Performance Comparison of Messaging Protocols and Serialization Formats for Digital Twins in IoV

Daniel Persson Proos
Linköping University, Sweden

Niklas Carlsson
Linköping University, Sweden

***Abstract*—This paper compares the performance tradeoffs of popular application-layer messaging protocols and binary serialization formats in the context of vehicle-to-cloud communication for maintaining digital twins. Of particular interest are solutions that enable emerging delay-sensitive Intelligent Transport System (ITS) features, while reducing data usage in mobile networks. The evaluated protocols are Constrained Application Protocol (CoAP), Advanced Message Queuing Protocol (AMQP), and Message Queuing Telemetry Transport (MQTT), and the serialization formats studied are Protobuf and Flatbuffers. The results show that CoAP – the only User Datagram Protocol (UDP) based protocol evaluated – has the lowest latency and overhead, but is not able to guarantee reliable transfer, even when using its confirmable message feature. For our context, the best performer that guarantees reliable transfer is MQTT. For the serialization formats, Protobuf is shown to have faster serialization speed and three times smaller serialized message size than Flatbuffers. In contrast, Flatbuffers uses less memory and has shorter deserialization time, making it an interesting alternative for applications where the vehicle is the receiver of time sensitive information. Finally, insights and implications on ITS communication are discussed.**

I. Introduction

The transport industry is headed into a connected future, sometimes referred to as the Internet of Vehicles (IoV) [1]. As part of this transition, many traditional car, truck, and bus manufacturers are aiming to collect and use detailed (sensory) data from each vehicle to improve the performance and use of both individual vehicles and entire vehicle fleets. For example, digital twins [2] of vehicles and other physical entities are expected to be stored and updated in the cloud. Using these digital twins, advanced data analytics are then expected to be used to improve system solutions that benefit the customer.

While this data today typically is extracted at a coarse time granularity, it is expected that data will be collected and used at increasingly finer time granularity. This trend, coupled with emerging time-sensitive applications of digital twins is expected to place increasingly strict demands on network communication between vehicles and cloud computing resources; e.g., with regards to both bandwidth and latencies [3].

Looking at the communication between vehicles and the cloud, a provider can take several steps to reduce overheads (expressed as memory and/or bandwidth usage) and end-to-end latencies [4]. In this paper, we focus primarily on the performance impact of the serialization format and application-layer protocol used to send messages, the impact of parameter choices within the protocols, and their performance tradeoffs. Of particular interest are scenarios with highly variable bandwidth conditions, as seen by vehicles [5]–[7].

ISBN 978-3-903176-28-7 © 2020 IFIP

Serialization formats: Data serialization plays an important role in reducing the message size. Reductions in the payload size help reduce transfer times, lower the risk of packets being dropped, and reduce transfer costs when using a mobile network [8]. Here, we evaluate two publicly available serialization formats: Protobuf and Flatbuffers. Protobuf, developed by Google, has become a standard format for binary data serialization in many applications because of its performance and widespread use. It uses predefined message schemas, defined in .proto files, known to both the sender and receiver. Flatbuffers, also developed by Google, is a zero-copy serialization format designed to have shorter deserialization time and use less memory than Protobuf [9], [10]. Compared with other formats, very limited amount of academic performance studies have been conducted on Flatbuffers. Here, we compare Flatbuffers and Protobuf using real production messages.

Messaging protocols: Hyper-Text Transport Protocol (HTTP), the de-facto protocol on the web, is typically considered too verbose for Internet of Things (IoT) communication. Therefore, many new, lightweight messaging protocols aimed at this market have been developed in the last two decades [4], [11]–[13]. Here, we use performance experiments to compare and contrast the tradeoffs associated with some of the most prevalent IoT messaging protocols: MQTT, AMQP, and CoAP. These protocols have different strengths and weaknesses, with AMQP being the most verbose and CoAP being most lightweight. While all three protocols have been developed for IoT, they have not been thoroughly tested in an IoV setting. In contrast to prior work, we compare the performance of these protocols under network conditions that mimic the unpredictable and highly time-varying conditions that mobile connections incur using a controlled test environment (allowing repeatable and fair head-to-head comparisons).

For our evaluation we use real production messages and bandwidth traces from mobile clients. Two primary use cases are considered. In the first use case, the vehicle sends *large chunks* of data to the cloud without any strict latency demands. In the second use case, the vehicle sends *small chunks* of data with an expectation of low latencies. For both use cases, we emulate the platform and network conditions seen by a vehicle driving along a road while using the mobile networks available. In particular, we (i) develop a network environment for sending and receiving messages using the different messaging protocols and data serialization formats interchangeably, (ii) setup a test framework that allows controlled and repeatable experiments in which bandwidth conditions are varied based on trace data, and (iii) evaluate the performance of each combination (serialization format + messaging protocol) and determine which is best suited for the considered use cases.

Our results provide quantitative comparisons and qualitative insights into which serialization formats and messaging protocols to use when optimizing the communication between mobile vehicles and cloud. This includes exploring quantitative tradeoffs capturing the performance of different formats and protocols. With the exception of scenarios where the memory footprint or deserialization times are the main bottleneck, Protobuf beats Flatbuffers, as it typically achieves three times smaller serialized message size and has faster serialization speed. This makes it the obvious choice for most vehicle-to-cloud communication, where there typically are more resource constraints on the sender. Among the messaging protocols, only the the TCP-based protocols (MQTT and AMQP) deliver all messages. Even when using the option of confirmable messages, CoAP was not able to achieve 100% delivery under high-loss scenarios. Of the TCP-based protocols, the lightweight nature of MQTT typically makes it the winner over AMQP, as it has much smaller overheads. CoAP is most suitable for delivery of small update messages that do not require guaranteed delivery, as it achieves the lowest latencies and overheads (at the cost of some failed messages).

Finally, when discussing our results, it should be noted that we do not consider the use of satellite links, alternative hardware architectures, or other alternative ways to reduce the end-to-end latencies (e.g., pushing computing to the "edge") and do not consider the impact that encryption may have.

Outline: Section II provides background on IoV, digital twins, and the serialization formats and messaging protocols evaluated. Related work is discussed in Section III. Section IV describes our methodology. Sections V and VI present our performance results. Conclusions are presented in Section VII.

