Safety Implications of Serialization Timing in Autonomous Vehicles

Zachary Pierce, Nathan Aschbacher
PolySync Technologies, Inc.

Introduction

It is a commonly held belief that time is money. True as that may be, when it comes to autonomous vehicles, a more apt aphorism may read "time is safety."

In emergent situations, the reaction speed of an autonomous vehicle determines whether a dangerous situation can be resolved safely. In normal operations, the amount of time a vehicle's software spends accomplishing its bare minimum objectives determines how much time remains to be spent ensuring the situation does not become a problem.

In this paper, we examine the possible effect that differences in the time budget consumption of cross-component message serialization–a core vehicular software function–have on overall system safety.

Background

Message encoding and decoding is essential every time a vehicular system component needs to communicate with another component. If we broadly divide an autonomous system's capabilities into the Sense-Plan-Act paradigm [1], it's clear that there's communication between each step. Sensors encode messages that are transmitting to the autonomous planning layer, which must decode that data, make decisions, and pass along command messages to the actuating components. Depending on the system's architecture, these phases may be implemented by multiple collaborating subcomponents rather than single monolithic compute units, requiring further coordination by some shared messaging approach. Even when hardware architecture centralizes computational work onto high-specification physical devices, it is still common for co-located processes to communicate through message passing.

The software industry has given rise to multiple message serialization technologies that are relevant to the autonomous vehicle use case. Compared to ad-hoc solutions, best practice message serialization approaches define standardized representations to improve the assurance of consistency across components. High-performance serialization formats of note include Apache Thrift [2], Cap'n Proto [3], Colfer [4], FlatBuffers [5], Lightweight Communications and Marshalling (LCM) [6], Protocol Buffers [7], Simple Binary Encoding [8], and the Object Management Group's Common Data Representation (CDR) [9]. Some formats are strongly linked to broader communications frameworks for embedded environments. For example, CDR is the default message format for the Data Distribution Service (DDS) real-time middleware system, and was recently adopted into a similar role by the Robot Operating System 2 (ROS2) [10]. Others, such as Protocol Buffers, Cap'n Proto, and Thrift were designed with data center remote procedure calls in mind [2], but nevertheless possess embedded-system compatible implementations and competitive performance.

Significance Argument from End-to-End Emergency Response

When a sudden development in the external environment causes an emergency situation, an autonomous vehicle's end-to-end reaction time is critical in determining the available set of safe responses and the success of their execution. A simple example is the need to rapidly decelerate from highway speeds to avoid a dangerous obstacle. Assuming a vehicle cruising at 60 miles per hour and an emergent situation requiring a total stop at maximum deceleration, every millisecond of end-to-end reaction delay extends the distance traveled by at least an inch. Anyone who has skidded to a stop inches from a collision will be able to appreciate the visceral significance of those milliseconds.

Significance Argument from More Thoughtful Autonomy Modules

Naturally, messaging itself is not the primary goal of an autonomous vehicle system. Nor is message serialization frequently the primary computational-time-consumer.

Environment perception and course planning are the headline features of modern autonomous vehicles, and traditionally the biggest spenders of allotted time budgets for computational units due to the inherent complexity of their tasks. Most algorithmic or machine learning techniques implemented within these autonomy systems are approximate, optimizing, or adaptational in nature. Often, the more computational resources that can be spent on executing said algorithms, the better of a job they can do at their essential roles. In that regard, computational time spent elsewhere–without explicit safety motivation–may be considered nonessential and ripe for minimization. Selecting a high-efficiency serialization technology can thus be cast as a matter of freeing up resources that can be spent on autonomy tasks, therefore improving the quality of operations.

Significance Argument from Smarter Distributed Systems Robustness

Minimizing the time-cost of messaging enables the use of distributed systems techniques for improving reliability and safety. Autonomous vehicle software systems are always made of collaborating parts that communicate with each other. However, they are not always architected with distributed systems techniques such that the collaborating components are redundant, capable of detecting flaws in other subsystems, and self-recovering from individual component failures. These desirable safety features are accomplishable at the cost of communication beyond the bare minimum.

Take, for example, a subsystem responsible for controlling the steering angle of the vehicle. To enable some safe redundancy, assume three physically separate computational units are running software components that should be capable of constructing and sending actuation-triggering messages. In order to prevent conflicting messaging, at any given time only a single one of those steering components is the "leader" of the group and responsible for actually sending actuation messages. All of the components talk to each other to collectively determine which one is presently the "leader" and check for faults. In the event that the "leader" component fails to do its job, due to some physical damage to the computational unit or a software error, the remaining components must decide which of them will become the new leader and take responsibility for the subsystem's role of sending steering actuation messages.

