Formal Verification and Performance Analysis of a Data Exchange Protocol for Connected Vehicles

Samir Chouali, Azzedine Boukerche, Ahmed Mostefaoui, Mohammed-Amine Merzoug

Formal Verification and Performance Analysis of a New Data Exchange Protocol for Connected Vehicles

Samir Chouali, Azzedine Boukerche, *Fellow, IEEE*, Ahmed Mostefaoui, and Mohammed Amine Merzoug

***Abstract*—In this article, we focus on the usage of MQTT (Message Queuing Telemetry Transport) within Connected Vehicles (CVs). Indeed, in the original version of MQTT protocol, the broker is responsible "only" for sending received data to subscribers; abstracting then the underlying mechanism of data exchange. However, within CVs context, subscribers (i.e., the processing infrastructure) may be overloaded with irrelevant data, in particular when the requirement is real or near real-time processing. To overcome this issue, we propose MQTT-CV; a new variant of MQTT protocol, in which the broker is able to perform local processing in order to reduce the workload at the infrastructure; i.e., filtering data before sending them. In this article, we first validate formally the correctness of MQTT-CV protocol (i.e., the three components of the proposed protocol are correctly interacting), through the use of Promela language and its system verification tool; the model checker SPIN. Secondly, using real-world data provided by our car manufacturer partner, we have conducted real implementation and experiments. The obtained results show the effectiveness of our approach in term of data workload reduction at the processing infrastructure. The mean improvement, besides the fact that it is dependent of the target application, was in general about 10 times less in comparison to native MQTT protocol.**

***Index Terms*—Connected vehicles, data filtration, formal analysis, formal verification, MQTT, promela, SPIN.**

I. Introduction

NOWADAYS, the Internet of Things (IoT) concept [1] is prevalent in various sectors such as automotive, domotics, health care, etc. This interesting technology connects several smart objects, usually able to collect, process and transmit data (environmental observations), that can be exploited by end applications and services. Because of the diversity of devises as well as the heterogeneity of their related software, the data communication protocol in these applications plays an important since it abstracts the data exchange between all the components.

Currently, the most widely adopted communication protocols in IoT systems are MQTT [2], XMPP [3], and others. In this paper, we focus on MQTT; an application layer protocol that is based on a publish/subscribe messaging model for distributing data between networked applications through a message broker. Given the high level of abstraction it offers, its processing lightness and its implementation easiness, MQTT has been used in many real applications and services including Connected Vehicles (CVs). Indeed, MQTT is one of the protocols used by PSA Group[1] to gather and leverage data from connected vehicles [4]. The authors in [4] assert that PSA Group vehicles can send roughly 170 different types of data ranging from vehicle identification number, GPS coordinates, engine rounds per minute to the current angle of the steering wheel, etc. This amount of data has great value for automotive manufacturers as well as for third parties since it allows the development of several applications and services in different domains (improve driver's safety, enhance mobility experience, personalize insurance costs, etc.).

In PSA experience [4], MQTT is used as a communication protocol that connects, through a broker, vehicles as publishers and PSA automotive infrastructure as a subscriber. Data is then collected from vehicles and processed in both off-line and on-line (real or near real-time) fashions by PSA Big Data infrastructure. As previously mentioned, the usage of MQTT has been motivated by the numerous features it presents in terms of fast communications, processing lightness and easiness of implementation; i.e, integration within a large platform. This last feature is particularly appreciable in industrial environments where software compatibility and rapid deployment are vital requirements.

Nevertheless, in the original version of MQTT, the role of the broker is limited to data forwarding (i.e., transmission) between publishers and subscribers. Hence, using it as it is in the context of CVs will lead to the following problematic issues:

- The number of CVs (publishers) is very huge (expected to be in the order of millions) and hence the infrastructure is supposed to support a very heavy workload since the

Samir Chouali is with the DISC Department, FEMTO-ST Institute/CNRS UMR 6174, University of Bourgogne Franche-Comte, Besançon 25000, France (e-mail: samir.chouali@univ-fcomte.fr).

Azzedine Boukerche is with PARADISE Research Lab., University of Ottawa, Ottawa, Ontario K1N 6N5, Canada (e-mail: boukerch@site.uottawa.ca).

Ahmed Mostefaoui is with the DISC Department, FEMTO-ST Institute/CNRS UMR 6174, University of Bourgogne Franche-Comte, Besançon 25000, France, and also with PARADISE Research Lab., University of Ottawa, Ottawa, Ontario K1N 6N5, Canada (e-mail: ahmed.mostefaoui@univ-fcomte.fr).

Mohammed Amine Merzoug is with the Computer Science Department, Faculty of Mathematics and Computer Science, University of Batna, Algeria, and also with DISC Department, FEMTO-ST Institute/CNRS UMR 6174, University of Bourgogne Franche-Comte, Besançon 25000, France (e-mail: amine.merzoug@gmail.com).

[1] PSA Group (Peugeot-Citroen) is the second-largest automobile manufacturer in Europe with about 3 million sold vehicles in 2015.

brokers are responsible for sending data without any processing or filtering. The processing task is then located at the infrastructure layer. Nevertheless, several applications are interested only in a particular part of the sent data and do not require it in its entirety. In some other cases, only data that is greater or less than a certain threshold is of interest. For instance, an after-sales application could be interested to track vehicles that have their engine's temperature exceeding a certain value. In this case, as all data is processed at the infrastructure layer, the latter has to support a huge workload.

- The interaction system, which is constituted of CVs, a broker, and the infrastructure, must be reliable because it involves very sensitive applications (e.g., emergency applications, drivers' safety applications, etc.). Hence, it is mandatory, in this context, to formally guarantee the correctness of any protocol responsible of connecting CVs to the infrastructure.[2]

In order to reduce the infrastructure's workload, two ways are possible: (a) the first one is filtering unnecessary data at the sources; i.e., at the vehicles. So, only *"valuable"* data are sent to the infrastructure. This supposes that an extra-software, controlled from the infrastructure, has to run on the vehicle in order to perform such processing. From real experience (i.e., Group PSA), this approach has not been considered for security reasons since vehicle are considered as sensitive hosts and hence the number as well as the complexity of running software have to be strictly controlled and reduced only to necessary ones. (b) The second way is to perform filtering at the intermediate layer (i.e., the broker). However, this was not possible in the original version of MQTT protocol.

In this paper, we focus on a new variant of the MQTT protocol, we named MQTT-CV (MQTT for Connected Vehicles). In our proposal, parts of the data processing are handled by the broker layer, leading thus (as proven by the real experiments we conducted) to a significant reduction in the workload that was addressed for the infrastructure. In fact, the infrastructure can define some conditions (mainly filtering) on the received data from the broker. In other words, the broker will send to the infrastructure only data that satisfies the defined conditions. By doing so, the workload of the infrastructure can be substantially reduced at the expense of a negligible processing cost at the broker, as stated by our experiments. This infrastructure workload reduction can have a significant impact on the overall performance of the system because of the huge number of CVs.

Furthermore, because of the sensitivity of several automotive applications, we provide in this paper formal proofs of the correctness of our proposed protocol (i.e., the three components will behave as they are supposed to do). To this end, we used components formal validation based on Promela language [6] and the model checker SPIN [7], [8].

[2]It is worthwhile to notice that some vulnerabilities have been identified in the original MQTT protocol [5]

To the best of our knowledge, this is the first research work addressing both formal verification and performance improvement of a data exchange protocol specifically tailored to CVs infrastructures.

The rest of the paper is organized, as follows: works related to our proposition are presented in Section II. In Section III, we succinctly present the MQTT protocol and provide a brief introduction to the model checker SPIN and Promela language. Section IV describes the proposed MQTT-CV protocol. Our formal analyzing approach of MQTT-CV is presented in Section V. Section VI presents and comments on the obtained experimental results. Finally, Section VII concludes the paper and gives some directions for future works.

