Timing Predictability for SOME/IP-based Service-Oriented Automotive In-Vehicle Networks

Enrico Fraccaroli*, Prachi Joshi†, Shengjie Xu*, Khaja Shazzad†, Markus Jochim†, and Samarjit Chakraborty*
*University of North Carolina at Chapel Hill, Chapel Hill, North Carolina, USA, email:{enrifrac, sxunique, samarjit}@unc.edu
†Research and Development, General Motors, Michigan, USA

***Abstract*—In-vehicle network architectures are evolving from a typical signal-based client-server paradigm to a service-oriented one, introducing flexibility for software updates and upgrades. While signal-based networks are static by nature, service-oriented ones can more easily evolve during and after the design phase. As a result, service-oriented protocols are becoming more prominent in automotive in-vehicle networks. While applications like infotainment are less sensitive to delays, others like sensing and control have more stringent timing and reliability requirements. Hence, wider adoption of service-oriented protocols requires addressing the timing analysis and predictability of such protocols, which is more challenging than in their signal-oriented counterparts. In service-oriented architectures, the *discovery phase* defines how clients find their required services. The time required to complete the discovery phase is an important parameter since it determines the readiness of a sub-system or even the vehicle. In this paper, we develop a formal timing analysis of the discovery phase of SOME/IP, which is an emerging service-oriented protocol being considered for adoption by several automotive Original Equipment Manufacturers (OEMs) and suppliers.**

***Index Terms*—Service-oriented architecture, SOME/IP, service discovery, worst-case timing analysis**

I. Introduction

Over the past few decades, automotive electrical and electronic (E/E) architectures have followed a trend depicted in Fig. 1. They have transitioned from being modular: where each function was allocated to a dedicated Electronic Control Unit (ECU), to becoming more integrated: where functionalities were combined into ECUs as domains. Serial data communication protocols such as CAN, CAN-FD, LIN, and FlexRay have played an important role in in-vehicle network communication for these architectures (using *signal-based* communication). More recently, the move to create a layered architecture where input/output devices and sensors/actuators are separated from the compute layer and are interfaced via "zonal gateways" is gaining momentum in the automotive industry. Additionally, *service-oriented* communication and Ethernet are replacing the underlying communication protocols for such architectures. The main motivation behind this trend is the growing demand for computation and communication bandwidth and the need for flexibility to add software. This transformation towards software-defined platforms, where innovation in software drives the main value-add, is now the key trend in the automotive industry [1]; and this is likely to remain in the near future.

Service-oriented communication & timing analysis: Here, service-oriented communication is key to flexibility for frequent software updates in an architecture. This is because

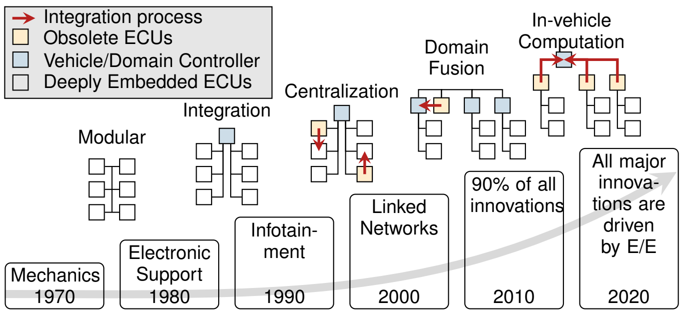


Figure 1: Trends in automotive E/E architectures [2].

services are independent software providing functionalities through implementation-independent interfaces. This decoupling between how functionality is implemented and how to gain access to it makes it scalable and easy to maintain. However, it also poses new challenges in analyzing its timing behavior. *Timing analysis* techniques are used to study the timing behavior of lower layers of the protocol stack, e.g., in CAN [3], [4] and Ethernet [5]–[7]. While well-developed for such signal-oriented protocols, they have not yet been sufficiently studied for service-oriented middleware such as the "Scalable service-Oriented MiddlewarE over IP (SOME/IP)" or "Data Distribution Service (DDS)".

This paper focuses on the specific middleware, SOME/IP. As the name implies, it is an application layer Ethernet-based protocol that can be paired with both TCP/IP and UDP protocols. In SOME/IP, clients and services that need to communicate must first discover each other through a process called *service discovery* [8]. This aspect of *dynamically* discovering entities that are to communicate and realize a particular service is a key difference from signal-based communication, where communicating entities are always pre-configured. Such *static* configuration in signal-oriented communication makes timing analysis much easier. In service-oriented communication, it is important to assess how long a *discovery phase* lasts since this duration determines the readiness of a component or subsystem within an automotive architecture. However, very little study has been devoted to analyzing this *discovery latency*. The work in [9] has been one of the first attempts toward timing analysis of the discovery phase in SOME/IP. But it does not suitably consider all the scenarios involved and, as a result, returns incorrect estimates of the discovery latency, as we show later in our paper. Other studies have attempted to reduce the discovery latency by optimally determining the SOME/IP parameters [10] or by starting clients and services in an optimal order [11].

