation Node
FAST	FIX Adapted for STreaming
FIX	Financial Information eXchange
Gbps	Gigabits per second
GIOP	General Inter-ORB Protocol
IDL	Interface Definition Language
JSON	JavaScript Object Notation
OASIS	Open Source Architecture for Software Instrumentation of Systems
OMG	Object Management Group
ORB	Object Request Broker
PDL	Probe Definition Language
UBJSON	Universal Binary JSON
UUID	Universally Unique Identifier

ABSTRACT

Satyanarayana, Geetha R. MS, Purdue University, August 2016. A Quantitative Comparison & Evaluation of Prominent Marshalling/Un-Marshalling Formats in Distributed Real-time & Embedded Systems. Major Professor: James H. Hill.

This thesis demonstrates a novel idea on how components in a distributed real-time & embedded (DRE) system can choose from different data interchange formats at run-time. It also quantitatively evaluates three binary data interchange protocols used in distributed real-time & embedded (DRE) systems: the Common Data Representation (CDR), which collects data "as-is" into a buffer; Binary JSON (BSON), which enables "on the fly" discovery of elements in a message; and FIX Adapted for Streaming (FAST), which is a binary compression algorithm popularly used for data exchange in financial stock market domain. We compare these three data exchange formats to determine if it is possible to minimize the data usage without compromising CPU processing times, data throughput, and data latency. The lack of such a study has made protocols such as CDR popular based on the assumption that collecting data "as-is" will consume less processing time and send with high throughput.

We perform the study in the context of an Open Source Architecture for Software Instrumentation of Systems (OASIS). To perform our study, we modified its existing data interchange framework to flexibly and seamlessly integrate either format, and let the components choose a format at run-time. The experiments from our study shows that as data size increases, the throughput of CDR, BSON, and FAST decreases by 96.16%, 97.23%, and 84.41%, respectively. The increase in packaging and un-packaging times are 1985.12% and 1642.28% for FAST, compared to 3158.96% and 2312.50% for CDR, and 5077.98% and 3686.48% for BSON.

1 INTRODUCTION

Distributed real-time & embedded (DRE) systems are composed of many components, which perform various tasks by interchanging data with each other over a network [1]. Although there are many data interchange protocols available on the market today, DRE system designers use the protocol provided by the architecture they use for the system. For example, *Common Data Representation* (CDR) [2] is a binary data interchange syntax defined by the *Object Management Group* (OMG) that maps OMG's Interface Definition Language (IDL) data types into a low-level binary representation. CDR is the default format used in the Common Object Request Broker Architecture (CORBA) [2] and Data Distribution Services (DDS) [3]. *Binary JSON* (BSON) [4] is a binary discovery interchange format, that enables "on the fly" discovery of elements when the sender and receiver have not agreed upon the elements in a message. BSON is popularly used in architectures that are based on MongoDB [5]. *FIX Adapted for Streaming* (FAST) [6] is a binary compression interchange format, developed by *FIX Protocol Ltd.* (FPL), and is widely used in the financial stock markets to interchange data with high throughput and low latency.

Instead of choosing the default formats specified by the architectures, DRE system designers and developers must gain in-depth knowledge about the available formats, do a quantitative evaluation of their performance, and be able to choose from them based on the requirements for their system. If given a choice, DRE system designers choose simple formats like CDR or Google's ProtoBuf [7], which append new data to a buffer and send it across the network. This popular choice is based on the assumption that such protocols will consume less CPU processing time and latency by simply collecting the data "as-is" and sending it. DRE system designers are not aware of frameworks that can let components choose from different formats at runtime.

To date, there exists no quantitative studies to determine whether binary data compression formats, such as FAST, or binary discovery formats, such as BSON, can perform better than traditional formats, such as CDR, and provide better processing times, throughput, and/or latency. With this understanding, the contributions of this thesis are:

- It details how to engineer a framework that allows DRE system designers to flexibly and seamlessly integrate different data interchange formats so that components can choose what format to use at configuration time.

- It evaluates and compares traditional formats, like CDR, against binary compression formats, like FAST, and binary discovery formats, like BSON, (1) to analyze if it is possible to minimize the data usage without compromising CPU processing times, data throughput, and data latency; and (2) to answer the question if a compression technique, like FAST, will make any difference in today's high speed Internet,which is in the range of Gigabits per second (Gbps).

We perform the quantitative evaluation in the context of an Open Source Architecture for Software Instrumentation of Systems (OASIS) [8]. OASIS is a framework that instruments distributed software systems without *a priori* knowledge of the instrumentation data transmitted over the network. To support our work, we modified the OASIS framework to enable seamless integration of either data exchange format. Lastly, our experiments show that with increase in the data size, the throughput of FAST decreases only by 84.41%, whereas for CDR and BSON it is 96.16% and 97.23% respectively. The latency to send each packet increases by 2510.73% for CDR, 3522.69% for BSON, and 541.3% for FAST. The increase in packaging and un-packaging times are 1985.12% and 1642.28% for FAST, compared to 3158.96% and 2312.50% for CDR, and 5077.98% and 3686.48% for BSON. Our results therefore suggest using compression algorithms like FAST decreases memory usage while not compromising on CPU and network times.

1.1 Thesis Organization

The rest of this thesis is organized as follows: Chapter 2 discusses some similar works that were done to compare different data interchange formats. Chapter 3 provides a brief introduction to OASIS, and the CDR, BSON and FAST data marshalling formats. Chapter 4 gives a detail description of how we re-engineered OASIS to support interchange of different data exchange formats. Chapter 5 discusses our experimental results. Lastly, Chapter 6 concludes this thesis with a brief summary and a future research directions.

2 RELATED WORKS

We discuss related works that compare different data interchange formats in this chapter.

2.1 Native Data Representation

Bustamante et. al. [9] propose a flexible messaging format called the *Native Data Representation* (NDR). In this format, a sender places the data on the wire in the format that it maintains, i.e., it provides a record like structure, and provides the names, the data types, the sizes, and the position of the elements in the record. The receiver too specifies such a structure, and compares the incoming message with its own structure by the element names, without giving much importance to the size or position of the element. Their implementation of NDR, Portable Binary I/O (PBIO), can perform translation if there are differences in byte ordering, size of data representations, or the compiler's structure layout.

They compare PBIO with traditional message passing formats used in high performance computing, such as Message Passing Interfaces (MPI) [10], Remote Procedure Call (RPC) [11], and XML-based marshalling. They make MPI as their baseline, and show that the encoding time of PBIO is better by three times, decoding by one time, than MPI and also show a 45% reduction in the round-trip time from the sender to the receiver.

However, the record format of PBIO is very similar to a FAST message, except the field names and types are exchanged before hand by the sender and receiver. FAST also contains a field presence map in its message that is very similar to NDR. Therefore, the size of message produced by FAST and the time taken to package and send it, will be less than PBIO.

2.2 Lightweight Communication & Marshalling

Huang et. al. [12] describe a lightweight communication and marshalling (LCM), a publisher-subscriber message passing system, which can simplify sending low latency messages in real-time robotics applications. LCM decouples message implementation from its description by providing a platform and language independent type-specification language. It provides easy to use and highly efficient tools for analyzing, marshalling and communicating data. An LCM encoded message begins with a message header, that contains a magic number, the version of the encoding protocol, sequence number, channel number that denotes the channel on which the message arrived, and a 8 byte hash value computed for all the field names, and field type tuples for validating on the receiver side to decide whether the message was correctly received.

In their paper, Huang et. al also examine the bandwidth, latency and message loss of the C and Java implementations of LCM, Inter-Process Communication (IPC) and the TCP transport of Robotic Operating System (ROS) [13]. They also compare the efficiency of the data interchange computations in these three formats, by sending data from a laser scanner, a grayscale image from a camera, a list of 50 points in a path.

Their results show that LCM performs better when both the amount of traffic and the number of subscribers are more. With only one subscriber, IPC performs well, but doesn't scale well when the number of subscribers increase. ROS has high throughput when its maximum queue length is infinity, but this costs message latency. Their encoding experiments showed that LCM C implementation performed better for messages that describe a path and images, whereas ROS C++ implementation was the best for sending data from a laser scanner.

LCM encoding is very similar to CDR encoding in the way it sends the header information. Although this research is very specific to robotics applciations, they provide very good performance comparisons. However, this is different form our

research because they are benchmarking LCM while comparing it to other message passing formats, whereas we are comparing only binary data interchange formats to suggest which one is better to use.

2.3 Self adaptive data marshalling

Andrade et. al. [14] perform a systematic evaluation of Google's Protobuf, LCM, and their own self-adaptive data marshalling approach. Their self adaptive data marshalling approaches uses a delta comparison to monitor two consecutive packets that are sent and only sends the difference. Their research goals were to find how marshalling formats can be systematically evaluated, and to know the performance of these formats under different application-independent contexts.

They perform a quantitative analysis of the three formats for integer and double fields to compare the data sizes of the three protocols. They suggest Google Proto-Buf is a better choice when using all integer fields. When the number of message fields increase LCM sends more compact data. Their self adaptive approach tends to perform better when there are large number of non-integer (double) fields because it assumes that the difference between two consecutive messages sent in a short span of time is very small.

However, Andrade et. al. do not provide any evaluation of performance, network delays, etc., only evaluating based on data sizes. An interesting observation from this research is that their self adaptive approach is very similar to field operators in FAST, which can perform delta, increment, or other operations on data.

Although these related works compare different data interchange formats, they propose a new data interchange format and benchmark them by comparing it with other formats. This is different from this thesis because we show how components can choose a data interchange format at run-time and quantitatively evaluate three existing formats to suggest when to use which format.

3 BACKGROUND

We provide a brief overview of OASIS, and the CDR, BSON and FAST data interchange formats that we used to perform this quantitative analysis.

3.1 Open Source Architecture for Software Instrumentation of Systems

Open Source Architecture for Software Instrumentation of Systems (OASIS) is a service-oriented architecture that instruments DRE systems without *a-priori* knowledge about them. Figure 3.1 gives a high-level overview of the different components in OASIS.

A **Software Probe** collects metrics of interest from DRE systems at the application or system level, such as processor or memory or network information, and sends

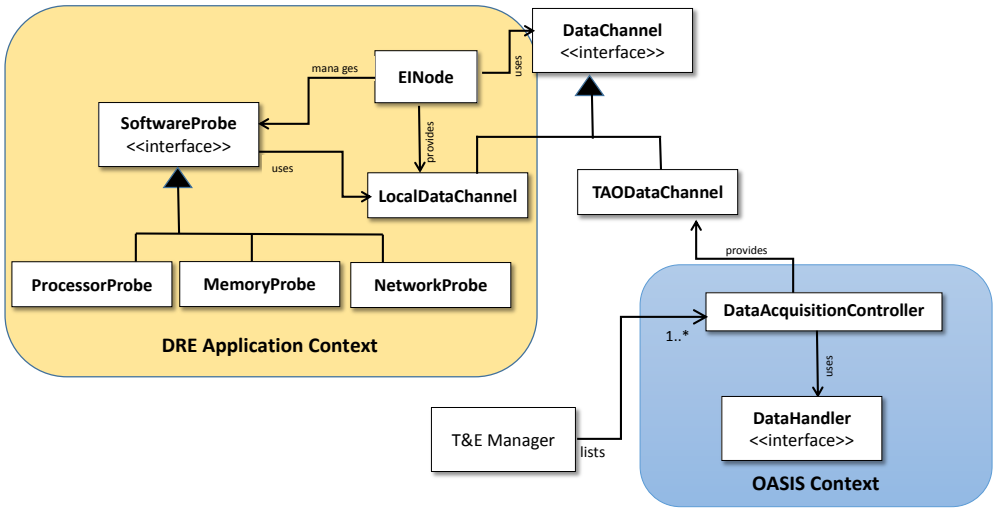

Figure 3.1.: High level overview of OASIS.

it to an Embedded Instrumentation Node (EINode). A *probe definition language* (PDL) compiler generates the stubs and skeletons required for marshalling the data and sending it across the network.

