DEVS OVER MQTT TO ENABLE DISTRIBUTED REAL-TIME SIMULATION

Román Cárdenas[a b], Patricia Arroba[a b], Segundo Esteban[c], and José L. Risco-Martín[c]

[a]Department of Electronics Engineering, Universidad Politécnica de Madrid, Spain
[b]Center for Computational Simulation, Universidad Politécnica de Madrid, Spain
[c]Department of Computer Architecture and Automation, Universidad Complutense de Madrid, Spain

ABSTRACT

This paper presents a novel approach to enabling distributed Real-Time (RT) simulation using the Discrete Event System Specification (DEVS) formalism over the MQTT protocol. DEVS is a well-established framework for modeling and simulating discrete event systems, while MQTT is a lightweight, publish-subscribe network protocol widely used in Internet of Things (IoT) applications. We propose a method integrating DEVS and MQTT to facilitate efficient event propagation across distributed systems, enabling RT interaction and integration with IoT devices. Our approach combines DEVS's hierarchical structure with MQTT's topic-based communication to achieve scalable and flexible environments. We demonstrate the effectiveness of this method through a case study involving a distributed RT simulation of a DEVS model, highlighting the potential for seamless integration with IoT applications. This work contributes to the field of modeling and simulation by providing a practical solution for distributed IoT RT simulations, improving the applicability of DEVS in modern, interconnected environments.

Keywords: DEVS, MQTT, distributed simulation, RT systems, IoT integration.

1 INTRODUCTION AND RELATED WORK

Distributed intelligent Internet of Things (IoT) applications present unique challenges, such as the need for robust, secure, and scalable solutions, and mechanisms for the early detection of misspecifications [1]. Examples include ensuring data privacy, managing heterogeneous device interoperability, and maintaining system reliability under dynamic network conditions [2]. Model-Based Systems Engineering (MBSE) methodologies integrate Modeling and Simulation (M&S) tools as part of the system development process to support the analysis, specification, design, and verification of the system in development [3]. As a result, MBSE improves the quality of the specification and design, reuse of the system specifications, and system evaluation, maintenance, and diagnostics [4].

MBSE relies on mathematical formalisms to provide more robust solutions. Among the different proposed methods, the Discrete EVent System specification (DEVS) formalism stands out as a particularly effective framework for the M&S of discrete event systems [5]. DEVS follows a hierarchical structure to encapsulate the behavioral and structural characteristics of systems using the principles of set theory. The effectiveness of DEVS as a universal modeling formalism has been acknowledged, which facilitates the completeness, verifiability, extensibility, and maintainability of the models [6].

In recent years, Real-Time (RT) simulation of DEVS models has garnered significant attention [7, 8]. The ability to simulate DEVS models in RT implies numerous opportunities, such as Hardware-In-the-Loop (HIL) or Human-In-The-Loop (HITL) simulations, as well as Digital Twins (DTs). For example, there are

Proc. of the 2025 Annual Modeling and Simulation Conference (ANNSIM'25)
May 26-29, 2025, Universidad Complutense de Madrid, Madrid, Spain
J.L. Risco-Martín, G. Rabadi, D. Cetinkaya, R. Cárdenas, S. Ferrero-Losada, and A. Bany Abdelnabi, eds.

works that integrate RT simulation of DEVS models within embedded systems to control a Proportional-Integral-Derivative (PID) system [9] or perform sensor fusion algorithms to increase the reliability of Cyber-Physical Systems (CPSs) [10].

There are research works that use DEVS as the main artifact to aid during the development of IoT applications. For example, Alavi and Wainer [11] model the behavior of IoT applications that use the MQTT protocol to interconnect the elements of the scenario under study. MQTT is a lightweight publish-subscribe network protocol widely used in IoT applications due to its efficiency and low overhead [12]. Their simple yet versatile properties have made MQTT a *de facto* standard in IoT [13].

Despite its potential, the current state of RT simulation in DEVS reveals gaps in the literature, particularly concerning the seamless integration of RT DEVS simulation with external agents [14]. Integrating RT simulations of DEVS models with other systems and technologies can facilitate communication between models, making it easier to work in hybrid scenarios that combine simulated components with real-world elements. This integration is particularly beneficial for applications such as HIL simulations, DTs, and educational tools, where the ability to seamlessly blend simulated and real elements without additional development effort is crucial.

This paper proposes a novel approach to allow distributed RT simulation using the DEVS formalism over the MQTT protocol. Our method combines the strengths of both DEVS and MQTT to create a flexible and scalable simulation environment. We demonstrate the effectiveness of our approach through a case study involving a distributed RT simulation of a DEVS model, highlighting the potential for seamless integration with IoT applications. Taking advantage of the characteristics of the MQTT protocol, our approach allows a smooth transition from a DEVS conceptual model to a real IoT system, enabling hybrid solutions where devices, people, and computer simulations interact without the need to adapt to the particularities of the other elements of the scenario.

The remainder of this paper is structured as follows. Section 2 provides background information on the DEVS formalism and the MQTT protocol. Section 3 details our proposed methodology for integrating DEVS with MQTT, and Section 4 presents a use case that illustrates the practical application of our approach. Finally, Section 5 concludes the paper and outlines the direction of future research.

2 BACKGROUND

This section presents the DEVS formalism and explains how the xDEVS simulation engine supports RT simulation for DEVS models. It also introduces the MQTT protocol, which highlights the principal characteristics that have made it a *de facto* communication standard in IoT applications.

2.1 The DEVS Formalism

In DEVS, the systems are modeled using atomic and coupled models [5]. Atomic models describe autonomous behavior through state transitions and reactions to external events. They are formally defined by the following tuple:

$$AM = \langle X, Y, S, \delta_{int}, \delta_{ext}, \delta_{con}, \lambda, ta \rangle, \tag{1}$$

where X and Y are input and output sets, S is the set of states, and ta is the time advance function. The functions δ_{int}, δ_{ext}, and δ_{con} govern state transitions, while λ defines output events.