Table I: Service-oriented communication taxonomy.

	One-to-One	One-to-Many
Sync.	Request/Response	–
Async.	Request/Async. Response, Fire-and-Forget	Publish/Subscribe

The improved timing model proposed in this paper would also impact these results.

This paper proposes a timing model to estimate the discovery latency of SOME/IP. It builds upon the work in [9] and addresses its shortcomings. In particular, it accounts for all the different scenarios arising in the discovery phase and their impact on the discovery latency. We believe the resulting timing model is more intuitive and can be easily validated under different scenarios, as shown in this paper.

II. Background

This section explains what a Service-Oriented Architecture (SOA) is and what is the timing analysis problem for it. The focus is on the *service discovery* phase of SOME/IP.

A. Service-Oriented Architecture (SOA)

All SOAs have one or more service providers and clients that need their service [12]. As shown in Table I, we can create a taxonomy of SOA communication patterns based on whether (1) one or many services handle a client request, and (2) the communication happens synchronously or asynchronously. When only one service handles a request, we are in a *one-to-one* configuration. Conversely, when multiple services handle it, we are in a *one-to-many* configuration. In *synchronous* communications, the client sends either a blocking or non-blocking request for the service and expects a timely response. In *asynchronous* ones, the client sends a non-blocking request to the service and does not expect an immediate response or any response at all.

The main pattern of synchronous communications is the *request/response* one. Here, the client makes a request to a service, which then answers the request and marks the end of the interaction. This pattern requires the client to know the address and port where the service resides. With asynchronous communication like the *request/async response* pattern, where responses are not expected to be received in a timely fashion, the client does not block its execution and handles responses as they arrive. In the *fire-and-forget* pattern, the client sends a request but does not expect a response.

But this paper focuses on protocols that follow the *Publish/Subscribe* pattern. Publish-subscribe is a data distribution architecture where a series of entities owning information (publishers) send it to those who want it (subscribers) through a middle-man (broker). Subscription and publishing are performed for a specific topic of interest to avoid broadcasting undesired data. Publish/subscribe architectures may be further categorized, e.g., as *brokered* vs. *broker-less*.

In the *brokered* scenario, both publishers and subscribers communicate through the broker. There are pros and cons of having a broker. With a broker, for instance, publishers do not need to know the address of all the subscribers but just that of the broker. They also do not need to communicate with every single member of a subscription group; the broker takes care of delivering the message. The broker can also deal with data availability from "offline publishers": even if a publisher goes offline, the messages sent up until that moment are safely stored inside the broker's queue. There are also drawbacks of this communication pattern. First, it is a traffic-intense architecture since all data published to the broker must then be re-sent to subscribers. Second, the broker might become a bottleneck or a single point of failure. To avoid bottlenecks and increase throughput and availability, a network can rely on multiple brokers, forming what is known as a *broker federation*.

In a *broker-less* scenario, messages go directly from publishers to subscribers, and do not go through a broker. One of the benefits of this architecture is a lighter network traffic with reasonable latency. Without a broker, there is no chance for it to become a bottleneck or a single point of failure. However, clients and services need to know each other's address, or rely on a *discovery mechanism* to find it [8]. Unlike the brokered pattern, both sender and receiver must be online and available for the message to be correctly delivered.

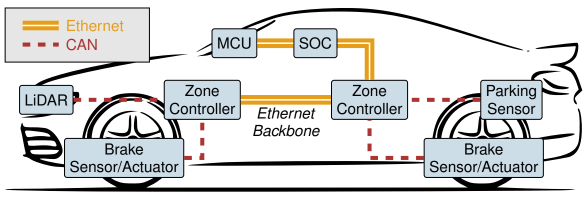
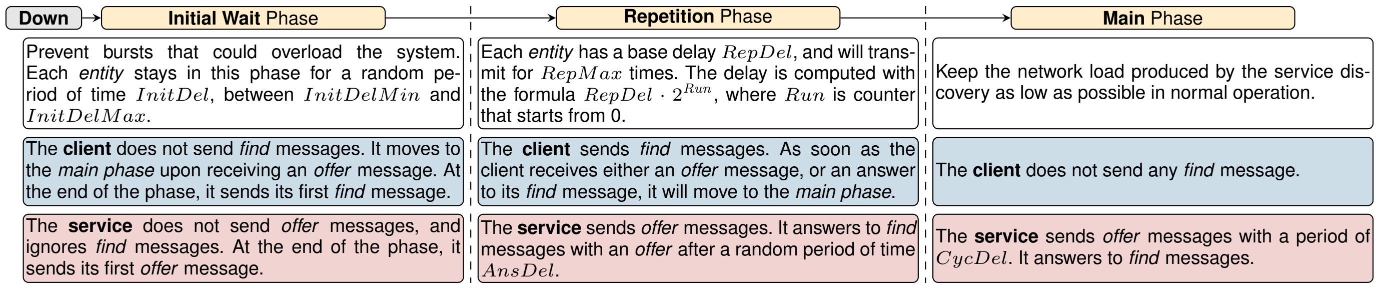


Figure 2: Example of In-Vehicle Network (IVN).