An **Embedded Instrumentation Node** (EINode) manages all the software probes in an application context. An EINode is identified by a *Universally Unique Identifier* (UUID), and a human readable name. In the current OASIS framework, the EINode provides the software probes with a CDR packager to marshall the data. We changed this framework to flexibly integrate other data interchange formats, and this is discussed in details in chapter 4.

A **Local Data Channel** is an abstraction that sends instrumentation data, locally from the different software probes to the EINode in their context. This abstraction allows implementation to strategize the data locally before sending it across the network, such as collecting the packets in a queue or speculating them.

A **Data Channel** abstraction allows the EINode to send instrumentation data to the Data and Acquisition Controller (DAC) over the network. DACs can choose from different middleware technologies, such as CORBA or DDS, to send the instrumented data. The `TAODataChannel` implementation shown in the Figure 3.1, is the CORBA implementation of the data channel.

The **DAC** receives the instrumented data from various EINodes, and controls access to it. External clients can connect to the DAC to get the collected instrumentation data. The DAC also manages various data handlers that un-package the data and store it in a file or database. *JavaScript Object Notation* (JSON) [15] data handler, Websocket [16] data handler, SQLite data handler are some of the data handlers that currently exist in OASIS.

Test and Execution Manager (TnE) is the main entry point for external clients to access the data from the DAC. It acts as naming service to list all the existing DACs.

3.2 Binary JSON

JSON is a data interchange format that transmits human-readable data in the form of attribute-value pairs. It has largely replaced XML in asynchronous communication systems, and enables humans to read and write data easily while also making it easy for machines to parse and generate it. A JSON documents consist of either attribute/value pairs of data or a collection of data, such as arrays or vectors, other embedded JSON documents, or a combination of them. The elements in a JSON document are differentiated by braces or brackets.

BSON is a lightweight binary-encoded data interchange format that serializes JSON documents, and is the primary storage and transfer format in MongoDB. Unlike JSON, elements in a BSON document are stored in an ordered list, consisting of its names, types and values and not differentiated by brackets. Table 3.1 [4] shows element types of various basic data types in BSON.

Element type	**Data type**
0x01	Double - 64-bit binary floating point
0x02	String
0x05	Binary data
0x08	Boolean
0x09	UTC datetime
0x10	32-bit integer
0x11	Timestamp
0x12	64-bit integer
0x0A	Null value
0x03	Embedded Document
0x04	Array

Table 3.1.: Element types in BSON for standard data types

BSON provides a wider range of data types than JSON, and also supports embedded documents and arrays. Custom data types can also be represented using BSON extensions. The "schema-less" property of BSON makes it more flexible when compared to formats such as Protobuf [7]. BSON minimizes the spatial overhead and the element scan speed considerably when compared to JSON—making it lightweight and easily traversable. Because it uses C-like data types, it performs efficient encoding and de-coding of data.

For example, to encode the value `system_time:1345`, where `system_time` is the key, and 1345 is the value, the BSON document would look as shown in Listing 3.1.

```
(0x16 0x00 0x00 0x00) 0x10 (s y s t e m _ t i m e 0x00)
(0x05 0x41 0x00 0x00) 0x00
```

Listing 3.1: Data marshalled using BSON

In Listing 3.1, the parenthesis are for illustration purposes only. The first set of parenthesis represent the size of the document, which is 22. The next byte represents the element type, which is 32-bit integer. The second set of parenthesis represent the null-terminated key `system_time`. The third set of parenthesis represent the value 1345 in hexadecimal. Finally, the document terminates with the null-byte 0x00.

The corresponding JSON document will look as shown in Listing 3.2.

```
{system_time : 1345}
```

Listing 3.2: Data marshalled using JSON

In Listing 3.2, the braces are part of the format. The BSON implementation that we integrated into OASIS is BiSON [17].

3.3 Common Data Representation

CDR is a data interchange format specified by the OMG that maps OMG's Interface Definition Language (IDL) data types into an octet stream for transfer between different Object Request Brokers (ORBs) and Inter-ORBs. Octets are defined as

8-bit values, and an octet stream is a memory buffer that can to be sent over the network. Octet streams have a well-defined beginning, are arbitrarily long, but finite and contain *n* octets, indexed from *0* to *n-1*.

CDR defines two kinds of octet streams: *messages* and *encapuslations*. The basic units of information exchange in General Inter ORB Protocol (GIOP) are messages. A message contains a message header followed by the message data. A message header contains the following fields:

- **Magic** that identifies GIOP messages, whose value is always the four uppercase characters "GIOP" encoded in ISO Latin-1 (8859.1).
- **Version** specifies the major and minor versions of GIOP that is being used.
- **Byte Ordering** specifies if the rest of the message follows big-endian representation, denoted by 0, or little-endian representation, denoted by 1.
- **Message type**: specifies the message type such as request, reply, etc. as an enum value.
- **Message size**: specifies the number of octets in the data that follows the header. This is also encoded by the byte ordering specified above.

Octet streams that are marshalled independently from any particular message context are called encapsulations. An encapsulated octet stream can be represented as the OMG IDL's opaque data type `sequence <octet>`, which can be included in another message or encapsulation for subsequent marshaling without changing the data or the byte ordering in the encapsulation's octet stream.

CDR moves values of primitive data types in and out of octet streams by aligning them at the size of their type. If necessary, the value of a primitive data type will be preceded by an alignment gap, *i.e.*, an unknown value of octets. The alignment gap is equal to the minimum size required to align the primitive that is being followed. Table 3.2 [**?**] shows the alignment requirements for the OMG IDL primitive

Primitive type	**Octet Alignment**
char	1
wchar	1
octet	1
short	2
unsigned short	2
long	4
unsigned long	4
long long	8
unsigned long long	8
float	4
double	8
long double	8
boolean	1
enum	4

Table 3.2.: Octet Alignment requirements of primitive data types in CDR.

types. Alignment permits efficient data handling by different architectures, since no information of the underlying types is given in the message and assumes that the sender and receiver have prior agreement on the types. However, CDR can enable communication between machines that have different byte ordering because it can translate from big-endian to little-endian or vice versa.

For constructed types, such as structures, unions, arrays, sequences, etc. there is no alignment specified except for those of the primitive types of which they are composed.

To send the message `idle_time:1234`, `system_time:3451`, `user_time:2341`, where the values are long (4-byte integer) values in CDR, it would be encoded in big-endian format as shown in Listing 3.3.

```
(00000000 00000000 00000100 11010010)
(00000000 00000000 00001101 01111011)
(00000000 00000000 00001001 00100101)
```

Listing 3.3: Data marshalled using CDR

The parenthesis are for illustration purposes and denote the values of `idle_time`, `system_time` and `user_time`, respectively, along with their alignment. Notice that there is no information about the types of data or the entities that they belong to present in the message. This implies that the sender and the receiver have to agree upon the entities, their types, and their order in the message before hand.

The Adaptive Communication Environment (ACE) [18] implementation of CDR is integrated into OASIS.

3.4 Financial Information eXchange (FIX) Adapted for STreaming

FAST is a binary compression algorithm developed by FIX Protocol Ltd. (FPL) to encode message oriented data streams. It was developed to transmit high volume market data with high throughput and low latency using significant compression.

Data interchange using FAST is performed based on an XML document, called the template, that defines how to encode an instance of a message or group of messages in an application into a stream of bytes. A template is similar to an agreement between the two parties that send and receive FAST encoded data. The template is identified by a name and an id. The template specifies a sequence of instructions, which can be of two types: *field instructions* and *template reference instructions*. Field instructions define the structure of a field (*i.e.*, name and type), the field operator, and the binary encoding representations. Fields can be defined as mandatory or optional. Template reference instructions provide a way to reference parts of a template by

other templates. The order of the instructions in the template is important as it corresponds to order of the data in the data stream.

FAST encoding compresses the data in two stages. First, field operators leverage data similarities in a stream and remove redundant data. This is done by maintaining a dictionary to store the previous values sent so that compressing based on the field operators specified. Next, the remaining data is serialized using binary encoding which uses self-describing field lengths and bit maps that indicate the presence, or absence, of fields. The following field operators can be specified in a FAST template, which help in compression.

- **Constant** operator specifies that a field value never changes.
- **Default** operator specifies to use the initial value, if a field value is not specified.
- **Copy** operator specifies to copy the previous value, if a field value is not present. If a field value is present, then it becomes the new previous value.
- **Increment** operator specifies to increment the previous value by one, if a field value is not present. If a field value is present, then it becomes the new previous value.
- **Delta** operator specifies that a delta value is present in the stream, and the field value can be calculated by adding or subtracting the previous value with the previous, or base value. For ASCII strings, the subtraction length of delta specifies the number of characters to be deleted from the end. If the length is negative, then characters are deleted from the front.
- **Tail** operator specifies that the field value is obtained by combining the tail value with the base value. It is applicable only for ASCII or Unicode strings and byte vectors. For ASCII strings, the length denotes the number of characters to be removed from the back of the base value and the tail value contains the characters to be appended at the end of the base string.

A FAST encoded data stream consists of sequences of messages or blocks, each containing one or more message segments. A message segment consists of a presence map, an optional template identifier, and a sequence of fields and subsegments, if applicable. All integer fields are represented in big endian format.

A key property of the FAST algorithm is stop bit encoding of entities, where the most significant bit in each byte tells whether the next byte is part of the current entity or not. If this bit is set, it means that this is last byte that belongs to the current entity. If it is not set, then the next byte belongs to the current entity. The seven bits after the stop bit are the significant data bits, and correspond to the entity value. The stop bit in each byte of an entity should be removed before calculating the entity value.

Listing 3.4 is a sample FAST template for the Processor probe in OASIS. It consists of three 64-bit unsigned integer fields, with the delta field operator. There can be one or more templates for a message being encoded, and the encoder can select one of them. The compression of a particular message is dependent on the template that is being encoded.

```
1 <?xml version=''1.0'' encoding=''UTF-8''?>
2 <templates xmlns=''http://www.fixprotocol.org/ns/fast/td/1.1''>
3   <template id=''1'' name=''ProcessorProbe''>
4         <uInt64 name=''idle_time''><delta/></uInt64>
5         <uInt64 name=''system_time''><delta/></uInt64>
6     <uInt64 name=''user_time''><delta/></uInt64>
7   </template>
8 </templates>
```

Listing 3.4: FAST template for Processor probe

To send the message `idle_time:1234`, `system_time:3451`, `user_time:2341` FAST would be encoded it as shown in listing 3.5.

1̲1111000 1̲0000001 0̲0001001 1̲1010010
0̲0011010 1̲1111011 0̲0010010 1̲0100101

Listing 3.5: Data marshalled using FAST

The underlined bits are for illustration purposes and represent the stopping bits. The first byte `11111000` represents the presence map of the stream. Here the first bit, 1, tells that it is the last byte in the presence map, the second bit indicates that a template id is present in the stream, and the rest of the bits correspond to each of the fields in the order specified in the template, 1 indicating that a field is present and 0 indicating that a field is absent. There are several rules that govern when fields occupy a bit in the presence map, and when they do not. We do not include the rules here since it is outside the scope of this thesis. You, however, can learn more about the rules in the FAST specification [6].