II. Background: Selected Formats and Protocols

The Internet of Vehicles (IoV) has evolved from primarily focusing on inter-vehicle mesh networks [14] and Vehicle Ad-hoc Networks (VANETs) to more complete solutions for vehicle networking, with recent trends heavily inspired by IoT. The purpose of digital twin platforms are plentiful. However, within IoV, the main uses of digital twins are typically centered around the idea that up-to-date digital twins (maintained in the cloud) will provide the customer with a clearer view of the state of their vehicle fleet (e.g., for predictive maintenance aiming to minimize unplanned stops due to breakdowns). In the paper, we primarily focus on vehicle-to-cloud communication (often called called Vehicle-to-Infrastructure (V2I) communication) over WiFi or cellular networks (e.g., 4G/LTE/5G) [15] targeting such scenarios. However, we also briefly discuss use cases involving HD maps and automated vehicle scenarios (e.g., to reduce road congestion by rerouting vehicles based on real-time information from other vehicles) that rely on V2I communication. Overall, these technologies are expected to help reduce fuel consumption, delivery times, emissions, and traffic jams.

A. Data Serialization

Data serialization is used to structure data in a streamlined format before storing or sending. Broadly, there are two serialization approaches: text based and binary. With text based serialization, the data is typically structured into key-value pairs in readable text format. With binary serialization, the key-value pairs are stored in binary format, typically reducing the space requirements. To further reduce space requirements, a schema can be used for both serialization and deserialization.

Choice of formats: While there are many serialization formats [8], [16]–[23], binary formats are typically desired as they provide smaller message size than text based formats like JSON and XML. Among the binary serialization formats, Protobuf was selected as a baseline as it already is used today for the chosen production messages used in this study, and since it had been suggested as the top candidate in some prior works [8], [24]. Flatbuffers was selected for its use of zero-copy serialization and since it has been shown to outperform Protobuf in certain situations [10], [25].

Protobuf: Protocol buffers, also called Protobuf, is a binary serialization format developed and used by Google. From version 2, this protocol is open-source and available to the public. It is described as *a language-neutral, platform-neutral, extensible way of serializing structured data* [16]. It works by using schemas to define the structure of the data that is to be sent. These schemas are then *compiled* into code in a specified programming language, which is then used to serialize and deserialize the defined data structure.

Flatbuffers: Like Protobuf, Flatbuffers is developed at Google [26] and uses a schema. (Protobuf schemas can even be converted to Flatbuffers schemas, using the command line tool `flatc`.) The Flatbuffers format emphasizes a zero-copy methodology based on the principle that the serialized data should be structured the same way as in working memory. In particular, the structure of the serialized data is pointer-based, with every field in the data having a specific offset in memory from the start of the buffer [27]. This removes the need for expensive encoding and decoding steps, allows direct reads of values contained inside a serialized data structure (without need to deserialize), and speeds up deserialization.

Other formats: There are also other zero-copy formats. Among these, Cap'n Proto (same developers as Protobuf) provides an attractive alternative, as it has shown to perform similarly to Flatbuffers [26], with a slight speed advantage for Cap'n Proto and slight message size advantage for Flatbuffers [28]. Here, we selected to use only Flatbuffers as it (in contrast to Cap'n Proto) allows us to use optional values and since the comparison still provides insights into the general tradeoffs between these zero-copy approaches and Protobuf.

Others have suggested that schema-less formats such as MessagePack can beat Protobuf in some instances (e.g., [17]).

However, while MessagePack allows for smaller serialized message size [24], Protobuf typically has faster serialization/deserialization speed [24]. Here, we selected to exclude schema-less formats for qualitative reasons, as they lack some of the flexibility of the other formats and since we found them much less practical for the vehicular production use cases targeted here (e.g., where the selected data may change substantially over time and between use cases). For example, since the developers themselves must optimize the layout of the serialized data in memory, updates are costly.

B. Messaging Protocols

Use-case driven protocol selection: With the exception of mobile aspects, the communication patterns to-and-from vehicles are often similar to the communication patterns seen in most other IoT applications and sensor networks [11]. Here, we consider two use cases: *large chunks* and *small chunks*. For the *large chunk* case, we primarily focus on IoV applications that need reliable transfer, but for which latency requirements are less strict. For this scenario, the TCP based protocols MQTT and AMQP were selected. MQTT is the more lightweight candidate, as it has low data overhead with proven performance and widespread use in IoT applications. AMQP was chosen for its richness of features, proven performance, and corporate adoption. Both protocols adopt a broker-based message-oriented middleware approach (pre. v1.0 for AMQP). For the *small chunk* case, we focus on on application scenarios with tighter latency demands, but without expectation of reliable transfer. Here, the UDP based CoAP protocol was selected. We next briefly discuss each protocol.

MQTT: MQTT [29] is designed for constrained environments with low bandwidth and uses a small code footprint. MQTT uses a publish/subscribe framework, in which a client can subscribe to a *topic* and receive notifications via a server whenever there is a new message published on that topic. An MQTT server acts as *message broker* between publishers and subscribers. MQTT uses three levels of message transmission reliability, each representing a different level of Quality of Service (QoS): at-most-once (QoS=0), at-least-once (QoS=1) and exactly-once (QoS=2). With QoS=0, messages are simply sent once and are not acknowledged. With QoS=1, acknowledgements are used and messages are re-sent if no acknowledgement is received before timeout. Finally, with QoS=2, a four-way handshake is used to guarantee that a message arrives exactly once.

AMQP: AMQP began its life at the investment bank J.P. Morgan Chase [30] and was developed to handle the high demands of stock trading systems on the New York Stock Exchange. Depending on the version, AMQP either specifies a broker architecture or not. The version studied here is 0-9-1. This is the most used version and is supported by most message brokers. Beyond using TCP, AMQP 0-9-1 uses no application-layer reliability measures (in terms of acknowledgements) to ensure reliable message transfer from the sender of a message to a broker. However, this functionality has been added to different brokers as *Protocol Extensions* [31].

CoAP: CoAP [32] implements the Representational State Transfer (REST) architecture in machine-to-machine (M2M) communication and communicates in a request-response manner. Like HTTP, CoAP works with URIs, resources, and supports resource discovery on a server. CoAP implements its own reliability models on-top of UDP that either use *confirmable* or *non-confirmable* messages. Confirmable messages are re-sent if an acknowledgement does not arrive within a timeout window. In contrast, non-confirmable messages do not require acknowledgements. In addition to the request-response approach, CoAP can also work in a publish/subscribe manner by using *observable resources* [33]. Using an extended GET-request, a client can tell a server that it wants a message whenever the resource is updated. A resource can have multiple observers and a list of all observers is kept in the server.

