UNIVERSITY OF GOTHENBURG

Evaluating Data Marshalling Approaches for Embedded Real-Time Systems on the Example of Autonomous Scaled Cars

Bachelor of Science Thesis in the Programme Software Engineering & Management

VICTOR BOROSEAN
SAULIUS EIDUKAS
ANDY DANG

University of Gothenburg
Chalmers University of Technology
Department of Computer Science and Engineering
Göteborg, Sweden, June 2015

The Author grants to Chalmers University of Technology and University of Gothenburg the non-exclusive right to publish the Work electronically and in a non-commercial purpose make it accessible on the Internet.
The Author warrants that he/she is the author to the Work, and warrants that the Work does not contain text, pictures or other material that violates copyright law.

The Author shall, when transferring the rights of the Work to a third party (for example a publisher or a company), acknowledge the third party about this agreement. If the Author has signed a copyright agreement with a third party regarding the Work, the Author warrants hereby that he/she has obtained any necessary permission from this third party to let Chalmers University of Technology and University of Gothenburg store the Work electronically and make it accessible on the Internet.

Evaluating Data Marshalling Approaches for Embedded Real-Time Systems on the Example of Autonomous Scaled Cars

VICTOR BOROSEAN
SAULIUS EIDUKAS
ANDY DANG

© VICTOR BOROSEAN, June 2015.
© SAULIUS EIDUKAS, June 2015.
© ANDY DANG, June 2015.

Examiner: JAN SCHRÖDER

University of Gothenburg
Chalmers University of Technology
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden June 2015

Contents

1 Introduction **2**

2 Related Work **4**

3 Methodology **5**

4 Environment **7**

5 Implementation **8**
5.1 Message Structure 8
5.1.1 Netstrings 8
5.1.2 LCM 9
5.1.3 Protobuf 10
5.1.4 ROS 11
5.2 Data Flow 12
5.2.1 Netstrings 12
5.2.2 LCM, Protobuf, and ROS 14

6 Data Collection **16**

7 Data Analysis **16**

8 Findings **17**
8.1 Descriptive Expressiveness 17
8.2 Performance 19
8.3 Serialized Message Size 23

9 Discussion and Conclusion **25**
9.1 Validity Threats 25

10 Future Work **26**

11 Summary **26**

12 Acknowledgments **27**

Abstract

This paper conducted a comparison between four different data marshaling approaches (Netstrings, LCM, Protobuf, and ROS) that were implemented in the OpenDaVINCI environment and evaluated from different perspectives. In this study we compared the data marshaling approaches on the low-level, focusing on the overhead in terms of control data and encoding/decoding for data structures that are common to OpenDaVINCI. Further, the data marshaling approaches were compared in terms of descriptive expressiveness with the current interface description language ODVD.

1 Introduction

One of the most important factors in designing complex systems is to divide it into different components in order to improve maintainability, fault isolation, and code reuse [5] [9]. For this reason, it is important to assure a reliable and fast exchange of information between the system components.

OpenDaVINCI [8] is a run-time environment that is built upon multiple components which require a continuous and robust connectivity between them. By default, it uses the Netstrings data marshaling approach. Marshaling is the process of converting data to a suitable format for sending data between different software components. Various marshaling approaches provides diverse features and performance rates.

The purpose of this study is to compare different data marshaling approaches in terms of controlling data and encoding/decoding for OpenDaVINCI ordinary data structures. Three different approaches were chosen as analogies to the current implementation (Netstrings) [6]: Lightweight Communications and Marshaling (LCM) [4], Google Protocol Buffer (Protobuf) [3], Robot Operating System (ROS) [10]. These three marshaling approaches were chosen based on their capabilities and use in similar to OpenDaVINCI environments. For instance, LCM was used in the 2007 DARPA Urban Challenge[2], ROS itself is an operating system for robots, and Protocol Buffers is data interchange format for resource-constrained, distributed and embedded real-time systems [11].

LCM is described as a new message passing system for inter-process communication that is specifically targeted for the development of real-time systems [4]. It places great emphasis on simplicity and usability from the perspective of a system designer [4]. LCM stands for Lightweight Communications and Marshaling, because of its functionality and simplicity in both usage and implementation [4]. It is very similar to XDR, but it is designed with an emphasis on type safety [4].

ROS is an open-source meta-operating system for robots, which provides services such as hardware abstraction, low-level control of the device, message passing between processes, package managements, and commonly-used functionality. ROS also provides libraries and tools for different robotics systems. It is widely used in different real-time systems similar to OpenDaVINCI. [10].

Protobuf is created and widely used by Google. It supports different variety of platforms and programming languages. Protobuf's main advantages, compared to XML are that it is simpler, smaller, faster, and less ambiguous. In order to see how it affects overall performance, we choose to evaluate it and compare it with other three approaches.

This study contributes by providing OpenDaVINCI three more solutions towards a better message marshaling capability. This comparison plays a key role in improving a system's robustness and effectiveness. It also provides an outline regarding performances towards serialization capabilities of each of the data marshaling approach that can be taken in consideration in similar practices.

The aim of this research is to find out which approach is most preferable for users in terms of control data and encoding/decoding for typical data structures on real-time systems. Furthermore, this research compares the descriptive expressiveness property of each approach with the ODVD language. ODVD is a descriptive language that defines the data types that can be expressed within OpenDaVINCI environment. Descriptive expressiveness is the property that describes what types of data can be expressed through the marshaling approach.

We have included two research question in our study:

RQ1: How efficient are the marshaling approaches (Netstrings, LCM, Protobuf, and ROS) for encoding/decoding messages to the use of the algorithms within simulation environment?

With the first research question, we were interested to assess efficiency. Efficiency is a criteria that includes the performance, in terms of time it takes to serialize/deserialize messages, and the size of the message after serialization process. These two aspects are important criteria for the data marshaling approach.

RQ2: How expressive are the interface description languages of the different marshaling approaches (LCM, Protobuf, and ROS) compared to the existing ODVD interface description language?

With the second research question, we were interested to find out about data types that marshaling approaches can express within their descriptive language. Furthermore, we are interested to compare their descriptive languages with the ODVD.

In this paper we discuss related works that have common ground with our intentions and have studied different properties of data marshaling approaches. Further, we describe the research methodology that involves techniques towards investigating problems and finding the solution. We describe the environment where we have later proceed with implementing the approaches. Later, this study describes the implementation and evaluation phases of the data marshaling approaches on the low-level in one single environment. We compare performances and properties of implemented data marshaling approaches (LCM, Protobuf, and ROS) with the current implementation (Netstrings) and have a discussion of the findings. We include a conclusion and list the threats that can influence the validity of our findings. Lastly, we include suggestions for a future work, and a summary of the study.

2 Related Work

Researches regarding communication between components gets, by each year, more studied in different applications and devices. Many studies describe different data marshaling approaches in order to bring up the differences between them, as well as their different possibilities for utilization [11][13]. After analyzing the "*LCM: Lightweight Communications and Marshalling*"[2] study we found a comparison in terms of performance between LCM, ROS, and IPC. In that study, authors examined the latency, bandwidth, and message loss of the three approaches under different conditions. In the same study, they described and compared the computational marshaling performance between LCM, ROS, and IPC. Three different message types were used to perform the comparison:

- a 640x480 grayscale camera image;
- a single scan of a planar laser range scanner;
- a list of 50 waypoints.