B. Timing analysis problems in Service-Oriented Architectures

As mentioned in the previous section, electronics and software are becoming an integral part of the automotive domain, boosting non-critical functionalities like infotainment and critical ones like assisted driving. Safety-critical functions involve sensors, actuators, and control units combined. Looking at Fig. 2, if the sensor data is made available by an ECU and sent to a System-on-Chip (SOC) or a Micro-Controller Unit (MCU) for processing, the time the data takes to get to the destination is critical. The path that leads from sensors to actuators (and back) involves different communication protocols (e.g., CAN, CAN-FD, LIN, and FlexRay), and computing sensor-to-actuator delays is a non-trivial endeavor. Knowing this delay is vital to design control software and to ensure stability and control performance of feedback controllers implementing safety-critical functions like brake control.

Deterministic sensor-to-actuator delays and their safe, but tight estimates, is paramount in the automotive domain. Time-critical functionalities require that the IVN transmits sensor and control data in a fixed amount of time to avoid uncontrolled behaviors. Bounded delays must be ensured throughout all the design phases of the vehicle architecture. Two prominent families of techniques aid designers in doing this: formal and simulation-based timing analysis. The former is an exhaustive analysis that can guarantee upper bounds on end-to-end latencies. Still, detailed models are hard to come by, and it often suffers from what is referred to as a "state-space explosion"



Prevent bursts that could overload the system. Each *entity* stays in this phase for a random period of time $InitDel$, between $InitDelMin$ and $InitDelMax$.

The **client** does not send *find* messages. It moves to the *main phase* upon receiving an *offer* message. At the end of the phase, it sends its first *find* message.

The **service** does not send *offer* messages, and ignores *find* messages. At the end of the phase, it sends its first *offer* message.

Each *entity* has a base delay $RepDel$, and will transmit for $RepMax$ times. The delay is computed with the formula $RepDel \cdot 2^{Run}$, where Run is counter that starts from 0.

The **client** sends *find* messages. As soon as the client receives either an *offer* message, or an answer to its *find* message, it will move to the *main phase*.

The **service** sends *offer* messages. It answers to *find* messages with an *offer* after a random period of time $AnsDel$.

Keep the network load produced by the service discovery as low as possible in normal operation.

The **client** does not send any *find* message.

The **service** sends *offer* messages with a period of $CycDel$. It answers to *find* messages.

Figure 3: Service discovery phases in SOME/IP.

Table II: SOME/IP configuration parameters mapping.

Parameter name from [14]	Symbol
`INITIAL_DELAY_MIN`	$InitDelMin$
`INITIAL_DELAY_MAX`	$InitDelMax$
`REPETITIONS_BASE_DELAY`	$RepDel$
`REPETITIONS_MAX`	$RepMax$
`REQUEST_RESPONSE_DELAY`	$AnsDel$
`CYCLIC_OFFER_DELAY`	$CycDel$

or lack of scalability. The latter approach does not guarantee safe upper bounds on end-to-end latencies but can discover timing issues faster and might be more scalable. In this paper, we propose a formal *timing analysis* to estimate the worst-case timing bounds on the service discovery phase of SOME/IP.

C. *SOME/IP service discovery*

SOME/IP is a middle-ware solution that provides service-oriented communication in automotive in-vehicle networks. The protocol is based on the concept of a *broker-less publisher/subscriber* architecture outlined above, i.e., the protocol defines how services can offer their data and how clients can find the desired data. How the discovery phase works is defined in AUTOSAR [13], while the specifics of the SOME/IP discovery phase are detailed in [14]. The discovery process is subdivided into three phases for both clients and services: *initial wait phase*, *repetition phase*, and *main phase*. Services send broadcast *offer* messages, specifying which services they provide. Clients send broadcast *find* messages specifying which service they require. Only during specific phases do services and clients send their messages. The behavior of clients and services is detailed in Fig. 3. For the rest of this paper, we are going to map the parameters found in the *SOME/IP Service Discovery Protocol Specification* [14] to the variables outlined in Table II and use them for bounding the SOME/IP discovery latency.