II. Related work

Besides the increasing number of data exchange protocols proposed for IoT applications, only few of them have been adopted in the context of CVs [4]. Furthermore, to the extent of our knowledge, only few research works have addressed formal verification of such protocols. The work presented in [9] for instance proposes to formally model the publish/subscribe protocols to specify their essential properties such as minimality and completeness. This work, however, has not considered the verification aspect. In [10], the authors have proposed a formal model, based on Petri nets, to specify the publish/subscribe protocols in the domain of Grid computation. In [11], Zigbee (which is widely used in IoT) has been formally modeled and verified using the Event-B formal method.

Concerning the security properties, the authors in [12] have presented a general discussion on the security issues/requirements of the publish/subscribe protocols in the field of Internet-based peer-to-peer systems. In the same context, in [5], the author has proposed to analyze the MQTT protocol using a formal approach based on timed-message passing process algebra. This approach, which focuses on verifying the security properties related to the protocol vulnerability against attackers, demonstrates that there are some scenarios in which MQTT fails to fulfill the QoS requirements. Some other performance evaluation methods, that assess the MQTT protocol with regards to its different QoS levels, were also proposed in [13], [14].

Several works which are close to our proposition were also proposed. In more exact words, these works have adopted a model checking technique to verify the reliability properties of the publish/subscribe systems. For instance, the solutions proposed in [15], [16] define a general framework that aims to verify the publish/subscribe systems by model checking. The main difference between these approaches and ours lies in the fact that while these solutions are general, our proposition focuses on MQTT-CV analysis and safety properties verification. In [17], the authors utilized a probabilistic model checking to model and validate the publish/subscribe systems. The validation in this work was carried out using the PRISM model checker. Another work which uses a probabilistic model checking was proposed in [18]. This work allows analyzing the quality of prediction in service-oriented architectures.

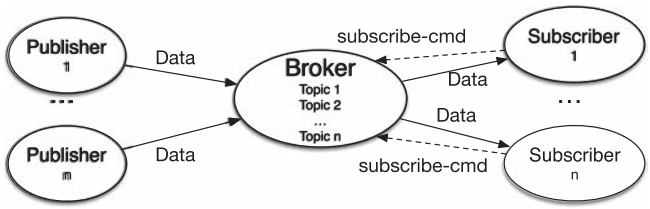

Fig. 1. MQTT: publishers send data to the broker which is in charge of forwarding it to subscribers on given topics.

To summarize, we can say that compared with state-of-the-art solutions, the originality and contribution of the work presented in this paper are (1) the proposition of MQTT-CV (a publish/subscribe protocol dedicated to connected vehicles), (2) its specification with Promela language, and (3) its analysis/verification with the SPIN model checker. Additionally, real implementation and experimentation have been conducted to demonstrate the effectiveness of our approach in terms of noticeable performance improvement (i.e., workload reduction) in the context of automotive big-data infrastructures.

III. Background: MQTT, SPIN and Promela

For the easiness of presentation, we start by introducing the original MQTT protocol as well as the model checker SPIN and its related language Promela.

A. MQTT Protocol

MQTT [2] is a publish-subscribe protocol designed to be open, simple, lightweight, and easy to implement. Moreover, MQTT is a machine-to-machine protocol designed to allow devices with small storage and processing power to communicate with each other over low-bandwidth and eventually unreliable networks (with a high abstraction regarding the underlying network functions). Before receiving data from other devices, a subscriber subscribes to a given topic at the broker by the mean of a subscription command. Then, at each time a data is published on that topic, it will be immediately forwarded to all subscribers. Similarly, a publisher can publish data by the mean of a publish command. Fig. 1 summarizes the MQTT communication paradigm.

In practice, MQTT gives the flexibility to connect multiple publishers to multiple subscribers via a main central entity called Broker. The number of connected devices to a given MQTT broker depends on the intrinsic capacity, in terms of computing power and network bandwidth, of the underlying platform which runs it. In the context of connected vehicles, millions of cars are expected to be connected. MQTT also handles the quality of service for the delivered messages. In fact, it offers three levels of QoS. The first level (QoS 0, called *at most once*) represents the case where the sender issues a message only once and does not wait for any acknowledgment. The second level (QoS 1, called *at least once*) guarantees the delivery of messages at least once by seeking acknowledgment for every sent message (a message can be sent/received multiple times). The third and last level (QoS 2, called *exactly once*) guarantees that each message is delivered only once to the recipient(s) in question.

B. Spin Model Checker and Promela Language

SPIN is one of the world's most popular, and arguably one of the world's most powerful, tools for detecting software defects in concurrent system designs [8]. More specifically, SPIN is an open-source tool that has been developed at Bell Labs by Gerard Holzmann, and since has been applied, so to speak, to everything; from the verification of complex call processing software (used in telephone exchanges) to the validation of intricate control software for interplanetary spacecraft [8].

In SPIN, a formal specification is built using Promela; an imperative language close to C programming language (variables declaration, data types, etc.). A Promela program is composed of a set of processes that are defined with the statement `proctype`. To perform system analysis and verification, SPIN transforms each process into an automaton. Promela supports nondeterminism and parallelism through:

1) The selection statement which describes a nondeterministic choice among those guarded conditions prefixed by '`::`'.
   `if :: sequence[::sequence]* fi`
2) The predefined operator; i.e., `run`, used to create a new process, which will be asynchronously executed with the currently active ones.
   `run` process_name ([argument list]) `;`

In Promela, repetitive instructions are expressed using a `do`-statement; `do :: sequence[::sequence]* od`, which is an `if`-statement caught in a cycle. Promela also allows describing communication between processes via an explicit message passing channel. Both synchronous and asynchronous communications are supported. In the former (i.e., synchronous communication), the channel works in a rendezvous mode with a zero capacity. Whereas, in the latter (i.e., asynchronous communication), the channel works as a FIFO buffer with a non-zero capacity. The send/receive operations are, respectively, denoted by:

- `name ! arguments` which sends messages to channel specified by `name`.
- `name ? arguments` which receives messages from channel specified by `name`.

Finally, as regards atomic (indivisible) sequences, they can be expressed using the `atomic {...}` or `d_step {...}` statements. More details about Promela grammar can be found in [6].

IV. Our Proposal: MQTT-CV

In the context of connected vehicles (CVs), the publish-subscribe paradigm is composed of vehicles (publishers), automotive infrastructures (subscribers), and a broker. In general, depending on the target application, infrastructures do not need

to process all the received data but only a sub-set of it. For instance, in a traffic congestion detection application, the infrastructures will be interested only in the positions of vehicles having their corresponding speed below 30 km/h. However, the current version of the MQTT broker does not support any filtering and consequently will forward all the received data to the infrastructures.

To overcome this issue which has a deep impact on the overall performance of automotive infrastructures, we propose MQTT-CV; an MQTT variant for connected vehicles. The key idea behind our proposal is to allow the broker performing (i.e., processing) some constraints/filtering tasks defined by the subscribers (i.e., infrastructures) on the received data before forwarding it. By doing so, we expect to reduce the infrastructures workload and hence improving the overall system performance. Our proposition raises however the following issues:

- *Protocol correctness:* given the sensitivity of several CV-applications related to driver's safety, the new proposed protocol must remain correct in the sense that the new introduced tasks in the broker will not alter the general behavior of the three components; i.e., publisher-broker-subscriber. Here, it is a matter of software verification.
- *Computing overhead at the broker:* because of the huge workload faced in CV-applications, any slight processing introduced in the platform hosting the broker must remain below a certain threshold. In other words, the new broker tasks must not slow down the overall system by causing delays on messages sent to the infrastructure.

Concerning the first issue, we have paid particular attention to it since we prove in this paper (through the use of well-known software verification techniques) that our proposal remains correct (see next Section). The second issue has been addressed through real implementation and experimental validation that uses realistic data sets.

MQTT-CV defines the interaction between vehicles, broker, and automotive infrastructures as presented by the three following steps:

1) First, vehicles send data to the broker about a specific topic. For example, a vehicle can publish, on the topic "temperature," its collected ambient temperature.
2) Second, infrastructures register their interests in certain topics available at the broker. Infrastructures can also impose conditions on the data values that must be sent to them according to their subscriptions. For example, an infrastructure can require only temperatures that exceed 40 °C.
3) Finally, before distributing data (received from vehicles) to infrastructures, the broker filters it by applying the conditions defined by infrastructures.

