Systematic Evaluation of Three Data Marshalling Approaches for Distributed Software Systems

Hugo Andrade   Federico Giaimo   Christian Berger   Ivica Crnkovic

Chalmers | University of Gothenburg, Sweden

{sica, giaimo, crnkovic}@chalmers.se, christian.berger@gu.se

Abstract

Cyber-physical systems like robots and self-driving vehicles comprise complex software systems. Their software is typically realized as distributed agents that are responsible for dedicated tasks like sensor data handling, sensor data fusion, or action planning. The modular design allows a flexible deployment as well as algorithm encapsulation to exchange software modules where needed. Such distributed software exchanges data using a data marshalling layer to serialize and deserialize data structures between a sending and receiving entity. In this article, we are systematically evaluating Google Protobuf, LCM, and our self-adaptive delta marshalling approach by using a generic description language, of which instances can be composed at runtime. Our results show that Google Protobuf performs well for small messages composed mainly by integral field types; the self-adaptive data marshalling approach is efficient if four or more fields of type double are present, and LCM outperforms both when a mix of many integral and double fields is used.

Categories and Subject Descriptors D.2.12 [*Software Engineering*]: Interoperability—Data mapping; E.4 [*Coding and Information Theory*]: Data compaction and compression; I.2.9 [*Robotics*]: Autonomous vehicles, Sensors

General Terms distributed software systems, data marshalling, self-adaptive data marshalling

Keywords distributed software systems, data marshalling, self-adaptive data marshalling

1. Introduction

Distributed software systems are powering complex cyber-physical systems like self-driving vehicles [3]. Also in the area of Internet-of-the-Things (IoT), where remote entities collecting data are connecting back to a cloud based data processing infrastructure, a distributed software system is present. Realizing such a distributed software allows system designers to flexibly deploy software components or to exchange the behavior of one component while preserving its public interfaces. This flexibility is achieved by standardizing the data modeling and exchange among them. Popular examples therefor are Google's Protobuf [1] or Lightweight Communication and Marshalling (LCM) [6] that is also being used in experimental vehicles.

1.1 Problem Domain & Motivation

Both aforementioned approaches offer a message modeling language and code generation engine with bindings to different languages. Thereby, a distributed software system can even be realized on different platforms using different languages. However, once a message's design is specified during development time, it cannot be modified during runtime anymore. In our previous study Giaimo et al. [5] on the example of a cyber-physical system, we showed that the data that is exchanged in practice allows for saving bandwidth consumption as the difference between two consecutively sent messages is typically small in such application contexts. Thus, this delta can be encoded more effectively resulting in a faster data exchange as fewer bytes needed to be sent.

1.2 Research Goal & Research Questions

The goal for this work is to systematically evaluate the performance of the three different data marshalling approaches: Google Protobuf, LCM, and our self-adaptive data marshalling approach. The following questions were of particular interest:

RQ-1: *How can different data marshalling approaches be systematically evaluated?*

RQ-2: *What is the performance of the respective data marshalling under various, application-independent conditions?*

1.3 Contributions of the Article

We present a generic message description language (DSL) that serves as a super-set to common language features of Google Protobuf and LCM. This DSL is used to systematically create different message structures at runtime by our C++ middleware OpenDaVINCI[1], where the native implementation of the respective data marshalling approaches was evaluated.

[1] http://opendavinci.cse.chalmers.se

1.4 Structure of the Article

The rest of the article is structured as follows: In Sec. 2, we are outlining related work. In Sec. 3, the design of the DSL as well as implementation details are presented, followed by the evaluation in Sec. 4. The article concludes in Sec. 5.

2. Related Work

Huang et al. present in their work [6] Lightweight Communication and Marshalling, which is a spin-off of their 2007 DARPA Urban Challenge competition vehicle. They describe the language features alongside with a performance comparison towards the Robot Operating System (ROS) [7]. LCM encodes a message starting with a 4 bytes magic number that also encodes the version of the protocol. Next, a 4 byte sequence number is part of a message header to describe fragmented messages. The third field in the header is a null-terminated string describing the channel number under which a message is transmitted. The last field of the header is an 8 byte hash value, which is iteratively computed for all <field name, field type> tuples constituting a message. This hash value allows the validation at receiver side whether the deserialization could successfully read back all data as the fields of a message are not separated by delimiters or identifiers.