III. Related Work

Data serialization: Recent work has compared different serialization formats. For example, Hamerski et al. [17] compared the performance of binary serialization formats in a highly resource restricted, embedded environment (with only a few kilobytes of memory). Petersen et al. [24] evaluated serialization formats in the context of smart electrical grids, with tests made on a Beaglebone Black (a small standalone computer with similar performance as a Raspberry Pi). Despite both studies including MessagePack and Protobuf, they came to different conclusions. For example, in the context of Hamerski et al. [17] study, MessagePack provides the largest size reduction and best serialization speeds, whereas in the context of Petersen et al. [24] Protobuf performed best. Neither of these studies evaluate Flatbuffers or any other zero-copy serialization format in comparison to more established formats, like Protobuf. Hamerski et al. [17] were not able to use the official Flatbuffers library due to it being too verbose for their extremely resource restricted environment, and Petersen et al. [24] instead investigated a plethora of different serialization formats. However, none of these were zero-copy formats. Finally, we also note that among the many studies comparing different formats using a similar methodology [8], [17], [20], [22], [24], only Petersen et al. [24] and Popić et al. [22] (similar to us) evaluate the performance using real messages (in our case CurrentStatus messages from Scania vehicles).

Messaging protocols: Various studies have qualitatively [13] or quantitatively [34]–[37] evaluated and compared the performance of different messaging protocols [4], [11], [12]. For example, Naik [13] presents a really nice overview of the key tradeoffs between CoAP, MQTT, AMQP, and HTTP. At a high level, it is argued that this ordering of the protocols (i.e., CoAP, MQTT, AMQP, and HTTP) corresponds to going (i) from smallest to largest message size, (ii) from smallest to largest message overhead, (iii) smallest to largest power consumption, (iv) smallest to largest resource requirements, (v) smallest to largest bandwidth consumption, and (vi) smallest to largest latency. For other metrics (e.g., reliability, interoperability, security, provisioning, M2M/IoT usage, and standardization) alternative orderings are provided. For example, from lowest to highest, Naik suggests the following reliability ordering: HTTP, CoAP, AMQP, and MQTT (and the reverse ordering for interoperability.)

Early work on CoAP studied the interoperability of specific implementations [38], [39], compared CoAP performance with HTTP [40], or evaluated its performance on different hardware architectures [41]. Thangavel et al. [36] developed middleware that can accommodate both CoAP and MQTT for the context of sensor networks using the publish/subscribe paradigm, and used it to provide early comparisons of the their performance.

Chen et al. [34] evaluate the performance under constrained conditions with poor network connections (e.g., low bandwidth, high latency, and/or high packet loss rate) and found advantages of TCP-based protocols (e.g, MQTT). Gundoğan et al. [37] compared the performance of CoAP and MQTT with that of named data networking (NDN) [42].

None of the above works consider mobile scenarios. Perhaps closest to ours in this regard, is the work by Luzuriaga et al. [35], which compares protocol performance in the context of access point migration, where a mobile user changes from WiFi access point. However, they also do not capture the end-to-end performance that a vehicle would experience in an IoV scenario. In contrast, we use traces to emulate the network conditions that such a vehicle may experience.

IV. Method: Evaluation Platform

Scania uses a message type called CurrentStatus to transfer sensor readings from the vehicle to the cloud. For testing the performance of the different serialization formats, this message type was used. The template for this message type is written and optimized for Protobuf, and therefore a challenge was rewriting the schema to optimize for Flatbuffers.

Default messages: For the serialization experiments, we use 5,739 procured CurrentStatus messages from Scania. These messages are in JSON format and provide a representation of unserialized data. Figure 1 shows the size distribution of these messages. We call these *large chunks*.

For the protocol experiments with *large chunks*, we used random CurrentStatus messages serialized with Protobuf (which provided the most compression of the serialized formats tested). For the protocol experiments with *small chunks*, we used 10 byte messages with the contents `message001` through `message100`. These two types represent two ends of the current spectrum. We also ran complementing experiments with intermediate messages sizes.

Serialization: The JSON files of the CurrentStatus messages were loaded into the serialization program and the content was serialized into Protobuf or Flatbuffers format. To measure the memory usage for serialization and deserialization, the function `getrusage` was used to get the peak *resident set size*, which is the amount of memory allocated for the program in RAM. Similar to Petersen et al. [24] we also measured the serialization and deserialization time, serialized size, and report averages as measured 1,000 times per message.

The measurements were made on an instance of Ubuntu 18.04 LTS running on a VirtualBox virtual machine (VM),

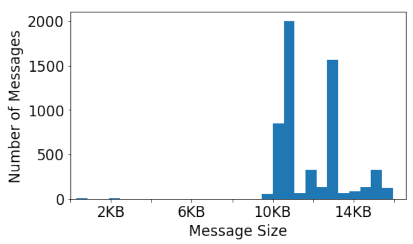


Fig. 1: Distribution of JSON message sizes.

on top of 64-bit Windows 10 build 1803. The hardware used was a laptop with an 8th gen, six core Intel Core i7 CPU running at 2.6 GHz, with 16GB RAM. The VM was allocated six physical cores and 8GB RAM. Nanosecond timestamps were extracted before and after every measurement using C++. For the serialization, we used the official C++ libraries for Flatbuffers (1.10.0), Protobuf (3.7.1), and JsonCpp (1.8.4). gcc (7.4.0) was used for compilation.

Messaging protocols: In our experiments, we place the clients on a Raspberry Pi (3B running Raspbian 9) and the server on the same VM and laptop as the serialization tests above. The machines were connected via Ethernet and messages are sent from the publishing/sending client on the Pi, to the broker/server on the laptop, and then back to the Pi to finally arrive at the consuming/receiving client. To capture the network communication we run Wireshark on the Pi and to emulate the network conditions we run NetEm on the laptop. Table I summarizes the different software that were used together with the testing of each protocol. The client and server were implemented in Java and the libraries used were Eclipse Californium for CoAP, RabbitMQ for AMQP, and for MQTT Eclipse Paho was used for client and server and VerneMQ for the broker. The versions used for the RabbitMQ and VerneMQ server software were 3.7.12 and 1.8.0, respectively.