Alternatively, coupled models represent systems as interconnected DEVS submodels, defined by:

$$CM = \langle X, Y, C, EIC, EOC, IC \rangle, \tag{2}$$

where C is the set of components, and EIC, EOC, and IC define the coupling relations between inputs, outputs, and internal connections of the components.

A common approach in DEVS modeling is to use ports to classify input and output events [15]. The input set X is divided into subsets, each assigned to a port, forming the input port set P_X. Each input event $x \in X$ is expressed as a tuple $\langle p, x_p \rangle$, where $p \in P_X$ is the port and x_p is the event value:

$$X = \{\langle p, x_p \rangle, p \in P_X\}. \tag{3}$$

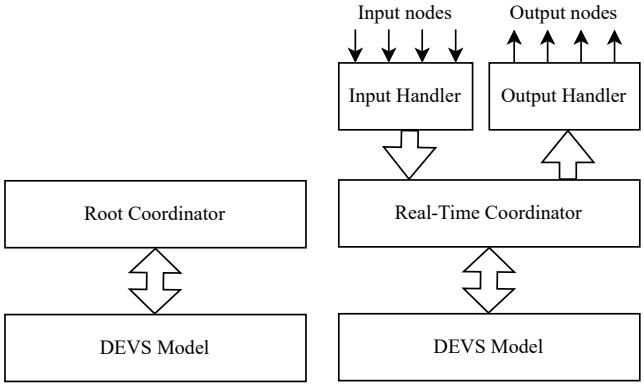
Similarly, the input set X_p for a port p is defined as the set of values for events assigned to that port:

$$X_p = \{x_p | \langle p, x_p \rangle \in X\}. \tag{4}$$

This scheme is also applied to the output set Y and facilitates the definition of coupled models by structuring event propagation through port couplings.

2.2 Real-Time DEVS Simulation in xDEVS

The xDEVS framework [16] has been improved to support RT simulation, allowing DEVS models to interact with external systems in a manner that aligns with real-world time constraints. This enhancement is crucial for applications requiring synchronization with physical processes, such as HIL simulations and DTs. Figure 1 compares the structure of a Virtual-Time (VT) simulation with a RT simulation using the xDEVS simulation engine.



(a) VT simulation. (b) RT simulation.

Figure 1: Comparison between VT and RT simulation on xDEVS.

In the RT simulation architecture shown in Figure 1b, the xDEVS framework introduces three key components: the RT coordinator, the input handler, and the output handler. These components work together to manage the flow of events and ensure that the simulation adheres to RT constraints. The **RT coordinator** is an extension of the traditional DEVS root coordinator shown in Figure 1a designed to handle RT operations, acting as an intermediary between the DEVS models and the external environment. It manages the execution of DEVS state transition functions while coordinating with the input and output handlers. Thus, it is responsible for translating between the virtual time of the simulation and the wall-clock time, ensuring that there is no drift between the two.

The input and output handlers facilitate communication between DEVS models and external systems. The **input handler** receives events from one or more external sources and injects them into the simulation. Injecting input events into xDEVS is implemented as a Multiple Producer Single Consumer (MPSC) queue,

in which the input handler is the consumer and the external event sources are producers that push incoming events to the queue. Alternatively, the **output handler** captures events from the simulation and transmits them to external systems. The propagation of output events in xDEVS corresponds to a broadcast queue, where the output handler pushes output events and the output nodes consume them, acting accordingly to propagate them to external systems.

Figure 2 shows the RT simulation workflow, illustrating how the RT coordinator interacts with the input and output handlers to process events while ensuring that simulation and wall-clock times are synchronized. First, the RT coordinator computes the next expected event in the simulation, *t_until*, which corresponds to the simulation time of the next expected event in the model, *t_next_internal* without exceeding the desired simulation stop time, *t_stop*. Then, the RT coordinator uses the input handler to wait the corresponding wall-clock time for any external input event. The input handler returns the new corresponding virtual simulation time, t. The input handler guarantees that t is always less than or equal to *t_until*. Otherwise, it would be a violation of the DEVS simulation algorithm. Given t and the state of the input ports of the model under study, the RT coordinator can determine whether an internal, external, or confluent transition has occurred. The output handler is run every time after the output function *lambda* to eject output events.

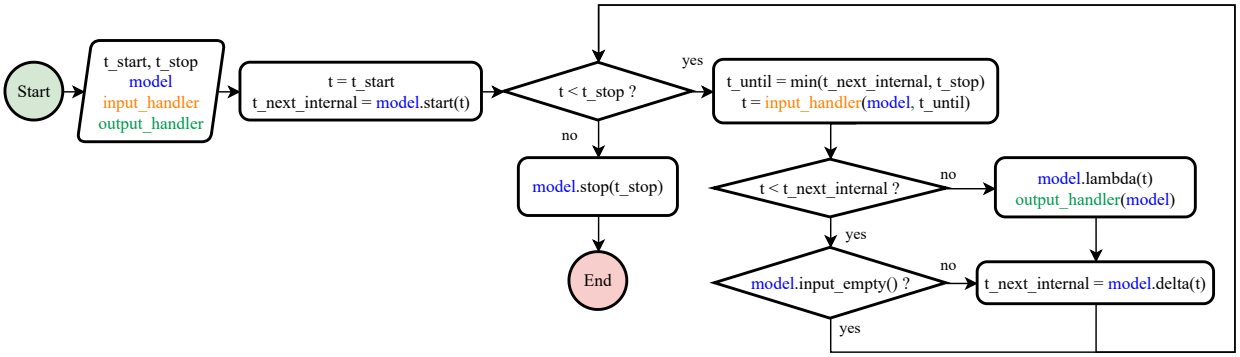


Figure 2: Workflow of RT simulation in xDEVS.