In the next byte `10000001`, the stop bit 1 indicates that this is the only byte with the template id, and the remaining significant bits denote the template id, 1, that is used. The next bytes in the stream represent the values of the fields, `idle_time`, `system_time`, `user_time`, in the order specified in the template.

To send a second message with the values, `idle_time:1256`, `system_time:3460`, `user_time:2356` it would be encoded as shown in Listing 3.6.

1̲0111000 1̲0010110 1̲0001001 1̲0001111}

Listing 3.6: Second message marshalled using FAST

Here, the first byte is the presence map, whose significant bits indicate that there is no template id present in the stream and the fields `idle_time`, `system_time`, `user_time` are present. The template id contains a copy field operator, which suggests to use the previous value if not present. Furthermore, the values of the fields are the corresponding differences between the values sent in the previous stream, which are `idle_time:12`, `system_time:9`, `user_time:15`.

We have integrated QuickFAST [19], a C++ implementation of FAST written by the OCI Web community, into OASIS. QuickFAST is freely available in open source format at https://github.com/objectcomputing/quickfast.

4 INTEGRATING VARIOUS DATA INTERCHANGE FORMATS INTO OASIS

This chapter discusses how we implemented a novel idea to allow components in a DRE system to choose from various data interchange formats available at run-time, in the context of OASIS. For completeness, we first provide a brief discussion of the current data interchange design in OASIS, and its shortcomings. We then introduce our new framework and how it can be used.

4.1 Challenges with the Existing Data Interchange Framework in OASIS

As discussed in chapter 3, CDR was the only data interchange format supported by OASIS. The UML class diagram of the existing data packaging framework is shown in Figure 4.1. As shown in the UML diagram, the `DataChannel` interface contains a pointer to the `SoftwareProbeDataPackager`. The child class of the `DataChannel`, the `LocalDataChannel`, creates a new `CDRSoftwareProbeDataPackager` and uses it.

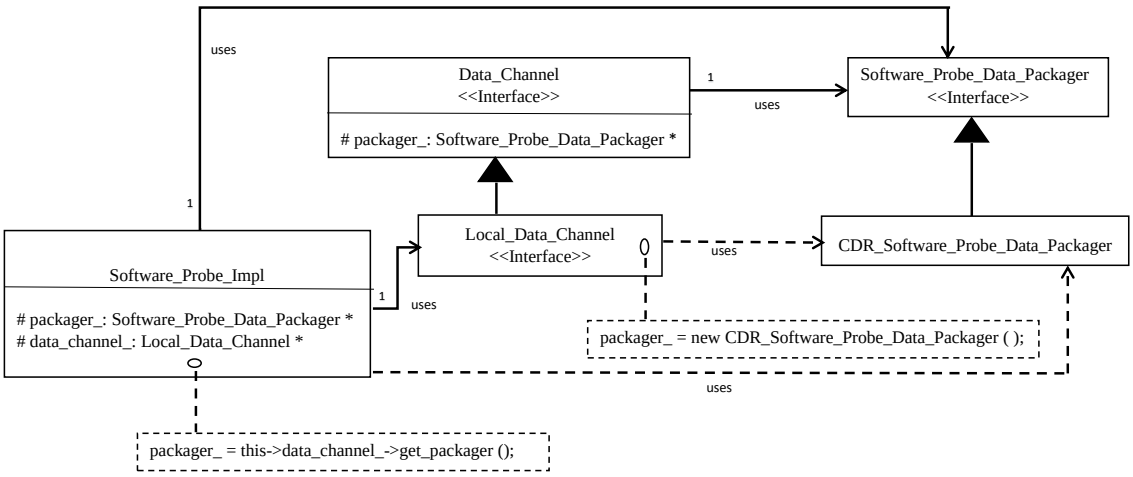

Figure 4.1.: Existing packaging framework in OASIS

The packager created by the `LocalDataChannel`, *i.e*, the `CDRSoftwareProbeDataPackager`, is used by the `SoftwareProbeImpl`, which is the parent class of all the probes in OASIS. When the instrumentation data is available to be sent, the probes use this packager to write the data to a CDR stream. Note that the `CDRSoftwareProbeDataPackager` class in the OASIS framework acts as a Wrapper Facadé [20] to the `ACEOutputCDR` class provided by the ACE framework. The shortcomings of this design are as follows:

1. The `DataChannel` dictates to the `SoftwareProbeImpl` which packager to use. In reality, the `DataChannel`, and its subclasses, must only be responsible for sending out the marshalled data. The two concerns, packaging the data and sending the data, must be independent of each other.

2. This design is not flexible enough to add new packaging formats. Although new packaging formats can inherit from the `SoftwareProbeDataPackager` interface to provide their own implementations, it is not easy for the probes to choose from different packagers available. The probes must be easily able to choose between different packaging formats and switch, if required, based on some run-time configuration.

The rigidity of design is a lot more during the un-packaging that happens while handling data by the DAC. As shown in Figure 4.2, the different handlers registered with the DAC use the `SoftwareProbe` class to un-package the data related to the different probes. The `SoftwareProbe` class and its child classes directly use the `ACEInputCDR`, provided by the ACE framework class to un-package the data.

This is not a hindering design because adding new un-packaging formats requires writing new concrete classes for each of the probes to directly use these formats, which is time-consuming and tedious. Also, because the packaging and un-packaging code is auto-generated by OASIS PDL (see Section 3.1), it would require updating OASIS PDL to support the different data interchange formats.

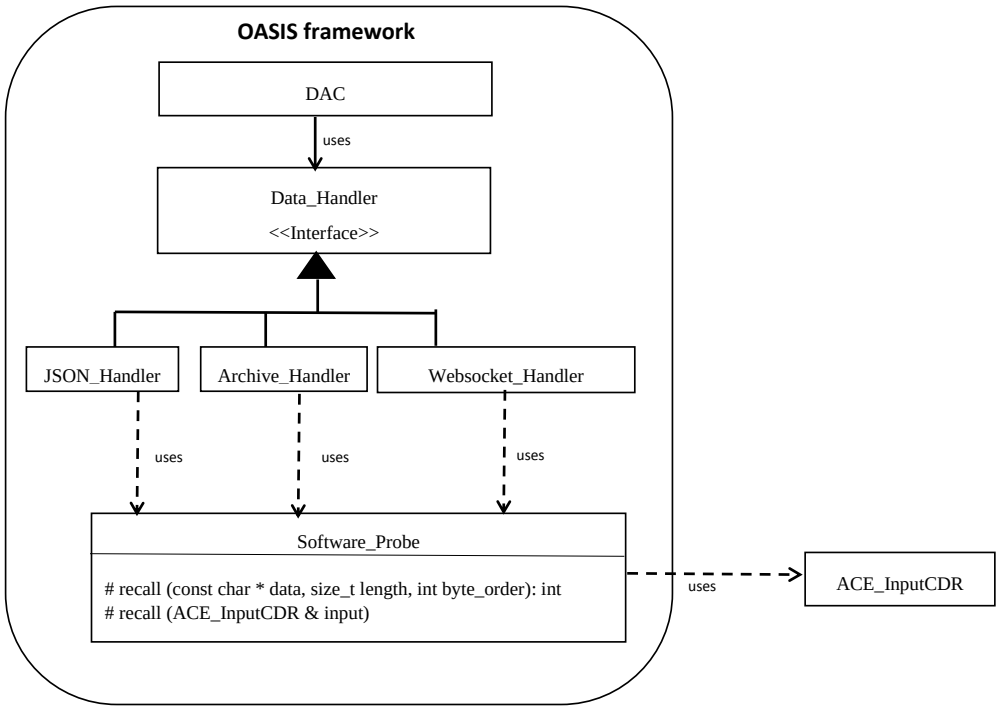

Figure 4.2.: Existing un-packaging framework in OASIS

4.2 The New Data Interchange Framework in OASIS

We redesigned the data interchange framework in OASIS using the strategy and abstract factory design patterns [20], as shown in Figure 4.3. Since the different data interchange formats vary only in the way they package data, *i.e*, their behavior, we chose the strategy pattern to implement them.

The interfaces `DataPackager` and `DataUnpackager` define the required methods for packaging and un-packaging different commonly used data types. These methods appear as writeXXX and readXXX methods where XXX denotes the data type of the element that is packaged. The concrete classes `CDRDataPackager`, `BSONDataPacakger` and `FASTDataPackager` implement its respective algorithm(s) to package data. Similarly, the concrete classes `CDRDataUnpackager`, `BSONDataUnpacakger` and `FASTDataUnpackager` implement its respective algorithm(s) to un-package data.

The abstract factory pattern is used to create the different packagers and un-packagers. The factory enables easy creation of a family of classes, like the different classes in a strategy. The interface `CodecFactory` declares methods to create references to the interfaces `DataPackager` and the `DataUnpackagers`. The concrete factories, *i.e.*, the `CDRCodecFactory`, `BSONCodecFactory` and the `FASTCodecFactory` create the respective concrete data packagers and un-packagers.

Note that the classes that work with the packagers and un-packagers in this system only interact with the `DataPacakger`, `DataUnpackager` and the `CodecsFactory` interfaces. In this way, we are depending on the abstractions, and not the concretions.

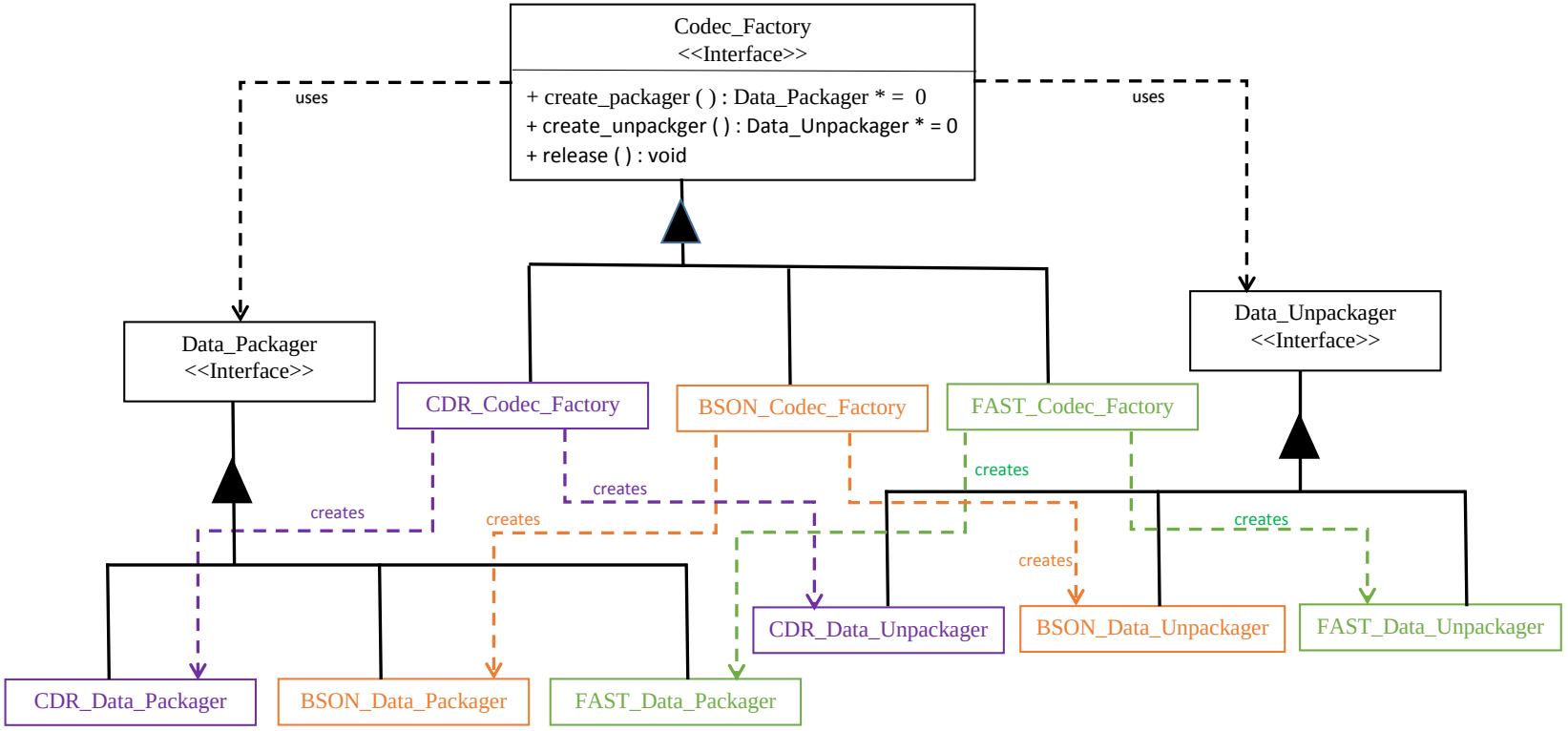

Figure 4.3.: New packaging/un-packaging framework in OASIS

The data interchange format to be used is decided at run-time based on the configuration specified in the EINode, to send the data, and the DAC, to receive the data. This enables the creation of the `DataPackager` and the `DataUnpackager` at run-time, and also acts as an agreement between the sender and receiver on the data interchange format to be used.