When evaluating AMQP and MQTT, we let one client subscribe to the same topic that the other client is publishing to. To ensure that we could separate the exchanges of different messages, serialized messages sent were separated by ten seconds. We used a similar set up when evaluating CoAP (but with had to use a larger time threshold when using confirmable messages under large loss rate scenarios). Here, the server (on the laptop) used an observable resource that was being observed by one of the clients, and the other client sent messages in POST requests to update this resource. These updates were then sent to the first client.

Network conditions: Network conditions for the link between clients and server were emulated using NetEm and the Linux Traffic Control tool (`tc`) on the laptop. For the bandwidth variations, we used LTE upload throughput traces by Raca et al. [6], recorded while driving a car. (See example trace in Figure 2.) We used RTT measurements by Huang et al. [3], from a LTE network, to generate 100,000 random RTT samples from a fitted Gaussian distribution (μ=69.5 ms, σ=5.6 ms). The samples were then used together with the throughput traces to update the parameters of NetEm on a per-second granularity. This allowed us to repeat the exact same network conditions for each of the protocols, QoS levels, and

TABLE I: Software used for the networking experiments.

	Raspberry Pi		Laptop
Protocol	Publishing/sending	Subscribing/receiving	Server/broker
AMQP	RabbitMQ Publishing Client	RabbitMQ Subscribing Client	RabbitMQ Broker
MQTT	Eclipse Paho Publishing Client	Eclipse Paho Subscribing Client	VerneMQ Broker
CoAP	Eclipse Californium Posting Client	Eclipse Californium Observing Client	Eclipse Californium Server

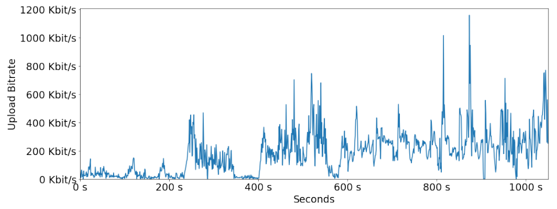


Fig. 2: Upload bandwidth trace used for the protocol tests.

messages. Finally, to test reliability, different packet loss rates were applied to the link: 0%, 1%, 5%, 10%, and 25%.

Limitations: While RabbitMQ runs on all common OS's, VerneMQ, only runs on Linux and MacOS. To ensure fair head-to-head comparisons, a computer running one of these OS:es is needed to run the server, in the way it is described here. Other broker software may be used for AMQP or MQTT. However, the performance of such brokers is not guaranteed to be the same as the ones used in our experiment.

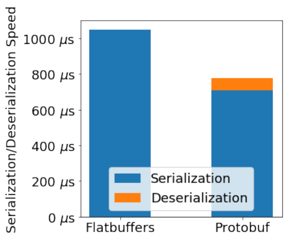
Here, we only consider non-encrypted traffic. Interesting future work could compare the impact of implementing encrypted versions of the messaging protocols or protocols aimed for secure communication [43].

The chosen throughput traces provide example V2I connections. However, measurements have shown that bandwidth conditions can vary significantly from location-to-location as well as within a location [44]. Furthermore, throughput, latency, and packet loss rate are not the only parameters affecting this type of connection and the example parameter values selected do not capture all possible scenarios that a V2I connection may experience. The use of throughput traces and representative RTTs are instead intended to provide an example comparison under relatively normal conditions. Finally, we note that the RTTs in practice would be related to the throughput and the distance to the cell tower, for example. Here, we simply assume that RTTs are independent, and draw random independent and identically distributed (i.i.d) samples.

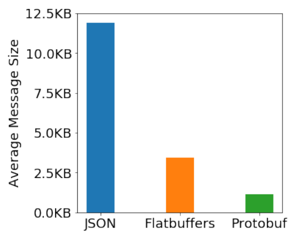
V. Performance Results: Serialization Formats

Speed: Figure 3a shows the measured speed of serialization and deserialization. For Flatbuffers, the serialization for one message took on average 1,048 μs and deserialization took only 0.09 μs. For Protobuf, these values were 708 μs and 69 μs, respectively. The serialization and deserialization times of Protobuf with the *optimize for speed* option set (results omitted from figure) were close to identical to the default settings (shown in Figure 3a): 702 μs and 69 μs, respectively. These results highlight that Protobuf is faster for serialization (the dominating time cost) and Flatbuffers is much faster at deserialization (time cost at receiver side).

Sizes: Figure 3b shows the message sizes before serialization (original JSON format) and after serialization when using Protobuf and Flatbuffers. Protobuf is the clear winner

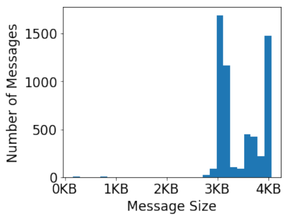


(a) Serialization speed.

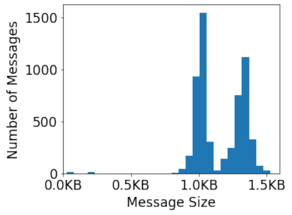


(b) Serialized message size.

Fig. 3: Serialization/deserialization speeds and serialized message sizes; averaged over all messages and 1000 runs/message.



(a) Flatbuffers.



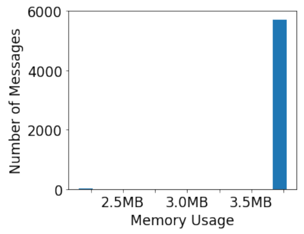
(b) Protobuf.

Fig. 4: Distributions of serialized message sizes.

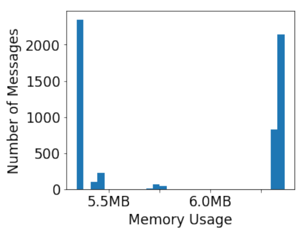
and managed to shrink the average message size to 1,157 bytes. This is more than a factor ten smaller compared to the original JSON size (average of 12,187 bytes) and a factor three smaller than the average serialized message size when using Flatbuffers (3,536 bytes). Upon further inspection, the messages serialized with Flatbuffers contained many zeros.

Figure 4 shows the distribution of serialized message sizes when using Flatbuffers and Protobuf. These distributions mimic the general shape of the original JSON message size distribution (Figure 1), including two peaks around the average message size and some smaller outliers with smaller sizes.