During all of this, the frequency and latency with which messaging can be executed between the collaborating components are key factors in settling how long it takes for the subsystem to recover from failures. Another critical factor is the selected consensus algorithm or protocol in use. If the computational cost of messaging were high, it would necessarily drive down the frequency with which components could check in on each other and detect potential faults. Similarly, the fewer messages that a collaborating system can afford to send, the more limited the options are for the choice of consensus algorithm. Thus shrinking the range of possible levels of providable assurance.

Once again, the time required to do message serialization and deserialization represents a plausible bound on the level of safety possible for an autonomous vehicle.

Methods

In order to assess the significance of message serialization technology selection on computational time consumption, we set out to benchmark numerous serialization formats for some representative scenarios, detailed below. All measurements were executed with unified test harnesses written in Rust, exercising optimized serialization codec code implemented in C, C++, or Rust. The tested codecs were Bincode [11], Cap'n Proto (both standard and packed variations of the Rust implementation [12]), Colfer, a commercial DDS CDR Stream, FastCDR (an open source DDS CDR implementation) [13], JSON through the serde_json library [14], LCM, the Prost Protocol Buffers implementation [15], and the Quick Protobuf Protocol Buffers implementation [16].

All measurements were based on a single message layout used in different manners. As sensor data represents a significant proportion of total message data within a typical autonomous vehicle, we selected an exemplar message schema based on a LiDAR message. The schema contained both a fixed-size metadata portion and a variable-length portion representing three-dimensional points with associated intensities. Every serialization format evaluated was capable of representing a sufficiently equivalent schema in its own Interface Definition Language.

A normalized in-memory representation of the message was implemented to enable confirmation of fully equivalent message content for serialization and deserialization across formats.

```rust
// Rust language normalized form
of message structures

#[repr(C)]

pub struct LiDARPointsMsg {
     pub msg_info: MsgInfo,
     pub points: Vec<LiDARPoint>,
}

#[repr(C)]
pub struct MsgInfo {
     pub kind: u64,
     pub timestamp: u64,
     pub guid: u64,
}


#[repr(C)]
pub struct LiDARPoint {
     pub position: [f32; 3],
     pub intensity: u8,
}
```

All use cases were measured and summary statistics gathered on a Lenovo Thinkpad T460 with an Intel(R) Core(TM) i5-6200U CPU at 2.30GHz and 12 GiB DDR3 RAM running the 4.10.0 Linux kernel. The use case experiments were repeated with varying numbers of points in the encoded or decoded data, allowing for evaluation of the impact of larger or smaller message contents.

Additionally, a "no-operation" codec implementation was constructed for each experiment to account for any overhead associated with the measurement harness itself.

Encoding Time from Full Normalized Form to Buffer

This experiment measured the time required to take an in-memory struct instance of the normalized form, LiDARPointsMsg and encode its data into a pre-allocated byte buffer and report the number of bytes encoded.

Some codecs made use of intermediate structures to hold the data, and in those cases the time for each phase was observed (normalized to intermediate as a first phase, and intermediate to encoded in byte buffer as a second). This distinction allows for evaluation of the effective encoding cost of working with application-domain representative structures (here, the normalized form) as opposed to a representation specialized to the serialization format.

Decoding Time from Buffer to Full Normalized Form

The natural parallel to the normalized encoding experiment, the normalized decoding experiment measured the time required to consume an in-memory byte buffer containing a message encoded in the appropriate format and decode that data into a newly instantiated struct of the normalized representation.

	Encode		Decode							
	Normalized		Normalized		Full Projection		Fixed Field		Repeating Field	
	mean	stdev	mean	stdev	mean	stdev	mean	stdev	mean	stdev
Bincode	263.2	7.0	344.7	13.8	350.8	12.0	336.2	12.9	336.3	12.9
CapnprotoPacked	786.2	31.6	1552.5	56.1	1440.8	58.2	427.1	16.9	427.6	16.8
CapnprotoStandard	346.9	15.1	1197.5	36.8	1063.9	40.1	55.0	2.6	45.3	1.1
Colfer	288.5	3.2	124.6	4.5	140.1	3.1	82.4	3.1	82.1	3.1
FastCdr	135.7	3.3	155.9	4.0	160.3	3.4	101.6	3.3	101.4	3.1
FlatBuffers	155.6	3.0	75.7	1.6	59.3	0.6	0.1	0.0	0.0	0.0
Json	3495.1	43.5	11452.8	79.3	11469.5	194.5	11446.2	153.6	11472.9	209.8
Lcm	113.5	1.1	89.5	1.4	106.7	1.2	48.8	0.8	48.7	0.7
ProstProtobuf	407.6	16.6	975.4	46.1	929.0	47.9	914.3	45.2	914.4	46.6
QuickProtobuf	914.4	48.6	469.6	15.2	392.1	14.6	376.5	15.0	378.0	15.0
Commercial DDS	1107.8	30.2	1129.5	51.9	863.3	13.6	808.6	22.8	778.9	12.6