Results showed that the LCM implementation in C language is the fastest when it was marshaling the 640x480 grayscale camera image, and ROS proved to be the fastest when marshaling the scan of a planar laser range scanner[2]. The comparison was focused on the transfer rate of messages and how reliable it is. Furthermore, the comparison was made by using different approaches implemented in different programming languages (Java, C, and C++). In the same study it was stated that the evaluation was performed for the specific scenarios. Therefore, performance may vary with the packet size and message structure[2]. In our case, we evaluated three different type messages which were sent during parking scenario, with a focus on time it takes to serialize and deserialize data. Thus, packet size and message structure differs from the study mentioned above.

The study "*Using Protocol Buffers for Resource-Constrained Distributed Embedded Systems*"[11] presented Protobuf-Embedded-C compiler. The main focus of this paper was to deploy Protobuf generated code on the embedded controllers with limited amount of memory. In case presented by the paper it was 40KB of static memory. It is partly related to our research, because we were interested in the *message size* after serializing data, which is part of efficiency criteria. Protobuf-Embedded-C compiler does not support some data types e.g. double, int32, bytes, uint32, uint64, fixed32, fixed64, at the current version 1.0. However, the aim of our study is to evaluate performance of Protobuf using these data types.

The study provided in this research is focused on comparing the Netstrings implementation with the further implemented LCM, Protobuf, and ROS serialization approaches. Since the main purpose of the study is to evaluate data marshaling approaches with data types that are specific for the OpenDaVINCI environment, it has a different implementation and message structure. All the

details regarding the implementation are described in the *Implementation* section. Furthermore, these approaches were evaluated from the descriptive expressiveness perspective, which is described in the respective section.

3 Methodology

In order to conduct our study, we have chosen to use, as the research methodology, the *design science*. Design science describes the designing and investigation of the artifacts in context [12]. It includes the necessary steps for identifying the problem in the current design of the environment and suggests a technique for implementing the best possible solution. The object of study in a design research is an artifact in a context, and the main activities in this research methodology is to analyze this artifact from the design perspective and investigating it with the same context [12]. In our case, we were implementing and evaluating by comparing four different data marshaling approaches, which serve as artifacts. The *design science* methodology is conducted inside a single environment, which serves as the context. We found evidence in one study[2] that only LCM and ROS were implemented and compared to each other in the same environment. However, we did not find any evidence regarding a comparison between all four aforementioned approaches in the same real-time environment.

In this research methodology the *designing cycle* is described as a cycle which is divided in three major steps:

- Problem investigation
- Treatment design
- Treatment validation

The *problem investigation* is described to be looking upon the phenomena that needs to be improved. It is also looking for the reason of this improvement action. *Treatment design* step is responsible for bringing in the possible solutions (artifacts) that would treat the problem. The *treatment validation* is investigating each artifact in order to choose the one that would treat the problem. These three steps are a part of a greater cycle which, with the help of these three steps, integrates the solution in the real environment. This entire cycle is called *engineering cycle* and it consists of five major steps:

- Problem investigation
- Treatment design
- Treatment validation
- Treatment implementation
- Implementation evaluation

The *treatment implementation* phase is treating the problem with the chosen artifact and the *implementation evaluation* phase is asking the same question as in *problem investigation*, but it already investigates the newly implemented artifact [12]. If the problem was not solved in the *implementation evaluation* phase, it starts all over again.

The design science methodology implies that the engineering cycle will continue looping until the initial problem, deducted from the *problem investigation* phase, will be solved. In our case, we have followed all the steps of the engineering cycle but we have not iterated the cycle more than one time. The reason behind it was the fact that we were limited in time and we could not perform more iterations of this cycle in order to find the perfect solution, which is the best substitution for Netstrings, in terms of efficiency and descriptive expressiveness.

We followed each phase according to the research methodology described above. In the *problem investigation* phase we have looked into the problem of improving the efficiency of the communication inside our environment. As it was stated in the *Introduction* section, our aim was to improve the efficiency of the data marshaling approach of OpenDaVINCI. The reasons behind it was to improve the communication between the running components of the system. The *treatment design* implies exploring different solutions towards solving the problem, therefore we started looking into substituting it with other data marshaling approaches, that would improve communication across components of OpenDaVINCI and to make it also more open to other environments. In the *treatment validation* step we have picked specifically LCM, Protobuf, and ROS marshaling approaches for their broad use, performance skill, and compatibility with other system such as ROS itself. Also, as it was stated in the "introduction", LCM and ROS are used in the systems alike OpenDaVINCI. *Descriptive expressiveness* perspective was also taken into consideration in this step. *Treatment implementation* step, consists of two parts, which are:

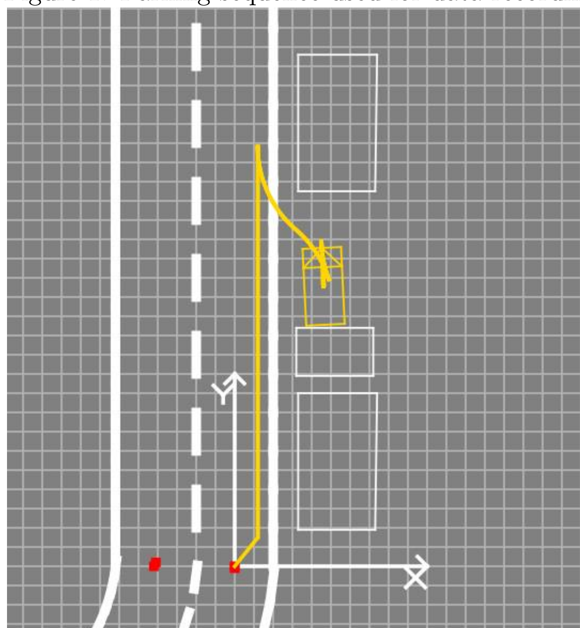
- Implementing each of the approach.
- Testing the implemented approaches.

Finally the last step, *implementation evaluation* consisted of a discussion based on findings, and concluding the results. It analyzed the properties of the implemented marshaling approaches (LCM, Protobuf, and ROS) and compare those with the properties of Netstrings, and generates a conclusion whether the implemented solutions perform better or worse compared to the original implementation(Netstrings) according to the criteria described in the *data analysis* section.

The *design science* research methodology is very close to the *case study*, but in a case study, we would have as a task to investigate in depth the current implementation or the reason it was decided to substitute one marshaling approach over another one [1]. In our case that is not suitable since we had the task also to investigate the possible solutions, implement those, test and analyze the results. An experiment would imply that we would have to apply some

treatments for the factors and afterwards investigate the behavior that is caused after it [1]. The downside of using an experiment, in our case, is the the fact that it requires more time than we had available. A survey would be suitable for a collection of quantitative data from multiple practitioners in order to find out what do they think upon a change that was made regarding substituting a data marshaling approach [1]. However, we were interested in qualitative data with a focus on performance rather than opinion of other users regarding each approach.

Figure 1: Parking sequence used for data recording



4 Environment

"OpenDaVINCI is a lean, portable, collaborative, and extensible C++-middleware that enables the development and agent-based, unattended testing of distributed cyber-physical system infrastructures. Its technical concepts were elaborated during the academic development of two autonomously driving vehicles: "Caroline" from Technische Universität Braunschweig, Germany and the experimental vehicle from "CHESS" (Center for Hybrid and Embedded Software Systems) at the University of California, Berkeley" [8].

The environment we used for the assessment is called *Open Source Development Architecture for Virtualization of Networked Cyber-Physical System Infrastructures (OpenDaVINCI)*. *OpenDaVINCI* is a software development and a run-time environment [8] for vehicles. It simulates real world environment for

the autonomous vehicles, by simulating real world scenarios of vehicle movement on the lane of the road and in parking areas. In our case, we picked a scenario where car was parked parallel to the road between two objects. In Figure 1 it is displayed the starting and the ending point of the parking sequence together with the path that the car is traveling from starting point till the end of the sequence. The scenario was performed ten times during data collection.