The RT simulation approach implemented in xDEVS has proven to solve previous issues with other proposals (e.g., costly modifications to the original DEVS simulation algorithm or requiring modification of the DEVS model under study), and it is being implemented in other DEVS-compliant simulators [17].

2.3 The MQTT Protocol

The MQTT protocol is a lightweight publish-subscribe network protocol designed for efficient message exchange in constrained environments, such as IoT applications [12]. Operating on top of the TCP/IP protocol suite, MQTT is known for its minimal overhead, making it ideal for scenarios where bandwidth and power consumption are critical. MQTT follows a client-server architecture, where the server is known as the broker, and the clients are devices that communicate through the broker. Communication is based on a publish-subscribe model, where clients publish messages to a topic, and other clients subscribe to these topics to receive messages. This decouples the message sender (publisher) from the message receiver (subscriber), allowing for flexible and scalable communication patterns.

The topics in MQTT are hierarchical, with levels separated by slashes, resembling file paths in a Unix-like system. Clients can subscribe to specific topics or use wildcards to subscribe to multiple topics at once. The "+" wildcard matches a single level, while the "#" wildcard matches multiple levels. Clients can share a subscription to a topic to receive messages in rotation, achieving load balancing. Clients can join a shared

subscription using the topic structure:

$$\$share/\langle GROUPID\rangle/\langle TOPIC\rangle, \tag{5}$$

where \$*share* indicates a shared subscription, $\langle GROUPID\rangle$ is the group, and $\langle TOPIC\rangle$ is the actual topic.

MQTT provides three levels of Quality of Service (QoS) to ensure message delivery according to application needs: QoS 0 (at most once), QoS 1 (at least once), and QoS 2 (exactly once). These levels offer varying degrees of reliability, from best-effort delivery to guaranteed delivery without duplication. Additional features include retained messages, where the broker stores the last message for each topic, and Last Will and Testament (LWT), allowing clients to specify a message that the broker will send if the client unexpectedly disconnects. These features enhance the utility of MQTT in IoT applications, making it a *de facto* standard for communication in such environments [12, 2].

As we demonstrate in this paper, DEVS models can efficiently propagate events across distributed systems using MQTT, enabling RT interaction and seamless integration with IoT devices. This capability is particularly beneficial in scenarios that require low latency and high reliability, such as the proposed DEVS over the MQTT framework.

3 ENABLING DISTRIBUTED REAL-TIME DEVS SIMULATIONS WITH MQTT

This section describes our proposal to use the MQTT protocol as an event propagation channel for distributed RT DEVS simulations. We illustrate this process using the Experimental Frame - Processor (*efp*) model depicted in Figure 3. This model is typical in DEVS academic works, as it is a simple but illustrative example of a DEVS model.

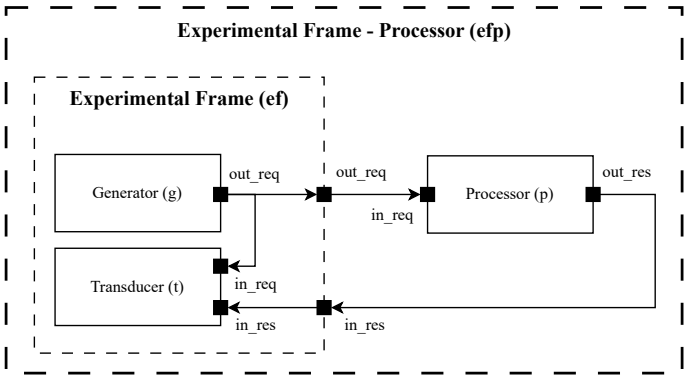


Figure 3: Structure of the Experimental Frame - Processor (EFP) model.

The Generator (g) atomic model sends requests periodically via its *out_req* output port. The Processor (p) atomic model receives these jobs through its *in_req* port and processes them on a first-come, first-served basis. When finished, p outputs the result via its *out_res* port. On the other hand, the Transducer (t) atomic model monitors the requests from g and responses from p via its *in_req* and *in_res* ports, respectively, to measure the processing time required by the p model. The g and t models are arranged in a coupled model called the Experimental Frame (*ef*) model, which, following the notation in (2), can be defined as follows:

$$ef = \langle \{in\_res\}, \{out\_req\}, \{g,t\}, \{\langle in\_res, t_{in\_res}\rangle\}, \{\langle g_{out\_req}, out\_req\rangle\}, \{\langle g_{out\_req}, t_{in\_req}\rangle\}\rangle. \tag{6}$$

Finally, the *ef* and p models are subcomponents of the *efp* model, which is defined as follows:

$$efp = \langle \emptyset, \emptyset, \{ef, p\}, \emptyset, \emptyset, \{\langle ef_{out\_req}, p_{in\_req}\rangle, \langle p_{out\_res}, ef_{in\_res}\rangle\}\rangle. \tag{7}$$

Let us assume that the *efp* model is a DEVS model of an IoT application under development. This application consists of three independent elements (the generator, the processor, and the transducer) that communicate with each other through an MQTT broker. In an early stage of the development process, we implemented the DEVS model of our application and used the xDEVS simulation engine to simulate how the system would behave. After the simulation results show that our system behaves as expected and meets all the system requirements, it is time to develop and integrate all the parts of our system.

In this paper, we propose using an incremental model-driven development process that, instead of jumping from simulation to implementation, provides a set of intermediate steps from computer simulation to the deployment of the final system. This process revolves around using MQTT as the communication means between the elements that make up our solution. Figure 4 represents the proposed approach. In this scenario, there are as many independent RT simulations as elements in our system. For the proposed use case, there are three independent RT simulations for the generator, processor, and transducer. Note that even if the original model has been fragmented into three submodels, these submodels do not need any modification.