III. Estimating the SOME/IP discovery latency

This section estimates the discovery latency for a single client/service pair, where the objective is for the client to discover the desired service. Typical scenarios have several services and clients, and the following analysis is a necessary step for handling those scenarios. The idea is that once we compute the discovery time for each client/service pair of the system, the maximum among those values represents the discovery time of our entire system. In the equations used in this section, we will use c to identify the properties of a *client*, s to identify those of a *service*, and e to identify those of a generic *entity*, which can be either be a *client* or a *service*. The communication delay t_c is computed using approaches like the ones in [6], [7]. Let us now define two variables:

$$c_{init} = c_{BootDel} + c_{InitDel} \ , \tag{1}$$

$$s_{init} = s_{BootDel} + s_{InitDel} \ , \tag{2}$$

where *BootDel* is the period of time when clients and services are *down*, and *InitDel* is the length of their *initial wait phase*. As mentioned above, the delays between messages sent by both client and service during the *repetition phase* are doubled with each consecutive message. We can compute the delay before sending the i-th message using the following:

$$d(e,i) = 2^i \cdot e_{RepDel} \ , \tag{3}$$

where e_{RepDel} represents the repetition delay. The actual size of the *repetition phase* is given by summing all those delays. We can compute its size, up to the n-th repetition message, by using the following function:

$$t_{rep}(e,n) = \sum_{i=0}^{n-1} d(e,i) \ . \tag{4}$$

Using properties of geometric series, we can transform this sum into a single expression:

$$t_{rep}(e,n) = (2^n - 1) \cdot e_{RepDel} \ . \tag{5}$$

A. *Computing the actual length of the repetition phase*

As seen above, the time window when either the client or service starts sending messages is the *repetition phase*. Depending on their respective configuration, we might end up with a service that enters the *repetition phase* before the client or the other way around. For the purpose of timing analysis, we need to compute the index of the first *find* or *offer* message that can reach its destination. To do that, we need to find the timespan in the *repetition phase*, during which the source node (either client or service) sends messages, but the destination node is not ready to receive them. We can call this timespan the *failure window*. However, we need to differentiate between two cases: z_s if the client enters the *repetition phase* before the service, and z_c if the service enters the *repetition phase* before the client. We can compute them as follows:

$$z_s = \begin{cases} s_{init} - c_{init} & \text{if } s_{init} > c_{init} \\ 0 & \text{otherwise} \end{cases} , \tag{6}$$

$$z_c = \begin{cases} c_{BootDel} - s_{init} & \text{if } s_{init} < c_{BootDel} \\ 0 & \text{otherwise} \end{cases} . \tag{7}$$

We use $c_{BootDel}$ instead of c_{init} to compute z_c because the client accepts *offer* messages even during the *initial wait phase*.

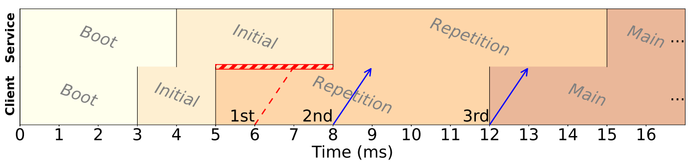


Figure 4: Example showing the *failure window* (red hatched band in the middle), and *repetition phase* messages that either fail (red dashed line) or succeed (blue arrow) to reach the service.

Once we have computed the size of the *failure window*, we need to find the index of the first message that can reach its destination within the *repetition phase*. Again, we need to differentiate between x_s for *find* messages and x_c for *offer* messages. We can compute these two values as follows:

$$x_s = \begin{cases} \left\lceil \log_2 \left(\frac{z_s - t_c}{c_{RepDel}} + 1 \right) \right\rceil & \text{if } z_s > t_c \ , \\ 0 & \text{otherwise} \end{cases} \tag{8}$$

$$x_c = \begin{cases} \left\lceil \log_2 \left(\frac{z_c - t_c}{s_{RepDel}} + 1 \right) \right\rceil & \text{if } z_c > t_c \ , \\ 0 & \text{otherwise} \end{cases} \tag{9}$$

where t_c represents the communication delay between the client/service pair, and *RepDel* represents the repetition base delay used during the *repetition phase*.