In Table 1 it is described hardware and software specifications of the system where data collection was performed.

Table 1: System information

Operating System	Ubuntu 15.04 64bit
CPU	Intel(R) Core(TM) i5-4300U CPU @ 1.90GHz
RAM	8GB DDR3L SDRAM 1600 MHz
Kernel version	3.19.0-18-generic

5 Implementation

In order to assess LCM, Protobuf, and ROS in the same conditions, we had to implement them in OpenDaVINCI environment and compare their performances along with the already implemented one (Netstrings). In this section we described the flow of the data passing through OpenDaVINCI along with: the description of the additionally implemented serializers, how they process the data, and how serializers were implemented.

In order to be able to compare each approach, the data flow had to be kept as close as possible to the Netstrings serialization sequence. Therefore, we have performed an analysis of the OpenDaVINCI environment and identified the main classes used in the data marshaling process. These classes will be discussed in the *Data Flow* subsection.

5.1 Message Structure

LCM, Protobuf, and ROS have unique way to structure their serialized payload. During implementation phase our goal was to implement each approach while keeping their original structure of serialized message. Thus, allowing each respective approach to communicate with third-party software using their approach to serialize message. In the sections bellow, we outlined and explained the structure of message serialized by Netstrings, LCM, Protobuf, and ROS.

5.1.1 Netstrings

The structure of Nestrings message is illustrated in Table 2.

The *Magic Nr.* is a number used to confirm that the data received has been serialized using Netstrings serializer, and that it is possible to decode data using Netstrings message structure.

Table 2: Structure of Netstrings message

Magic Nr.	Length	Payload	Comma
16 bit integer	32 bit integer		8 bit value

The *Length* provides information regarding the length of the serialized message. It is used during the deserialization process for identifying the end of the payload, which is followed by a comma.

The *Payload* is the serialized data. The data fields of a serializable class will be serialized and written in the same order in which the *write functions* were being called. Every variable that gets serialized also gets an ID provided to it. The ID is used later to identify the location of the variable in the process of reading the entire message. It is written to a buffer which holds the entire serialized message. When the data is serialized using Netstrings, it first encodes the provided ID in network byte order (Big-endian). *Network byte order* implies that the bytes of the encoded data are reversed while being encoded. It is the most common convention used in data networking, therefore it is safer to send the data in network byte order to avoid performance issues. The size of the data is converted to network byte order and written to the buffer as a 32 bit size value. Further the data, which is usually a variable, is also encoded in network byte order and written into the buffer. The payload will have a structure as illustrated in Table 3.

The *Comma* indicates the end of the message and it is used to check for the message data corruptness. When the full message is read, but no *Comma* was found at the end of the stream, it would mean that the data got corrupted, and some data may have been lost.

Table 3: Payload structure of Netstrings messages implemented in OpenDaVINCI

ID	Size	Variable	ID	Size	Variable	ID	...
32 bit integer	32 bit integer		32 bit integer	32 bit integer		32bit integer	... ...

ID is being used in order to pair values with their representative ID, and making it easier to find during the deserialization process. *Size* provides information for deserializer regarding the number of the bytes it took to write the value of *variable.*

5.1.2 LCM

The structure of messages from the LCM are illustrated in Table 4.

LCM serialization/deserialization uses network byte order(Big-endian), and uses UDP for the communication. The *Magic Nr.* has the same purpose as in Netstrings; it is used for checking if the received message has been serialized into an LCM structure. This *Magic Nr.* is the same number as the number that

Table 4: Structure of LCM messages

Magic Nr.	Sequence Nr.	Channel Name	Null Terminator	Hash	Payload
32 bit integer	32 bit integer	String	8 bit integer	64 bit integer	

is used in the third-party LCM system. This is done in order for other systems to be able to identify which serializer has been used to serialize the message.

The *sequence number* is a number that, in the LCM system, is increased for each message sent to know if a message has been dropped.

The *channel name* is the channel in the UDP stream where messages related to the channel name are being sent e.g. data related to the vehicle control, would be sent to channel "VehicleControl".

A *null terminator* is placed after the channel name in order to know when the channel name ends, because maximum size of the channel can be 256 characters.

Hash is a number that is generated based on the names of the member variables and the types of these variables. It is used in order to ensure that both sides of system agreed on same data types in the message.

The *payload* is the serialized data. The data fields of a serializable class will be serialized and written in the same order that functions are being called. The payload is encoded in network byte order and written to a buffer. Unlike the implementation of Netstrings, there is no ID marking of the locations of the variables in the message and this means the data must be read in the same order as it was written.

5.1.3 Protobuf

Table 5: Structure of Protobuf message

Magic Nr.	Payload
16 bit integer	

Full message structure of Protobuf is displayed in Table 5. Protobuf provides specifications for the structure of the payload as displayed in Table 6, and the structure of the message is up to the user to decide, therefore it can be adapted based on systems requirements and specifications. In our case, message header included just *Magic Nr.* Protobuf encodes data using little-endian byte order.

Table 6: Payload structure of Protobuf message

Size of payload	Key	Value	Key	Value	...
32 bit integer	32 bit integer		32 bit integer		... ...

Protobuf does not have any indicator for the end of the payload, thus in order to make sure it reads all bytes, it keeps *size of the payload* in front. Therefore, when a message is being deserialized, Protobuf knows how many bytes it has to read in total.

Protobuf uses WIRE TYPE and PROTO TYPE data types to classify variables into categories. WIRE TYPE contains: varint, bit 64, length delimited, bit 32, and other. Serialization type depends on variables WIRE TYPE and PROTO TYPE. PROTO TYPE contains information about variables types. Key is formed from those types (WIRE TYPE and PROTO TYPE). *Key* stores information about variable size and type, and field ID which is defined in the message's structure. Each *Key* is paired with a value which stored in least amount of bytes it requires to store it.

5.1.4 ROS

ROS provides two different message structures based on the type of the message and how the message is being transferred (UDP and TCP). ROS UDP specifications met our system specification best, therefore the focus was just set on the message structure, which is sent on UDP stream. ROS's UDP and TCP differs in the header only, with the payload being the same for both. The following structure can be seen in the Table 7 bellow.

Table 7: Structure of ROS UDP message

Connection ID	Opcode	Message ID	Block Nr.	Payload
32 bit integer	8 bit integer	8 bit integer	16 bit integer	

ROS messages are serialized in the little-endian format, which means that least significant byte is written in the smallest address. *Connection ID* represents the connection name to which messages are sent. This ID has to be agreed when the connection is established. The *opcode* defines the message types. While ROS system supports different message types, in our case we were sending one type of messages.

Message ID is an 8-bit value which is used to identify if there is a message loss. It is done by reading each message ID and checking if values are in correct order.

ROS divides one big message into few smaller messages and sent as fragment. *Block Nr.* is used to store the number of messages after they were fragmented. OpenDaVINCI uses non-fragmented messages.

