# DEVS BASED ROBUST COMMUNICATION PROTOCOL FOR INTER-SIMULATION COMMUNICATION IN CADMIUM

Sasisekhar Govind[a] and Gabriel A. Wainer[a]
[a]Advanced Real-Time Simulation Laboratory, Carleton University, Canada
*sasisekharmangalamgo@cunet.carleton.ca, gwainer@sce.carleton.ca*

## ABSTRACT

Intercommunication between processes in a distributed hard real-time system have to meet extremely tight timing deadlines. One of the bottlenecks in the communication process is the protocol used within the system. This paper presents a formal definition and implementation of a communication protocol that is reliable and predictable for use with distributed hard real-time systems and by extension, distributed real-time simulations. The protocol presented in this paper was modeled using the Discrete Event System specification and simulated using the Cadmium simulator. We present a case study wherein we distribute a centralized model using the protocol. Upon simulation, the behavior of the system before and after distribution were observed and their congruence determined.

**Keywords:** DEVS, distributed systems, RT-DEVS, communication protocol, distributed simulation.

## 1 INTRODUCTION

Hard real-time (RT) distributed systems are decentralized over multiple inter-connected compute nodes, wherein, each node runs various processes, each of which need to be completed within an explicit deadline. These processes may or may not be dependent on the output produced by another node within the system. When two processes running on different nodes are inter-dependent on each other, one of the prime factors that obstruct the system's capability of meeting the deadlines is the communication network employed within the system. One of the major steps to be taken to improve the communication network, would be to develop a robust, deterministic, and reliable communication protocol.

In the industry, where hard RT distributed systems are prevalent, this issue was overcome by using simple yet robust communication protocols, like the Fieldbus or CAN protocols. However, since Ethernet has become the standard communication protocol for businesses and educational institutions, the cost of setting up an Ethernet-based communication network has dropped significantly. With the infrastructure widely and cheaply available, there is incentive to move forward from outdated protocols like RS485[1].

However, Ethernet as a protocol was not developed with hard RT communication in mind. Hence, there is a need to slightly modify the protocol stack to mold it to the needs of a hard real time distributed system. The authors in [3] go into detail about the various paths to be taken to develop such an RT protocol by removing or modifying a few layers of the Ethernet communication stack.

The modification of the Ethernet protocol allows us to establish RT communication between various nodes in a distributed system. Nevertheless, the development of a communication protocol that can transfer data deterministically and reliably requires the developer to follow a rich modelling formalism. Modelling allows the developer to simulate the protocol per the environment it will be deployed in, enabling the developer to verify and validate the performance of their protocol. This would allow for the development of a deterministic and reliable protocol, wherein, the developer is completely aware of the possible

*Proc. of the 2024 Annual Simulation Conference (ANNSIM'24), May 20-23, 2024, American University, DC, USA*
*C. Ruiz-Martin, B. Oakes and R. Cárdenas, eds.*

behaviors of their model, hence enabling one to develop a protocol that is not only real-time, but also predictable in its faults.

This thought process can be extended towards the development of complex distributed systems. As systems increase in complexity, modelling the same leads to simpler and broader understanding of the system, hence the prevalence of UML diagrams etc. Despite this, there always seems to exist a disconnect between the model and the implementation of the system. Hence, this leads us to find a rich modelling formalism that can accurately and simply describe the behavior and structure of a system, so that the disconnect between implementation and modelling can be completely neglected [4].

One such modelling formalism is the Discrete Event System Specification (DEVS), which deals with systems that are continuous in time, but expect discrete events [5]. Modeling and Simulation Based Engineering (MSBE) is a methodology that draws on the DEVS formalism to create a development paradigm for embedded and cyber physical systems [6].

This paper aims to portray the development of a robust, deterministic, and reliable communication protocol based on the DEVS formalism to facilitate communication between different DEVS models running on different execution nodes. The DEVS-Inter Atomic Communication (DEVS-IAC) protocol borrows wisdom from the Ethernet communication stack, and formalizes the layers using the DEVS formalism, improving and modifying the protocol to fit the needs of DEVS models, and implementing the same using MSBE in a tool called Cadmium, on the ESP32S3 Microcontroller platform.

This paper is divided into four sections. Section 2 goes in depth regarding the previous works done in this field, and other background information necessary for understanding the methodology. Section 3 defines the DEVS-IAC protocol, its various layers, and its formal definition in DEVS. Section 4 provides a case study that implements the DEVS-IAC protocol and produces results based on various simulations. Finally Section 5 concludes the paper with relevant discussions and future work pertaining to the research presented in this paper.