We point out that in this paper, we focus on one message processing within the broker (more precisely message filtering operations). The processing of more complex operations such as messages aggregation, complex request processing and composition, etc. is left for future work because of its complexity. We also mention that whilst our proposal targets specifically CV-applications (big-data context), it can be easily deployed/adopted in other contexts where the broker has to filter unnecessary data to relieve the burden on subscribers.

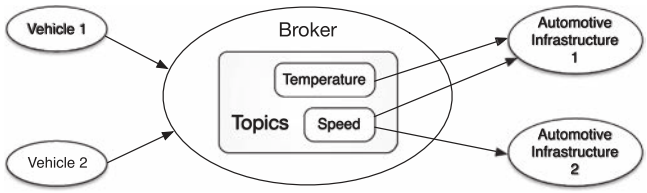

Fig. 2. MQTT-CV illustration.

V. MQTT-CV Correctness

In this section, we provide correctness proofs of our proposal MQTT-CV. To this end, we used the well-know SPIN model checker and its associated Promela language. Generally, the proofs are depicted in four steps: (a) the use of a representative case study, (b) this case study is modeled using UML language and (c) implemented in the Promela language. (d) Finally, formal verification is performed using SPIN. The aforementioned steps are described in the following subsections.

A. Case Study

We use a case study in which there are two different vehicles[3] (vehicle 1 and vehicle 2), two automotive infrastructures (infrastructure 1 and infrastructure 2), and one broker (see Fig. 2). We suppose that the two vehicles send data about two topics related to ambient temperature and vehicle speed, whereas infrastructure 1 has subscribed for both topics (i.e., temperature and vehicle speed), and infrastructure 2 has registered only for the speed topic. We also suppose that infrastructure 2 has not defined any conditions/restrictions on the received data from the broker (condition set to true) and infrastructure 1 has imposed the following conditions on data: (a) temperature must exceed 40 °C and (b) vehicle speed must be less than 30 km/h.

Concretely, the automotive infrastructures in this example might exploit the collected ambient temperatures to analyze temperature evolution in certain geographic zones of interest. Concerning the collected vehicles' speeds, they might be exploited, for example, to decide to broadcast an alert to drivers in the case where the speed of a set of other drivers (in their direction) has suddenly, and considerably decreased. The goal here is to inform the concerned drivers about a possible accident or road obstacle.

B. UML Modeling

Before formally analyzing MQTT-CV, we propose a UML sequence diagram (Fig. 3) that models a scenario of interaction between the main components implementing this protocol; that is, broker, vehicles, and automotive infrastructures. This model

[3]From a verification point of view, the number of vehicles does not matter.

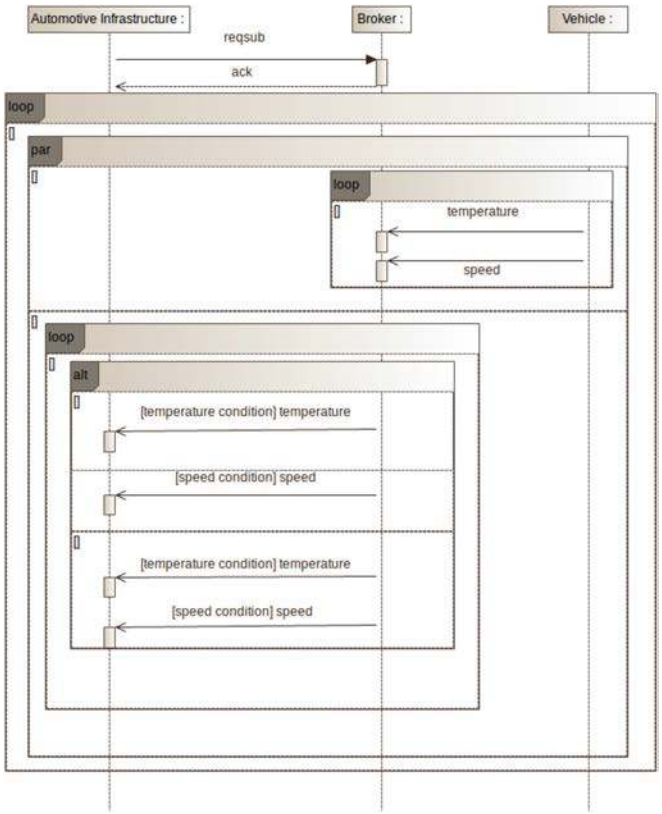

Fig. 3. MQTT-CV modeling with a UML sequence diagram.

will be utilized later in Section V-C to implement MQTT-CV using Promela language.

As Fig. 3 demonstrates each of the three components is modeled with a lifeline. The infrastructure starts interactions by sending a `reqsub` message that allows it to subscribe to a specific topic. The broker responds by sending an acknowledgment `ack`. After that, two interactions occur in parallel (see the UML parallel combined fragment). In the first interaction, the vehicle sends data related to both topics; temperature and vehicle speed. Whilst in the second one, the broker sends the received data while (1) respecting the infrastructure conditions and (2) taking into account its chosen topics. Actually, for this last interaction, the broker considers three possibilities (defined by topics to which the infrastructure has registered): the infrastructure can choose the temperature topic, the speed one, or both.

C. Promela Implementation

To be able to use the model checker SPIN, MQTT-CV must be implemented in Promela language. To do so, we consider the UML protocol specification described in Fig. 3 and the case study presented in Section V-A and Fig. 2. In more specific words, we associate a Promela process (`proctype` statement) to each interacting component (one broker, two vehicles, and two automotive infrastructures). In SPIN, these five processes will be executed in a parallel fashion and will be launched by the `init` process:

```
init{
    atomic{
        run Broker();

        run Vehicle1();
        run Vehicle2();

        run Subscriber1(3);
        run Subscriber2(2);
    }
}
```

The interactions between these concurrent processes can simply be implemented via the communicating channels allowing to send/receive integers. Indeed, this abstraction of data exchange is sufficient to simulate and verify the proposed protocol. The channels we have defined to implement MQTT-CV are:

```
/* Channels declaration */

/* Subscriptions: */
/* (1) from subscribers to broker */
chan chan_reqsub1 = [1] of {int};
chan chan_reqsub2 = [1] of {int};

/* (2) from broker to subscribers */
chan chan_acksub1 = [1] of {int};
chan chan_acksub2 = [1] of {int};

/* Data communication: */
/* from connected vehicles to broker */
chan chan_cvbroker_tmp = [1] of {int};
chan chan_cvbroker_spd = [1] of {int};

/* from broker to subscriber_1 */
chan chan_brokersub1_tmp = [1] of {int};
chan chan_brokersub1_spd = [1] of {int};

/* from broker to subscriber_2 */
chan chan_brokersub2_tmp = [1] of {int};
chan chan_brokersub2_spd = [1] of {int};
```

- `chan_reqsub1` and `chan_reqsub2`: are, respectively, used by the first and second subscribers to request a subscription (from the broker) on one or many topics. The subscriber sends an integer to indicate the desired topics.
- `chan_acksub1` and `chan_acksub2`: used by the broker to, respectively, send an acknowledgment to the first and second subscribers after their subscription requests.
- `chan_cvbroker_tmp` and `chan_cvbroker_spd`: used by connected vehicles to send data about topics defined in the broker. As their names indicate, these two channels are, respectively, dedicated to the temperature and speed topics.
- `chan_brokersub1_tmp` and `chan_brokersub1_spd`: used by the broker to send data (i.e., temperature and speed) to the first subscriber (automotive infrastructure 1).
- `chan_brokersub2_tmp` and `chan_brokersub2_spd`: used by the broker to send data (i.e., temperature and speed) to the second subscriber (infrastructure 2).

In the next subsections, we provide and discuss the Promela code that we propose to implement the subscribers, publishers, and broker.