Table 8: Payload structure of ROS message

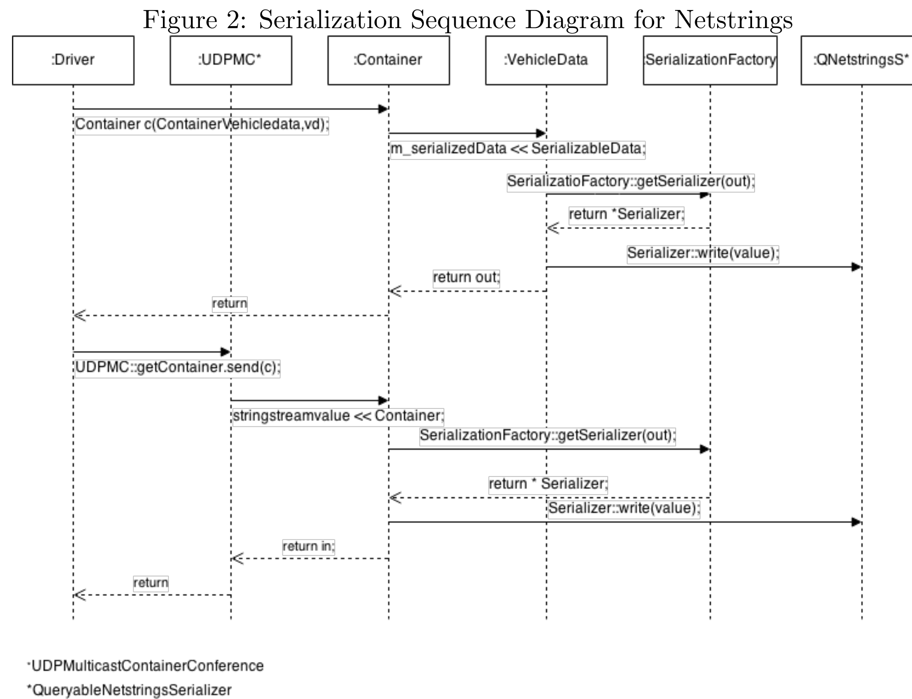
Size of payload	Value	Value	...
32 bit integer			...

Payload structure is displayed in Table 8 and is very simple, it first writes size of the payload, and the values are written after.

5.2 Data Flow

Before starting the implementation of the serialization part of each approach, we studied the data flow for the Netstrings serialization in order to find out how it was implemented and how we should implement the other three approaches. Since Netstrings is the default serialization/deserialization approach, we have implemented the other three approaches to have the same procedure when serializing and deserializing.

Figure 2: Serialization Sequence Diagram for Netstrings

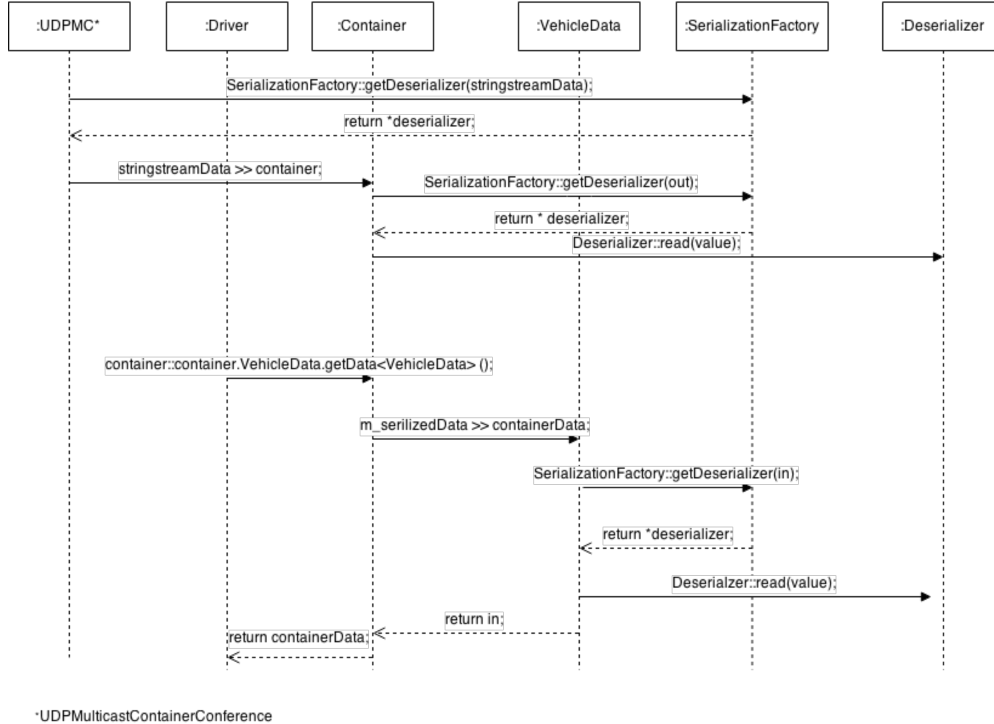


5.2.1 Netstrings

The Figure 2 is the sequence diagram that illustrates the data flow and the way it is being serialized and sent to the UDP multi-cast container conference, which further sends the message to the UDP stream.

The Figure 3 is a sequence diagram that illustrates the data flow from when it has been received from the UDP stream and the way it is being deserialized and read.

Figure 3: Deserialization Sequence Diagram for Netstrings



In the Figure 2 the flow starts when all the data fields of a message is set by a class and a container is created which will contain the message. We drew the diagrams on the example of the driver class. This class is used to control a vehicle automatically and it sets the data fields for the *Vehicle Data* class, which has fields for different information about the vehicle e.g. speed, steering wheel angle, and acceleration.

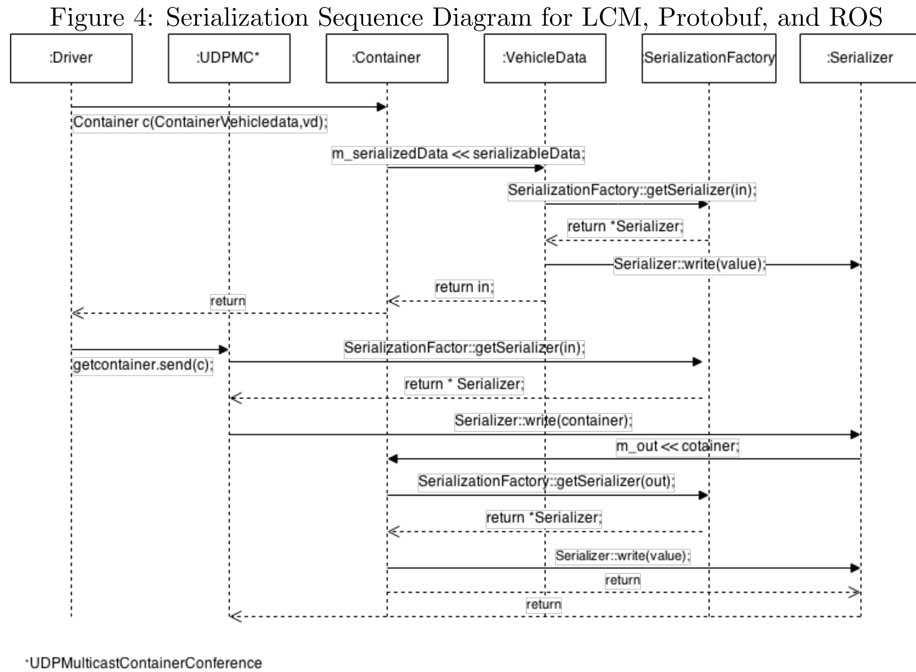
As the container object gets created with a *data type* which is a value describing what data this container will contain, and an instance of vehicle data. All values inside of the instance gets serialized by calling predefined serializer functions in the Vehicle data class. In order to serialize the data, Vehicle Data asks a Serialization Factory for a serializer that will be used in order to serialize data. A Serialization Factory is a class where the serializers and deserializers are created, and it is created every time a serializer or deserializer is needed.