Schwitzer and Popa [9] present an implementation of Google Protobuf [1] for resource constrained devices. Their C-based implementation of a self-contained serialization approach enables the use of the protocol in the domain of Internet-of-the-Things (IoT). The fundamental idea behind Protobuf is to encode integral data types not on their type as defined during design time but on their concrete value at runtime; thus, even fields with an `uint32_t` field that would consume 4 bytes, could only occupy 1 byte if the value of that field is smaller than 128. This approach is called *variable-length quantity* (VLQ).

The general structure of a Protobuf message can start with a magic number followed by the length of the message; both attributes are encoded as VLQ. Next, the fields are encoded as tuples in sequence as they are specified. First, the field identifier in combination with the field's data type, which is consuming 3 bits, is stored in the first byte; the key/type combination is encoded using VLQ as well. Next, the value is written to the byte sequence. For any integral type, VLQ is used. Floating point types are encoded either as 4-byte floats or 8-byte double fields. Strings or raw byte fields are encoded including their respective data length.

For processes running on the same computation node, low-level inter-process communication (IPC) as defined in POSIX like message queues or shared memory can be used. As these means would allow a higher performance between communicating processes, truly distributed software entities running on different nodes would not be supported. Thus, IPC is not considered in this study.

```
message automotive.VehicleData [id = 39] {
    double heading;
    double absoluteTraveledPath;
    double relativeTraveledPath;
    double speed;
    double temp;
}

message automotive.VehicleControl [id = 41] {
    double speed;
    double acceleration;
    double steeringWheelAngle;
    bool brakeLights;
    bool flashingLightsLeft;
    bool flashingLightsRight;
}
```

Figure 1. Example of a message description file from OpenDaVINCI.

3. Generic Message Description and Self-Adaptive Marshalling

Our framework OpenDaVINCI is used on different cyber-physical experimentation platforms such as scaled self-driving vehicles [2] and for distributed simulations [4]. It allows the realization of distributed software systems by providing different communication patterns like publish/subscribe or centralized hub-based communication, as well as centralized scheduling for algorithms running on distributed nodes.

Messages to be exchanged among the interacting software agents can be modeled at design time by a data description language specified with Eclipse Xtext. An accompanying code generator to our C++ environment was realized with Xtend. The data description language exhibits the following language features:

- Scalar types like uint8, float, double;
- Definition of initialization values;
- Nested types;
- Enumeration types and constants;
- Lists, fixed size arrays, and maps;
- Specialization via message inheritance.

An example of a design time artifact is depicted in Fig. 1. The language itself has similarities with Google Protobuf and LCM but also provides further concepts, like inheritance to describe relations of an application domain.

At design time, concrete message classes are derived from the given specification file providing methods to access the data fields and to serialize and deserialize the message. In addition, every class also implements the interface *Visitable* allowing a *Visitor* to query the data fields of a message without knowing the concrete type of given object at runtime.

The concept of visiting any message in an abstract way was also used to realize a generic message representation by transforming a given data structure into a list representation of its attributes. In this case, an attribute comprises an identifier,

its type, and the current value. This generic representation of any message while preserving its properties like nested types allows the definition of model transformations at runtime by defining appropriate visitors.

This concept was used twofold: Firstly, it was used to realize the actual mapping of a given data structure from the OpenDaVINCI environment to Google Protobuf and LCM, respectively. Therefore, the visitor for the target language was instantiated at runtime to visit a data structure to serialize the containing data into the corresponding byte representation. For this purpose, Google Protobuf as well as LCM were natively implemented in OpenDaVINCI allowing a transparent data exchange between the three environments.

Furthermore, the generic visitor approach also allows the realization of an adaptive data serialization approach. As we have pointed out in [5], the differences between two consecutively sent messages in cyber-physical systems, which are interacting with their environment based on data perceived by sensors, is rather small. Moreover, sensors for control tasks are typically sampled with higher frequencies resulting in only small delta increments between two consecutive sampling time points.

The aforementioned domain properties can be used to exchange data more effectively between interacting software agents. Therefore, the difference between the current message to be sent and its preceding message is calculated and only its difference values are communicated to avoid consuming bandwidth for redundant information that can be safely reconstructed at receiver side.

Fig. 2 depicts the delta-based deserialization process using the generic runtime message description: An application realized with OpenDaVINCI receives a new message encapsulated in a *Container*. The container contains the actual serialized message from the sender as payload, and meta-information as time stamp when the container left the sender, and time stamp when it arrived at the receiver. By using the container identifier, the application starts to access the serialized content in the container with the given type of the expected message.