A sample EINode configuration file is shown in Listing 4.1. This file specifies the name and UUID of the EINode, the packager to be used, the data channel that connects to the DAC, and the location of the DAC. Specifically, the lines 4 and 5 inform the EINode to load the `__make_fast_codec_factory__()` symbol from the `OASISCodecs` library. The arguments –template= ./FASTCodecsTemplate.xml –template_id=2 –fields=10 are initializing parameters for the packagers. Here, these parameters inform the `FASTDataPackager` the name of the XML template file, the template Id to be used, and the number of fields in the template. Currently, CDR and BSON packagers do not require any initializing parameters.

```
1  Name: Daemon_EINode
2  Uuid: 6429FC2A-0A40-4BB5-BF7F-A0FB19B1BFA8
3
4  Codec: OASIS_Codecs:__make_fast_codec_factory__()
5  "--template=./FASTCodecsTemplate.xml --template_id=2 --fields=10"
6
7  DataChannel:
8  dynamic TaoChannel Service_Object *
9  OASIS_Tao_DataChannel_Client:_make_OASIS_Tao_Data_Channel_Client ()
10 "-ORBInitRef DataChannel=corbaloc:iiop:192.168.2.101:30000/DataChannel --run-orb"
```

Listing 4.1: EINode configuration file

Similarly, the DAC configuration file, as shown in listing 4.2, tells the DAC the un-packager to be used, the registered data handlers and the data channels to be used. The grammar for specifying the un-packager on the DAC is similar to that of the EINode.

```
1  Codec: OASIS_Codecs:__make_fast_codec_factory__()
2  "--template=./FASTCodecsTemplate.xml --template_id=2 --fields=10"
3
4  section DataHandlers:
```

```
5  dynamic JsonFlatfilePublisherService Service_Object *
6  OASIS_DAC_Json_Flatfile_Publisher_Service:
7  _make_OASIS_DAC_Json_Flatfile_Publisher_Service() ""
8
9  section DataChannels:
10 dynamic TaoChannel Service_Object * OASIS_Tao_DataChannel:
11 _make_OASIS_Tao_Data_Channel_Service() ""
```

Listing 4.2: DAC configuration file

The EINode and the DAC use `loadcodecsfactory ()` method in the `CodecsLoader` helper class to load the codec factory that is specified in its configuration, as shown in Figure 4.4. This method takes as parameters the name of the library, *i.e.*, the `OASISCodecs` library, and the name of the method that creates the required factory. It dynamically loads the factory creating method from the library, executes it, and returns the object of the factory that is created. Here the method `__make_fast_codec_factory__()` is a C-style method that returns a new object to the `FASTCodecFactory`. The EINode and the DAC use the factory returned by the `loadcodecsfactory ()` to create the corresponding concrete packager or the un-packager.

Since the EINode is responsible for managing the software probes, as discussed in Section 3.1, it provides the data packager to be used to the `SoftwareProbeImpl` class, and its child classes. Similarly, the DAC passes on the data un-pacakger to be used while notifying the handlers to handle the instrumentation data that is received.

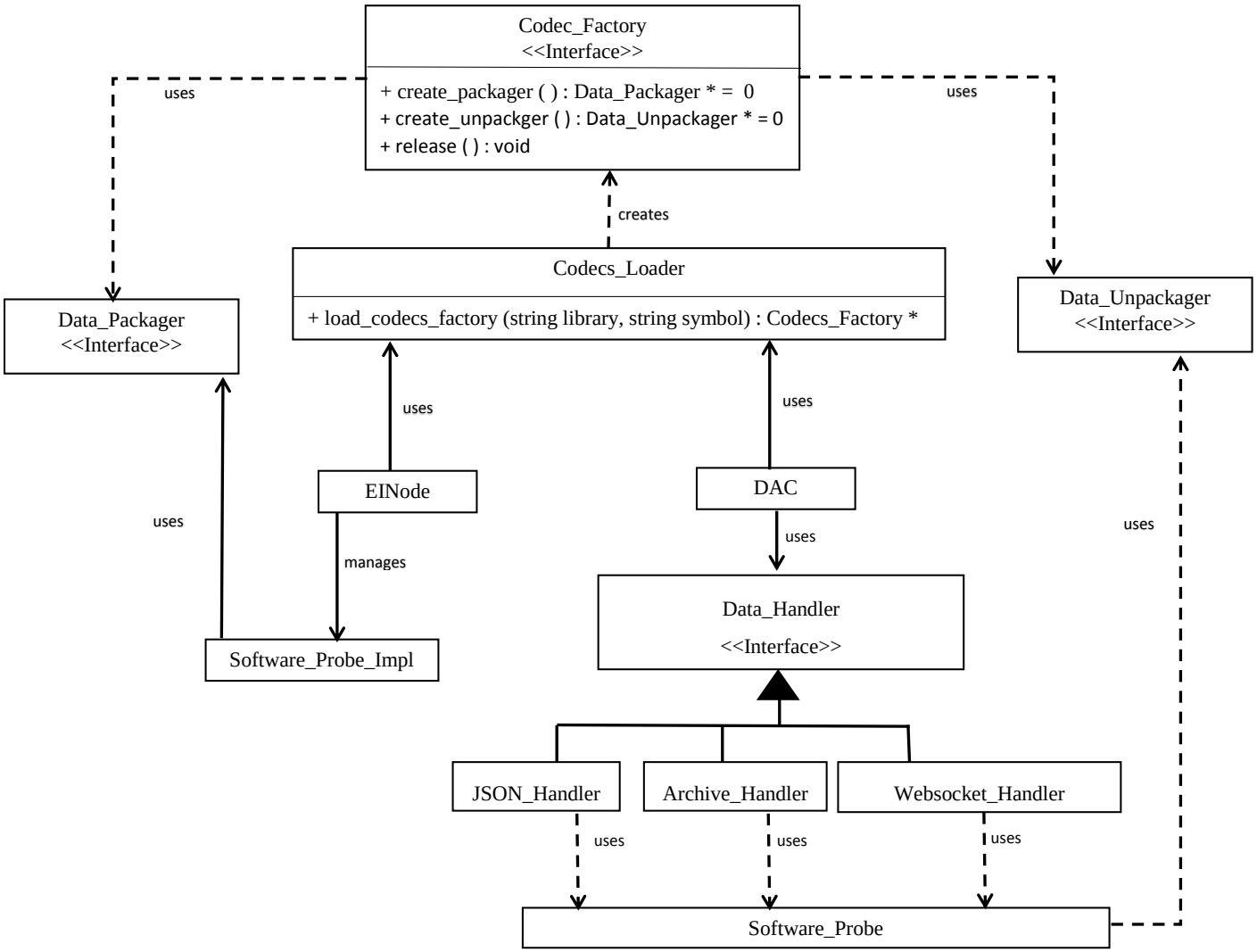

Figure 4.4.: Using the new packaging/un-packaging framework

This design overcomes the two challenges that were inherent in the previous design. It decouples the behaviour for sending the data and packaging the data, making the data channels only responsible for sending the marshalled data across the network. Secondly, it enables the data interchange format to be chosen based on configuration. The strategy pattern for integrating new data interchange formats made it very easy to integrate the FAST and the BSON formats into OASIS.

4.3 Sending Data Encapsulations

OASIS uses CDR messages to send instrumentation data, both locally and over the network, and its contents was discussed in detail in Section 3.3. When CDR messages are used, data marshalled in any format is again marshalled using CDR, adding an extra indirection while sending data. To overcome this challenge, we modified the OASIS framework to send data encapsulations instead of CDR messages.

Two new abstractions, `DataPacket` and `NodePacket`, were defined to send the data. A `DataPacket` contains the data that is locally sent from a probe to the EINode. A `NodePacket` contains one more `DataPacket`(s) sent from the EINode to the DAC, after adding information specific to the EINode. This enables the EINode to collect data packets from one or more probes into a `NodePacket` and send it at once to the DAC. Although the current EINode only sends one data packet per node packet, a future extension can be easily made to collect a few data packets and send it at once.

A data packet contains (a) UUID of the software probe, (b) time stamp when the data packet was sent, (c) sequence number of the packet, (d) the instance name of the software probe, (e) the size of data, and (f) the packaged data, which is the data encapsulation. Similarly, a node packet contains the UUID of EINode that is sending the node packet, the time stamp of when the node packet was sent, and one or more data packets.

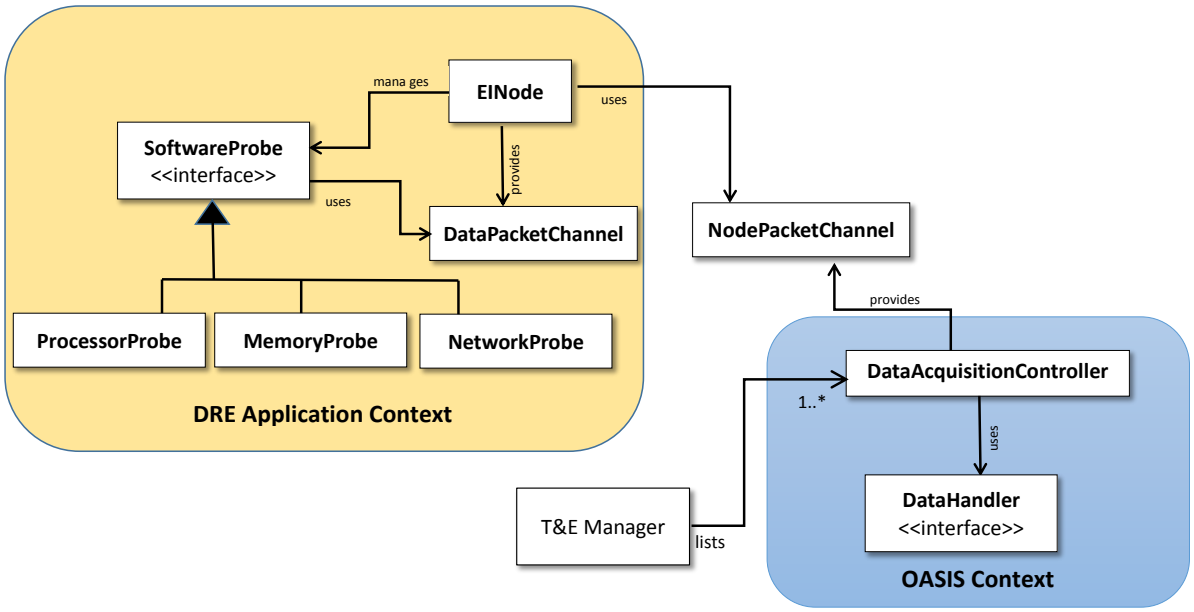

Figure 4.5.: Data Packet Channel & Node Packet Channel

To support these abstractions, and differentiate between sending a data packet and node packet, we removed the `DataChannel` abstraction, and created a `DataPacketChannel` that locally sends a data packet, and a `NodePacketChannel` that sends a node packet across the network, as shown in Figure 4.5.