Let us analyze these two equations with the help of Fig. 4. In this example, t_c=1, c_{RepDel}=1, c_{RepMax}=3, and z_s=3. Now, from Eq. (5), we know how to compute the length of the *repetition phase* given that we are sending the n-th message. Conversely, here we want to compute the index of the first message that successfully reaches the destination, after the *failure window*. As already mentioned, every message sent inside the *failure window* is not received by the destination node. However, we need to take into account the communication delay t_c. Because, as shown in Fig. 4, the message is still delivered if the instant when the message is sent is inside the *failure window*, but the instant when it is received is not, i.e., after t_c. That is why the actual size we need to consider in our equations is $z_s - t_c$. We can derive Eq. (8) starting from Eq. (5), as follows:

$$\begin{aligned} t_{rep}(c, x_s) &= z_s - t_c \ , \\ (2^{x_s} - 1) \cdot c_{RepDel} &= z_s - t_c \ , \\ 2^{x_s} &= \frac{z_s - t_c}{c_{RepDel}} + 1 \ , \\ x_s &= \log_2 \left(\frac{z_s - t_c}{c_{RepDel}} + 1 \right) \ . \end{aligned}$$

Computed in this manner, x_s would return a real number, and not the index we would expect. For instance, if we plug in the values from the scenario shown in Fig. 4, the expression would evaluate to 1.58. Applying the *ceil* function, like we do in Eqs. (8) and (9), allows us to compute the actual index of the successfully received message, which in the example is 2. We can derive Eq. (9) in a similar manner.

To avoid overestimating the value of x, we can define the following two support variables:

$$\hat{x}_s = \min(c_{RepMax}, x_s) \ , \tag{10}$$

$$\hat{x}_c = \min(s_{RepMax}, x_c) \ . \tag{11}$$

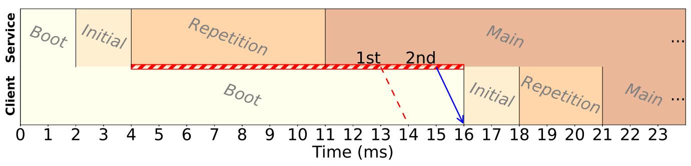


Figure 5: Example showing *main phase* messages that either fail (red dashed line) or succeed (blue arrow) to reach the client.

B. Computing the length of the main phase

Services periodically send *offer* messages during the *main phase*, with a fixed period of s_{CycDel}. There might be a scenario where the client misses all the messages from the service *repetition phase* and can receive only those from the *main phase*, as shown in Fig. 5. As such, we need to compute the number of messages sent during this phase, and given that number, we need to compute its length in time.

Let us start by computing y, as the index of the first periodic message to be successfully delivered during the *main phase*:

$$y = \left\lceil \frac{z_c - (2^{s_{RepMax}} - 1) \cdot s_{RepDel} - t_c}{s_{CycDel}} \right\rceil \ . \tag{12}$$

The z_c tells us the size of the entire *failure window*, which spans over all phases, including the *main phase*; however, here, we are only interested in this last one. From that whole window, we need to remove the length of the *repetition phase*, and the communication delay t_c. What is left is the length of the *failure window* during the *main phase*. If we divide that length by the period with which we send periodic messages, we can compute the index of the first periodic message that is successfully received by the client. To avoid wrong estimates, we can define:

$$\hat{y} = \begin{cases} y & \text{if } (y \geq 0) \wedge (\hat{x}_c \geq s_{RepMax}) \\ 0 & \text{otherwise} \end{cases} \ . \tag{13}$$

Now, we can define a function that computes the timespan from the start of the *main phase* up to when the n-th message is sent:

$$t_{main}(n) = n \cdot s_{CycDel} \ . \tag{14}$$

C. Computing the discovery time

There are three main scenarios we need to consider:

(a) Service sends *offer* messages and client is silent;
(b) Service is silent and client sends *find* messages;
(c) Both service and client send messages.

We compute the discovery time for scenario (a) as follows:

$$t_w^a = s_{init} + t_{rep}(s, \hat{x}_c) + t_{main}(\hat{y}) + t_c \ , \tag{15}$$

which comprises the initial delay, the timespan before the first *offer* messages is successfully received (i.e., $t_{rep}(s, \hat{x}_c)$), the same timespan for the messages during the *main phase* (i.e., $t_{main}(\hat{y})$), and the communication delay t_c.

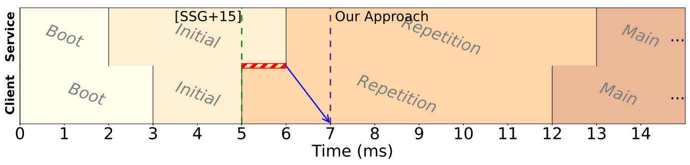
We compute the discovery time for scenario (b) as follows:

$$t_w^b = c_{init} + t_{rep}(c, \hat{x}_s) + t_c + s_{AnsDel} + t_c \ , \tag{16}$$

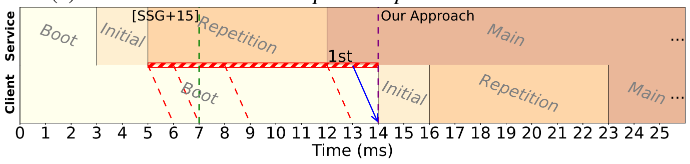
which comprises the initial delay, the timespan before the first *find* messages is successfully received (i.e., $t_{rep}(s, \hat{x}_s)$), the communication delay t_c for the *find* message, the amount of time the service will wait before answering (i.e., s_{AnsDel}), and the communication delay for the answer t_c.