Table 1. Summary of timings for key metrics, in microseconds, for the 10,000 points-per-message case. Includes encoding time from normalized form, decoding to normalized form, decoding to full message projection and direct consumption, projection of a fixed region field, and projection of a variable region member field.

Again, some codecs made use of intermediate structures to hold the data, and in those cases the time for each phase was observed (byte buffer to intermediate as the first phase and intermediate to normalized form as the second).

Decoding and Consuming Full Data Projection

This experiment measured the time it took to consume an in-memory byte buffer containing an encoded message and decode the message enough to extract every numeric value from every sub-field and sub-structure. Said values were cast to a float representation and summed.

This decoding use case is perhaps the most representative of a fully optimized application consuming a tailor-made message with a given serialization format. It bypasses any cost associated with incidental mismatches between the data as decodable and the data in the normalized form. It also requires the decoding and usage of the entire message contents, elucidating the complete cost of data interpretation for formats where the intermediate form may have a heavily deferred interpretation strategy.

Decoding Time from Buffer to Fixed-Region Projection

This experiment measured the time it took each codec to take an in-memory byte buffer containing a message encoded in the appropriate format and extract a single value from a single field in the fixed-length region of the message. In particular, implementations extracted the guid field of the msg_info metadata. Due to the extremely low time required for such a projection, repeated executions were done in some cases, accumulating the sum of the extracted values for correctness-checking.

Decoding Time from Buffer to Variable-Region Projection

This final experiment measured the time it took each codec to take an in-memory byte buffer containing a message encoded in the appropriate format and extract a float value from the middlemost point. Implementations extracted the y-dimension value (index 1) of the position field of the LiDAR point equidistant from the beginning and the end of the set of available points. Due to the extremely low time required for such a projection in some cases, repeated executions were done, accumulating the sum of the extracted values for correctness-checking.

Results

Table 1 shows the overall results of the experiments, highlighting 10,000 points-per-message case. JSON is immediately noteworthy, as it demonstrates slower performance by a factor of 10 compared to the next-worst entry for every measurement except encoding, where that factor drops to 3. As the sole text-based serialization format in a field of binary formats, this difference is not suprising. For the sake of improving clarity by reducing scale skew, JSON is excluded from subsequent figures and tables. Along those lines, the "packed" representation variation of the Cap'n Proto serialization format is universally outperformed by its "standard" representation alternative, and is henceforth omitted to minimize clutter.

For several of the gathered measurements, there is a cluster of high performers hovering within a range of approximately 1.3 from the fastest option. When measuring encoding from normalized representation, LCM has the fastest mean time, followed closely by FlatBuffers and FastCDR. Similarly, for the decoding to normalized form case and the decoding to full data projection cases, LCM, FlatBuffers, and FastCDR are joined by Colfer. Considering all tested point ranges, Cap'n Proto, the commercial DDS implementation, and the Protocol Buffers implementations frequently do not approach within even a factor of 2 of the slowest of the top tier.

Figure 1 emphasizes this gap in performance, as well as identifies the serialization technologies that are most challenging to a two-tier classification. Figure 1 visualizes the sum of duration of encoding and decoding to and from the fully normalized form, with the Protocol Buffers options, DDS and Cap'n Proto separated by a gap of over 700 microseconds from LCM, FlatBuffers, and FastCDR to the left. Bincode and Colfer straddle the middle, over 2 times slower than the fastest normalized round-trip technology (LCM) but twice as fast as the DDS implementation, Cap'n Proto, and Protocol Buffers group.