1) Subscribers Code: The Promela process modeling infrastructure 1 can be implemented using the `proctype` statement, as follows:

```
proctype Subscriber1(int topicsub){
 int tempsub1, spdsub1, respsub1 = 0;

 chan_reqsub1!topicsub;
 chan_acksub1?respsub1;

 (respsub1 == 1) ->
 do
 ::if
   ::topicsub == 1 ->
                  chan_brokersub1_tmp?tempsub1;

   ::topicsub == 2 ->
                  chan_brokersub1_spd?spdsub1;

   ::topicsub == 3 ->
                  chan_brokersub1_tmp?tempsub1;
                  chan_brokersub1_spd?spdsub1;
   fi
 od
}
```

The process parameter `topicsub` determines topics to which this first infrastructure requests to subscribe. More exactly, the value 1 (`topicsub == 1`) means that the subscription will be done for the temperature topic. The value `2` is for the speed one, and `3` is to indicate both topics. The two first internal variables `tempsub1` and `spdsub1` are used to, respectively, receive temperature and vehicle speed from the broker. As a matter of fact, after receiving the expected message from the broker (through `chan_acksub1` channel) and finding that `respsub1` is equal to `1`, `Subscriber1` process will wait for data corresponding to the topic(s) in which it has subscribed. As specified in the UML sequence diagram depicted in Fig. 3, this last step will be executed in a repetitive way using the `do`-statement (second inner-loop of Fig. 3).

Fig. 4 shows the automaton generated by SPIN after executing `proctype Subscriber1(3)`. As indicated above, in this case, the subscription is requested for both the temperature and speed topics (`topicsub == 3`). The generated automaton, demonstrated in Fig. 4, describes the scheduling of infrastructure 1 actions when it interacts with the broker through the corresponding communication channels. These different actions are defined by the transitions labels. More specifically, according to this automaton, infrastructure 1 starts by sending a message that requests a subscription from the broker. After receiving a response from the latter, infrastructure 1 will reach a state in which it will wait for data reception. Note that there are no deadlock states in this automaton.

The Promela code of infrastructure 2 is relatively close to the first one.

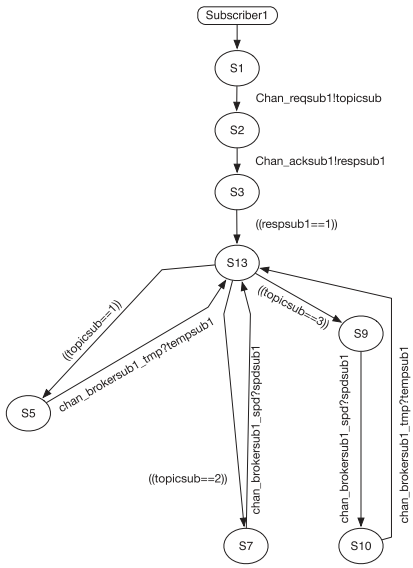

Fig. 4. Automotive infrastructure 1 automaton.

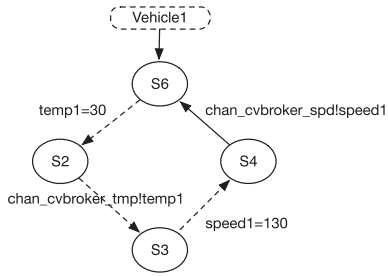

Fig. 5. Vehicle 1 automaton.

2) Publishers Code: The following Promela code shows the implementation of the first considered connected vehicle:

```
proctype Vehicle1(){
 int temp1, speed1;

 do
 ::atomic{
            temp1 = 30;
            chan_cvbroker_tmp!temp1;

            speed1 = 130;
            chan_cvbroker_spd!speed1;
   }
 od
}
```

The above code corresponds to the first inner-loop of Fig. 3. Note that this process defines two internal parameters; `temp1` and `speed1`. These two variables, which represents temperature and speed, are, respectively, initialized to 30 and 130, and will be sent to the broker through their corresponding channels. The scheduling of vehicle 1 actions is described in Fig. 5.

We mention that in the Promela code of the second connected vehicle (which has not been shown in this paper), the temperature has been set to 45 and speed to 20.

3) Broker Code: The Promela code that implements an example of the proposed MQTT-CV broker is presented below.

```
proctype Broker() {
 int temp = 0, speed = 0;
 int topicsub1 = 0, topicsub2 = 0;

 /* Handle subscriptions */
 atomic{
        chan_reqsub1?topicsub1;
        chan_acksub1!1;
 }

 atomic{
        chan_reqsub2?topicsub2;
        chan_acksub2!1;
 }

 do
 ::atomic{/* Receive data from vehicles */
          chan_cvbroker_tmp?temp;
          chan_cvbroker_spd?speed;
   }

   /* Send received data to Subscriber_1 */
   if
   ::topicsub1 == 1 ->
     if
     ::(temp <= 40) -> skip;
     ::(temp > 40) -> chan_brokersub1_tmp!temp;
     fi

   ::topicsub1 == 2 ->
     if
     ::(speed >= 30) -> skip;
     ::(speed < 30) -> chan_brokersub1_spd!speed;
     fi

   ::topicsub1 == 3 ->
     atomic{
            if
            ::(temp <= 40) -> skip;
            ::(temp > 40) ->
                  chan_brokersub1_tmp!temp;
            fi
            if
            ::(speed >= 30) -> skip;
            ::(speed < 30) ->
                  chan_brokersub1_spd!speed;
            fi
     }

   :: else -> skip;
   fi

   /* Send received data to Subscriber_2 */
   if
   ::topicsub2 == 1 -> chan_brokersub2_tmp!temp;
   ::topicsub2 == 2 -> chan_brokersub2_spd!speed;
   ::topicsub2 == 3 -> atomic{
                        chan_brokersub2_tmp!temp;
                        chan_brokersub2_spd!speed;
                       }
   ::else -> skip;
   fi

 od
}
```

As this code clearly states, after receiving subscription requests from both infrastructures, the broker will repeatedly receive data from vehicles and forward it to those subscribers. Note that the broker can utilize conditions and filter data sent to the subscribed infrastructures. Indeed, as previously mentioned, in the considered case study, the conditions concern only data that will be sent to Subscriber 1 (which has registered for both

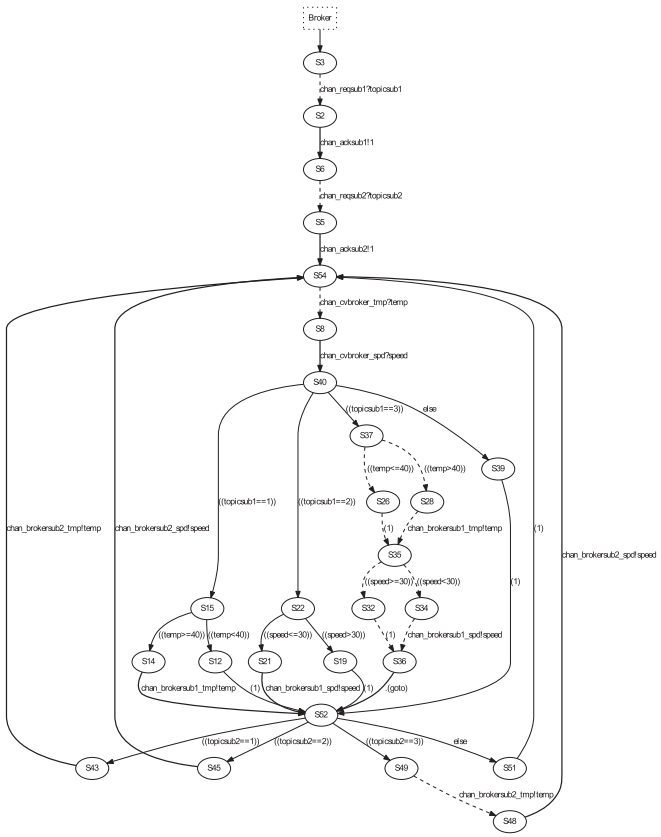

Fig. 6. Broker automaton.

temperature and speed topics). More exactly, the broker will provide infrastructure 1 with only temperatures that have exceeded 40 and speed values that are lower than 30 km/h. For instance, in the latter scenario, that is, when infrastructure 1 receives, in the same time, many of these speed values from vehicles that are located in the same geographic zone, it can deduce that there is an obstacle (or maybe an accident) preventing vehicles from normally flowing.