Once the container is created with serialized data, the data is sent to the UDP multi-cast container conference. Additional information is added, including data type of container, serialized data to send and a time stamp, marking the time when the container was sent. The container with all this additional information is serialized again, and a header is added to the front of message,

and sent to UDP stream.

In Figure 3 the flow of the deserialization part starts in the UDP multi-cast container conference. Whenever data is received from the UDP stream, the container is deserialized and stored. Once a class, in this case the Driver class, wants to access the data, the stored container is acquired by providing the data type value for Vehicle Data when creating a container. The data in the container is deserialized, read, and returned to the Driver class.

Figure 4: Serialization Sequence Diagram for LCM, Protobuf, and ROS

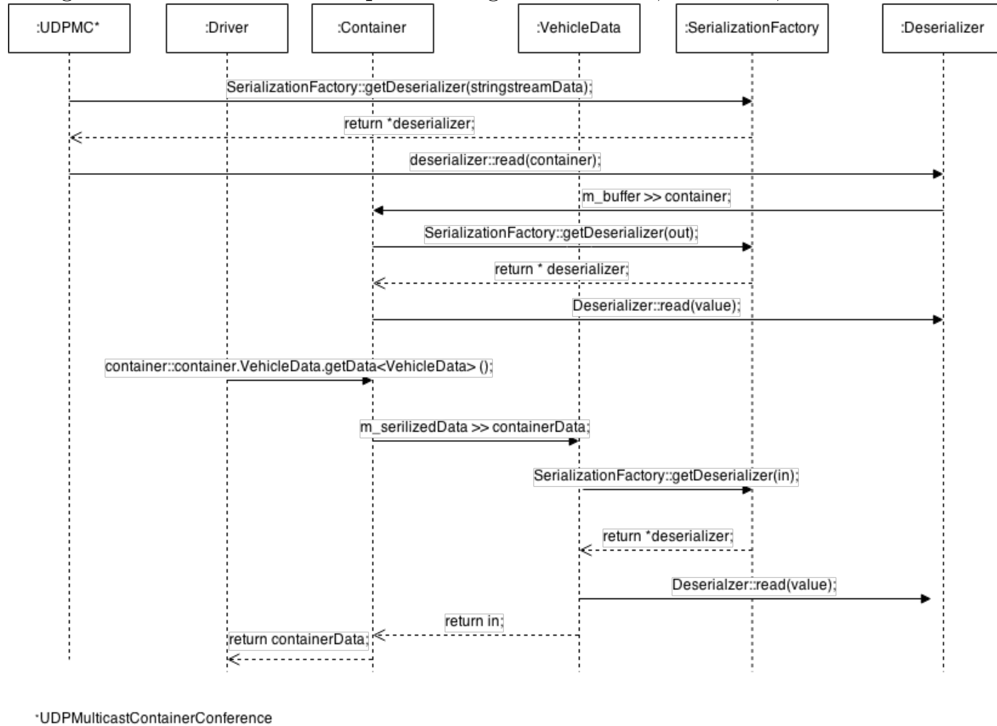


5.2.2 LCM, Protobuf, and ROS

In the case of LCM, Protobuf, and ROS, serializers and deserializers have the same data flow, which is almost the same in the Netstrings serialization. Figure 4 illustrates the new flow on the example of LCM serializer and Figure 5 displays example of LCM deserializer.

The difference is the addition of a *write(container)* and a *read(container)* function to the serializers and deserializers respectively, which are called from the UDP multi-cast container conference. These functions writes/reads the header to the container before sending it to the UDP stream. They are required because OpenDaVINCI is designed to operate in such way. There are at least two serializers that are being created during the serialization process and every time a serializer is created, a header is added to the message. If the messages

Figure 5: Deserialization Sequence Diagram for LCM, Protobuf, and ROS



are to be sent to systems outside OpenDaVINCI, they can't contain more than one header. That is why a separate function for adding the header is required instead of having the serializer to add it.

6 Data Collection

After the implementation phase, we collected the data by recording message serialization/deserialization time values and size values of the same message. We chose Vehicle Control, Vehicle Data, and Sensor Board Data containers to collect our data. Each container had different structure. Vehicle Control contains three *double* type and three *boolean* type variables. Vehicle Data contains two *Point3* type and seven *double* type variables. Sensor Board Data contains one *string*.

Values of each container performance were used to assess each serialization approach. The assessment included the following aspects:

- size of the serialized message;
- serialization/deserialization speed of the messages;

In order to collect all the required data for the following assessment, we went through each one of these aspects in part.

The *size of the serialized message* was collected after serialization by calculating the size of the header separately and then adding the size of the payload, which is the actual container that was serialized.

The *serialization/deserialization speed of the messages* was collected by adding time stamps before and after the message was serialized/deserialized.

The size of the serialized message and the time it took to serialize it has been collected from two classes.

In the *UDPMulticastContainerConference* we have extracted the size of the serialized container while running the system with having each of the approaches implemented each at a time. Also, in the same file we have set time stamps before and after the serialization of the container. We used the time stamps for measuring the time it took for serializing the container. Deserialization time of the header was measured in the same file.

The payload size was measured in the *Container.cpp*. In the same file we measured the time of payload serialization using time stamps. The deserialization time of the payload was measured in the *Container.h*. The deserializing function *m_serializedData* is called from the header file by default.

We ran each approach with the same exact data in order to evaluate them objectively. We have collected all the data while running a parallel parking algorithm in the simulation environment provided by OpenDaVINCI. The parking sequence was performed ten times for each collection. We were collecting over 25 000 values each time the sequence was initiated. The collected data was stored in output files for the further analysis.

7 Data Analysis

Before conducting an analysis the raw data was averaged out in order to be able to measure the collected results and to illustrate the performance differences between the approaches with bar charts.

The analysis itself was divided into two phases. The first phase included a comparison between the approaches in terms of performance. The performance comparison took part across all four data marshaling approaches and consisted of the following parts:

- time wise comparison to serialize the message;
- time wise comparison to deserialize the message;
- size comparison of the serialized message;

We compared message size of all implemented approaches. This included a separate comparison of the serialized payload size and the serialized container size. The further step included the comparison of the time it took for the message to be serialized using a given approach. This aspect also included the timings of the serialized payload separately. Lastly, the performance comparison included the difference of time required by each approach to deserialize the entire message and the payload separately.

The second phase included an analysis of the results in compliance with the descriptive expressiveness. The outcome of this phase was to generate the final conclusion regarding the effectiveness of each approach and to identify the advantages and disadvantages of the newly implemented approaches over the Netstrings.

8 Findings

8.1 Descriptive Expressiveness

Marshaling approaches compared in this study were evaluated in terms performance and descriptive expressiveness. Descriptive expressiveness is responsible for indicating the level of the types of data a language supports and is able to express. In this section we describe the similarities, differences, and specifics of each of the three approaches compared with the ODVD language.

ODVD is the language that describes all the different types of data used to communicate within the OpenDaVINCI environment. Besides primitive data types, ODVD and the descriptive languages of the approaches that were implemented, use utilities that generate functions for marshaling and unmarshaling custom data types. In ODVD, these custom data types are: messages, lists, maps, fixed arrays and enumerations. Messages are compound data types that consist of other data types. They are used for the exchange of the information in the data streams. All the included approaches support communication through message exchange.

In ROS, for instance, a message is a sequence of different values that are arranged in an order, which is defined by a type descriptor (Topic and Service). We will mainly refer to topic descriptor in this paper since it is uni-directional, thus it requires only the definition of a message type. The service descriptor is used for a request/response method[13].