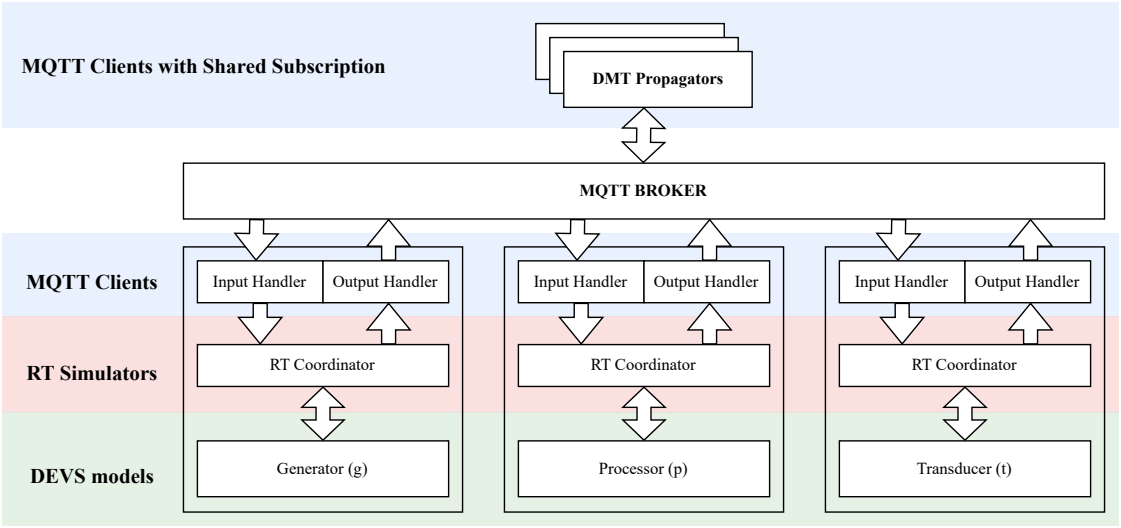


Figure 4: Architecture of the proposed RT simulation over MQTT.

There is one independent simulation per submodel. Thus, in contrast to VT simulation, the propagation of events is not handled by a coordinator. Instead, this task is managed by the input and output handlers. These handlers are clients of the same MQTT broker. Input handlers are subscribers that inject events into their corresponding simulation as they receive messages from the broker. Alternatively, output handlers are publishers that send messages to the broker in response to output events in their respective model. Finally, DEVS Model Tree (DMT) propagators are MQTT clients in charge of republishing messages from source topics to destination topics according to the model hierarchy. These clients are necessary to ensure communication between models while leaving input and output handlers unaware of the rest of the system. The remainder of this section describes these elements in more detail.

3.1 The DEVS Model Tree Notation

If we want to effectively use MQTT as a communication means for distributed RT simulation, we must adopt an idiomatic approach that is natural for MQTT. To achieve this goal, we propose the DEVS Model Tree (DMT) notation. A DMT is a mechanism to describe the topology of a DEVS model as a hierarchical tree, similar to file paths in a Unix-like system. A DMT only shows which components comprise a model, their input and output ports, and the couplings that interconnect them. Thus, it does not specify how these

components behave. Figure 5 illustrates the DMT corresponding to the *efp* model. It is a hierarchical representation of (7) and (6) that explicitly specifies the input and output ports of each component. For coupled models, it also lists their components and their couplings.

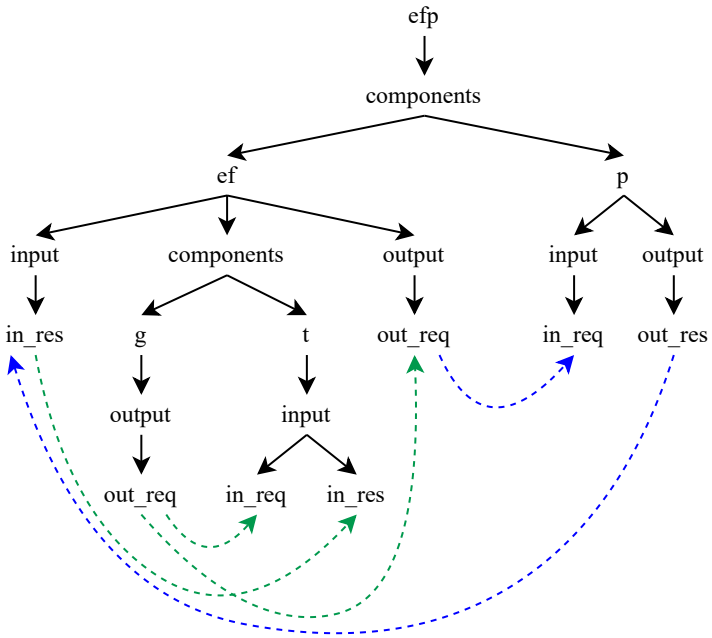


Figure 5: DEVS Model Tree of the EFP model.

At the top of the figure, we see the *efp* model. We refer to this model as the *root* model, mimicking the nomenclature followed in Unix-like file systems. In this case, the root model does not have any input or output port, as shown in (7). Alternatively, it has two components: the *ef* and the *p* models. We can refer to them as *efp/components/ef* and *efp/components/p*, respectively. The *ef* model has one input port, *in_res*, and one output port, *out_req*. We can refer to them as *efp/components/ef/input/in_res* and *efp/components/ef/output/out_req*, respectively.