Memory usage: Figures 5 and 6 show the memory usage for the programs used to serialize and deserialize the messages, respectively. We note that (i) these measures include the memory taken by all libraries and data structures used by the programs, and (ii) both the program used for Flatbuffers and Protobuf mostly use the same libraries. The Flatbuffers program uses JsonCpp, the official Flatbuffers C++ library and the official Protobuf library (for parsing of RFC 3339 timestamp strings). The Protobuf program uses all these libraries with the exception of the Flatbuffers library. Both programs also had to load the JSON data structure containing the unserialized messages. For these reasons, the reported memory usage results should be seen as a relative comparison of the two processes rather than as exact measures of their individual memory footprints. In general, the average memory consumption for Flatbuffers is lower than for Protobuf. For example, the average memory usage during serialization were 3.76 MB and 5.87 MB, respectively, and the corresponding numbers for deserialization were 2.04 MB and 5.28 MB.

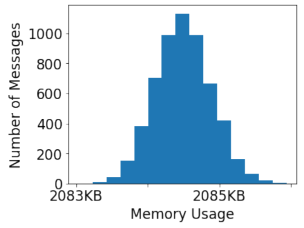


(a) Flatbuffers.

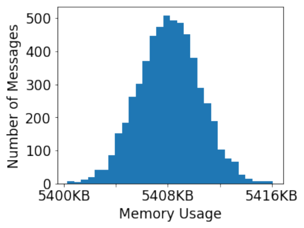


(b) Protobuf.

Fig. 5: Distributions of serialization memory usage, averaged over 1000 serializations for each message.



(a) Flatbuffers.



(b) Protobuf.

Fig. 6: Distributions of deserialization memory usage, averaged over 1000 serializations for each message.

As with all compression, serialization requires more resources to achieve a better compression rate. This is evident in the case of Protobuf, in comparison to Flatbuffers. For example, the average memory footprints (reported above) suggest that Protobuf uses more than 50% more memory for serialization than for Flatbuffers and 150% more for deserialization. As these numbers also include the consumed memory for all used (and mostly shared) libraries, the 50% and 150% differences in memory consumption are only lower bounds. In memory critical situations, this is an important consideration to make, especially since the message sizes to be serialized/deserialized would grow as more data is sent from/to vehicles. The above results therefore show an important tradeoff between serialization speed (Protobuf winner), message size (Protobuf winner), and memory usage (Flatbuffer winner).

Impact of message size on the memory usage: First, for larger message sizes, the memory usage for Protobuf serialization depends more on the message sizes than the memory usage of Flatbuffers. For example, note that the memory usage distribution for Protobuf serialization (Figure 5b) has a similar bimodular distribution as the size distribution of the unserialized messages (Figure 1) and the serialized Protobuf messages (Figure 4b). In contrast, Flatbuffers does not show this same correlation, with over 5,700 out of the 5,739 messages residing in the dominant right bin in Figure 5a.

Second, Flatbuffers is better able to reduce its footprint when serving small messages. For example, see the small peak at 2.2 MB in Figure 5a (which captures the smaller outliers). In contrast, Protobuf never reduces its memory usage below 5,400 KB (Figure 5b). This floor is mirrored in its deserialization memory usage (Figure 6) and suggests that Protobuf may not be able to reduce its memory footprint when there are large size variations or there are many smaller messages sent.

Finally, the memory usages for deserialization (Figure 6) do not seem to depend on the message sizes. For both formats, the memory usage during deserialization appears to be Gaussian, with a relatively low variance.

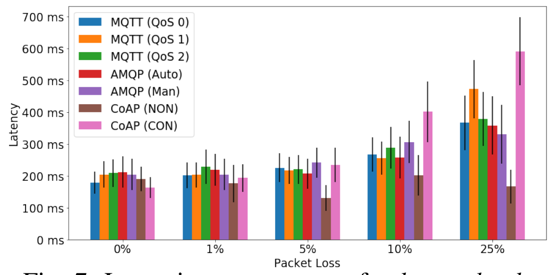


Fig. 7: Latencies per message for *large chunks*.

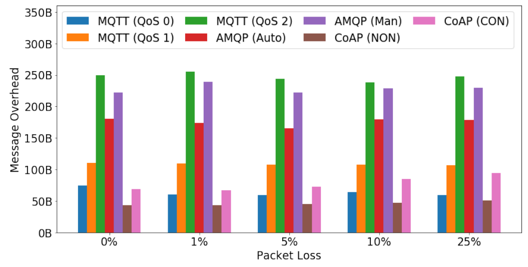


Fig. 8: Message overhead for *large chunks*.

Discussion: Protobuf achieves greater size reductions than Flatbuffers. This provides Protobuf with a big advantage in end-to-end communication scenarios, as both latencies and bandwidth usage are directly related to the message size. Most of this size difference can be explained by the padding generally needed in zero-copy serialization formats [26], for the fast lookup times that signify this type of serialization format. While these features provide Flatbuffers with the opportunities to do quick lookups on the receiver side (e.g., in the cloud in the typical digital twin scenario, on vehicles in a platoon [45], or on an individual vehicle in the case up-to-date HD maps that are distributed to enhance self-driving capabilities of a vehicle) it results in higher processing and serialization times on the sender side (e.g., a vehicle, the lead vehicle, or the cloud for the three cases above). Based on these observations, we argue that Protobuf typically is the desirable format for the main digital twin scenarios considered here, but see Flatbuffers as a good candidate for cloud-to-vehicle communication where the vehicle needs quick access to the data (e.g., as in HD map scenarios where the maps may be updated in real-time so to provide traffic data and other information that may benefit the vehicle).

VI. Performance Results: Protocol Comparison

A. *Large Chunks Case*

Latency: Figure 7 shows the average latencies (with two-sided 95% confidence intervals using student-t) for the different protocols and QoS settings, at different packet loss rates. We note that CoAP generally has the lowest latency when using non-confirmable packets, especially for higher packet loss rates, as it consistently achieves latencies at around 150-200 ms. This is perhaps not surprising, since CoAP with this configuration has no mechanism to handle packet loss.

Message overhead: Figure 8 shows the average overhead per message, measured in additional bytes *on the wire* for

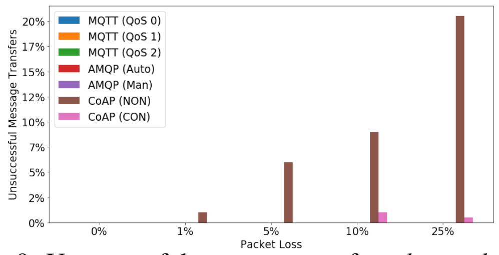


Fig. 9: Unsuccessful message transfers, *large chunks*.

the application layer packets. This metric was calculated by summing up the sizes of all link-layer frames associated with a message, and then subtracting the size of the message. The sizes of packets that did not refer to a successful message exchange, such as heartbeat packets and failed transfers, were distributed equally across the individual messages' overhead.