The Broker process actions are scheduled by the automaton depicted in Fig. 6. Note that there are no deadlocks in this automaton. Note also that the size of this automaton is important when compared with those of vehicles and subscribers. This shows that it is difficult to manually analyze the broker behavior (or, in general, the behavior of any other complex system). That is why it is very interesting to exploit SPIN to automatically simulate and verify this protocol. In fact, this will be the object of the next section.

D. SPIN Formal Verification of MQTT-CV

After implementing MQTT-CV components using Promela language, we focus, in this subsection, on SPIN verification of our proposal.

1) MQTT-CV Simulation With SPIN: Fig. 7 represents an extract of a random MQTT-CV simulation performed using SPIN. As shown in this figure, the different processes interact by sending messages using communicating channels which are defined as integers. For example, the broker starts by receiving

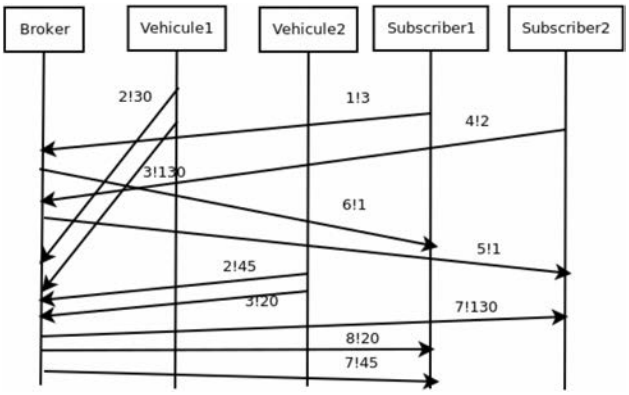

Fig. 7. Extract of MQTT-CV simulation with SPIN.

the message 3 in channel 1 from Subscriber 1 (`1!3`). This indicates that Subscriber 1 is requesting a subscription to both topics; temperature and vehicles' speed. The broker responds by sending message 1 through channel 6 to Subscriber 1. The interactions continue by sending data from Vehicle 1 and Vehicle 2 to the broker, and then, from the broker to Subscriber 1 and Subscriber 2. Note that the broker applies a filter before sending data to Subscriber 1. As Fig. 7 shows, Subscriber 1 receives only the value 20 for speed (less than 30 km/h condition) and 45 for temperature (greater than 40 °C condition).

We mention that SPIN also offers the possibility to execute guided simulations (i.e., steps, that will be executed, can be chosen in advance).

The simulation results, we obtained, have shown that the proposed MQTT-CV protocol behaves correctly in some specific interaction scenarios. However, to prove that the protocol is reliable in general, one must decide on the validity of all possible behavior cases and scenarios. In other words, to prove its correctness, MQTT-CV must be formally verified. This can be done by specifying and verifying the properties that this protocol must satisfy.

2) Safety Properties Verification: In this section, we verify the safety properties which confirm that MQTT-CV always stays in the allowed states in which nothing abnormal would happen. More exactly, we focus on the safety properties related to deadlock states. The latter are states from which a system cannot progress (i.e., from which no transitions are enabled). Actually, in general, the reachability of deadlock states is a consequence of a wrong system specification. For example, in our publish-subscribe system, a deadlock state might be a scenario in which the broker sends a message to a subscriber, while the latter is not ready to receive it.

It is worth mentioning that the absence of deadlocks in each process of the system (automata presented in Fig. 4, 5, and 6) does not guarantee the absence of deadlocks in the whole system. In other terms, this means that the safety verification must consist of checking the non-reachability of deadlock states in the automaton corresponding to the whole system (i.e., the transition system corresponding to the interaction between all processes). This automaton, provided by SPIN, is obtained by calculating the asynchronous product of the automata corresponding to each considered process (broker, subscribers, and publishers).

After using SPIN to verify the property of deadlocks absence in MQTT-CV, we obtained the result shown hereafter.

```
(Spin Version 6.4.5 -- 1 January 2016)
+ Partial Order Reduction

Full statespace search for:
never claim         - (not selected)
assertion violations - (disabled by -A flag)
cycle checks        - (disabled by -DSAFETY)
invalid end states +

State-vector 204 byte, depth reached 962, errors:0
6185 states, stored
6528 states, matched
12713 transitions (= stored+matched)
6547 atomic steps
hash conflicts: 0 (resolved)
...
```

This result demonstrates that the interacting MQTT-CV processes do not reach any deadlock states (`errors: 0`), and also shows information on the generated automaton (size, etc.).

E. Formal Verification of Temporal Properties

The Promela specification of MQTT-CV protocol described in the previous section allows only the simulation with SPIN of MQTT-CV behaviors and the verification of a part of its safety properties, for example those related to deadlock states. In this section, we focus on the specification and the verification of MQTT-CV liveness properties. Informally, a liveness property asserts that program execution eventually reaches some desirable states, which also means that system eventually will do something good, for example by producing desired outputs [19].

In our case, to prove that our protocol behaves correctly, by allowing the broker to send only the required data by the infrastructures, we need to verify some liveness properties, like:

- $p1$: always when a broker receives a temperature t, where $t \geq 40$ °C, this data will be sent to infrastructures.
- $p2$: always when a broker receives a temperature t, where $t < 40$ °C, this data will be sent to infrastructures (this property should not be satisfied).
- $p3$: always when a broker receives a speed value v, where $v \leq 30$ km/h, it will send it to the infrastructures.
- $p4$: always when the broker receives a speed value v, where $t > 30$ km/h, this data will be sent to infrastructures (this property should not be satisfied).
- $p5$: always when a broker is active and the received temperature is greater or equal to 40 °C (in our case it is equal to 45 °C), then infrastructure 1 will receive this data.
- $p6$: always when a broker is active and the received temperature is less than 40 °C (in our case it is equal to 30 °C), then infrastructure 1 will receive this data (this property should not be satisfied).
- $p7$: always when the broker is active and the received speed is less or equal to 30 km/h (in our case is equal to 20 km/h), then infrastructure 1 will receive this data.
- $p8$: always when the broker is active and the received speed is greater than 30 km/h (in our case it is equal to

130 km/h), then infrastructure 1 will receive this data (this property should not be satisfied).

Notice that the properties $p1$ to $p4$ concern the broker behavior, and $p5$ to $p6$ are related to the interactions between the broker and infrastructure 1[4] (or Subscriber 1). In order to prove the correctness of MQTT-CV, we propose to verify whether the broker behaves correctly by applying the filter and sending the right data, and whether the subscribers (infrastructures) receive right data from the broker. We notice also that in the section V-D we verified a global safety property related to deadlock lock states. So we proved that all processes (broker, subscribers, vehicles) behave without deadlock.

The verification of these properties will ensures that our protocol is correct with regard to the filtering of data that will be sent to infrastructures. Nonetheless, to proceed with their verification with SPIN, it is necessary to specify them with Linear Temporal Logic (LTL) [20]. It is a mathematical logic with modalities referring to time, which allows to express temporal properties on the system behaviors. Moreover in Promela, we need to determine which process (Vehicles, Broker, infrastructures) is sending/receiving what to/from whom at any time of execution. Furthermore, we need also to know what message is exchanged between the processes. In other word, it is necessary to be aware of all event occurrences during the process interactions.

However, with the implementation of MQTT-CV protocol as it is proposed in the precedent section, the system state does not change when a message exchange holds between processes through the channels. To overcome this issue, we propose to associate flags (a boolean variable) to each sending and receiving event. This allows to keep track of the actions performed by the processes and their environment reactions. By doing so, SPIN will generate transition system corresponding to Promela processes in which, transition will be enabled by sending and receiving messages, and each state will be specified by the flags that indicate: the entity (process) that performs the last action, the last performed actions, the message used in the last action, the entity to/from which the message was sent/received. So, in Promela, for each process, each message, and send/receive events, a flag is declared. These flags are updated together with each send/receive event, using a $atomic$ statement to ensure the values assignment in one execution step. For example, in the following we present the corresponding Promela code of the process Broker, enriched with the flags.

Send and receive flags indicate that the process is respectively sending or receiving messages. For example, the flags msg_speed, and msg_tmp refer respectively the last speed and temperature message sent. And the flags lf_broker, lf_v1, lf_inf1, indicate the last process (in this case broker, vehicle1, and infrastructure1) sending or receiving data. These flags are update at each sending and receiving events.