Figure 2 illuminates the performance of serialization technologies as the number of data points encoded per message increases. Predictably, duration required for encoding and decoding tends to rise as the points increase. Most serialization formats appear to display similar relative speed at different sizes, with some exceptions. FlatBuffers begins with comparatively high overhead and subpar performance for very small messages, but migrates towards the front of the class as the points-per-message crosses 10. The commercial DDS implementation demonstrates exceptionally poor performance whenever

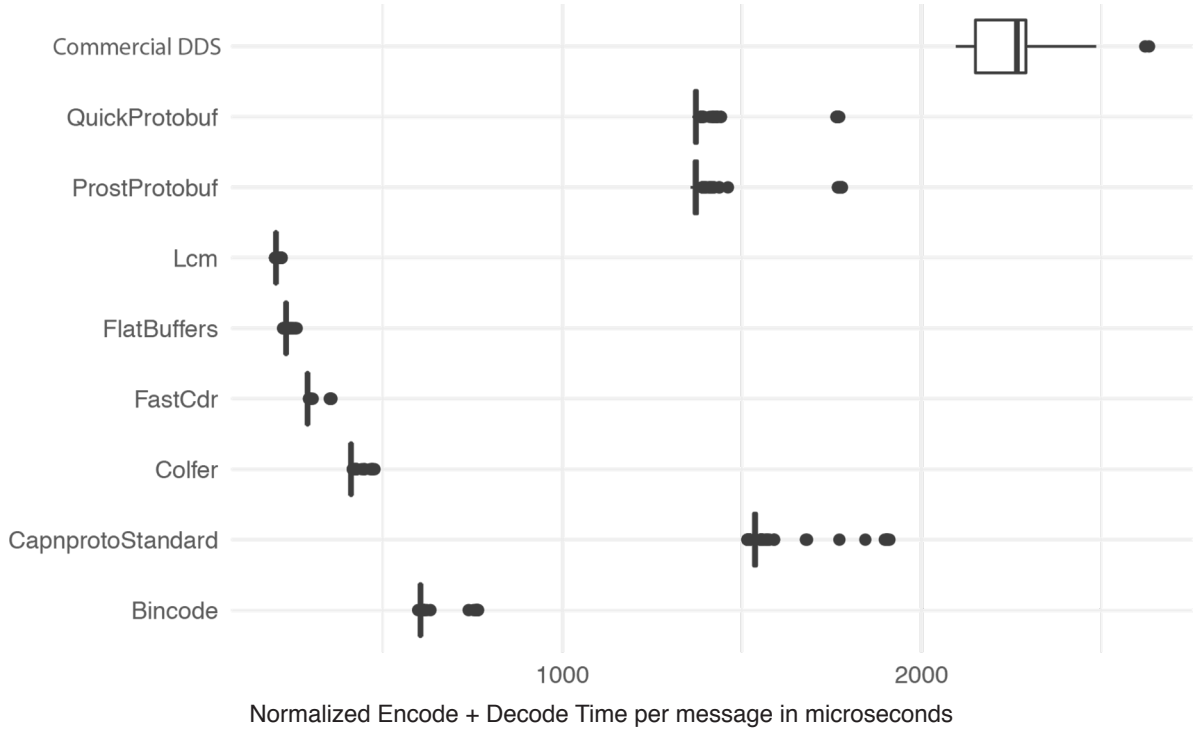

Figure 1. Round trip to normalized form serialization time for the 10,000 points−per−message case.