## 2 RELATED WORKS

As mentioned earlier, the author of [1] describes multiple reasons as to why the industry should move on from Fieldbus protocols to protocols like Ethernet. One of the main incentives is the abundance of Ethernet infrastructure available as well as scenarios wherein the plants are physically separated over a wide geographical area. Adoption of a heavily standardized protocol like Ethernet allows the implementation of wide area networks which would prove extremely useful for several use cases like remote supervisory control. The authors of [15] talk about this case scenario in a real-time co-simulation of models geographically distributed over multiple laboratories in Europe. The final solution involved using the Internet as an interface to transmit data amongst the various servers performing the simulation.

However, for hard RT systems, Ethernet has a substantial number of ‘layers’ for it to be feasible as a RT communication protocol [3]. The Open Systems Interconnection (OSI) Basic Reference Model or the ISO 7498 standard provides a framework for developing communication protocols [11]. Each layer preforms a very specific task. The physical layer is responsible for converting the binary data (1s and 0s) into mechanical, electrical, or functional representations that can be decoded by the physical layer of another system. The prevalent physical layer protocols are Low Voltage Differential Signaling (LVDS), Unipolar encoding, Manchester encoding etc.

LVDS is the physical layer protocol used in protocols like HDMI [12], USB, and later versions of Ethernet physical layer protocols like 100Base/T. But, since this protocol is aimed to work for all embedded platforms, Manchester encoding was chosen as the physical layer protocol. Manchester encoding itself was part of the Ethernet stack IEEE 802.3-2002/05 [2], and has better performance in relation to power spectral density as compared to a protocol like Unipolar encoding[13]. For these reasons, in this paper, we have used Manchester Encoding as our physical layer protocol.

This research implements two more layers apart from the physical layer, the Data Link layer (DLL) and the Network layer (NL). The DLL is responsible for error detection and ensuring the physical layer gets the appropriate amount of data bits. Further, the DLL also ensures that there is no collision in the between packets sent by different transmission nodes. The system implemented in this research is not in an environment that will lead to collision, and hence this function is ignored. This is one of the modifications to be made to Ethernet. The Ethernet protocol implements Carrier Sense Multiple Access/Collision Detection (CSMA/CD) which is part of its DLL and should be removed for RT purposes. The authors in [3] explain how to modify the protocol to remove CSMA/CD.

The Network layer is responsible for dividing incoming data into appropriately sized packets and to route these packets efficiently in the physical network. Looking back at the Ethernet protocol, this layer along with the Transport layer comprises the TCP/IP protocol stack. However, this paper implements a system that will not be affected by various routing methods, and hence the main task of the network layer in DEVS-IAC is to packetize the incoming data into byte sized chunks that the DLL can work with.

Moreover, apart from modifying this protocol as is suggested by [3] in this research we follow a MSBE approach defined in [9]. This methodology seamlessly incorporates the formal rigor of modelling and simulation into the development process of embedded systems. As mentioned earlier, modelling a system before implementation allows one to predict, verify and validate the behavior of a system before implementing it in the real world. The foundation of the MSBE methodology is the DEVS formalism [6].

DEVS is a modelling formalism devised for analyzing the performance of continuous time, discrete event, dynamic system. DEVS is composed of two variants of models, *Atomic* and *Coupled*. The *Atomic* model presents the base behavior of a small part of the entire system while a *Coupled* model composed of other *Atomic* and/or *Coupled* models that describe the system behavior at a higher level [5]. A description of *Atomic* Models can be found in Appendix 1.

These formal specifications can be used by the modeler to develop models that can be simulated using any DEVS simulator. In this research the authors have used the Cadmium simulator. Cadmium is a header only C++ library that can be used to develop DEVS models and provides a framework for simulating the same [6]. Real-time Cadmium is a branch of Cadmium that enables one to develop DEVS models that are implementable on an embedded platform [8].

Various research efforts point towards formalizing the distributed system. The authors of [13] and [16] use the concept of Functional Mockup Interface (FMI) for Co-Simulation (CS). They approach distributed simulation by creating FMI compliant simulation Units (FMU) for CS. The FMU-CS units are generated for all the models across all the simulation platforms, and then simulated using the High-Level Architecture (HLA) standard. This simulation using HLA, however, is conservatively time stepped, the authors call it Adapter-based Hybrid Federate (A-HF). This implies that the A-HF method of distributed simulation cannot be used for accurately simulating continuous time systems, where an event may occur at any point of time. Hence in our research, the usage of DEVS allows us to implement distributed simulation while maintaining event-based simulation, further, since the entire simulation framework is still in Cadmium, the distributed simulation can be run in RT.

Going back to [15], we see that one of the main issues that they faced when running a distributed simulation over a wide geographical area was latency. Although this paper does not directly solve the problem of latency, it acts as a foundation for the work presented in [17].