We compute the discovery time for scenario (c) as follows:

$$t_w^c = \min(t_w^a, t_w^b) \ . \tag{17}$$

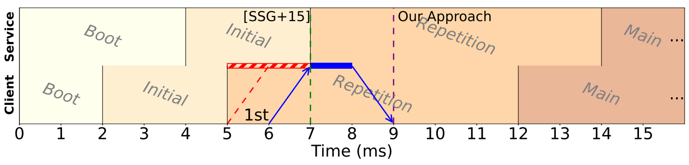


(a) The client enters the *repetition phase* before the service.

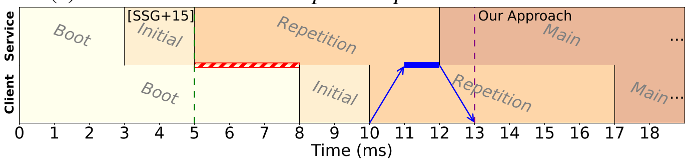


(b) The service enters the *repetition phase* before the client.

Figure 6: The service sends *offer* messages and the client is silent.



(a) The client enters the *repetition phase* before the service.



(b) The service enters the *repetition phase* before the client.

Figure 7: The service is silent and the client sends *find* messages.

IV. Illustrative Examples

This section validates the SOME/IP discovery phase timing analysis proposed in this paper in the three scenarios listed in Section III-C, both numerically and intuitively (or visually) for easy understanding. These examples help uncover the weaknesses in prior studies. Each example (1) provides the configuration required to reproduce the experiments, and (2) compares our results with those computed using the approach in [9], that is referred to as [SSG+15] in Figs. 6 and 7.

For simplicity and without loss of generality, we consider *milliseconds* as the unit of measure throughout the section. All scenarios consider a communication delay t_c=1ms, both client and service have a repetition delay $RepDel$=1ms, and a maximum number of repetition messages $RepMax$=3. Services have a cyclic delay $CycDel$=1ms, and answer requests with a delay $AnsDel$=1ms, the horizontal blue band in the figures. For each scenario, we have two cases: (1) when the client enters the *repetition phase* before the service, and (2) vice versa.

A. The service sends offer messages and the client is silent

In scenario (a), only the service sends *offer* messages. In the first case, shown in Fig. 6a, the message that reaches its destination is the one sent at the end of the *initial wait phase* (i.e., $\hat{x}_c$=0). We compute the discovery latency as follows:

$$\begin{aligned} t^a_w &= s_{init} + t_{rep}(s, \hat{x}_c) + t_{main}(\hat{y}) + t_c \ , \\ &= 6 + t_{rep}(s, 0) + t_{main}(0) + 1 = 6 + 0 + 0 + 1 = \mathbf{7} \ . \end{aligned}$$

The service sends its first *offer* message at the end of the *initial wait phase* at 6ms, which is received at 7ms, after 1ms of delay, ending the discovery phase. The methods proposed in [9], on the other hand, return a latency of 5ms. In the second case, shown in Fig. 6b, the message that reaches its destination is the one sent in the *main phase* with index 1 (i.e., $\hat{x}_c$=3 and $\hat{y}$=1). We compute the discovery time as follows:

$$\begin{aligned} t^a_w &= s_{init} + t_{rep}(s, \hat{x}_c) + t_{main}(\hat{y}) + t_c \ , \\ &= 5 + t_{rep}(s, 2) + t_{main}(0) + 1 \ , \\ &= 5 + ((2^3 - 1) \cdot s_{RepDel}) + 1 \cdot s_{CycDel} + 1 \\ &= 5 + 7 + 1 + 1 = \mathbf{14} \ . \end{aligned}$$

The first message reaching an active client is the one sent in the *main phase* at 13ms, which arrives at 14ms ending the discovery phase. But the approach in [9], returns a discovery latency of 7ms. In both cases, it underestimates the discovery latency because it (1) does not add s_{init} to the discovery time, and (2) z_c assumes negative values.

B. The service is silent and the client sends find messages

In scenario (b), only the client sends *find* messages. In the first case, shown in Fig. 7a, the message that reaches its destination is the one sent in the *repetition phase* with index 1 (i.e., $\hat{x}_s = 1$). We compute the discovery time as follows:

$$\begin{aligned} t^b_w &= c_{init} + t_{rep}(c, \hat{x}_s) + t_c + s_{AnsDel} + t_c \ , \\ &= 5 + t_{rep}(c, 1) + 1 + 1 + 1 \ , \\ &= 5 + ((2^1 - 1) \cdot c_{RepDel}) + 3 = 5 + 1 + 3 = \mathbf{9} \end{aligned}$$

The first message to reach the service is sent at 6ms and received at 7ms after a 1ms delay. Then, the service processes the request for 1ms and sends the response at 8ms, which is received at 9ms after a 1ms delay, ending the discovery phase. But the approach in [9], returns a discovery latency of 7ms.