[4]It is sufficient to specify and verify properties only on infrastructure 1, without considering those of infrastructure 2, because both infrastructures have the same behavior.

```
/***FLAGS***/
/*Last performed action*/
bit send=0;
bit receive=0;
/*Message used in the last action*/
bit msg_speed=0;
bit msg_tmp=0;
bit msg_respsub1;
bit msg_respsub2;
bit msg_topicsub;

/*Lifeline that performed the last action*/
bit lf_broker=0;
bit lf_v1=0;
bit lf_v2=0;
bit lf_inf1=0;
bit lf_inf2=0;

....
/***process Broker***/
proctype Broker() {
atomic{ chan_reqsub1?topicsub1; receive=1; send=0; lf_broker=1;}
atomic{chan_acksub1!1; send=1; receive=0; msg_tmp=0;lf_broker=1;}
atomic{chan_reqsub1?topicsub1;receive=1; send=0; lf_broker=1; }
atomic{chan_reqsub1?topicsub1; chan_acksub1!1;send=1; receive=0;
msg_tmp=0;lf_broker=1;}

do
::
  atomic{receive=1; send=0; lf_broker=1;msg_tmp=1;msg_speed=0;
chan_cvbroker_tmp?temp;...}
atomic{ receive=1; send=0; msg_speed=1;msg_tmp=0; lf_broker=1;
chan_cvbroker_spd?speed;}

if
 ::topicsub1==1->
          if
           :: (temp< 40) -> atomic{send=0; receive=0; lf_broker=1;
              lf_v1=0;msg_tmp=0; msg_speed=0;temp=0;}
           :: (temp>=40 )-> atomic{chan_brokersub1_tmp!temp; send=1;
            receive=0; lf_broker=0; msg_tmp=1; msg_speed=0;temp=0;}
    fi

fi...
```

To verify the LTL properties $p\_1 - p\_8$ expressed informally above, we specify them with LTL and Promela as described in the following listing:

```
ltl p1 {[](( lf_broker==1 && temp < 40 && receive==1 ) ->
<> (lf_broker==1 && send==1 && receive==0 && msg_tmp==1))};

ltl p2 {[](lf_broker==1&& temp >= 40 && receive==1 ) ->
 <>( lf_broker==1 && send==1 && receive==0 && msg_tmp==1 ) )};

ltl p3 {[](( lf_broker==1 && speed <= 30 && receive==1 ) -> <>
(lf_broker==1 && send==1 && receive==0 && msg_speed==1))};

ltl p4 {[](( lf_broker==1 && speed > 30 && receive==1 ) ->
 <>( lf_broker==1 && send==1 && receive==0 && msg_speed==1 ) )};

 ltl p5 { [](temp>=40 && lf_broker==1 && receive==1)->
 <>( lf_inf1==1 && receive==1 && tempsub1==45)) };

 ltl p6 { [](temp<40 && lf_broker==1 && receive==1 )->
 <>( lf_inf1==1 && receive==1 && tempsub1==30)) };

  ltl p7 { [](speed>30 && lf_broker==1 && receive==1)->
  <>( lf_inf1==1 && receive==1 && spdsub1==130)) };

  ltl p8 { [](speed<=30 && lf_broker==1 && receive==1 )->
  <>( lf_inf1==1 && receive==1 && spdsub1==20)) };
```

We obtained the following results after their verification with SPIN:

- $p1$ and $p3$ are verified, which confirms that the broker send the received data that meet the conditions of the specified filter.
- however, $p2$ and $p4$ are not verified, which insures that the broker does not send the data that are not required by the infrastructures.
- $p5$ and $p7$ are verified, which confirms that infrastructure 1 receives data sent by the broker which respect the specified filter.
- however, $p6$ and $p8$ are not verified, which insures that infrastructure 1 does not receive the data that do not respect the specified filter.

VI. Experimental Validation of MQTT-CV

To conduct our experiments and assess both MQTT and the proposed MQTT-CV solution, we have opted for Eclipse Mosquitto [21]. In a nutshell, this tool, written in C, is a message broker that implements the MQTT protocol versions 3.1 and 3.1.1. Actually, given its lightweight feature (i.e., its lightweight technique of carrying out messaging using the publish/subscribe model), Mosquitto is also suitable for IoT messaging and low-power single-board devices (such as low-power sensors, mobile devices, cell phones, embedded microcontrollers, etc.). Furthermore, Mosquitto, which is free, open-source, and available for both Linux and Windows platforms, provides a C library for implementing and launching MQTT publishers and subscribers through their respective `mosquitto_pub` and `mosquitto_sub` command lines. This gives developers the ability and freedom to completely modify/adapt the system behavior according to their needs and preferences. Finally, like any other MQTT broker, Mosquitto allows creating and connecting several publish/subscribe clients.

To ease its reading and understanding, the remainder of this section has been organized in three subsections, as follows. First, Section VI-A details the experimental environment configuration (vehicles, broker, and infrastructure settings,...). Second, Section VI-B provides the evaluation criteria according to which MQTT-CV and MQTT will be compared. Finally, Section VI-C presents the obtained results along with their corresponding interpretation and analysis.

A. Validation Settings and Parameters

For the sake of comparison and evaluation, besides the original MQTT functionalities offered by Mosquitto, we have reused the provided C source code to implement our proposed MQTT-CV broker and its corresponding subscribers/publishers. More in detail, in order to (1) check the proper operation of our solution, and (2) to compare its performance with that of the basic MQTT broker, we have considered a connected vehicles scenario in which there are one broker (MQTT or MQTT-CV), one automotive infrastructure (i.e., one subscriber), and n vehicles (publish clients).

The validation environment has been set in such a way that the broker, automotive infrastructure and vehicles will be executed separately. In other terms, we have used three physical machines. The first one was used to run the broker (alternatively MQTT or MQTT-CV). The second one was used as the automotive infrastructure. The third one was utilized to run the implemented processes (`mosquitto_pub`) that simulate vehicles. The details of the three utilized computers are depicted in Table I. The major objective behind executing the broker (MQTT-CV or MQTT) and its corresponding clients (i.e., infrastructure and vehicle processes) on different physical machines was to avoid affecting the obtained results (presented in Section VI-C).

The three following points will briefly talk about the specifics of the three main components of our validation experiments; namely, the vehicles, automotive infrastructure, and broker.

- First, note that temperatures collected and sent by our virtual vehicles (processes) to the broker have not been generated through a random process, but are, in fact, real data that has been collected by real vehicles. This temperature data set, provided by PSA Group, is related to the external sensed temperature and internal oil temperature of vehicles.
- Second, no specific tasks were performed by the implemented automotive infrastructure client, except printing the received temperature to the screen.
- Third, and finally, as for the implemented MQTT-CV broker, it proposes several predefined filtering functions. It also gives subscribers the ability to formulate their own requests/conditions on data they desire to receive (`less-than`, `greater-than`, ...). For instance, a subscriber can express its interest in receiving only data that exceeds a certain threshold (e.g., temperatures that are higher than 36 °C). In our experiments, two scenarios have been considered. In the first one, the infrastructure requires only temperatures that exceed 8 °C (which represents 50% of the total data received by the broker). While in the second scenario, the infrastructure demands only temperatures that surpass 14 °C (which represents 2% of the total data received by the broker).

The validation scenario starts as follows. First, the broker (MQTT-CV or MQTT) is launched (respectively through the `mosquitto-cv` or `mosquitto` command lines). Second, the automotive infrastructure (the only subscriber in the system) is created and connected to the broker through the `mosquitto_sub` command:

```
$ mosquitto_sub -v -id autoInfra -h
IP_address -t temperature/# -q 0
```

Finally, processes that represent vehicles are created, connected to the broker, and their corresponding implemented publishing mechanism starts reading and reporting real data. In order to evaluate the scalability of both MQTT-CV and MQTT brokers, the number of participating vehicles was progressively increased through the gradual increase of Mosquitto processes that simulate them. During this whole working phase, the considered evaluation metrics (detailed below in Section VI-B) were continuously recorded at the level of both the broker and infrastructure. The goal is to evaluate the effect of the proposed solution on these two entities. Note that there is no difference in the vehicles' performance in both architectures; MQTT and

TABLE I
HARDWARE SETTINGS AND DETAILS

Characteristics	Vehicles	Broker (MQTT/MQTT-CV)	Automotive infrastructure
Processor (GHz)	Intel i5-3337U @1.80GHz x 4	AMD E1-1200 @1.40GHz	Intel i3-370M @2.40GHz x 4
RAM Memory (GB)	7.7	1.6	3.7
Operating System	Ubuntu 16.04	Ubuntu 16.04	Ubuntu 18.04
OS Type	64-bit	64-bit	64-bit

MQTT-CV. In both cases, vehicles read and report the same data for fair comparison purposes.