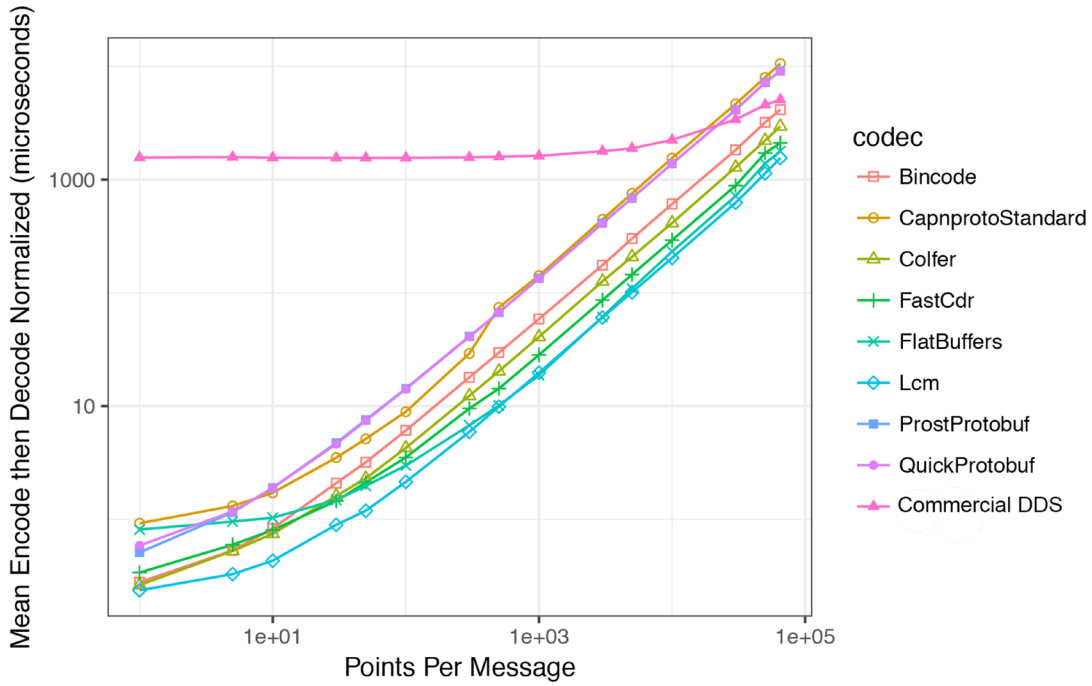

Figure 2. Round trip to normalized form time, per message.

Figure 2: Round trip to decoded projection summation time, per message.

the points-per-message differs in order of magnitude from the maximum allowed per its implementation schema. Once the number of points tested matched the schema's theoretical maximum, DDS' performance shifted from abysmal to second-tier.

Conclusions

The results demonstrate clearly that some formats are more time-efficient than others, often by wide margins. Given that some of the differences between formats are more than two orders of magnitude apart, it is difficult to argue that format selection won't affect the bottom line of time available for accomplishing safety-critical tasks. The fact that technologies not necessarily designed for the embedded space consistently ranked among the fastest performers for each experiment strongly suggests that there's room for reconsideration of extant serialization choices.

The poor showing of both Protocol Buffers implementations measured merits a recommendation against that format's use. This result is in contrast to prior experimentation which demonstrated a Protocol Buffer implementation performing comparably to LCM [17], one of the fastest formats presently under test. Given that it was designed to be a faster version of Protocol Buffers by focusing on fewer copies [3], Cap'n Proto also underperformed expectations. Especially considering how FlatBuffers, another low-copy-oriented format, excelled in most categories. The relatively new and unknown format Colfer did well in decoding tasks, up there with more familiar embedded-space formats like CDR and LCM.

Since time is safety in this industry, and serialization formats display radically different profiles for time consumed, there is significant opportunity to optimize reliability by selecting a top-tier format.

References

[1] DiClemente J, Mogos S, Wang R (2014) Autonomous Car Policy Report. Available at: https:// www.cmu. edu/epp/people/faculty/course-reports/Autonomous%20Car%20Final%20Report.pdf.

[2] Slee M, Agarwal A, Kwiatkowski M (2007) Thrift: Scalable Cross-Language Services. Available at: http://thrift.apache.org/static/files/thrift-20070401.pdf.

[3] Varda K (2013) Bincode: A binary encoder / decoder implementation in Rust. Available at: https: //capnproto.org.

[4] de Kloe P (2016) Colfer binary serialization format. Available at: https://github.com/pascaldekloe/colfer.

[5] FlatBuffers: Memory Efficient Serialization Library. Available at: https://google.github.io/flatbuffers.

[6] Huang A, Olson E, Moore D (2010) LCM: Lightweight Communications and Marshalling. Intelligent Robots and Systems (IROS), 2010 IEEE/RSJ International Conference on. doi:10.1109/IROS.2010.5649358.

[7] Protocol Buffers. Available at: https://developers.google.com/protocol-buffers/.

[8] Montgomery T, Thompson M, Deheurles O, Warburton R, Segall B (2014) Simple Binary Encoding: High Performance Message Codec. Available at: https://github.com/real-logic/simple-binary-encoding.

[9] DDS Interoperability Wire Protocol Available at: http://www.omg.org/spec/DDSI-RTPS/.

[10] ROS on DDS (2015) Available at: http://design.ros2.org/articles/ros_on_dds.html.

[11] Overby T (2014) Bincode: A binary encoder / decoder implementation in Rust. Available at: https: //github.com/TyOverby/bincode.

[12] Renshaw D Cap'n Proto for Rust. Available at: https://github.com/capnproto/capnproto-rust.

[13] Fast-CDR. Available at: https://github.com/eProsima/Fast-CDR.

[14] Serde JSON (2014) Available at: https://github.com/serde-rs/json.

[15] Burkert D PROST! a Protocol Buffers implementation for the Rust Language. Available at: https: //github.com/danburkert/prost/.

[16] Tuffe J Quick-Protobuf: A rust implementation of protobuf parser. Available at: https://github.com/ tafia/quick-protobuf.

[17] Borosean V, Eidukas S, Dang A (2015) Evaluating Data Marshalling Approaches for Embedded Real-Time Systems on the Example of Autonomous Scaled Cars.