Table 9: Compound data types

Compound data types	ODVD	LCM	Proto	ROS
Message	x	x	x	x
List	x			x
Map	x		x	x
Fixed Array	x	x		x
Enumerations	x		x	x

The topic message descriptor in ROS is written in a file with the .msg extension and its name is the name of the type described inside the file. It supports the nested data structure as well: in each line both primitive types and another message type descriptor. It can also support another compound types such as arrays [13]. In LCM the principle is the same, assuming that is has structs that contain other different types. It can also contain different structs. Thus, we can also consider that it supports nested data [4]. In Protobuf, there is the same principle as well. It has the .proto files that contain structs which can contain not only scalar types but also composite types such as enumerations or other structs. In Protobuf it is possible to define more messages types in one single .proto file.

Table 10: Primitive data types supported by ODVD

Boolean	Char	Int32	Uint32	Double	Float	String

Regarding other compound and custom data types, among the approaches discussed in this study, *list* types are supported only by ROS currently. *Map* types are supported by Protobuf and ROS. *Fixed array* type is supported only by LCM and ROS. *Arrays* are not supported by Protobuf directly[3].

For the unsupported types, LCM provides users with simple declarations of constants that serves to substitute other data fields. *Enumerations*, *magic numbers* or bit fields, for instance, can be expressed through constants [4].

An overview of the support of the complex data types are listed in the Table 9.

Table 11: Primitive data types

Primitive data types	ODVD	LCM	Proto	ROS
bool	x	x	x	uint8
char	x	int8	byte	x
int32	x	x	x	x
uint32	x		x	x
float	x	x	x	x
double	x	x	x	x
string	x	x	x	x

In terms of primitive data types, the ODVD supports the primitive data types listed in Table 10.

All of the approaches support all of these types listed above, except for the *char* that is not supported directly by neither LCM nor Protobuf. Instead, LCM uses *int8* to express a char type. For any data that is expressed as byte sequence Protobuf uses *bytes* to encode it. For enumerations Protobuf uses integers encoding method. All values has to be in a rage of 32 bit integer otherwise it is not efficient and not recommended [3]. LCM also does not support unsigned integers in order to avoid problems when it is being used with programming languages that do not support unsigned integers, like JAVA.

The *boolean* type is expressed through an *uint8* type in ROS.

The overview of the compatibility of the primitive types of Protobuf, LCM and ROS with the descriptive language ODVD is displayed in the Table 11.

8.2 Performance

In this section we wrote the performance results in terms of serializing/deserializing times and also in terms of size of the messages being serialized. In the Figure 6 is displayed the serialization time of the entire container. The structure of the container is illustrated in the Table 12.

Table 12: Structure of the container

Data type	Data	Sent time	Received time
Uint32	String	Time stamp	Time stamp

The container is being used for wrapping the data that is going to be sent, this data is called *serializable data* in OpenDaVINCI environment. The structure of the container is never changed since the *serializable data* is contained as a string in this container. Therefore, in the Figure 6 and Figure 7 we can observe that each serializer has performed roughly the same when it came to time to serialize and deserialize these containers. The difference is more noticeable when different approaches are serializing/deserializing the same container.

Figure 6: Serialization time for the entire container (in microseconds)

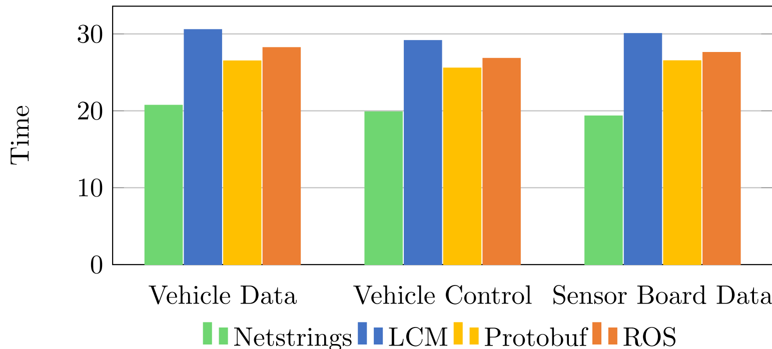
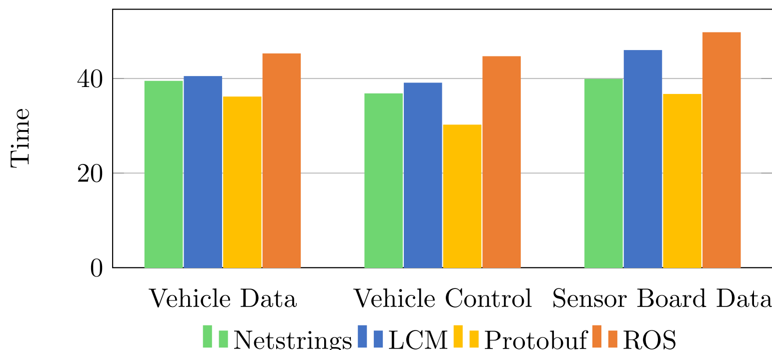


Figure 7: Deserialization time for the entire container (in microseconds)



From the Figure 6 we observe how different the approaches are handling this container while serializing it.

LCM was the slowest one because it calculates the hash while serializing the container. Netstrings serializes the container the fastest because it is not strict regarding the payload structure. Therefore it was adjusted to the OpenDaVINCI design. Protobuf and ROS perform very close to each other. ROS was slightly slower since its header is by six bytes bigger than the two-byte header of Protobuf.

The Figure 7 displays the time in which data marshaling approaches read the containers. In this case, Protobuf has performed the fastest since the container was smaller, because Protobuf is writing only the bytes that contains information, the rest of the bytes are dropped. An important factor is also the fact that it has the smallest header (2 bytes). LCM was performing closely to Netstrings, but slightly slower since it also spends time to deserialize the hash. ROS proved to be the slowest.

In the Table 13 we showed the same difference as it is in the Figure 6, and

7 but now we measured the time differences in percentages of serialization and deserialization procedure. Thus, we can state that when it comes to serialization, Netstrings outperformed all other approaches, where in the deserialization process, Protobuf takes the lead.

Table 13: Container serialization/deserialization time comparison to Netstrings

Serialization	Netstrings	LCM	Protobuf	ROS
Vehicle Data	100%	+47.5%	+27.9%	+36.2%
Vehicle Control	100%	+46.7%	+28.6%	+35%
Sensor Board Data	100%	+55.5%	+37.2%	+42.8%
Deserialization				
Vehicle Data	100%	+2.5%	-8.4%	+14.7%
Vehicle Control	100%	+6.2%	-18%	+21.4%
Sensor Board Data	100%	+15.3%	-8%	+24.8%

In order to send the container with data, we needed to have this data serialized first. The Figure 10 displays how each approach performed when it came serializing three different container types, which are: vehicle data, vehicle control, and sensor board data.

First, we will display the content of the data that is being serialized in the Figures 8, 9. *Sensor board data* contains only a single string.

Figure 8: Vehicle Data (VD) structure

point3	double	double	double	point3	double	double	double	double

Point3 data type is specific for OpenCV [7]. It contains three *double* data types inside.