The amount of data sent by each vehicle is set to 727 178 items. Of course, vehicles report this data (to the broker) with a certain data rate of x messages/second. However, in order to be more accurate, rather than tracking and reporting the number of connected vehicles or their data rate (i.e., number of published messages per second), we have measured the broker load in terms of received "publish" messages over one minute. In Mosquitto, this number (moving average) of published messages can be obtained by simply creating a client and subscribing it to the following provided system topic:

```
$mosquitto_sub -v -id bLoad -h IP_address
-t
\ $SYS/broker/load/publish/received/1min-q
0
```

As a last point in this first subsection, we mention that as was the case for the automotive infrastructure during its subscription (`-t topic -q 0`), each vehicle in the system also specifies the lowest available Quality of Service (i.e., 0) to publish its data (using the `mosquitto_publish()` function). In other words, in these experiments, the only considered QoS is 0. Recall that MQTT offers three levels of QoS (0, 1, and 2). A higher QoS is more reliable but requires a higher latency and higher bandwidth. These different levels of QoS determine how the publisher-broker and broker-subscriber communications will take place. In more specific words, 0 means that packets will be delivered once (no confirmation), 1 means that they will be delivered at least once (confirmation required), and finally, 2 means they will be delivered exactly once (a four-step handshake).

B. Validation Metrics

The two main criteria we used for the evaluation of MQTT-CV and MQTT are the CPU and the RAM usage.

- **CPU usage:** this criterion, which is expressed in percentage, represents the processor load in both the broker and infrastructure machines. The goal is to compare our MQTT-CV approach with MQTT and measure their effect on the broker and infrastructure performances (advantages and disadvantages).
  To collect this first information (i.e., CPU load), we have considered several Linux tools/commands such as the well-known `top`, `ps`, and *PowerShell Core* (`Get-Process`, etc.).
- **RAM usage:** this evaluation metric measures the quantity of memory that has been consumed by both the broker and infrastructure entities during their working stages. We mention that this information has been also collected using the Linux tools mentioned above. The RAM usage, collected by these tools, is expressed in percentage.

The obtained results, depicted in the next (last) subsection, show the effect of both MQTT-CV and MQTT on resources' consumption. In fact, the obtained results can be used as a gauge to estimate the impact that the additional functionalities (filtering operations,...) of MQTT-CV solution can have on the complexity/performance of both the broker and infrastructure. For instance:

- Will the MQTT-CV broker consume less or more resources (CPU and RAM) due to its specific data processing tasks? This question can be asked differently. Which task consumes more CPU and RAM: (1) transmissions (i.e., creating and sending packets) or (2) local data processing?
- As regards the automotive infrastructure, intuitively speaking, fewer received messages mean less processing. As was the case for the previous point, this one also remains to be confirmed through the obtained results.

These last two addressed points are very crucial because, in addition to the Connected Vehicles context, the MQTT-CV logic can be also considered in many other publish/subscribe models. For example, in IoT applications, the broker can be installed on a low-power device (e.g., a low-power sensor device, embedded microcontroller,...). The same applies to subscribed clients. Therefore, on the one hand, it would be very beneficial not to burden these low-power clients with useless data. Rather, the broker must feed them with only data that interests them. On the other hand, data processing (filtering, etc.) tasks performed by the broker must not move the burden from clients to the broker itself. In other words, these tasks must not drastically augment resources consumption at the broker level.

C. Validation Results

In the following, we start by depicting and discussing the obtained broker results. Then, we move to the automotive infrastructures' performance. Note that the obtained results (whether those relating to the broker or infrastructure) were plotted against the broker load which is expressed in million publish messages/minute.

1) Brokers Results: Fig. 8 shows the amount of CPU consumed by MQTT-CV and MQTT brokers to perform their respective tasks. As previously mentioned, we have considered two MQTT-CV scenarios. In the first one, the MQTT-CV broker sends only temperatures that exceed 8 °C, which represents 50% of the total received data from vehicles. This broker is denoted as "MQTT-CV Broker with 50% filter" in Fig. 8. In the second

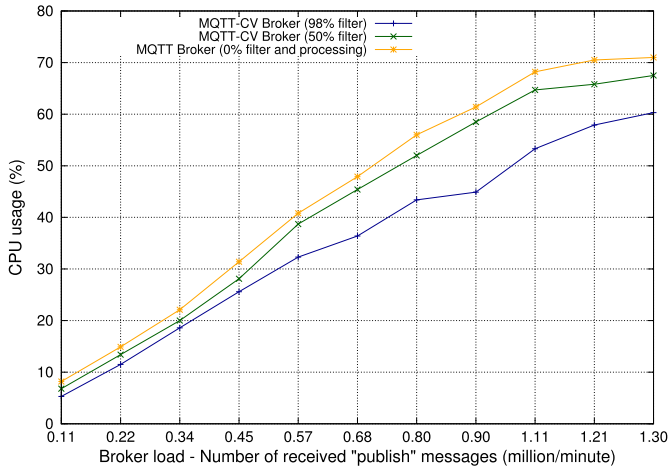

Fig. 8. MQTT-CV and MQTT brokers: comparison in terms of CPU usage (%).

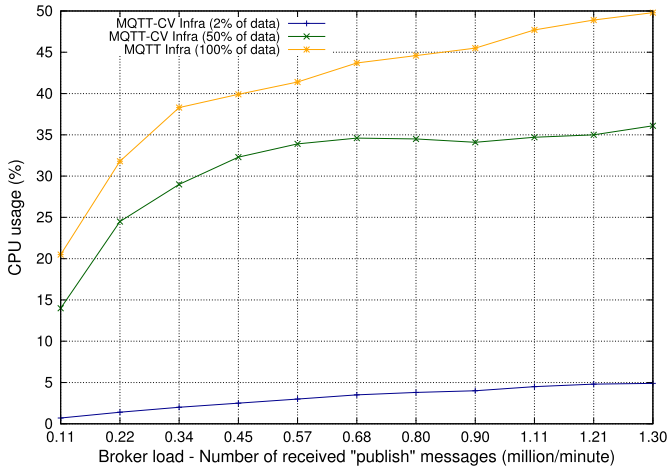

Fig. 9. MQTT-CV and MQTT infrastructures: comparison in terms of CPU usage (%).

scenario, the MQTT-CV broker forwards only temperatures that exceed 14 °C, which represents 2% of the total data received by the broker (denoted as "MQTT-CV Broker-98% filter"). Indeed, this high value of 98% has been chosen in order to more closely monitor the MQTT-CV broker behavior in those extreme circumstances and to show the filtering effect on it.

To summarize, the main difference between MQTT-CV and MQTT resides in the fact that the MQTT broker does not inspect the received data or apply any treatment to it. This basic broker acts just like a relay or bridge for subscribers that are interested in that data. As for our MQTT-CV broker, it processes data before forwarding it to subscribers. As explained above, in this simulation scenario, MQTT-CV forwards only data that is larger than a certain threshold imposed by the automotive infrastructure. In general, we can say that MQTT sends more packets, whereas MQTT-CV performs local computations and sends fewer messages.

Fig. 8 reveals that the applied filter does not augment the MQTT-CV resources' consumption, but on the contrary, it reduces it. First, according to Fig. 8, we can say that the more a broker sends data the more CPU it will consume, and vice versa. Second, note that the filter threshold specified by the infrastructure considerably affects the MQTT-CV broker performance because, in fact, it is tightly related to the number of sent packets (to infrastructure). In other terms, depending on the set threshold, less or more messages will be sent to the infrastructure. And, more sent packets means more CPU consumption.