During the first communication cycle, a complete message is exchanged between the sender and receiver and thus, the contained data, in our case for instance a *VehicleData* message, is deserialized. The content of that message is stored to serve as basis in the case that the sender would send a delta message only in the next communication cycle. In that case, a *DeltaDeserializerVisitor* is instantiated to read the difference information from the received container. Next, the corresponding previous message is restored so that *DeltaReconstructor* can calculate the new complete *VehicleData* resulting from the differences applied to this message's predecessor.

The process described above is realized in the lower communication layer of OpenDaVINCI to encapsulate it from the user. Thus, the adaptive data marshalling is fully transparent to the user. Furthermore, this adaptation layer can also apply different delta strategies by not only considering just the previously received message as basis for reconstruction. Here, further properties of the application domain could be considered in the design of data reconstruction algorithms.

```
1  message EvaluationMessage {
2      uint16 uiAtt1 [default = 1234, id = 11];
3      uint16 uiAtt2 [default = 1234, id = 12];
4      uint16 uiAtt3 [default = 1234, id = 13];
5      ...
6      uint16 uiAttN [default = 1234, id = 10+N];
7
8      double dAtt1 [default = 1.2, id = 31];
9      double dAtt2 [default = 1.2, id = 32];
10     double dAtt3 [default = 1.2, id = 33];
11     ...
12     double dAttN [default = 1.2, id = 30+N];
13 }
```

Figure 3. Structure of the evaluation message, which is dynamically composed runtime.

4. Evaluation

The evaluation of a given data marshalling approach typically depends on the intended application context. To circumvent this issue and properly evaluate the aforementioned marshalling approaches, we have designed the data collection step in a systematic way, as described in the remainder of this section. Further, the experimentation procedure is intended to be reproducible, i.e., the evaluation could be repeated with both the same setting or with different parameters, so other scenarios can also be considered in the future.

4.1 Experimentation Procedure

Instead of choosing a specific scenario in which the different approaches are evaluated, we decided to use the generic message description feature to dynamically create different message types at runtime. Thereby, we could systematically vary a message's parameters influencing the performance of the respective approaches.

Due to the nature of the exchanged messages, the integer values had generally orders of magnitude of at most 10^3, while the floating point values were usually in the 10^0 magnitude and increasing or decreasing by the centesimal digit. For these reasons, in the study we have selected the following parameters:

- Varying the number of integral data fields (16 bit integer) in a message from 0 to 10;
- Varying the number of double data fields in a message from 0 to 10;
- Increasing the value for the integral data fields from an order of magnitude from 10^0 to 10^3;