4.4 Optimizing the Integration of QuickFAST for Better Performance

FAST is a data compression algorithm, designed to send data with high throughput, low latency and minimal CPU costs, as discussed in Section 3.4. However, the `Message` class, used to encode data, provided by the QuickFAST implementation was not designed efficiently to minimize the CPU costs. Our hypothesis and most of our experiments failed, because of excessive memory allocation that increased the CPU time a lot more than CDR and BSON.

After doing some research over the Internet, we found that other QuickFAST users had similar issues while encoding using the `Message` class [21]. The QuickFAST developers acknowledge this issue and state that this design was developed as a generic solution that fits all FAST messages that might be sent using QuickFAST. To overcome this, they suggest not to use the `Message` class that they provide, and instead develop our own class that implements their `MessageAccessor` interface and provide the encoder with the field values, when it calls the getXXX methods of the `MessageAccessor`. By doing this, applications can directly supply values to the encoder from its own data structure, without needing to use the generic QuickFAST message. Therefore, the design of `FASTDataPackager` class, discussed in Section 4.2, was changed to inherit from both the `QuickFAST::MessageAccessor` and the `OASIS::DataPackager` classes to integrate this new design into OASIS.

The probe in OASIS does not use any kind of data structure to relate between the name of a particular element and its value. It directly provides the name of the element (as a string) to be packaged, and its value to the packager by calling the writeXXX methods, making the integration of this new design challenging. To overcome this, the `FASTDataPackager` class in OASIS had to maintain a mapping between the element name and its value, which was flexible enough to hold a value of any data type, or a sequence, or a group of values.

To overcome this challenge, we designed a `FASTField` interface that defines methods for accessing the values of different data types. The concrete implementations of the `FASTField` class contains a reference to a probe variable, and override the accessor methods for the corresponding data type, as shown in Figure 4.6. Currently, the support for FAST sequences and groups is not integrated into this design, but can be done so easily by inheriting from the `FASTField` class and providing appropriate implementation in the `FASTDataPackager`.

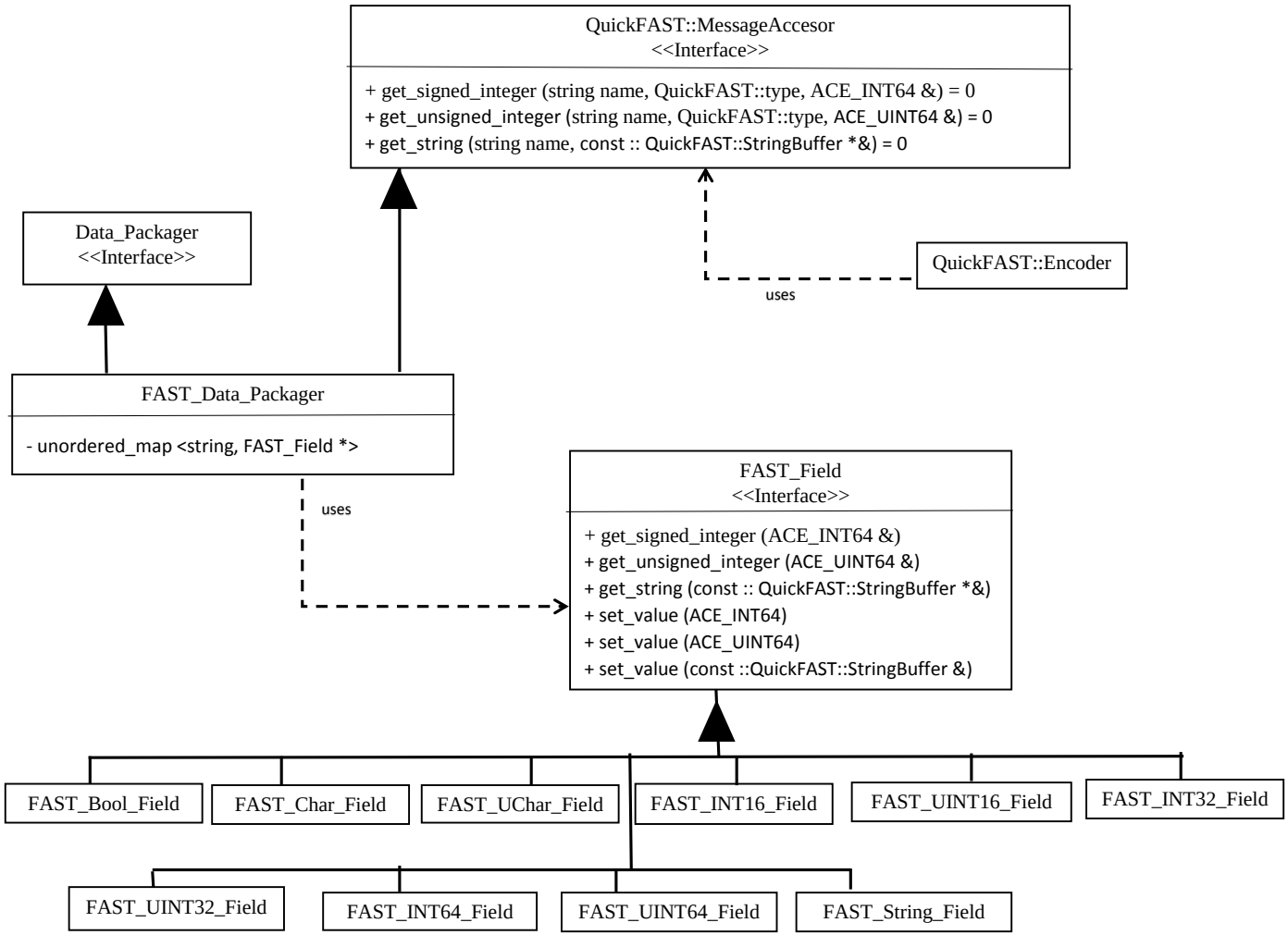

Figure 4.6.: Framework Changes to improve performance of FAST

Finally, we introduced `initXXX` methods in the `DataPackager` class to reduce the excessive memory allocations that take place when software probes call the writeXX methods for every message. The software probes can call the initXXX methods for each of their probe variables to initialize the `DataPackager` with their references. At this time, the `FASTDataPackager` creates appropriate concrete implementation of the `FASTField`, based on the data type of the probe variable, and initializes it with the reference to the probe variable. After this one time initialization, the software probes need not make calls to the writeXXX methods in the `FASTDataPackager` every time new data has to be packaged. When the probe needs to send the packaged data, it calls the `getdata ()` method on the packager, which triggers the FAST encoding process to take place. When the QuickFAST encoder calls the `FASTDataPackager` to provide the values of various fields, it looks up in its map for the field name and provides the value to the encoder. The encoded data is then provided to the probe to send it across the network. A similar framework is designed for the FASTDataUnpackager.

Since CDR and BSON marshall based on copying the values, this one time initialization is not possible and every time a new value has to be written, the software probe needs to call the corresponding writeXXX on their packagers.

To summarize, in this chapter, we discussed the challenges in integrating different data interchange formats into OASIS and how it is possible for components in a DRE system to choose from different formats based on configuration.

5 EXPERIMENTATION, ANALYSIS & DISCUSSION

In this chapter, we discuss some qualitative differences between CDR, BSON and FAST, the experiments that we performed to quantitatively evaluate them and analyze the results from our experiments.

5.1 Qualitative differences between CDR, BSON & FAST

In applications like OASIS, CDR is the most prominently used data interchange formats because it just appends byte data to a buffer and sends it across the network, and is assumed to take less CPU time. However, CDR has various shortcomings because packaging happens based on alignment of the buffer at boundaries of different types.

Consider the example of the CDR message discussed in Section 3.3. To send message `idle_time`:1234, `system_time`:3451, `user_time`:2341, where the values are long (4-byte integer) values, the bytes received by the receiver are shown in Listing 5.1.

```
(00000000 00000000 00000100 11010010)
(00000000 00000000 00001101 01111011)
(00000000 00000000 00001001 00100101)
```

Listing 5.1: Data marshalled using CDR

The challenges while sending and un-packaging these bytes are:

1. The number of bytes sent are more than the number of bytes that actually represent the data. This is because of the boundary alignment of CDR. In this example, although only 6 bytes represent the actual values, the sender sends 4 bytes for each of the three 4-byte integer fields, *i.e*, a total of 12 bytes. This is more when the integer bytes are represented as 8 byte integers, but only 1 or

2 bytes represent actual data. This difference makes a significant slow down in the network when a huge amount of data has to be sent across the network.

2. There is no way the receiver can correspond the received values to the elements they represent, unless the sender and receiver have a prior agreement on the order of value and the data type of values, and their correspondence. In this example, if the order of un-packaging is different than the packaging, then the values of `idle_time`, `system_time`, `user_time` can be interchanged. Furthermore, if they are of different types, there may be exceptions while un-packaging because the un-packager assumes the data is not properly aligned.

BSON eliminates one of the challenges in CDR, by enabling binary discovery. The data sent using the BSON packager contains the name of the element and its value, hence giving correspondence between them and eliminating the chance of misinterpreting the values to other elements. However, this takes up some extra CPU time, and also increases the amount of data that has to be sent across the network because it sends the extra null bytes when the number of bytes in the value is less than the size of the data type, and also adds the field names, data types, etc., as discussed in section 3.2.

FAST compresses the data significantly, sending the bytes that only correspond to the actual values and also compressing the data based on the templates, as discussed in detail in section 3.4, overcoming both the challenges in CDR. However, this compression might take more CPU time, which might have been the reason why it is not as popular as CDR.

5.2 Experimental Setup

Our experiments were run on emulab [22], a network test-bed and emulator, so that we can easily emulate real-time distributed systems. Our experiments consists of two nodes, each running UBUNTU-12 64 bit systems. The EINode runs on one of the nodes where the software probes run as threads to collect different instrumentation

data. The DAC runs on the other node, and receives the data sent by the EINode from the various probes.