The DMT terminology follows the topic structure in MQTT. This is convenient from a software development point of view, as it is possible to implement input and output handlers that, given a DMT, can automatically publish and subscribe to MQTT topics without requiring the adaptation of the model under study to the proposed scheme. For example, the input handler for the *p* model would subscribe to the *efp/components/p/input/in_req* topic. Then, every time it receives a message from the MQTT broker, the input handler parses it and injects an input event to the *in_req* port of the *p* model. Alternatively, its output handler would publish to the *efp/components/p/output/out_res* topic a serialized version of all the output events found in the *out_req* port of the model.

DMTs must also specify the couplings between ports. This is required to effectively propagate messages throughout the MQTT topic tree. Figure 5 represents couplings with dashed arrows. The blue arrows correspond to the couplings in the *efp* model, while the green arrows specify the couplings in the *ef* model. In this case, according to the couplings in the *efp* model, messages published in the *efp/components/ef/output/out_req* topic must be propagated to the *efp/components/p/input/in_req* topic. Furthermore, messages published in the *efp/components/p/output/out_res* topic must be propagated to the *efp/components/ef/input/in_res* topic.

DMTs can be serialized in plain text. This is required to develop software that, given a DMT text file, can perform all the actions to enable distributed RT simulation over an MQTT connection. In this work, we propose using the JavaScript Object Notation (JSON) format to describe a DMT. Listing 1 corresponds to the JSON file that describes the DMT of the *efp* model.

Listing 1: DEVS Model Tree of the EFP model represented in JSON format.

```
{
    "name": "efp",
    "model": {
        "components": {
            "ef": {
                "input": ["in_res"], "output": ["out_req"],
                "components": {
                    "g": { "output": ["out_req"] },
                    "t": { "input": ["in_req", "in_res"] }
                },
                "couplings": [ ... ]
            },
            "p": { "input": ["in_req"], "output": ["out_res"] }
        },
        "couplings": [
            {
                "component_from": "ef",  "port_from": "out_req",
                "component_to": "p", "port_to": "in_req"
            },
            {
                "component_from": "p", "port_from": "out_res",
                "component_to": "ef", "port_to": "in_res"
            }
        ]
    }
}
```

3.2 DEVS Model Tree Propagators

DMT propagators are MQTT clients whose job is to propagate messages from one topic to another according to the DMT of the model under study. A DMT propagator parses the DMT JSON file and detects all output ports of atomic components. In Listing 1, the *g*, *p*, and *t* models are considered atomic components, since they only specify the input and output ports. In the case of the *efp* model, there are only two output ports of atomic components corresponding to the *out_req* and *out_res* ports of the *g* and *p* models, respectively. Thus, DMT propagators will subscribe to the *efp/components/ef/components/g/output/out_req* and *efp/components/p/output/out_res* topics to receive all the output events of leaf components.

Next, for each output port of atomic components, DMT propagators infer all the destination ports of the models comprising the system under study. For example, in the *efp* model, the *out_res* port of the *p* model is coupled to the *in_res* port of the *ef* model, which in turn is coupled to the *in_res* port of the *t* model. Given this information, every time a DMT propagator receives a message from the *efp/components/p/output/out_res* topic, it will republish the content of the message in the *efp/components/ef/input/in_res* and *efp/components/ef/components/t/input/in_res* topics. In this way, the input handler of the *t* model will indirectly receive messages from the *p* model. Note that while it is not strictly necessary to publish events in the *efp/components/ef/input/in_res* topic, it is useful for monitoring and debugging the RT simulation.

In a standard MQTT subscription, each subscriber receives a copy of every message published on a topic. In simple models with few couplings and output event rates, a single DMT propagator may be enough to ensure the correct behavior of the proposed solution. However, in more complex scenarios with potentially hundreds of messages per second, a single DMT can become a bottleneck of the system, as well as a single point of failure. To address this issue, we can use MQTT shared subscriptions, described in Section 2.3. Therefore, in the case of the *efp* model, we can run several DMT propagators that subscribe to the *$share/dmtp/efp/components/ef/components/g/output/out_req* and *$share/dmtp/efp/components/p/output/out_res* topics to share the load.

4 USE CASE

This section illustrates how the proposed methodology can be applied to run distributed RT simulations of DEVS models using MQTT as a communication means. We use the previously explained *efp* model as a use case. First, we performed a regular VT simulation of the DEVS model. Then, we ran a RT simulation of the entire model without using input nor output handlers to communicate with external elements. Next, we used the architecture shown in Figure 4 to run a distributed RT simulation of the *efp* model that uses MQTT as a communication means. Finally, we replaced the *p* model with an IoT device that runs a simple program that emulates the expected behavior of this model.

The *efp* model has been implemented using the Rust version of xDEVS [18]. The MQTT broker used in these experiments is rumqttd 0.19.0 [19]. This broker supports shared subscriptions, which is required to illustrate how to perform load balancing with multiple DMT propagators. The code required to reproduce the experiments, as well as the numerical results of the experiments, is available in a public GitHub repository [20]. All elements of the scenarios, including simulations, DMT propagators, and the MQTT broker, were run on the same machine, a MacBook Pro (14-inch, 2023) with MacOS Sequoia 15.2, an Apple M2 Pro CPU, and 32 GB of memory. Communication between elements was performed using the loopback network interface. The only external element is the implementation of the *p* model, which was run on an Espressif ESP32-C6-DevKitM-1-N4 evaluation board [21].

4.1 Experimental Results

Each scenario consisted of a time interval of 60 seconds, after which the simulation ended. During this interval, the *g* model generates a new request every 5 seconds, and the *p* model takes 3 seconds to process a new request. The *t* model monitors the elapsed time since it detects a new request from the *g* model until it receives the response from the *p* model. All the simulations presented on this paper used the exact same model, and only the simulation engine changed from one scenario to another. We repeated all experiments 10 times to get better insight into the execution time. Table 1 gathers the results of all the experiments. This table also shows a confidence interval of the 95 % considering that the samples follow a t-distribution.