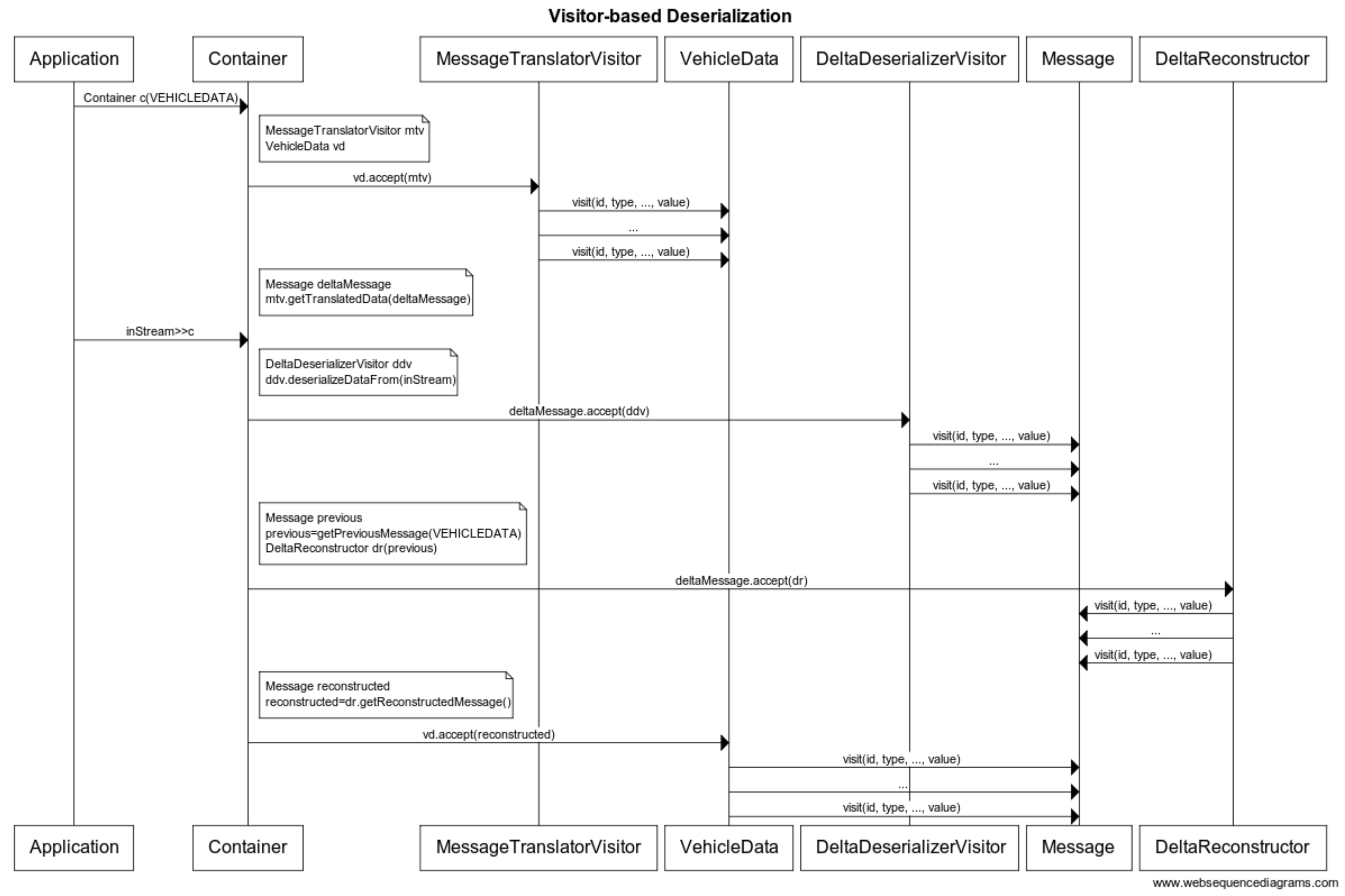

Figure 2. Sequence chart of the visitor-based delta deserialization.

- Increasing the value for the double data fields from 1.2 to 1.3 using the steps 0 (no increment), 0.01, 0.02, 0.05, and 0.1.

The steps in the floating point numbers were defined based on our domain experience focusing on high frequency data exchange. In total, 100 different message attribute fields times four different order of magnitudes for the integral types times five different double differences, resulting in $2,000$ different message configurations, were evaluated with all three data marshalling approaches. The overall structure of the evaluation message generated at runtime is depicted in Fig. 3.

4.2 Results

After systematically iterating all parameters using the generic message representation, we obtained the following results. The smallest message could be created with Google Protobuf consuming 6 bytes only for just one integral data field with values up to an order of magnitude of 10^2. The largest message created with Protobuf occupied 139 bytes for 10 integral data fields for data up to an order of magnitude of 10^3 and 10 double fields.

The smallest message with LCM consumed 21 bytes for 1 integral data field up to an order of magnitude of 10^3; the largest message with LCM covering 10 fields of both types consumes 119 bytes and thus, approximately 15% less then Protobuf. The reason therefor is that the internal structure of Protobuf uses a key field while LCM simply writes the data in sequence without further control data.

The smallest delta message with 22 bytes - and thus, approximately 3.5 times larger than the smallest Protobuf message - was obtained for a message carrying 1 integral field only. The largest delta message with 156 bytes was created for a message having all 10 integral data fields and all 10 double data fields.

A chart depicting the increasing amount of bytes required to store a message of the respective type is depicted in Fig. 4. Defining a message with just 4 data fields of type double results in a serialized message of the same length of 44 bytes for Google Protobuf and the self-adaptive data marshalling. Having already 2 additional data fields of an integral type resulted in the self-adaptive approach to fall behind LCM and Google Protobuf. And finally, having 7 fields of an integral