## 3 METHODOLOGY

### 3.1 Communication Protocol

As discussed earlier, we have defined a communication protocol that is loosely based on the layer system described by the OSI model. The DEVS-IAC protocol includes three major layers: Physical (PHY), DLL

and NL. The following sub-sections describe the definition and implementation of each of these layers. Each layer includes their own Protocol Data Unit (PDU) that consists of the payload (that is, the data itself) and the header (which contains information that is useful for transmission and reception). The header fields contain useful information that help to deconstruct the data to form the PDU, or to reconstruct the data from the PDU and format it such that it is comprehensible by the layers upstream and/or downstream. Layer 1 is similar to the NL, but it also includes some tasks originally defined in higher layers in the OSI model. This layer takes incoming data, regardless of datatype and converts it into a vector of bytes. This vector, along with the initial size of the data (in case the data is not byte aligned) is stored as its PDU. This PDU is transmitted to layers downstream. When this PDU is received from a downstream layer, the vector of bytes, along with the size parameter, is used to reconstruct the data in memory. This allows Upstream blocks in the design to send and receive data of any datatype without hinderance.

Layer 2 is similar to the DLL in that it divides the upstream data into packets that the following layer (physical) can understand. This layer computes a checksum of the entire data so that the corresponding layer in the receiver's end can ensure error free transmission.

The length/ size of the data coming from the layer upstream is often vastly different from the expected data length of the PHY downstream. The PHY expects 64 bits of data at its upstream input, hence, it falls upon this layer to divide the data bytes into 'Frames'. Each frame contains the data itself, of 32 bits, and relevant metadata, called the header, for reconstruction of the initial data sequence.

Since the input data is grouped into frames, 'Number of Frames' and 'Current Frame Number' are two parameters that are required to reconstruct the original bitstream upon reception. The total number of frames can be calculated as:

$$N = \left\lceil \frac{n}{32} \right\rceil, \tag{1}$$

where:

- $N$ is the total number of frames
- $n$ is the length of the byte vector received from the upstream layer

However, if the data received from the upstream layer is less than 32 bits wide, only a single frame would be required to transmit the data but will have to be padded with 0s to meet the 32-bit width constraint imposed by this layer's PDU specification. This padding would have to be undone when re-constructing the byte vector upon reception. Instead of assigning more bits of the header to indicate size before padding, a singular bit is used as a flag to select if the frame number field would contain the field number or the data length before padding.

A checksum calculation is performed by adding up each bit of the PDU, including the various header fields. Upon reception, this layer recalculates the checksum for the PDU and compares it to the checksum in the header field. This allows the system to ensure error-free data transmission. This can be formalized as:

$$C_i = \sum_{j=0}^{31} D_{ij} + \sum_{j=0}^{4} f_{ij} + \sum_{j=0}^{4} N_j + f\_sel_i \Big| \forall\, i \in [0, N], \tag{2}$$

where:

- $N$ is the total number of frames.
- $C_i$ is the checksum of the $i^{th}$ frame.
- $D_{ij}$ is the $j^{th}$ bit of the data in the $i^{th}$ frame.
- $f_{ij}$ is the $j^{th}$ bit of the frame number in the $i^{th}$ frame ($f_i = i$).
- $N_j$ is the $j^{th}$ bit of N.
- $f\_sel$ is the frame number/data length select bit.

The layout of these fields and how they come together as a single frame is represented by the diagram in Figure 1.

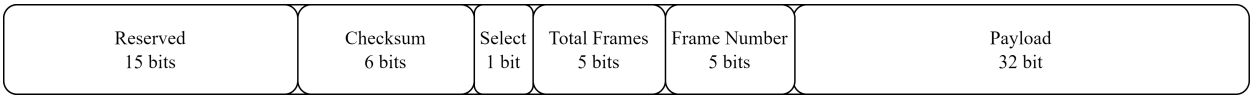


Figure 1: Layer 2 PDU.

Figure 1 shows the layout of the Layer2 PDU called Frame. Going from right to left, we see the fields Payload, Frame number, Total frames, Select, Checksum and Reserved, with each of their bit widths in parenthesis.

The Physical Layer is responsible for converting binary data symbols (1s and 0s) into tangible representations in the real world. For example, the simplest physical layer would assign different voltage levels to the two binary symbols, say 5 volts for binary 1 and 0 volts for binary 0. Although this simple representation would suffice for most low-speed applications, it is plagued with a plethora of disadvantages that make it unusable for the implementation at hand.

Popular physical layer protocols include LVDS, 100Base-t, 10Base-t (IEEE 802.3-2002/05)[2]. These protocols were evaluated as mentioned in previous sections; and Manchester encoding was chosen as the appropriate physical layer protocol for implementation on the ESP32.