Figure 9: Vehicle Control (VC) structure

double	double	double	boolean	boolean	boolean

Protobuf proved to have the best performance in serializing *vehicle data* and *vehicle control*, and third in serializing *sensor board data*. This is due to the fact that Protobuf reduces message size while writing as little byte amount as needed to express the value. Since sensor board data is a long string, it could not reduce the amount of bytes needed to write. Since Netstrings writes all bytes of each variable while serializing it, came into second place for *vehicle data*, *vehicle control* and first when it came to the *sensor board data*. LCM performed slower than Netstrings because it was affected by hash generation. ROS proved to be the slowest in this case.

Figure 11 displays the time it takes to read different serializable data with different approaches. When it came to reading Vehicle Data, LCM performed

Figure 10: Serialization time for the serializable data (in microseconds)

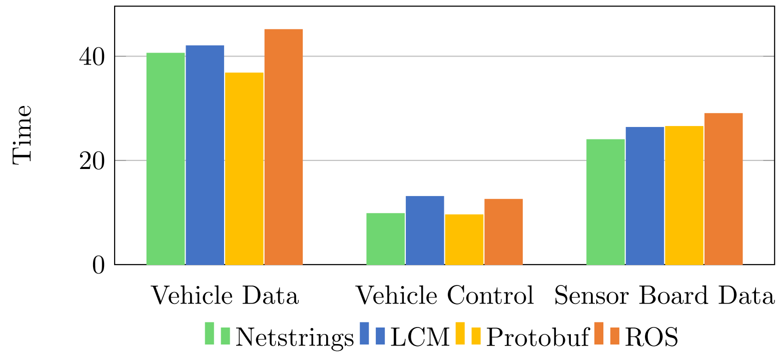
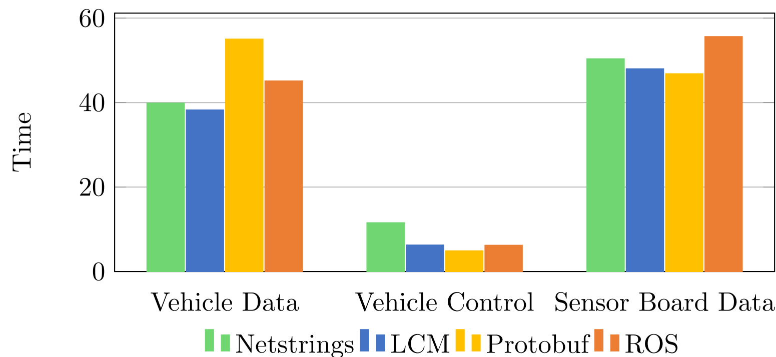


Figure 11: Deserialization time for the serializable data (in microseconds)



the fastest. This is because the hash value is ignored by the deserializer, since it is not used inside OpenDaVINCI, making LCM faster compared to the other three when reading many different variables. Protobuf is being the slowest, even though the size is the same as for LCM and ROS. This means Protobuf is slow at reading strings, but in the Vehicle Control, it is faster, even though the sizes are the same. ROS has a pretty straight forward way of reading strings and it performs average in terms of reading time. Netstrings is a little bit slower than LCM, because it assign the IDs in the structure. This fact slows Netstrings down when it comes to reading smaller strings.

In Table 14 we have showed an overall difference between Netstrings and other implemented approaches when it comes to time required for serializing and deserializing the serializable data, such as *vehicle data*, *vehicle control* and *sensor board data*.

Table 14: Serializable data serialization/deserialization time compared to Netstrings

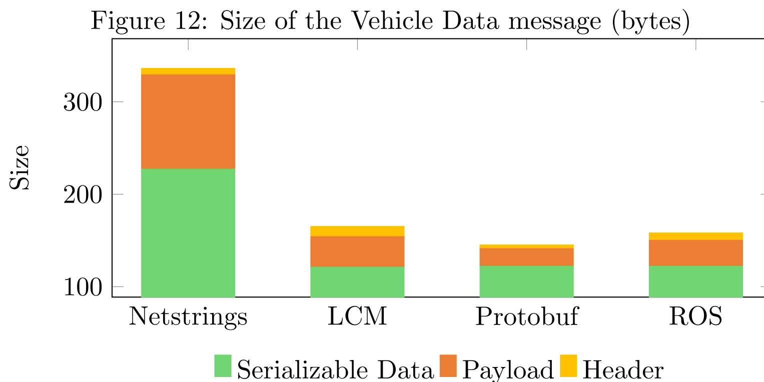
Serialization	Netstrings	LCM	Proto	ROS
Vehicle Data	100%	+3.5%	-9.4%	+11.2%
Vehicle Control	100%	+33.5%	-2.5%	+27.9%
Sensor Board Data	100%	+9.8%	+10.6%	+20.9%
Deserialization				
Vehicle Data	100%	-4.1%	+38%	+13.2%
Vehicle Control	100%	-45.8%	-57.6%	-46.3%
Sensor Board Data	100%	-4.7%	-7%	+10.4%

8.3 Serialized Message Size

In Figures 12, 13, and 14 we have illustrated the size of the entire message being serialized containing the *serialized data*, *payload* of the message, and the *header*. We measure the serialized message in three cases:

- containing *vehicle data* as the serializable data;
- containing *vehicle control* as the serializable data;
- containing *sensor board data* as the serializable data.

Figure 12: Size of the Vehicle Data message (bytes)



One thing to notice is that in all cases we addressed to *vehicle data*, vehicle control, and *sensor board data* as the *serializable data*, but in this case it is already serialized and included in the payload as a string of bytes. In these figures we illustrated how the newly implemented approaches (LCM, Protobuf, and ROS) serialized the messages in terms of size, compared to Netstrings. Netstrings as a result proved to have largest serialized messages. This happens due to the Netstring's method of serializing the message. This structure is

illustrated in the Table 3. Otherwise Protobuf is clearly the winner in this competition, regardless of the serialized data. It is due to the fact that Protobuf writes smaller numbers in less bytes. LCM and ROS are performing very closely to each other, since they write all bytes based on variables data size.

Figure 13: Size of the Vehicle Control message (bytes)

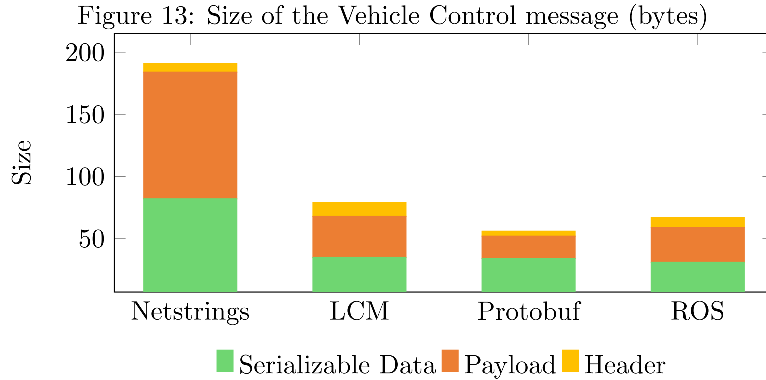
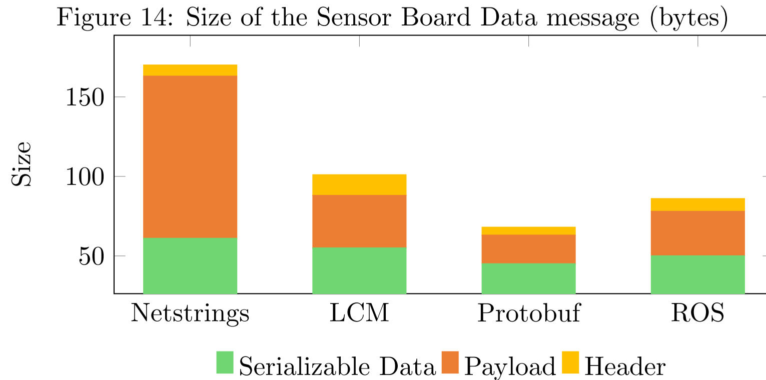


Figure 14: Size of the Sensor Board Data message (bytes)



In the Table 15 it is shown the size differences of the serialized messages LCM, Protobuf, and ROS compared with the serialized message of Netstrings.