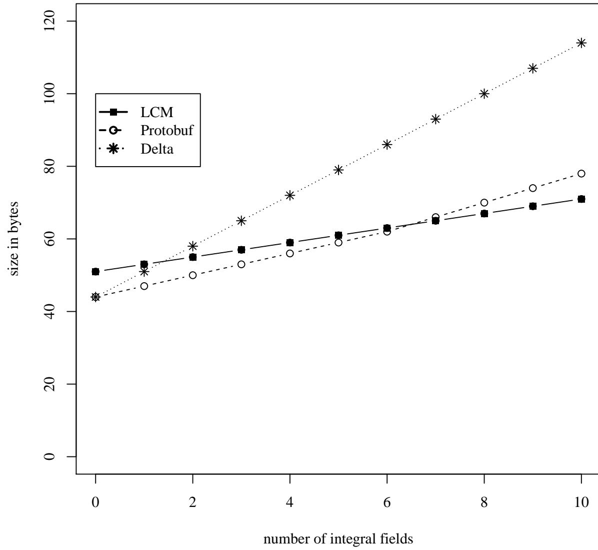

Figure 4. Increasing amount of bytes to store a message with 4 double data fields and and increasing amount of integral data types: Google Protobuf and the self-adaptive delta approach are identical with 44 bytes for just 4 double fields. Having already 2 additional integral type fields, LCM and Protobuf consume less than the delta approach, and with 7 integral data types, LCM outperforms all other approaches due to its design.

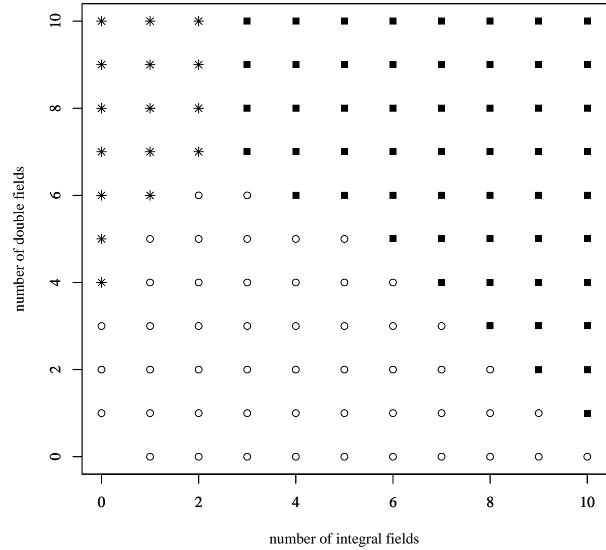

Figure 5. Matrix of the performance evaluation of the different approaches for a message with different configurations for integral types up to an order of magnitude 10^3 and a varying range for the double values from 0 to 0.1: On the x-axis, the number of additional integral data fields in the message is depicted and on the y-axis, the number of double data fields in the message is shown. In cells with a circle, Google Protobuf would result in the smallest byte sequence; cells with a star show the message configurations where the self-adaptive data marshalling approach would outperform the other approaches, and cells with a square represent LCM trumping the other data marshalling concepts.

type resulting in 65 bytes for LCM and 66 bytes for Protobuf at an order of magnitude of 10^3.

Fig. 5 summarizes the result for all message configurations where the integral data field can hold values up to an order of magnitude of 10^3. The circles represent configurations where Google Protobuf is the best choice in terms of the shortest resulting serialization byte sequence; squares show configurations, where LCM would trump over the other approaches; and the stars show message configurations where the self-adaptive data serialization would be the best choice.

4.3 Discussion

In order to obtain a systematic evaluation of the examined marshalling approaches, auto-generated messages were used, and they were populated with a varying number of integer and floating point variables, containing values with fixed ranges and increments. From the results of this evaluation, as shown in the charts, it is apparent that in case of messages with only integral types, Google Protobuf is the best choice; however, the more data fields a message contains, the more compact data format of LCM is paying off resulting in a growing number of squares in Fig. 5. By design, the self-adaptive delta approach is paying off for messages with a higher number of non-integral data fields as its concept aims for reducing the amount of data by assuming that the difference between two consecutively sent messages with only a short time gap in between is rather small.

Another observation from Fig. 5 is that once a specific design decision is made regarding the use of a specific serialization approach, the message design during system development should obey the identified boundaries for data fields to avoid unwanted side-effects at runtime affecting the performance of the data marshalling.

4.4 Threats to validity

In this section, we discuss the validity of this study according to four perspectives [8].

Considering construct validity, the design of the study by using a generic message representation that can be systematically defined allows for a scenario-independent analysis of the performance of the three approaches. Thus, no specific application domain is favored during the experimentation design.

Regarding internal validity, four different aspects influencing a serialized message's length were identified and systematically varied. All identified factors directly contribute to the message size; however, we did not study a random ordering of the attribute fields.

Concerning external validity, the comparison of our self-adaptive data marshalling approach with Google Protobuf and LCM can be considered as relevant as both other approaches are widely used and have shown the applicability in the domain of cyber-physical systems. Thus, the findings presented in this study have an impact on the design of such systems.