The standard system probes in OASIS, like processor probe, memory probe, etc., contain only a few probe elements, 3 to 10 elements, mostly of integer data type. To expand the scope of our experiments, we created some test probes that contain many probe elements, in the range of 10 - 500 elements, of different data types, such as strings, integers, booleans, characters, etc. The stubs for these probes were automatically generated using the PDL files that were written for each of them. Note that the PDL application was changed to generate the new data interchange framework discussed in Chapter 4. We also created the FAST templates for these probes, and each element had one of the various field operations discussed in Section 3.4. Listings 5.2 and 5.3 show a sample PDL file and the corresponding FAST template for a test probe with 10 elements.

```
1  module CodecsTest {
2  [uuid (0D1F63DC-0AC4-43C5-A366-C89D0A5F3ACD); version(1.0)]
3  probe Codecs_Test {
4          boolean bool_value_1;
5          string string_value_1;
6          int8 char_value_1;
7          uint8 uchar_value_1;
8          int16 int16_value_1;
9          uint16 uint16_value_1;
10         int32 int32_value_1;
11         uint32 uint32_value_1;
12         int64 int64_value_1;
13         uint64 uint64_value_1;
14 };
15 }
```

Listing 5.2: PDL file Codecs Test probe

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <templates>
3  <template id="2" name="template2">
4          <int8 name="bool_value_1">
5                  <copy value = "0"/>
6          </int8>
7          <string charset="ascii" name="string_value_1">
```