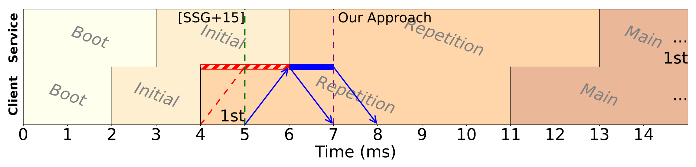
In the second case, shown in Fig. 7b, the message that reaches its destination is the one sent at the end of the *initial wait phase* ($\hat{x}_s = 0$). We compute the latency as follows:

$$\begin{aligned} t^b_w &= c_{init} + t_{rep}(c, \hat{x}_s) + t_c + s_{AnsDel} + t_c \ , \\ &= 10 + t_{rep}(c, 0) + 1 + 1 + 1 = 10 + 0 + 3 = \mathbf{13} \end{aligned}$$

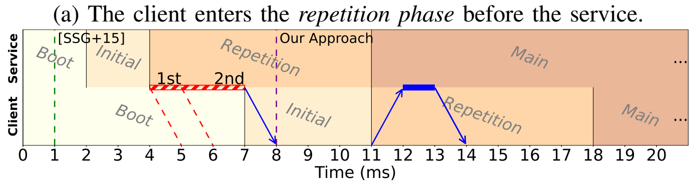
The first message sent by the client at 10ms is received at 11ms after a 1ms delay. The service waits for 1ms before sending the response at 12ms, which reaches the client at 13ms, ending the discovery phase. The approach in [9], returns a discovery latency of 5ms. The underestimations we see here are caused by the fact that it subtracts $c_{BootDel}$ from the computed discovery time. If we add $c_{BootDel}$ to its results, we obtain the correct ones. This also explains why the experiments shown in [9] are correct when $c_{BootDel}$ is set to 0 in all scenarios.

C. Both service and client are sending messages

In scenario (c), both the service and the client are sending messages. In the first case, shown in Fig. 8a, the client can successfully send its message during the *repetition phase* ($\hat{x}_s$=1), while the service with the one sent at the end of the



(a) The client enters the *repetition phase* before the service.



(b) The service enters the *repetition phase* before the client.

Figure 8: Both service and client are sending messages.

initial wait phase ($\hat{x}_c$=0). The discovery time for service and client is computed as follows:

$$\begin{aligned} t_w^a &= s_{init} + t_{rep}(s, \hat{x}_c) + t_{main}(\hat{y}) + t_c \ , \\ &= 6 + t_{rep}(s, 0) + t_{main}(0) + 1 = 6 + 0 + 0 + 1 = \mathbf{7} \ , \\ t_w^b &= c_{init} + t_{rep}(c, \hat{x}_s) + t_c + s_{AnsDel} + t_c \ , \\ &= 4 + t_{rep}(c, 1) + 1 + 1 + 1 \ , \\ &= 4 + ((2^1 - 1) \cdot c_{RepDel}) + 3 = 4 + 1 + 3 = \mathbf{8} \ , \end{aligned}$$

which leads to evaluate the overall discovery time as follows:

$$t_w^c = \min(t_w^a, t_w^b) = \min(7, 8) = \mathbf{7}$$

The service sends its *offer* message at $6ms$, which is received at $7ms$, ending the discovery phase. The client-side discovery activity starts earlier at $5ms$. However, after factoring in $1ms$ of delay, plus $1ms$ of answer delay, and again $1ms$ of communication delay, it completes the discovery procedure at $8ms$. Thus, the discovery latency is $7ms$, which is in line with our estimate above but not with the $5ms$ computed in [9].

In the second case, shown in Fig. 8b, the client successfully sends its first message at the end of its *initial wait phase* ($\hat{x}_s = 0$), while the service with the one sent with index two during the *repetition phase* ($\hat{x}_c = 2$). The discovery time for service and client is computed as follows:

$$\begin{aligned} t_w^a &= s_{init} + t_{rep}(s, \hat{x}_c) + t_{main}(\hat{y}) + t_c \ , \\ &= 4 + t_{rep}(s, 2) + t_{main}(0) + 1 \ , \\ &= 4 + ((2^2 - 1) \cdot s_{RepDel}) + 0 + 1 = 4 + 3 + 0 + 1 = \mathbf{8} \\ t_w^b &= c_{init} + t_{rep}(c, \hat{x}_s) + t_c + s_{AnsDel} + t_c \ , \\ &= 11 + t_{rep}(c, 0) + 1 + 1 + 1 = 11 + 0 + 3 = \mathbf{14} \end{aligned}$$