Regarding RAM usage, the obtained results (not depicted here given their similarity) have shown that RAM consumption is constant and very low for both MQTT and MQTT-CV. In more exact words, the recorded RAM usage is 0.05% for MQTT and MQTT-CV regardless of the considered load and applied filters. We conclude that neither the local processing nor the huge number of sent packets has affected the RAM consumption.

2) Infrastructures Results: The objective behind these performance results is to confirm the benefits that the MQTT-CV broker would allow the infrastructure to gain. As previously stated, fewer received messages will (without a doubt) relieve the infrastructure and allow it to consume less CPU. Fig. 9 confirms this intuitive expectation and shows that the added MQTT-CV functionalities relieve the infrastructure and allow it consume less CPU when compared with the MQTT infrastructure.

As was the case for the brokers, the obtained results (not depicted here given their similarity) have also shown that space complexity (RAM consumption) is constant throughout the operation of both MQTT and MQTT-CV infrastructures. The recorded RAM usage was 0.25% regardless of the received data quantity.

Based on the obtained results, we conclude that MQTT-CV is more efficient than MQTT in terms of CPU usage. First, the MQTT-CV broker consumes less CPU because it sends fewer messages. Second, the MQTT-CV infrastructure also consumes less CPU because it receives fewer messages. These results can be also interpreted as follows: local data processing is more efficient (and less energy-consuming) than transmissions. Moreover, we also conclude that the added filtering functions (e.g., greater than, less than, etc.) do not augment RAM and CPU consumption. Instead, they allow the broker to reduce its need for resources (by reducing the number of sent packets). Finally, we estimate that with QoS 1 and QoS 2, MQTT performance will be worse because, in this case, more messages will be sent.

VII. Conclusion and Future Works

This paper presented a proposal that aims to overcome the MQTT protocol drawbacks in the context of connected vehicles. These drawbacks are mainly related to (1) the huge volume of data sent by connected vehicles to automotive infrastructures (through a broker), and (2) the protocol reliability regarding the safety properties. To remedy these limitations, first, we have proposed a variant of MQTT, named MQTT-CV, which aims to alleviate automotive infrastructures (subscribers) in terms of data that will be sent to them by the broker (according to their topic subscriptions). In other words, these infrastructures will store/process only data that is important to them. Second, to ensure the reliability of MQTT-CV, which is a critical system that involves the drivers' safety, we have formally analyzed it.

More in detail, to specify the interaction between its different components (vehicles, broker, and automotive infrastructures), we have modeled MQTT-CV using the UML sequence diagram. After that, we have (1) implemented MQTT-CV using Promela language, and (2) utilized SPIN to perform simulations that allow analyzing some iteration scenarios. Finally, using SPIN, we have verified that MQTT-CV satisfies the safety property related to deadlock states, and liveness properties that express temporal constraints on MQTT-CV behaviors. In more exact words, we have proven that the broker, vehicles, and automotive infrastructures behave correctly and that MQTT-CV will never enter a deadlock situation. Moreover, we proved also, that the broker ensures data filtering before their sent.

As future works, we intend to improve MQTT-CV and make it able to handle more complex conditions that automotive infrastructures can express on proposed topics. In addition to this, we plan to consider the imposed conditions at the verification level.

ACKNOWLEDGMENT

The authors would like to thank A. Haroun and F. Dessables from PSA Group for their valuable help in providing us with realistic vehicular data sets.

REFERENCES

[1] J. Gubbi, R. Buyya, S. Marusic, and M. Palaniswami, "Internet of Things (IoT): A vision, architectural elements, and future directions," *Future Gener. Comput. Syst.*, vol. 29, no. 7, pp. 1645–1660, 2013.

[2] D. Locke, "MQ telemetry transport (MQTT) v3. 1 protocol specification," IBM, 2010. [Online]. Available: http://www.ibm.com/developerworks/webservices/library/ws-mqtt/index.html

[3] P. Saint-Andre, K. Smith, and R. Troncon, *XMPP: The Definitive Guide: Building Real-time Applications With Jabber*, 1st ed. Sebastopol, CA, USA: O'Reilly Media, 2009.

[4] A. Haroun, A. Mostefaoui, and F. Dessables, "A big data architecture for automotive applications: PSA group deployment experience," in *Proc. 17th IEEE/ACM Int. Symp. Cluster, Cloud Grid Comput.*, Madrid, Spain, 2017, pp. 921–928.

[5] B. Aziz, "A formal model and analysis of an IoT protocol," *Ad Hoc Netw.*, vol. 36, pp. 49–57, 2016.

[6] "Verifying multi-threaded software with spin," 2019. [Online]. Available: http://spinroot.com/

[7] G. J. Holzmann, "The model checker spin," *IEEE Trans. Softw. Eng.*, vol. 23, no. 5, pp. 279–295, May 1997.

[8] G. Holzmann, *Spin Model Checker, the: Primer and Reference Manual*, 1st ed., Reading, MA, USA: Addison-Wesley Professional, 2003.

[9] R. Baldoni, M. Contenti, S. T. Piergiovanni, and A. Virgillito, "Modeling publish/subscribe communication systems: Towards a formal approach," in *Proc. 8th Int. Workshop Object-Oriented Real-Time Dependable Syst.*, Jan. 2003, pp. 304–311.

[10] L. Abidi, C. Cerin, and S. Evangelista, "A petri-net model for the publish-subscribe paradigm and its application for the verification of the bonjourgrid middleware," in *Proc. IEEE Int. Conf. Serv. Comput.*, Jul. 2011, pp. 496–503.

[11] A. Gawanmeh, "Embedding and verification of zigbee protocol stack in event-b," *Procedia Comput. Sci.*, vol. 5, pp. 736–741, 2011.

[12] C. Wang, A. Carzaniga, D. Evans, and A. L. Wolf, "Security issues and requirements for internet-scale publish-subscribe systems," in *Proc. 35th Annu. Hawaii Int. Conf. Syst. Sci.*, Jan. 2002, pp. 3940–3947.

[13] S. Lee, H. Kim, D. k. Hong, and H. Ju, "Correlation analysis of mqtt loss and delay according to qos level," in *Proc. Int. Conf. Inf. Netw.*, Jan. 2013, pp. 714–717.

[14] D. Thangavel, X. Ma, A. Valera, H. X. Tan, and C. K. Y. Tan, "Performance evaluation of mqtt and coap via a common middleware," in *Proc. IEEE 9th Int. Conf. Intell. Sensors, Sensor Netw. Inf. Process.*, Apr. 2014, pp. 1–6.

[15] D. Garlan, S. Khersonsky, and J. S. Kim, "Model checking publish-subscribe systems," *Model Checking Software*, T. Ball and S. K. Rajamani, Eds., Berlin, Germany: Springer, 2003, pp. 166–180.

[16] L. Baresi, C. Ghezzi, and L. Mottola, "On accurate automatic verification of publish-subscribe architectures," in *Proc. 29th Int. Conf. Softw. Eng.*, May 2007, pp. 199–208.

[17] F. He, L. Baresi, C. Ghezzi, and P. Spoletini, "Formal analysis of publish-subscribe systems by probabilistic timed automata," in *Proc. Int. Conf. Formal Techn. Networked Distrib. Syst.*, Berlin, Heidelberg: Springer-Verlag, 2007, pp. 247–262.

[18] S. Gallotti, C. Ghezzi, R. Mirandola, and G. Tamburrelli, "Quality prediction of service compositions through probabilistic model checking," in *Proc. 4th Int. Conf. Qual. Softw.-Architectures: Models Architectures*, ser. QoSA '08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 119–134.

[19] S. S. Owicki and L. Lamport, "Proving liveness properties of concurrent programs," *ACM Trans. Program. Lang. Syst.*, vol. 4, no. 3, pp. 455–495, 1982.

[20] A. Pnueli, "The temporal logic of programs," in *Proc.18th Annu. Symp. Found. Comput. Sci.*, Oct. 1977, pp. 46–57.

[21] "Eclipse Mosquitto," Mar. 2019. [Online]. Available: https://mosquitto.org/