```
8                   <constant value = "Hello World!"/>
9           </string>
10          <string charset="ascii" name="char_value_1">
11          </string>
12          <string charset="ascii" name="uchar_value_1">
13          </string>
14          <int16 name="int16_value_1">
15                  <delta/>
16          </int16>
17          <uInt16 name="uint16_value_1">
18                  <delta/>
19          </uInt16>
20          <int32 name="int32_value_1">
21          </int32>
22          <uInt32 name="uint32_value_1">
23                  <copy value = "0"/>
24          </uInt32>
25          <int64 name="int64_value_1">
26                  <increment value = "-100"/>
27          </int64>
28          <uInt64 name="uint64_value_1">
29                  <delta/>
30          </uInt64>
31          </template>
32  </templates>
```

Listing 5.3: FAST template for Codecs Test probe

We created a test probe, that inherits from the stubs generated, to randomly change the values of all the elements in the probe. This emulates real-time system instrumentation and sends different element values each time.

Finally, we created a script to perform different experiments, that runs continuously for a specified amount of time to collect the data, package it and send it across the network.

5.3 Experiment 1: Size of packaged data

The goal of this experiment to quantify the amount of data sent over the network in various data interchange formats, and compare the compression rate of FAST with the data sizes of CDR and BSON.

5.3.1 Setup & Results

We setup the experiment as discussed in Section 5.2, and measured the size of the packaged data after the probe packages the data. We ran the experiment in a loop to send 10000 probe packets, to get the size of the data. The results are shown in Figure 5.1.

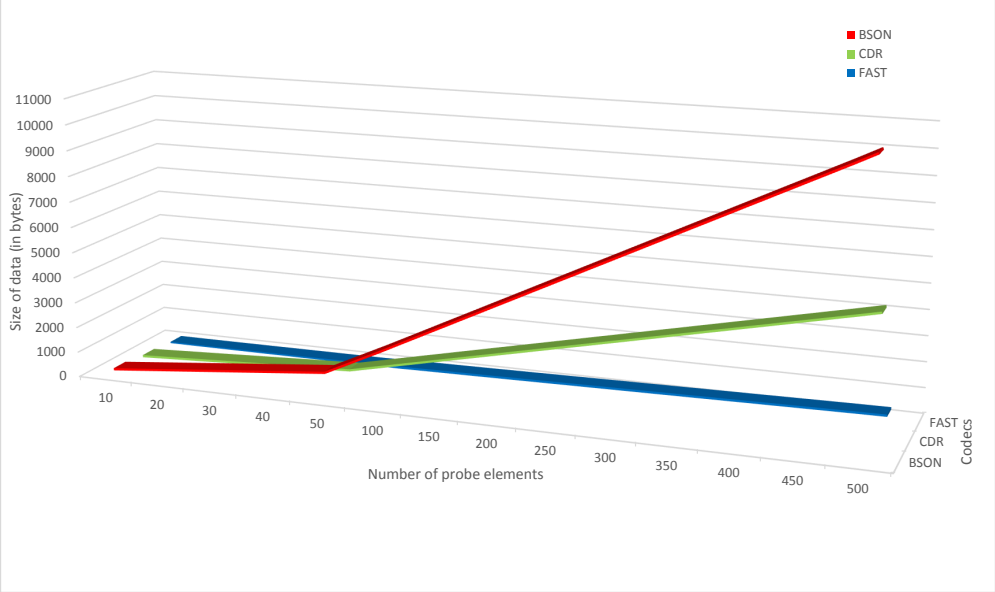

Figure 5.1.: Packaged data size (in bytes) for each of the data interchange for varying number of probe elements.

5.3.2 Analysis

For every probe with *'n'* elements, the data size for CDR and BSON is a constant value for each message packaged. However, the data sizes for FAST vary depending on the values and field operators in the template. For example, an increment field operator on an element does not send anything if its current value = previous value + 1. Otherwise, it sends the new value as the value for that field. Also, the first packet sent using FAST contains all the initial values to initialize the un-packager with. Even if no field operators are specified for one or all of the fields in the FAST template, FAST compresses data considerably because of significant bit encoding that was discussed in Section 3.4. However, not using any field operators defeats the purpose of using an algorithm like FAST. Therefore, the values for FAST shown in the graph represent the maximum number of bytes that were sent for a probe with *'n'* elements, excluding the first packet.

Our results show that the number of bytes sent using CDR is directly proportional to the total size of all the data types of the probe elements. The size for BSON increases quickly because of the addition of field names. However, for FAST, the maximum amount of data sent is compressed, as expected. For smaller amounts of data, *i.e.*, 10 probe elements, when CDR sends 96 bytes of data, BSON sends 215 bytes of data, and FAST only sends 10 bytes, which is 10% of CDR and 4.6% of BSON. For larger number of probe elements, *i.e.*, 500 probe elements, when CDR sends almost 4800 bytes of data and BSON sends 10915 bytes of data, FAST compresses it to send only 279 bytes, which is only 5.8% of CDR and 2.5% of BSON. As discussed above, although the compression is based on the templates and the field operators, the results from this experiment shows that the compression is more when the data is more, because the amount of data for CDR and BSON is proportional to the number of elements.

Our next experiments show how this increase in data relates to sending the data over the network and the CPU processing times.

5.4 Experiment 2: Throughput & Latency

The goal of this experiment is to quantitatively measure how the size of the data, discussed in section 5.3, relates to sending it over the network. For this, we selected the two measures throughput, which denotes the number of messages sent in a given amount of time, and latency, which denotes the time taken to send one message.

5.4.1 Setup & Results

We setup the experiment as discussed in Section 5.2. We emulated the network to send data over TCP/IP at a speed of 8 Mbps. The DAC data handlers were disabled so that the CORBA call made by the EINode to send the message returns back indicating receipt of a message, and for the EINode to continue sending further messages. We did this to emulate real-time systems where the sender does not have to worry about how the receiver handles the data, and is only concerned about the receipt of the packet. We ran the experiments five times for five minutes, and measured the throughput and latency.

Figures 5.2 and 5.3 show the throughput and latency (in microseconds) for each of the data interchange for varying number of probe elements.

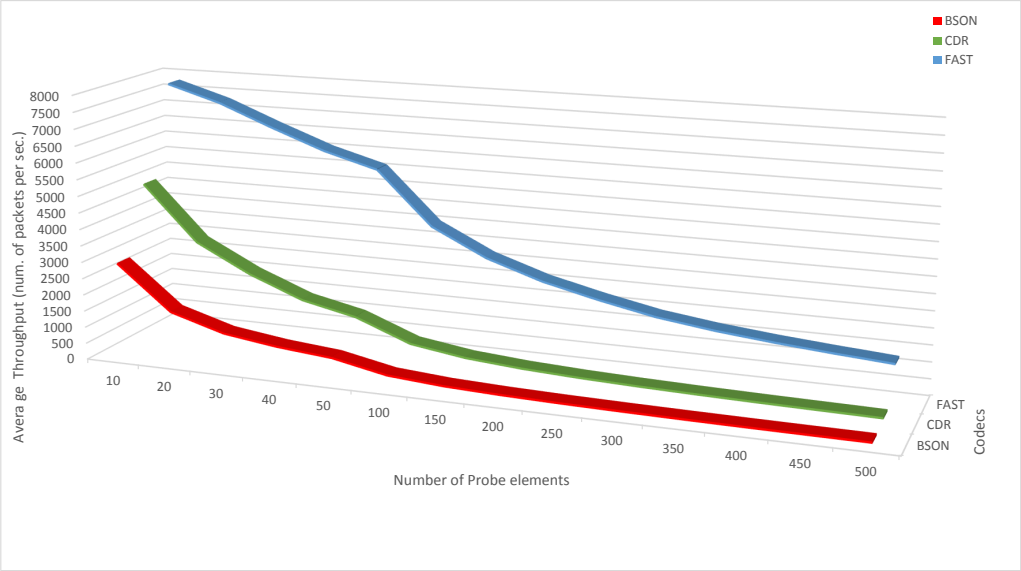

Figure 5.2.: Average Throughput for each of the data interchange for varying number of probe elements.

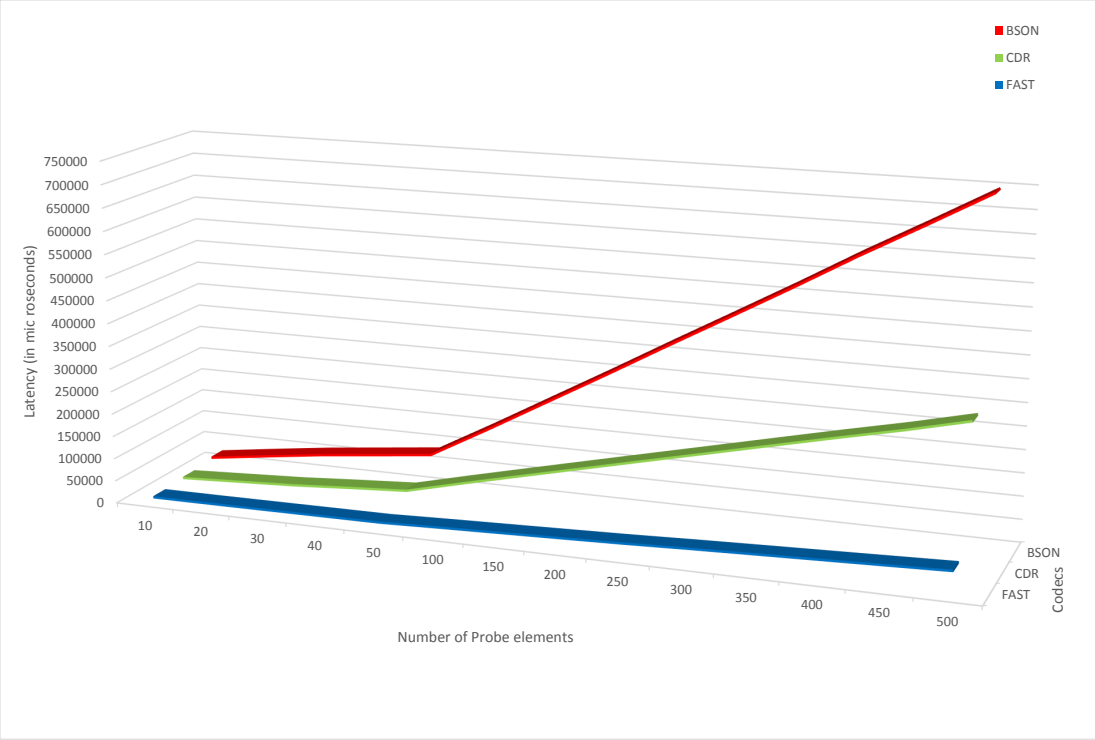

Figure 5.3.: Average Latency (in microseconds) for each of the data interchange for varying number of probe elements.

5.4.2 Analysis

As seen in Figure 5.2, the throughput is higher for FAST as compared to CDR or BSON. The throughput decreases very rapidly for CDR and BSON because of the rapid increase in the data size. However, for FAST the decrease is slow. For 10 probe elements, the throughput is 4959 for CDR, 2911 for BSON and 7740 for FAST. For 500 probe elements, the throughput values decrease to 190 for CDR, 80 for BSON and 1206 for FAST. From this, we calculated the percentage decrease in throughput to be 96.16% for CDR, 97.23% for BSON and only 84.41% for FAST, for the corresponding increase in data sizes from 10 probe elements to 250 probe elements.

Since latency is the inverse of throughput, latency increases very rapidly for BSON and CDR when compared to FAST, as shown in Figure 5.3. The percentage increase in latency for 2510.73% for CDR, 3522.69% for BSON, and 541.3% for FAST.

This experiment shows that an increase in the data size has a direct impact on network, increasing the time taken send each packet, and thus decreasing the number of packets for sent in a given unit of time.

5.5 Experiment 3: Processing times

The goal of this experiment is to evaluate the cost for CPU processing time for packaging and un-packaging the data, for the compression, high throughput and low latency possible through FAST in comparison to CDR and BSON.

5.5.1 Setup & Results

We setup the experiment as discussed in Section 5.2 and calculated the time for the software probe to package its instrumentation data.

To calculate the un-packaging time, we created a no operation data handler for the DAC, that only un-packages the received data and does not perform any other operation such as writing to a file or a database. This makes it easy for us to calculate the time only for un-packaging the data, without including the times for writing to file or database or perform other operations, which was common in other data handlers that were already integrated into OASIS.

We ran the experiment in a loop continuously to send 10000 data packets, similar to calculating the data sizes, and the median of packaging and un-packaging times is shown in Figures 5.4 and 5.5.

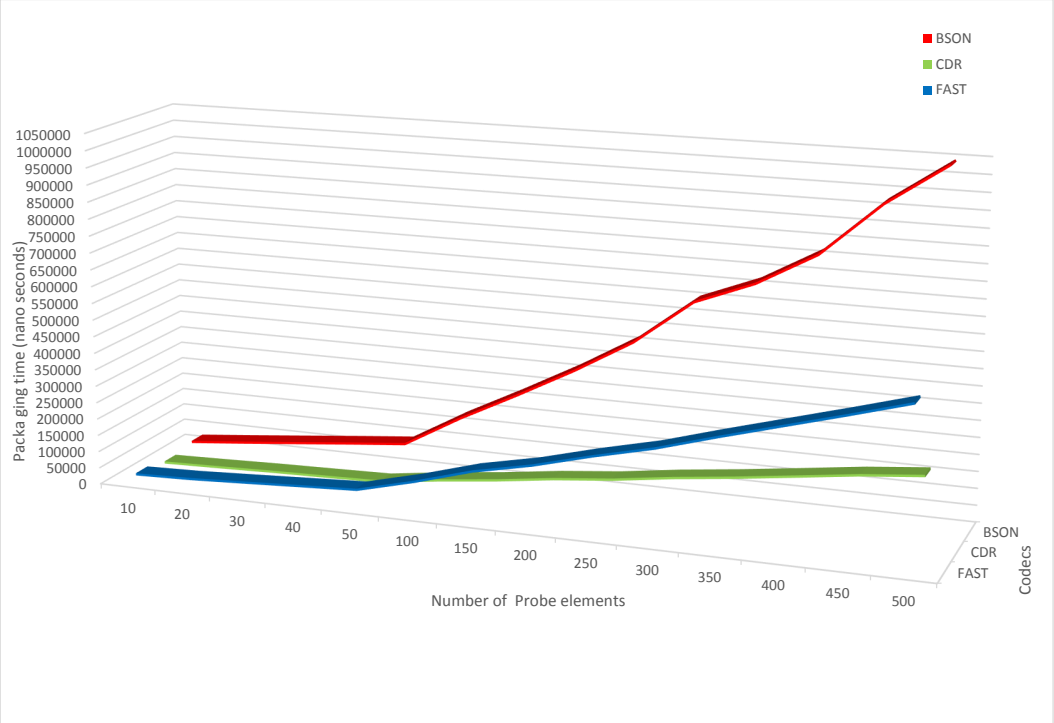

Figure 5.4.: Average Packaging time (in nanoseconds) for each of the data interchange formats for varying number of probe elements.

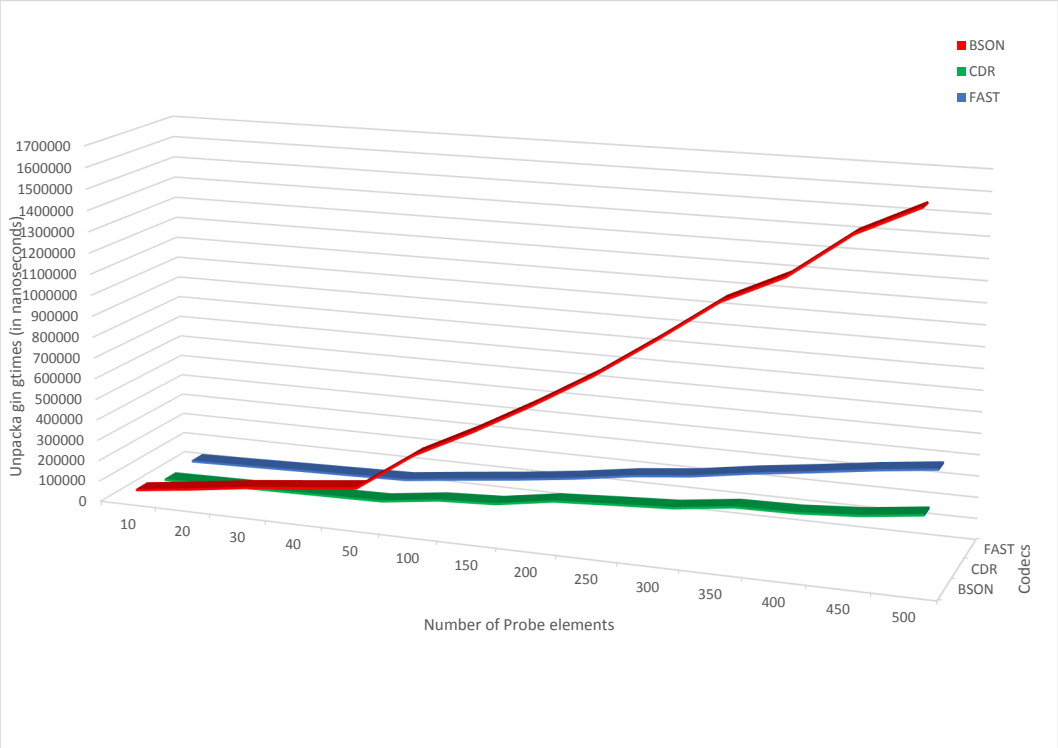

Figure 5.5.: Average Un-Packaging time (in nanoseconds) for each of the data interchange formats for varying number of probe elements.

5.5.2 Analysis

Figure 5.4 shows the packaging times, in nanoseconds, for various data interchange formats for varying data sizes. It shows that CDR takes the least amount of CPU time, since it just appends the data to a buffer. FAST takes a little more time than CDR, but considerably less than BSON. Moreover, our data points show that at 10 elements, the time for packaging in FAST is 3.1 times the time for packaging in CDR. However, this time difference rapidly decrease as the data size increases, making the time for packaging using FAST is 2.1 times that of using CDR for 500 elements. Based on our results, the calculated percentage increase in packaging times from 10

probe elements to 500 elements for CDR, BSON, and FAST are 3158.96%, 5077.98% and 1985.12% respectively.

Figure 5.5 shows the un-packaging times, in nanoseconds, for various data interchange formats for varying data sizes. Similar to the packaging times, CDR takes the least amount of time to un-package the data. However, the percentage increase in un-packaging time with the increase in data size is 2312.50% for CDR, 3686.48% for BSON, 1642.28% and BSON.

By extrapolating the results, we infer that when the data size increases considerably, the time to package and un-package using FAST will become less than that of CDR.

5.6 Discussion 1: Packaging Time Vs. Throughput

From our experiments, we concluded that the packaging time increases while throughput decreases. We wanted to further compare the rate of increase in packaging time with the rate of decrease int he throughput to get a better idea. Figure 5.6 shows a graph between the average packaging time (in nanoseconds) and the throughput (number of packets per second) for different data interchange formats for varying probe number of probe elements.

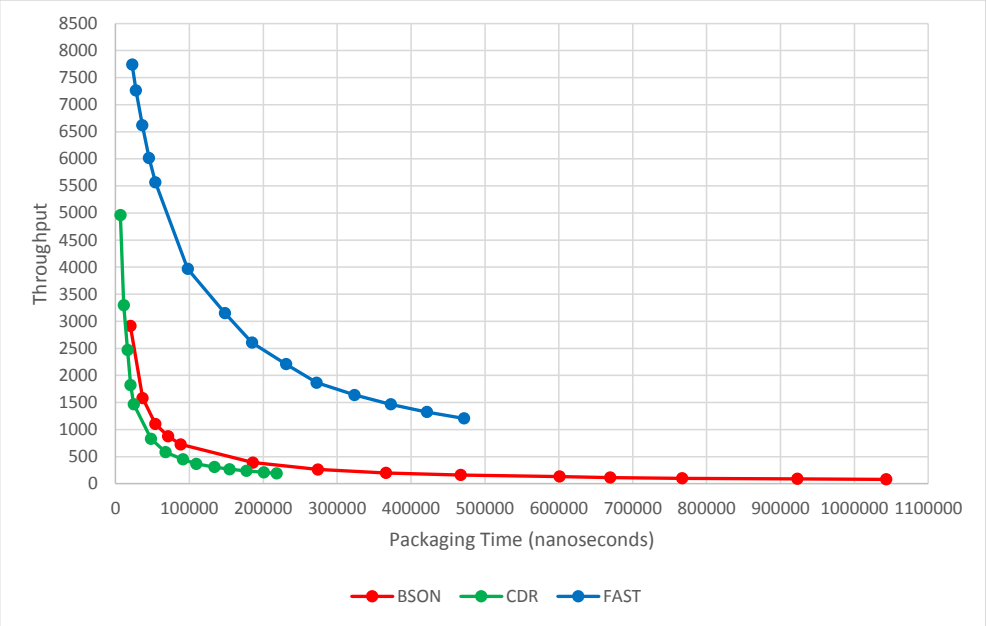

Figure 5.6.: Average Packaging time (in nanoseconds) Vs. Throughput for each of the data interchange formats

Figure 5.6 shows that there is a sharp decrease in the throughput of CDR with slight increase in the packaging time. For FAST, although the packaging time is quickly increasing there is a steady decrease in the throughput. BSON performs the worst with high processing time and very low throughput.

For probe with 10 elements, the ratios of packaging time to throughput are 1.34 : 1 for CDR, 6.9 : 1 for BSON and 2.92 : 1 for FAST. For probe with 500 elements, the same ratios are 1147.62 : 1 for CDR, 12977.15 : 1 for BSON and 391.17 : 1 for FAST.

We can, therefore, conclude that although the packaging time for FAST is more, the throughput is very high when compared to CDR or BSON. Also, the rate of decrease in throughput with increase in packaging time is very less for FAST, when compared to CDR and BSON.

5.7 Discussion 2: Packaging Time Vs. Latency

Similar to the previous discussion, we also wanted to compare the rate of increase in the packaging time with the rate of increase in latency. Figure 5.7 shows a graph between packaging time (in nanoseconds) and latency (in microseconds) for different data interchange formats for varying probe number of probe elements.