which leads to evaluate the overall discovery time as follows:

$$t_w^c = \min(t_w^a, t_w^b) = \min(8, 14) = \mathbf{8}$$

Similar to the first case, the fastest to complete the discovery phase is the service by sending its *offer* message at $7ms$, received at $8ms$. As per the protocol, the client should not even start the discovery procedure at $11ms$ because it already knows of the existence of the service at $8ms$. Thus, the discovery time is $8ms$, which corresponds to the value obtained by our model too. However, the estimate obtained in [9] in $1ms$. The discrepancy we see here with [9] is due to a combination of the ones affecting the previous scenarios: (1) it uses $c_{BootDel}$ instead of c_{init}, (2) it subtracts z_c, and (3) it does not add s_{init} to the discovery time. All the proposed analyses shown here are available in a repository as Python code[1]. The repository also contains the code for reproducing the proposed scenarios and parametric sweeps that we used to check the approach's correctness in various configurations.

V. Concluding Remarks

We proposed a formal timing analysis of the *discovery phase* of SOME/IP, which is a well-known automotive service-oriented protocol developed for AUTOSAR applications. SOME/IP allows high customization of clients and services behaviors during the *discovery phase* through a series of parameters. However, such high customization also necessitates a formal timing model for the protocol to be considered and adapted to different scenarios. We validated our analysis on a series of SOME/IP scenarios and compared our results with prior approaches. In the process, we also explained how previous approaches underestimated the discovery latencies in various scenarios.

References

[1] M. Rumez, D. Grimm, R. Kriesten, and E. Sax, "An overview of automotive service-oriented architectures and implications for security countermeasures," *IEEE access*, vol. 8, pp. 221 852–221 870, 2020.

[2] T. Scharnhorst, "Autosar adaptive platform – progress on the software framework for intelligent safe and secure mobility," Keynote at Automotive E/E Architecture Technology China Conference (AEATC), 2018.

[3] P. M. Yomsi, D. Bertrand, N. Navet, and R. I. Davis, "Controller area network (CAN): Response time analysis with offsets," in *9th IEEE International Workshop on Factory Communication Systems*, 2012.

[4] R. I. Davis, A. Burns, R. J. Bril, and J. J. Lukkien, "Controller area network (CAN) schedulability analysis: Refuted, revisited and revised," *Real-Time Systems*, vol. 35, no. 3, pp. 239–272, 2007.

[5] S. Thangamuthu *et al.*, "Analysis of Ethernet-switch traffic shapers for in-vehicle networking applications," in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2015.

[6] D. Thiele, R. Ernst, and J. Diemer, "Formal worst-case timing analysis of Ethernet TSN's time-aware and peristaltic shapers," in *IEEE Vehicular Networking Conference (VNC)*, 2015.

[7] J. Diemer, D. Thiele, and R. Ernst, "Formal worst-case timing analysis of Ethernet topologies with strict-priority and AVB switching," in *7th IEEE International Symposium on Industrial Embedded Systems (SIES'12)*. IEEE, 2012, pp. 1–10.

[8] E. Guttman, "Service location protocol: Automatic discovery of IP network services," *IEEE Internet computing*, vol. 3, no. 4, pp. 71–80, 1999.

[9] J. R. Seyler, T. Streichert, M. Glaß, N. Navet, and J. Teich, "Formal analysis of the startup delay of SOME/IP service discovery," in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2015.

[10] J. R. Seyler, N. Navet, and L. Fejoz, "Insights on the configuration and performances of SOME/IP service discovery," *SAE International Journal of Passenger Cars-Electronic and Electrical Systems*, vol. 8, no. 1, pp. 124–129, 2015.

[11] B. Saydam, "The improvement of SOME/IP service discovery via association rule mining," in *2022 9th International Conference on Electrical and Electronics Engineering (ICEEE)*. IEEE, 2022, pp. 285–289.

[12] C. Richardson and F. Smith. Designing and Deploying Microservices. [Online]. Available: https://www.nginx.com/resources/library/designing-deploying-microservices/

[13] AUTOSAR. Specification of Service Discovery. [Online]. Available: https://www.autosar.org/fileadmin/user_upload/standards/classic/21-11/AUTOSAR_SWS_ServiceDiscovery.pdf

[14] AUTOSAR SOME/IP Service Discovery Protocol Specification. [Online]. Available: \\https://www.autosar.org/fileadmin/user_upload/standards/foundation/21-11/AUTOSAR_PRS_SOMEIPServiceDiscoveryProtocol.pdf

[1]https://github.com/Galfurian/someip_timing_analysis