Table 1: Simulation results.

	VT	**RT**	**RT + MQTT**	**RT + MQTT + ESP32**
Sim. time (s)	98.57 ± 1.13 µs	60.046 ± 0.001	60.007 ± 0.008	60.004 ± 0.000
Proc. time estimate (s)	3	3	3.003 ± 0.054	3.103 ± 0.057

First, we simulated the *efp* model using a VT simulation. In all simulations, the transducer always measured a processing time of exactly 3 seconds, which corresponds to the specification of the model. Compared to the rest of the scenarios, the simulation time was negligible, with an average execution time of 98.57 µs.

Next, we ran a set of RT simulations of the *efp* model without using any input or output handler. As in the previous case, the transducer always measured a processing time of exactly 3 seconds. However, since these were RT simulations, the average execution time was 60.046 s (i.e., 0.08 % higher than the simulated time).

The third experiment consisted of running a distributed RT simulation of the *efp* model using MQTT as the communication means for the models. The experimental setup is similar to Figure 4. In this case, we ran five independent processes simultaneously: an MQTT broker, two DMT propagators, and three simulations for the generator, processor, and transducer models. Note that the model under study is simple and that a single DMT propagator would suffice in this case. We used two DMT propagators to show that it is possible to balance the load in more complex scenarios. Figure 6 shows the execution traces of the experiment.

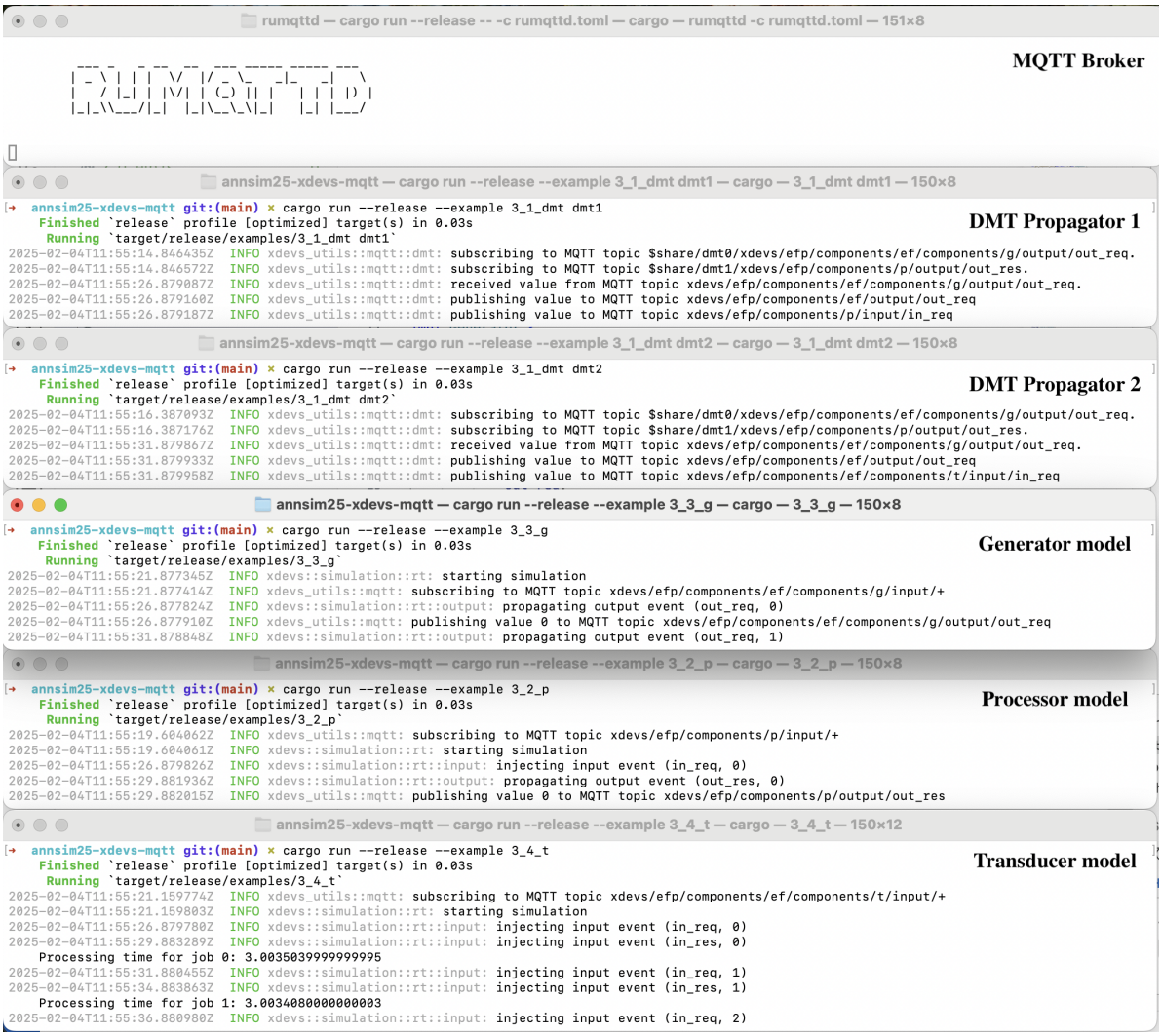


Figure 6: Terminals running a distributed RT simulation of the EFP model over MQTT.

The two DMT propagators in this scenario use a shared subscription to divide the message propagation workload. This is implied in Figure 6, since the topics the DMT propagators are subscribing to start with a *$share/dmtX* prefix. The execution traces also show that DMT propagators receive only half of the messages. For example, DMT propagator 1 receives a first message from the *out_req* port of the generator at t =

11:55:26, while DMT propagator 2 receives a first message from the *out_req* port of the generator five seconds after at *t = 11:55:31* (that is, the next request sent by the generator).