As expected, the protocols that provide higher reliability and QoS have higher overhead. For example, MQTT (QoS 2) is the configuration with the highest overhead (approximately 250 bytes per message), closely followed by AMQP using manual acks. These configurations have an average overhead of 50-100 bytes higher per message than the closest configuration AMQP (Auto). Both AMQP configurations have a large overhead compared to the MQTT configurations that use equivalent reliability levels (i.e., with QoS 0 and QoS 1).

At the other end of the spectrum, CoAP using non-confirmable messages (NON) consistently has the lowest overheads, followed by MQTT set to QoS 0 and CoAP using confirmable messages (CON). The consistently higher overhead with CoAP using confirmable messages (CON) than using non-confirmable messages (NON) is partially due to all transferred packets with this protocol option receiving an acknowledgement packet. We also note that both CoAP using confirmable messages and MQTT (QoS 1), which have very comparable overheads to each other for lower packet loss rates, use at-least-once delivery mechanisms.

The slightly bigger correlation between message overhead and packet loss rate at the higher loss rates seen for CoAP using confirmable messages is likely due to the extra messaging done at the application layer to transfer all messages also at larger loss rates. For most other protocols the (measured) overheads are relatively insensitive to the packet loss rates. While we have observed many retransmissions at the transport layer for all TCP-based protocols (when operating at high loss rates), these retransmissions are not captured by the application-level overhead metric. It is therefore not surprising that we only observe a slight correlation between overheads and packet loss rates for CoAP using confirmable messages.

Unsuccessful message transfers: Figure 9 shows the percentage of unsuccessful message transfers. As expected, all protocols using TCP successfully deliver all packets. In contrast, the UDP-based CoAP protocol sees unsuccessful message exchanges even when using confirmable messages. While it is expected that the percentages of unsuccessfull messages are similar to the packet loss rates when using non-confirmable messages, it is interesting that we also see non-negligible percentages when using confirmable messages.

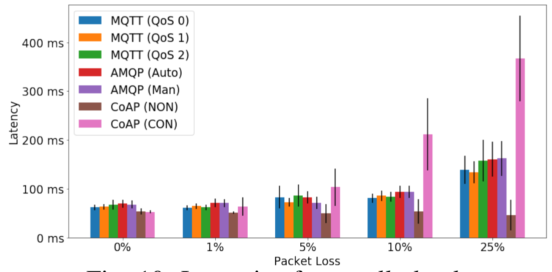


Fig. 10: Latencies for *small chunks*.

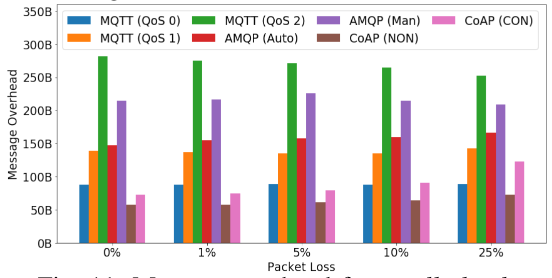


Fig. 11: Message overhead for *small chunks*.

Discussion: Since even CoAP with confirmable messages results in non-delivered messages, only the TCP-based protocols can be considered reasonable candidates for the *large chunks* scenario. Of the TCP-based protocols, MQTT (QoS 0) has by far the lowest overhead. MQTT (QoS 1-2) and AMQP have added reliability but comes at much higher overhead per message. The extra overhead of MQTT (QoS 1-2) comes from additional control messages. In contrast, the extra overhead associated with AMQP comes exclusively from use of larger packet headers. Of course, in the case that the reliability offered by TCP (which is all that MQTT (QoS 0) and AMQP (Auto) use) is not enough, MQTT (QoS 1-2) and AMQP using manual acks are the only remaining options. Considering the excessive overhead of MQTT (QoS 2) and AMQP using manual acks, and the fact that even MQTT (QoS 0) managed to transfer all messages successfully, we believe that MQTT (QoS 0) should be enough in this case.

B. *Small Chunks Case*

Figure 10 shows the latency results for the *small chunks* experiments. As expected, the delays for the *small chunks* are consistently lower than for the *large chunks* case, and again CoAP using non-confirmable messages is least affected by packet losses, and CoAP using confirmable messages is the most effected by packet loss. However, these performance differences are much more noticeable for the *small chunks* case than for the *large chunks* case. For example, with CoAP using confirmable messages the average latency increases almost seven times when the packet losses increase from 0-to-25%.

In general, the overheads per message are very similar for the *small chunk* case (Figure 11) and *large chunk* case (Figure 8), with a few notable differences. First, the overheads for both AMQP configurations are consistently lower for the *small chunk* case. Second, the overheads for MQTT with QoS 1 and QoS 2 are consistently higher. Nevertheless, for both CoAP versions and MQTT (QoS 0) the results are very similar for the two cases.

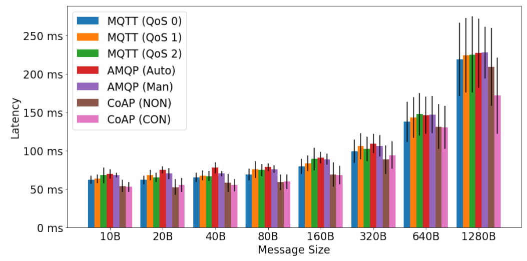


Fig. 12: Average latencies for different message sizes.

Finally, we note that the rate of unsuccessful messages (results not shown) were very similar for the *small chunk* case as for the *large chunk* case, with the only exception that CoAP using confirmable messages only saw unsuccessful message exchanges when there was a 25% packet loss rate (not also for the 10% case). Otherwise, CoAP with non-confirmable messages saw percentages similar to the packet loss rates and the TCP-based protocols succesfully delivered all messages.

C. Cross-case Observations

CoAP with non-confirmable messages consistently has lower message overhead than the other protocols. Furthermore, with the exception of at higher packet loss rates, CoAP using confirmable messages generally has similar latencies to the TCP-based protocols. The spike in delays for higher packet loss rates is especially interesting when compared to MQTT (QoS 0) and AMQP (Auto), that use no application layer reliability mechanisms. This hints that the transport layer reliability mechanisms offered by TCP are much more efficient at keeping latency down, than those offered by CoAP. This is especially true when considering that CoAP, using confirmable messages, did not get all messages through at higher packet loss rates while all TCP based configurations did so, even MQTT (QoS 0) and AMQP (Auto).