Regarding reliability, the range for the floats used in the evaluation was inspired by applications in the self-driving vehicles domain. As the goal for the delta approach was to address high frequency data exchanges, the motivation for the increment values was due to the small numeric difference between values in consecutive packets.

5. Conclusion and Future Work

Distributed software systems with interacting agents base on communication protocols to exchange information to act properly or to synchronize tasks. At a system's development time, domain specific languages assist the developers to quickly specify messages to be exchanged between the interacting system entities. While such DSL facilitates faster system development and modularization, our study shows that the composition of a message at design time can negatively influence the performance of a distributed software system.

One way to systematically evaluate different marshalling approaches is to have the components of the distributed system to automatically generate a number of messages that will be exchanged using one of the marshalling techniques that are analyzed. In this way, the total number of bytes exchanged in the system will act as one of the main parameters to compare the different marshalling approaches and their efficiency. The messages were generated containing an incrementing number of integer and floating point variables, and their values were incremented by predefined steps.

We experimentally showed that Google Protobuf is well suited for compact messages with few data fields focusing primarily on integral types. LCM in contrast is paying off as the number of message fields increases. Our self-adaptive data marshalling approach, which is making use of the practical fact that the difference between two consecutively sent messages is rather small, is beneficial in the case of messages that are heavy on non-integral data fields.

Future work would include extending the comparison to more marshalling approaches in order to get an even broader view of their performances and how these are affected by the nature of the processed messages in terms of number and type of contained variables. More efforts would also be required to further improve the self-adaptive data marshalling approach, especially for messages where the floating point types are not the predominant ones, since the result of the evaluation clearly shows the boundaries in which this technique has to become more efficient. Studying the way other approaches successfully process messages with integer variables exchanging smaller amounts of bytes will prove beneficial in this sense. Furthermore, alternative algorithms to model the hysteresis of previously received messages considering more messages than only the previously received one need to be explored to maker better use of the delta marshalling approach in different application contexts.

References

[1] Google protocol buffers. https://github.com/google/protobuf. *Accessed 2015-04-28*.

[2] C. Berger. From a Competition for Self-Driving Miniature Cars to a Standardized Experimental Platform: Concept, Models, Architecture, and Evaluation. *Journal of Software Engineering for Robotics*, 5(1):63–79, June 2014. URL `http://arxiv.org/abs/1406.7768`.

[3] C. Berger and M. Dukaczewski. Comparison of Architectural Design Decisions for Resource-Constrained Self-Driving Cars - A Multiple Case-Study. In E. Plödereder, L. Grunske, E. Schneider, and D. Ull, editors, *Proceedings of the INFORMATIK 2014*, pages 2157–2168, Stuttgart, Germany, Sept. 2014. Gesellschaft für Informatik e.V. (GI).

[4] C. Berger, M. Chaudron, R. Heldal, O. Landsiedel, and E. M. Schiller. Model-based, Composable Simulation for the Development of Autonomous Miniature Vehicles. In *Proceedings of the SCS/IEEE Symposium on Theory of Modeling and Simulation*, page 7, San Diego, CA, USA, Apr. 2013. URL `http://dl.acm.org/citation.cfm?id=2499651`.

[5] F. Giaimo, H. Andrade, C. Berger, and I. Crnkovic. Improving Bandwidth Efficiency with Self-Adaptation for Data Marshalling on the Example of a Self-Driving Miniature Car. In *Proceedings of the 1st International Workshop on Software Architectures for Next-generation Cyber-physical Systems (SANCS)*, page 6, Sept. 2015.

[6] A. S. Huang, E. Olson, and D. C. Moore. LCM: Lightweight Communications and Marshalling. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 4057–4062, Taipei, Taiwan, Oct. 2010. IEEE. ISBN 978-1-4244-6674-0. . URL `http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5649358`.

[7] M. Quigley, B. Gerkey, K. Conley, J. Faust, T. Foote, J. Leibs, E. Berger, R. Wheeler, and A. Ng. ROS : an open-source Robot Operating System. In *Open-Source Software workshop of the International Conference on Robotics and Automation (ICRA)*, number Figure 1, Kobe, Japan, May 2009.

[8] P. Runeson and M. Höst. Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering*, 14(2):131–164, Dec. 2008. ISSN 1382-3256. . URL `http://link.springer.com/10.1007/s10664-008-9102-8`.

[9] W. Schwitzer and V. Popa. Using Protocol Buffers for Resource-Constrained Distributed Embedded Systems. Technical report, Institut für Information, Munich, Germany, Nov. 2011.