Table 15: Serialized message size compared to Netstrings

Serialization	Netstrings	LCM	Protobuf	ROS
Vehicle Data	100%	-50.9%	-56.8%	-53%
Vehicle Control	100%	-58.6%	-70.7%	-64.9%
Sensor Board Data	100%	-40.6%	-60 %	-49.4%

9 Discussion and Conclusion

After mentioning the differences regarding the performances and descriptive expressiveness, we used this section to outline the findings regarding performance and to make a parallel of those to the descriptive expressiveness propriety.

Performance wise, we have seen the fact that Netstrings, the initial serialization approach used in OpenDaVINCI, overall outperformed every other implemented approach. It is due to the fact that it is flexible in terms of payload structure and therefore it was shaped to fit the design of the environment. One problem of Netstrings is the fact the serialized messages become much bigger compared to the rest of the approaches. In this field Protobuf's ability to generate serialized message in less bytes helped to generate the smallest serialized messages. LCM and ROS performed very close to each other in terms of performance, both for serialization time and message size wise. LCM is recommended when it is important to make sure, that both sides of communications agreed of message structure. Protobuf excels in environment where size of message is limited and size has to be as small as possible. Moreover, Protobuf showed good performance when messages had numeric values. ROS should be picked when main system requirement is compatibility with its ecosystem, as it displayed average performance compared to LCM and Protobuf.

Regarding descriptive expressiveness, each of the newly implemented approach is different compared to the ODVD descriptive language. Not so much when it comes to the primitive data types, with some small exceptions. The difference was noticed when it came to the compound data types. LCM proved to be the least descriptive approach compared to ODVD. ROS, on the other hand, has proved to be the most expressive. It supports all the compound data types that ODVD provides. The Protobuf supports more than LCM, but less than ROS, making it possible to be placed in the middle.

Overall we did not find any ideal substitution for Netstrings. Everything depends on the intention of the practitioners. Protobuf's strongest side is the small size of the message. This feature is very useful when it comes to connection with a limited bandwidth. LCM is designed for a secure connection between components. And finally, ROS has an enormous ecosystem, and elaborating a connection with it opens a larger window for future development.

During our research, main limitation we faced were time frame. Because of the time restriction for this research, we had time just to evaluate all four approaches within simulation environment. By expanding time frame it would be possible to get more accurate data by measuring performance on real-time system, in our case autonomous self-driving car.

9.1 Validity Threats

A validity threat is the threat that can affect the validity of our collected data. In our research we have identified few of such threats that could influence our findings.

One of the threats, we assume, is the fact that we have kept the same

structure of the payload for all the approaches. The results could have changed in the case when we would personalize the payload to the marshaling approach. For example, in the case of LCM, the *Data Type* of the *Container* could have been removed from the payload since we had it already as the *Channel Name* in the header. In the case of ROS, the *Data Type* of the *Container* could have been moved from the payload and written in the header as the *Connection ID*. That would decrease both the size and the serialization/deserialization time of the payload for ROS and LCM approaches.

The second threat to our validity is the fact that we were not able to verify 100% the structure of the serialized messages with third party systems ROS and Protobuf. The reason of that is because Protobuf provides just tools and structure to generate payload, leaving communication for the user. Having time constraints, we did not have enough time to create communication system outside OpenDaVINCI. Same reason is for not having ROS implementation tested with the third-party system. However, we confirmed the structure of simple message based on raw data found in literature [11] [13].

10 Future Work

Our research showed that in most cases when it came to serializing speed Netstrings outperformed LCM, Protobuf, and ROS. Nevertheless, there are still many different marshaling approaches that can be tested and compared with. However, optimizing each approach for OpenDaVINCI, should show improvements in the performance. The next steps are to design and implement a robust way to switch between different marshaling approaches. That would provide more flexibility and freedom for the users of OpenDaVINCI. Furthermore, this research was focused on evaluating performance of each given approach, so in order to find the reasons of performance, another research with a focus on finding out why each approach performs this way. This information would allow to improve and find best ways to marshal messages.

11 Summary

In this research paper we have discussed about different marshaling approaches that were implemented and evaluated in the OpenDaVINCI environment. In the beginning of the study we have introduced the definition of data marshaling and its importance for the real-time systems. We have talked about the marshaling approaches that were chosen to be implemented in the environment along with reason behind this choice. We have then listed two research questions which are describing the goals of our research. We described the research methodology which guided our study and described the environment we have been tested the newly implemented marshaling approaches. In this paper we have also discussed the method of collecting the necessary data that was required for the further intended analysis. The implementation procedure was described that included

some insights regarding the serialization design of OpenDaVINCI environment. Later we have included the finding in terms of how all approaches performed and illustrated the results in charts and tables. That included both the time of serializing/deserializing and the size of the serialized messages the approaches produce. Also we have showed the differences between them regarding the descriptive expressiveness. There we analyzed both the primitive and compound data types that are supported in the newly implemented serializing approaches and compare these with the descriptive language ODVD. We performed the discussion where we talked about the fields in which LCM, Protobuf, and ROS would be more useful. Lastly we a suggested for the future contributors to keep improving OpenDaVINCI environment by trying other approaches and improve the flexibility of its design.

12 Acknowledgments

The authors of this paper would like to thank Dr. rer. nat. Christian Berger for collaboration and interest in our research. Furthermore, we would like to thank Hugo Sica de Andrade and Federico Giaimo for reading, reviewing, and providing feedback for this study.

References

[1] John W Creswell. *Research design: Qualitative, quantitative, and mixed methods approaches*. Sage publications, 2013.

[2] 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*. IEEE. 2010, pp. 4057–4062.

[3] *Language Guide (proto2)*. URL: https://developers.google.com/protocol-buffers/docs/proto (visited on 03/15/2015).

[4] *LCM: LCM Type Specification Language*. URL: http://lcm-proj.github.io/type_specification.html (visited on 04/17/2015).

[5] Bertrand Meyer. *Object-oriented software construction*. Vol. 2. Prentice hall New York, 1988.

[6] *Netstrings*. URL: http://cr.yp.to/proto/netstrings.txt (visited on 04/27/2015).

[7] *OpenCV — OpenCV*. URL: http://opencv.org/ (visited on 04/24/2015).

[8] *OpenDaVINCI - Open Source Development Architecture for Virtualization of Networked Cyber-Physical System Infrastructures*. URL: http://www.cse.chalmers.se/~bergerc/opendavinci/ (visited on 04/27/2015).

[9] David Lorge Parnas. "On the criteria to be used in decomposing systems into modules". In: *Communications of the ACM* 15.12 (1972), pp. 1053–1058.

[10] *ROS.org — Powering the world's robots*. URL: `http://www.ros.org/` (visited on 04/27/2015).

[11] Wolfgang Schwitzer and Vlad Popa. *Using Protocol Buffers for Resource-Constrained Distributed Embedded Systems*. Tech. rep. TUM-I1120. 2011.

[12] Roel J Wieringa. *Design science methodology for information systems and software engineering*. Springer, 2014.

[13] Andrea Zoppi. "A lightweight open source communication framework for native integration of resource constrained robotics devices with ROS". In: (2013).