The lowest delay can be seen using CoAP with non-confirmable messages. If a low delay is the only important consideration when choosing a configuration, this one will be the best choice. However, if reliability is any consideration, this configuration should be avoided. Looking at TCP based protocols, all configurations keep a respectably good delay which does not seem affected too much by higher packet loss rates. When comparing overheads, CoAP using non-confirmable messages had the lowest, followed by MQTT (QoS 0). This is reasonable considering they do not have any application layer reliability mechanisms. What is surprising is that AMQP (Auto) had a higher overhead than both MQTT (QoS 1) and CoAP, using confirmable messages, even though it did not use any such mechanisms. The difference relative to CoAP using confirmable messages can perhaps be explained by the use of TCP. However, intuitively, from a reliability perspective alone, AMQP with automatic acks may be expected to have similar overhead to MQTT (QoS 0) and AMQP using manual acks similar to MQTT (QoS 1). Instead, the AMQP versions have noticeably higher overheads than their MQTT counterparts. This highlights AMQP's more verbose nature and that the protocols target different scenarios.

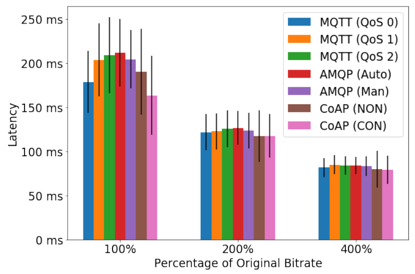


Fig. 13: Average latencies when keeping the loss rate at 0% and scaling the emulated bitrate for *large chunks*.

D. Impact of Message Sizes and Throughput

Figure 12 shows latency results for intermediate packet sizes (with logarithmic x-scale) between the *small chunk* and *large chunk* cases. Note that the relative differences between the protocols remain fairly consistent across the full range.

Figure 13 shows the latencies as a function of the available bitrate, as defined here by the factor that we scaled the bandwidth of the original bandwidth traces before using them for bandwidth emulation in our experiments. These experiments where done with the original *large chunks*, and again the results are shown consistent across bandwidths.

E. Impact of Broker Choices

Finally, we performed an experiment to better understand how much of the latencies are due to the added transmission logic introduced by the used protocol solutions. This experiment was run with the best possible network conditions (i.e., no introduced delays or packet losses, and the maximum possible bandwidth) and the same 10 byte payload used for the *small chunk* experiments. (Since the message payloads are much smaller than the headers processed by the brokers, they should not contribute much to the results.) The latencies from this experiment are shown in Figure 14 and are in the range 4.25-7.47 ms. This can be compared to the average ping time 2.52 ms (σ=1.37), across 200 pings, between the Pi and the laptop under the same conditions. These results suggest that the different protocol solutions on average introduce roughly 2-5 milliseconds of additional processing compared to a ping message. This is substantially less than the typical latencies reported earlier in this section, suggesting that additional improvements to the processing speed of the protocol solutions themselves (e.g., by optimizing brokers for MQTT and AMQP, or sending messages directly with CoAP) likely would not substantially improve the results. Instead, the latencies are mainly due to the factors already considered in the experiments.

VII. Conclusion

This paper evaluates the performance of different serialization formats and application layer messaging protocols in the context of vehicle-to-cloud communication intended for digital twins. For the serialization formats, Protobuf is shown to have three times smaller serialized message size than Flatbuffers and also faster serialization speed. Flatbuffers is the winner in the case of memory use and deserialization time, which could make up for the poor performance in other aspects of data

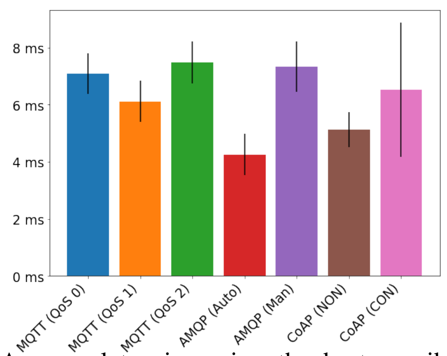


Fig. 14: Average latencies using the best possible network conditions with *small chunks*.

processing in the cloud (or more likely on the client, in the case communication is reversed, as in the case of distribution of HD maps, for example). In the context of the messaging protocols, the main tradeoffs have been found to be between message latency, reliability, and message overheads. For example, CoAP using non-confirmable messages has the lowest latencies, but does not guarantee delivery. In fact, even with confirmable messages CoAP was not able to achieve 100% delivery under high-loss scenarios. In contrast, the TCP-based protocols (MQTT and AMQP) deliver all messages. Of these, AMQP typically had much higher overheads than MQTT. Despite this, and the evident correlation between message size and latency, AMQP achieved comparable latencies as MQTT. However, assuming reliable delivery is expected (as in the *large chunks* case considered earlier in this paper), MQTT with QoS=0 provides the most attractive tradeoff between latency and transmitted data volume. On the other hand, for cases in which reliability is not a concern, CoAP is the best choice.

References

[1] T. S. Darwish and K. Abu Bakar, "Fog based intelligent transportation big data analytics in the internet of vehicles environment: Motivations, architecture, challenges, and critical issues," *IEEE Access*, vol. 6, 2018.

[2] D. Bolton. (2016) What are digital twins and why will they be integral to the internet of things? [Online]. Available: https://www.applause.com/blog/digital-twins-iot-faq/

[3] J. Huang *et al.*, "A close examination of performance and power characteristics of 4G LTE networks," in *Proc. ACM MobiSys*, 2012.

[4] J. Contreras-Castillo *et al.*, "Internet of vehicles: Architecture, protocols, and security," *IEEE Internet of Things Journal*, vol. 5, no. 5, pp. 3701–3709, 10 2018.

[5] E. Soltanmohammadi *et al.*, "A survey of traffic issues in machine-to-machine communications over LTE," *IEEE Internet of Things Journal*, vol. 3, no. 6, pp. 865–884, 12 2016.

[6] D. Raca *et al.*, "Beyond throughput: a 4G LTE dataset with channel and context metrics," in *Proc. ACM MMSys*, 2018.