In this case, the transducer model did not capture a processing time of exactly 3 s. Instead, the average processing time captured by the transducer was 3.003 s. This overhead of 0.1 % corresponds to the communication time imposed by the MQTT broker, in which, in contrast with a monolithic simulation, event propagation from one model to another is not 0. This overhead is subject to worsening depending on the quality of the connection between the simulators and the MQTT broker. The average execution time of the transducer model was 60.007 s, 0.01 % higher than the simulated time.

In the last experiment, we replaced the processor model with an IoT device with a similar behavior. The IoT device is an ESP32-C6-DevKitM-1-N4 evaluation board, which is based on the RISC-V Instruction Set Architecture (ISA). Figure 7 shows a schematic of the last experimental setup. This IoT device is an MQTT client connected to the same broker as the rest of the elements (i.e., the generator model, the transducer model, and the two DMT propagators).

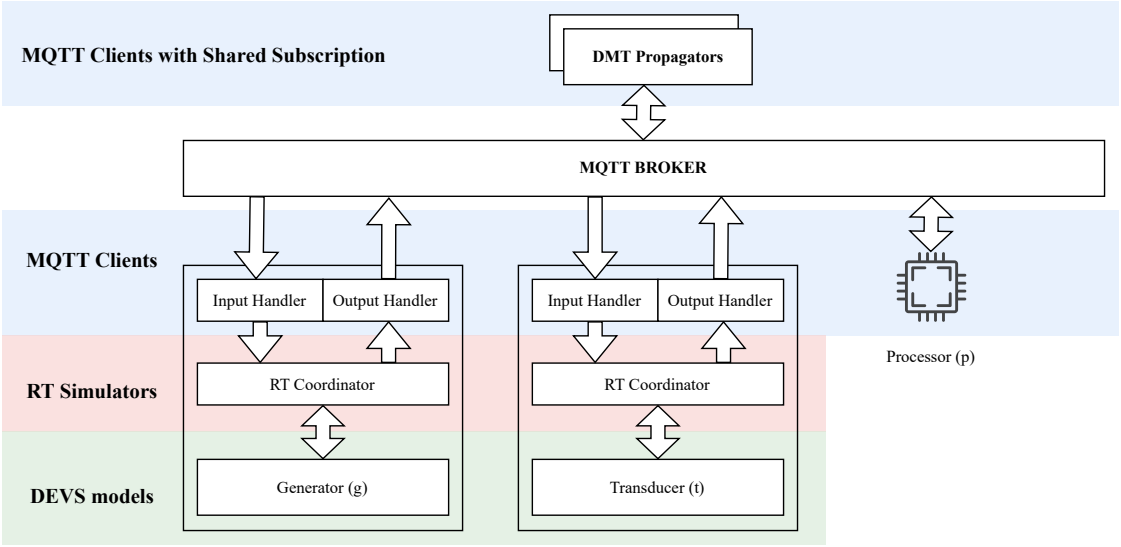


Figure 7: Schematic of RT simulation over MQTT.

The observed behavior was similar to the previous case study in which all the elements were DEVS models. However, the transducer captured an average processing time of 3.103 s. The reason for this 3.43 % increase in perceived response time is again the communication delay between elements. Now, the processor is not a DEVS simulation connected via the loopback network interface. Instead, it is an IoT device that uses WiFi to connect to the same Wireless Local Access Network (WLAN) where the computer runs the simulation and the MQTT broker are.

Although these results are similar to the ideal case simulated in the first VT setup, the importance of connection quality in aspects related to IoT deployments is evident. The preliminary results presented here have been performed in controlled scenarios with low delay and good connection quality. However, a real IoT scenario may face more complex situations, with limited network access and risk of congestion in the MQTT broker. In these cases, the error between the simulation and the real application can be significant, putting at risk the achievement of the technical requirements of a system. We must integrate these aspects in the early modeling phases and study the possible impact of the quality of service of MQTT connections on the behavior of the system in different scenarios.

5 CONCLUSIONS AND FUTURE WORK

In this paper, we present a novel approach to enable distributed RT simulation using the DEVS formalism over the MQTT protocol. Our method leverages the lightweight and efficient nature of MQTT to facilitate the propagation of events across distributed systems, enabling RT interaction and integration with IoT devices and people. Using DMTs and propagators, we demonstrate how to effectively manage communication between distributed DEVS models without requiring modifications to the original models.

The experimental results show that our approach maintains the expected behavior of the DEVS models while introducing minimal overhead due to network communication. The distributed RT simulation over MQTT achieve processing times close to the specified model requirements, with only a slight increase in execution time due to message propagation. Furthermore, replacement of the processor model with an actual IoT device validate the applicability of our method in co-simulated real-world scenarios, achieving similar results in terms of execution and processing times.

Future work will focus on exploring the scalability of the approach in more complex systems with higher message rates. Along the same lines, we will study the effect that MQTT communication characteristics such as QoS levels in subscriptions and publications, the quality of connection between devices, or network congestion can have on the presented proposal. We also plan to investigate the integration of other communication protocols. This work proves the applicability of DEVS in modern interconnected environments.

ACKNOWLEDGMENTS

This work has been supported by the Research Projects SMART-BLOOMS (TED2021-130123B-I00) by MCIN/AEI/10.13039/501100011033 and the European Union NextGenerationEU/PRTR.