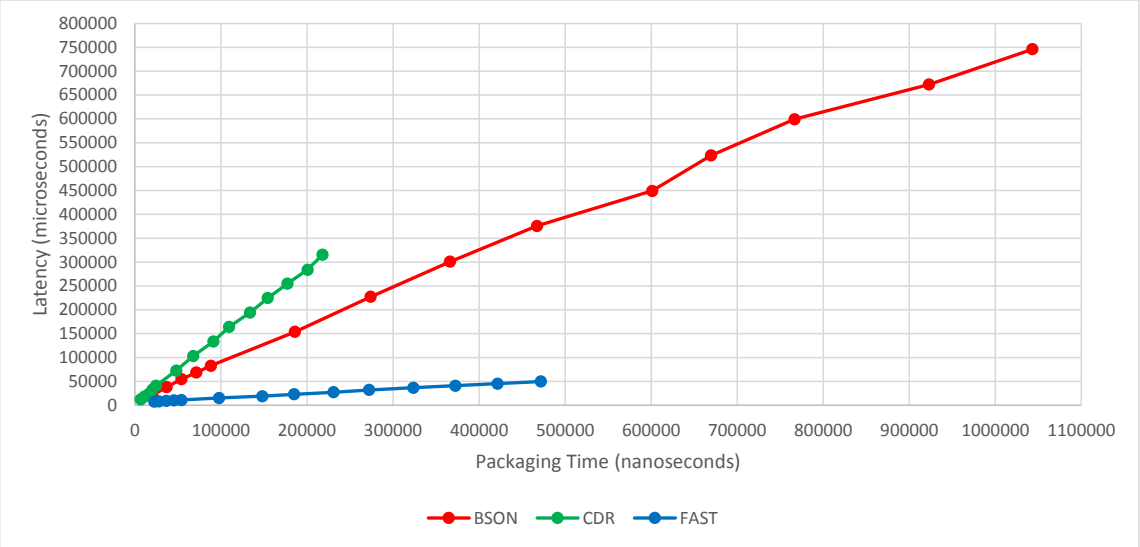

Figure 5.7.: Average Packaging time (in nanoseconds) Vs. Latency (in microseconds) for each of the data interchange formats

In section 5.4 we saw that latency increases as the data size increases. From Figure 5.7, we observe that latency increases with packaging time too. The increase in latency for FAST with increase in packaging time is smaller than that in CDR and BSON. For probe with 10 elements, the ratios of latency to packaging time (both in microseconds) is 1806.93 : 1 for CDR, 1022.10 : 1 for BSON, and 342.21 for FAST. The ratios for probe with 500 elements are 1447.52 : 1 for CDR, 715.09 : 1 for BSON and 105.25 : 1 for FAST.

From the ratios, it is clear that although the packaging time is more for FAST, the increase in latency with increase in packaging time is minimal when compared to

CDR and BSON. We can, therefore, say that FAST sends data with low latency even if the packaging time is higher than CDR.

5.8 Discussion 3: Extrapolation of Results

Although, we did not have enough data to send very large sizes of data and did not have enough resources to emulate very high speed network connections, in the range gigabytes per second, we can infer some valid conclusions for such kind of data and networks from the results that are summarized above.

Consider a probe with 15,625,000 elements, all as 64 bit integers, which implies the actual size of data to be 1,000,000,000 bits (1 Gb). This acts as a good example to discuss the impact of sending such huge amount of data over very high speed network, *for e.g*, 1 Gigabit per second.

Based on the current percentages of probe with 500 elements, CDR sends almost 1.2 Gb of data, and BSON sends almost 2.73 Gb, whereas FAST sends only 66.52 Mb of data. Based on result 4, we infer that the time for packaging/un-packaging each packet with CDR becomes considerably slower than FAST. With the speed of 1 Gbps, CDR will take almost 1.2 seconds to send those packets, and BSON will take a little over 2.7 seconds, whereas FAST will send it across in 66.5 milliseconds. Note that the packaging time and size of data is based on the current percentages, and the actual sizes may slightly vary due to the rate of increase.

From this, we infer that although CDR conserves processor time to marshall smaller amounts of data, the network latency decreases. For huge data sizes, the processing performance of CDR also starts to decrease, along with the decrease in network latency. However, CDR can be used when one wants to send data as is, and does not mind paying for the extra CPU and network costs. Similarly, although BSON enables "on the fly" discovery of elements, it also consumes too much of CPU and network for any size of data. However, from our experiments we can conclude that binary compression algorithms like FAST perform better, conserving the mem-

ory, processor and network, when huge amounts of data have to be sent across the network.

5.9 Summary of Experimental Results

To conclude this chapter, we provide a brief summary of the results and discussion.

1. As the data size increases, the compression of data in FAST increases in comparison to CDR and BSON.

2. As the data size increases, the throughput decreases. However, the rate of decrease is significantly less in FAST than in CDR and BSON.

3. As the data size increases, the latency increases. However, the rate of increase in latency using FAST is significantly less than the rate of increase using CDR and BSON.

4. Although the packaging and un-packaging times for FAST is more than CDR, our experiments show that the rate of increase in these times for FAST is significantly less than CDR and BSON.

5. The rate of decrease in throughput with increase in packaging time is very less for FAST, when compared to CDR and BSON.

6. For FAST, the increase in latency with the increase in packaging time is minimal when compared to CDR and BSON.

7. By extrapolation, we can say that FAST conserves memory, processor and network when huge amounts of data has to be sent over high speed network.

6 CONCLUSION & FUTURE WORKS

Although there are many data interchange protocols in market today, DRE system designers use the ones provided by the architecture they are using or use simple formats such as CDR that append data to a buffer "as-is". Instead DRE system designers, must understand the protocols, and be able choose them based on a quantitative analysis. To help DRE system designers with making such choices this thesis has implemented a novel idea that can allow components in a DRE system to choose a data interchange format at run-time and has quantitatively compared traditional formats like CDR, with binary discovery formats like BSON, and binary compression formats like FAST.

Our experiments showed that with increase in the data size, the throughput of FAST decreases only by 84.41%, whereas for CDR and BSON it is 96.16% and 97.23% respectively. The latency to send each packet increases by 2510.73% for CDR, 3522.69% for BSON, and 541.3% for FAST. The increase in packaging and unpackaging times are 1985.12% and 1642.28% for FAST, compared to 3158.96% and 2312.50% for CDR, and 5077.98% and 3686.48% for BSON. Therefore, our results suggest using compression algorithms such as FAST decreases memory usage, while not compromising on CPU and network times.

FAST has revolutionized the way in which financial institutions send high volumes of market data, and this should also be the "go-to" format for DRE system developers if they want to minimize data usage, without compromising on the CPU, network and memory usage provided by formats such as CDR. It is worth mentioning that various commercial implementations of the FAST marshalling format, other than the open source QuickFAST that we have used, provide better performance, and may out perform CDR better than the QuickFAST implementation.

However, formats such as CDR and BSON are useful in certain scenarios. Our experiments show that CDR marshalling processes smaller data sizes quicker than FAST, although it takes slightly more memory and network. On the other hand, BSON is a very good marshalling format when there is no *a priori* agreement between the sender and receiver about the messages that are being sent. It learns about the data "on the fly", and thus no two packets have to be the same.

6.1 Future Works

Based on the contribution from this thesis, we can move ahead in the following research direction.

6.1.1 Algorithm to choose from different formats based on the data

Our research showed that some data interchange protocols, such as CDR, are better for smaller sizes of data, while protocols such as BSON enable binary discovery, and protocols like FAST provide significant compression. Although FAST out performs CDR and BSON, these formats should be chosen depending on the nature of the application, and of the data being sent.

An interesting research direction is to develop an algorithm that can switch between different formats based on the different properties of data. The algorithm can also learn about the data for a period of time, and choose the best one. This can be done by comparing the processing times, memory usage, and network latency that we suggested.

6.1.2 Comparison of different binary compression formats

Our research concluded that sending compressed data, using algorithms such as FAST, minimizes data usage while not compromising on the processing times and network latency. Although, we used FAST in our research study, there are several other

compression formats such as UBJSON [23] that compress data. It would be an interesting research to understand more about such formats and perform a quantitative analysis about them.

REFERENCES

REFERENCES

[1] George F Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed systems: concepts and design*. pearson education, 2005.

[2] Object Management Group. *The Common Object Request Broker: Architecture and Specification Version 3.1, Part 1: CORBA Interfaces*, OMG Document formal/2008-01-04 edition, January 2008.

[3] Object Management Group. *Data Distribution Service for Real-time Systems Specification*, 1.2 edition, January 2007.

[4] Bson. http://bsonspec.org. Last Accessed: June, 2016.

[5] Mongodb. https://www.mongodb.com/. Last Accessed: July, 2016.

[6] Fast spcification. http://www.fixtradingcommunity.org/pg/structure/tech-specs/fast-protocol. Last Accessed: July, 2016.

[7] Google's protobuf. https://github.com/google/protobuf/. Last Accessed: June, 2016.

[8] James H. Hill, Hunt Sutherland, Paul Staudinger, Thomas Silveria, Douglas C. Schmidt, John M. Slaby, and Nikita A. Visnevski. OASIS: A Service-Oriented Architecture for Dynamic Instrumentation of Enterprise Distributed Real-time and Embedded Systems. In *Proceedings of 13th IEEE International Symposium on Object/Component/Service-oriented Real-time Distributed Computing (ISORC)*, Carmona, Spain, May 2010.

[9] Fabian Bustamante, Greg Eisenhauer, Karsten Schwan, and Patrick Widener. Efficient wire formats for high performance computing. In *Supercomputing, ACM/IEEE 2000 Conference*, pages 39–39. IEEE, 2000.

[10] Ewing Lusk, S Huss, B Saphir, and M Snir. Mpi: A message-passing interface standard, 2009.

[11] Bruce Jay Nelson. Remote procedure call. Technical report, Carnegie-Mellon Univ. Dept. Comput. Sci., 1981.

[12] Albert S Huang, Edwin Olson, and David C Moore. Lcm: Lightweight communications and marshalling. In *Intelligent robots and systems (IROS), 2010 IEEE/RSJ international conference on*, pages 4057–4062. IEEE, 2010.

[13] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y Ng. Ros: an open-source robot operating system. In *ICRA workshop on open source software*, volume 3, page 5. Kobe, Japan, 2009.

[14] Hugo Andrade, Federico Giaimo, Christian Berger, and Ivica Crnkovic. Systematic evaluation of three data marshalling approaches for distributed software systems. In *Proceedings of the Workshop on Domain-Specific Modeling*, pages 71–76. ACM, 2015.

[15] D. Crockford. JSON: Javascript object notation, 2006.

[16] P. Lubbers and F. Greco. HTML5 Websockets: A Quantum Leap in Scalability for the Web, 2011.

[17] Bison. http://github.com/seds/bison. Last Accessed: July, 2016.

[18] Stephen D. Huston, James C. E. Johnson, and Umar Syyid. *The ACE Programmer's Guide*. Addison-Wesley, Boston, 2002.

[19] Quickfast. http://www.ociweb.com/products/quickfast/. Last Accessed: April, 2016.

[20] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1997.

[21] Quickfast google group discussion. https://groups.google.com/forum/#!topic/quickfast_users/PVg34WsBckE. Last Accessed: May, 2016.

[22] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. An integrated experimental environment for distributed systems and networks. pages 255–270, Boston, MA, December 2002.

[23] Ubjson. http://ubjson.org/. Last Accessed: July, 2016.