[7] H. Riiser *et al.*, "Commute path bandwidth traces from 3G networks: analysis and applications," in *Proc. ACM MMSys*, 2013.

[8] A. Sumaray and S. Makki, "A comparison of data serialization formats for optimal efficiency on a mobile platform," in *Proc. ICUIMC*, 2012.

[9] Google. Flatbuffers. [Online]. Available: https://google.github.io/flatbuffers/

[10] K. Khare. (2018) JSON vs Protocol Buffers vs FlatBuffers. [Online]. Available: https://codeburst.io/json-vs-protocol-buffers-vs-flatbuffers-a4247f8bda6f

[11] V. Karagiannis *et al.*, "A survey on application layer protocols for the internet of things," *Trans. on IoT and Cloud Computing*, pp. 1–10, 2015.

[12] A. Čolaković and M. Hadžialić, "Internet of things (iot): A review of enabling technologies, challenges, and open research issues," *Computer Networks*, vol. 144, pp. 17–39, 10 2018.

[13] N. Naik, "Choice of effective messaging protocols for IoT systems: MQTT, CoAP, AMQP and HTTP," in *Proc. IEEE ISSE*, 2017.

[14] F. Yang, S. Wang, J. Li, Z. Liu, and Q. Sun, "An overview of internet of vehicles," *China Communications*, vol. 11, no. 10, pp. 1–15, 10 2014.

[15] O. Kaiwartya *et al.*, "Internet of vehicles: Motivation, layered architecture, network model, challenges, and future aspects," *IEEE Access*, vol. 4, pp. 5356–5373, 9 2016.

[16] Google. Protocol Buffers Developer Guide - Overview. [Online]. Available: https://developers.google.com/protocol-buffers/docs/overview

[17] J. C. Hamerski *et al.*, "Evaluating serialization for a publish-subscribe based middleware for MPSoCs," in *Proc. IEEE ICEC)*, 2018.

[18] G. Kaur and M. M. Fuad, "An evaluation of protocol buffer," in *Proc. IEEE SoutheastCon*, 2010.

[19] N. Gligorić *et al.*, "Performance evaluation of compact binary XML representation for constrained devices," in *Proc. DCOSS*, 2011.

[20] K. Maeda, "Performance evaluation of object serialization libraries in xml, json and binary formats," in *Proc. DICTAP*, 2012.

[21] L. Tuyisenge *et al.*, "Network architectures in internet of vehicles (IoV): Review, protocols analysis, challenges and issues," in *Proc. IOV*, 2018.

[22] S. Popić *et al.*, "Performance evaluation of using protocol buffers in the internet of things communication," in *Proc. SST*, 2016.

[23] ITU, "Abstract syntax notation one (ASN.1): Specification of basic notation," 2015. [Online]. Available: www.itu.int/rec/T-REC-X.680/

[24] B. Petersen *et al.*, "Smart grid serialization comparison: Comparision of serialization for distributed control in the context of the internet of things," in *Proceedings of Computing Conference*, 2018.

[25] Flatbuffers. C++ Benchmarks. [Online]. Available: https://google.github.io/flatbuffers/flatbuffers_benchmarks.html

[26] K. Varda. (2014) Cap'n Proto, FlatBuffers, and SBE. [Online]. Available: https://capnproto.org/news/2014-06-17-capnproto-flatbuffers-sbe.html

[27] Flatbuffers. Flatbuffer Internals. [Online]. Available: https://google.github.io/flatbuffers/flatbuffers_internals.html

[28] K. Sorokin. Benchmark comparing various data serialization libraries (thrift, protobuf etc.) for C++. [Online]. Available: https://github.com/thekvs/cpp-serializers

[29] OASIS. (2014) Mqtt version 3.1.1 documentation. [Online]. Available: http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html

[30] S. Vinoski, "Advanced message queuing protocol," *IEEE Internet Computing*, vol. 10, pp. 87–89, 11 2006.

[31] RabbitMQ. Introducing publisher confirms. [Online]. Available: https://www.rabbitmq.com/blog/2011/02/10/introducing-publisher-confirms/

[32] Z. Shelby, K. Hartke, and C. Bormann, "The constrained application protocol (CoAP)," RFC 7252, IETF, 2014.

[33] K. Hartke, "Observing resources in the constrained application protocol (coap)," RFC 7641, IETF, 2015.

[34] Y. Chen and T. Kunz, "Performance evaluation of iot protocols under a constrained wireless access network," in *Proc. MoWNeT*, 2016.

[35] J. Luzuriaga *et al.*, "A comparative evaluation of AMQP and MQTT protocols over unstable and mobile networks," in *IEEE CCNC*, 2015.

[36] D. Thangavel *et al.*, "Performance evaluation of MQTT and CoAP via a common middleware," in *Proc. IEEE ISSNIP*, 2014.

[37] C. Gündoğan *et al.*, "NDN, CoAP, and MQTT: A comparative measurement study in the IoT," in *Proc. ACM ICN*, 2018.

[38] C. Lerche *et al.*, "Industry adoption of the internet of things: A constrained application protocol survey," in *Proc. IEEE ETFA*, 2012.

[39] B. C. Villaverde *et al.*, "Constrained application protocol for low power embedded networks: A survey," in *Proc. IEEE IMIS*, 2012.

[40] A. Ludovici *et al.*, "TinyCoAP: A novel constrained application protocol (CoAP) implementation for embedding RESTful web services in wireless sensor networks based on TinyOS," *J. Sensor and Actuator Networks*, vol. 2, p. 288–315, 2013.

[41] C. P. Kruger and G. P. Hancke, "Benchmarking internet of things devices," in *Proc. IEEE INDIN*, 2014.

[42] L. Zhang *et al.*, "Named data networking," *ACM SIGCOMM CCR*, no. 44, pp. 66—-73, 2014.

[43] T. Limbasiya and D. Das, "Lightweight secure message broadcasting protocol for vehicle-to-vehicle communication," *IEEE Sys. J.*, vol. 14, pp. 520–529, 2020.

[44] T. Linder *et al.*, "On using crowd-sourced network measurements for performance prediction," in *Proc. IEEE/IFIP WONS*, 2016.

[45] C. Campolo *et al.*, "Better platooning control toward autonomous driving: An lte device-to-device communications strategy that meets ultralow latency requirements," *IEEE Vehicular Technology Magazine*, vol. 12, no. 1, pp. 30–38, 3 2017.