REFERENCES

[1] T. Alsboui, Y. Qin, R. Hill, and H. Al-Aqrabi, "Distributed Intelligence in the Internet of Things: Challenges and Opportunities," *SN Computer Science*, vol. 2, no. 4, p. 277, Jul. 2021. [Online]. Available: https://link.springer.com/10.1007/s42979-021-00677-7

[2] L. Atzori, A. Iera, and G. Morabito, "The internet of things: A survey," *Computer networks*, vol. 54, no. 15, pp. 2787–2805, 2010.

[3] A. W. Wymore, *Model-Based Systems Engineering*, 1st ed. CRC Press, May 2018. [Online]. Available: https://www.taylorfrancis.com/books/9781351431095

[4] M. Russell, "Using MBSE to enhance system design decision making," *Procedia Computer Science*, vol. 8, pp. 188–193, 2012.

[5] B. P. Zeigler, A. Muzy, and E. Kofman, *Theory of modeling and simulation: discrete event & iterative system computational foundations*. Academic Press, 2018.

[6] H. Vangheluwe, "DEVS as a common denominator for multi-formalism hybrid systems modelling," in *Proceedings of the IEEE International Symposium on Computer-Aided Control System Design*. Anchorage, AK, USA: IEEE, 2000, pp. 129–134.

[7] B. Earle, K. Bjornson, C. Ruiz-Martin, and G. Wainer, "Development of A Real-Time DEVS Kernel: RT-Cadmium," in *2020 Spring Simulation Conference (SpringSim)*, 2020, pp. 1–12.

[8] O. Fernandez-Sebastian, R. Cardenas, P. Arroba, and J. L. Risco-Martin, "A Novel Real-Time DEVS Simulation Architecture with Hardware-In-The-Loop Capabilities," in *2024 Annual Modeling and Simulation Conference (ANNSIM)*, 2024, pp. 1–13.

[9] M. Shahmohammadimehrjardi, G. Wainer, M. Wu, and X. Zhang, "PID Control Using Quantized State Systems in RT-DEVS Frameworks," in *2023 Annual Modeling and Simulation Conference (ANNSIM)*, 2023, pp. 171–183.

[10] G. Wainer, J. Boi-Ukeme, and V. Paranjape, "Sensor Fusion DEVS for Angle Estimation on Inertial Measurement Unit," in *2023 Winter Simulation Conference (WSC)*, 2023, pp. 2825–2536.

[11] I. Alavi Fazel and G. Wainer, "Discrete Event System Specification for IoT Applications," *Sensors*, vol. 24, no. 23, 2024. [Online]. Available: https://www.mdpi.com/1424-8220/24/23/7784

[12] M. Bender, E. Kirdan, M.-O. Pahl, and G. Carle, "Open-source MQTT evaluation," in *2021 IEEE 18th Annual Consumer Communications & Networking Conference*. IEEE, 2021, pp. 1–4.

[13] N. Naik, "Choice of effective messaging protocols for IoT systems: MQTT, CoAP, AMQP and HTTP," in *2017 IEEE International Systems Engineering Symposium (ISSE)*, 2017, pp. 1–7.

[14] G. A. Wainer and P. J. Mosterman, *Discrete-event modeling and simulation: theory and applications*. CRC press, 2018.

[15] R. Cárdenas, "Integrative Modeling, Simulation, and Optimization Techniques for Efficient Data-Intensive Applications in Edge Computing Infrastructures," Ph.D. dissertation, Carleton University; Universidad Politécnica de Madrid, 2024.

[16] J. L. Risco-Martín, S. Mittal, K. Henares, R. Cardenas, and P. Arroba, "xDEVS: a toolkit for interoperable modeling and simulation of formal discrete event systems," *Software: Practice and Experience*, pp. 1–42, 2023.

[17] S. Govind and G. Wainer, "Handling Asynchronous Inputs in DEVS Based Real-Time Kernels," in *2024 Winter Simulation Conference (WSC)*, 2024, pp. 2277–2288.

[18] R. Cárdenas, "xdevs.rs: Version of the xDEVS simulator for Rust projects," https://github.com/iscar-ucm/xdevs.rs, 2025, accessed February 4, 2025.

[19] Bytebeam, "rumqtt: the MQTT Ecosystem in Rust," https://github.com/bytebeamio/rumqtt, 2025, accessed February 3, 2025.

[20] R. Cárdenas, "Distributed Real-Time Simulations for ANNSIM'25," https://github.com/iscar-ucm/annsim25-xdevs-mqtt, 2025, accessed March 20, 2025.

[21] Espressif Systems Co., "ESP32-C6-DevKitM-1," https://docs.espressif.com/projects/esp-dev-kits/en/latest/esp32c6/esp32-c6-devkitm-1/user_guide.html, 2025, accessed February 4, 2025.

AUTHOR BIOGRAPHIES

ROMÁN CÁRDENAS is an Assistant Professor at Universidad Politécnica de Madrid (UPM), Spain. He obtained a Ph.D. in Electronic Systems Engineering in Cotutelle modality at UPM and Carleton University. His research interests include M&S in the IoT domain. His email address is r.cardenas@upm.es.

PATRICIA ARROBA is an Associate Professor at Universidad Politécnica de Madrid (UPM), Spain, where she received her Ph.D. in Telecommunication Engineering. Her research interests include energy and thermal-aware M&S&O for Cloud and Edge infrastructures. She can be reached at p.arroba@upm.es.

SEGUNDO ESTEBAN is an Associate Professor in the Department of Computer Architecture and Automation at Complutense University of Madrid (UCM). He received his Ph.D. in Physics from the UCM in 2002. His research is focused in practical aspect of modeling, simulation and control of dynamic systems. His email address is sesteban@ucm.es.

JOSÉ L. RISCO-MARTÍN received his Ph.D. from Universidad Complutense de Madrid (UCM), Spain, where he currently is a Full Professor. His research interests include computer-aided design and modeling, simulation, and optimization of complex systems. His email address is jlrisco@ucm.es.