The RMT peripheral of the ESP32 provides encoder and decoder templates that define the symbol parameters for each bit. A symbol, as described above, is the physical representation of a binary digit. Hence, per the template, an encoder and decoder were implemented to following the guidelines of Manchester coding. The presence of this peripheral allows the developer to transfer the message to the transmit buffer, after which point, the hardware peripheral transmits it per the encoder without requiring extra CPU cycles, freeing it up for execution.

Multiple transmission rates and packet widths were tested and finally, it was decided that 64 bits, with 200 nano seconds per bit gave the best Bit Error Ratio (BER). BER is defined as the ratio of erroneous bits to the total number of bits.

### 3.2 Modelling

The communication protocol was implemented using the DEVS methodology in the Cadmium simulator. The Cadmium V2 simulator (as mentioned in the previous section) supports real time simulation and execution on various hardware that supports C++. Modification to a few parts of the simulator [7] allows one to simulate/execute the models directly in hardware. Further details on the modelling framework is elaborated by the authors of [18]. The implementation of each of these layers in the models using the DEVS formalism is described in this subsection.

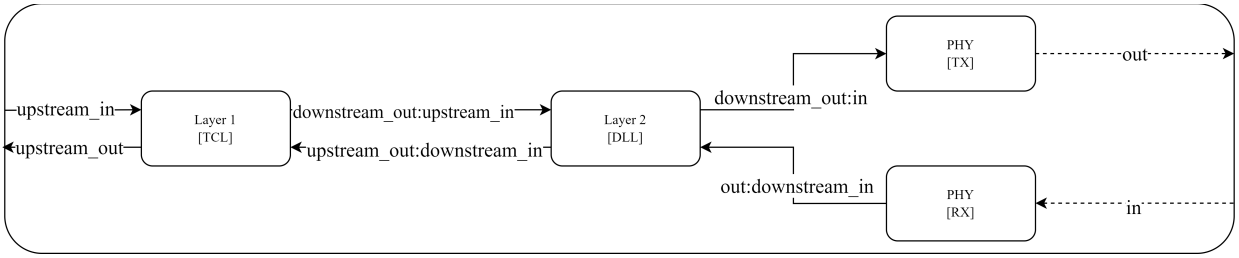


Figure 2: DEVS graph of the communication model.

Figure 2 shows the block diagram of the communication protocol implemented as a DEVS model. Going from left to right, we see that Layer 1 has an input named upstream_in and an output named upstream_out. These inputs and outputs (I/Os) communicate with atomic/ coupled models that lay outside the protocol. Layer 1 also has two more I/Os to the right; downstream_out that goes to upstream_in of Layer 2 and downstream_in that comes from upstream_out of Layer 2. As we proceed, we see Layer 2, and then the two remaining I/Os of Layer 2, that is downstream_out and downstream_in. Further, these I/Os are connected

to the I/Os of the two PHY models. The Transmit PHY, labelled PHY with TX in square brackets has an input port ‘in’ that collects data from downstream_out of Layer 2 and an ‘out’ port that transmits data into the real world. Finally, we have PHY [RX] with an output port ‘out’ that connects to the input ‘downstream_in’ of Layer 2 and an input port ‘in’ that collects data from the real world.

When observing Figure 2 we see that there is a contradiction between the definition explained earlier and the model implementation in DEVS of the PHY layer. The PHY layer has been broken down into the transmit module and receive module PHY [Tx] and PHY [Rx] respectively, since, in many use cases, only a simplex connection, that is, one way communication, is required. Splitting the functionality of PHY allows developers to deploy the appropriate atomic model per their requirement.

The formal definition of Layers 1, 2 and PHY are found in Appendix 2.

## 4 CASE STUDY: TRAFFIC LIGHT CONTROLLER

This case study describes a simple use case for the communication protocol described above. This section will go into detail regarding the implementation of the traffic light model on a single instance of the simulator, and then showcases the same over multiple instances of the simulator. This distributed implementation combines the traffic light model with the DEVS-IAC model. These instances are on the ESP32S3 microcontroller. The development of these models was done using Cadmium V2, compiled on the GNU-C++ compiler: g++ (Ubuntu 13.2.0-4ubuntu3) 13.2.0.

A simple traffic light model can be created using a multitude of modelling paradigms. We have focused on formalizing the traffic light model using DEVS. The traffic light model has three states, namely Red, Amber and Green. Each of these states have a time interval after which it expires, called lifetime (LT). Through observation, we are aware of the state transition cycle, we have assumed that our traffic light transitions from Red to Amber to Green. With these observations and assumptions, we can formalize the system using the following DEVS graph.

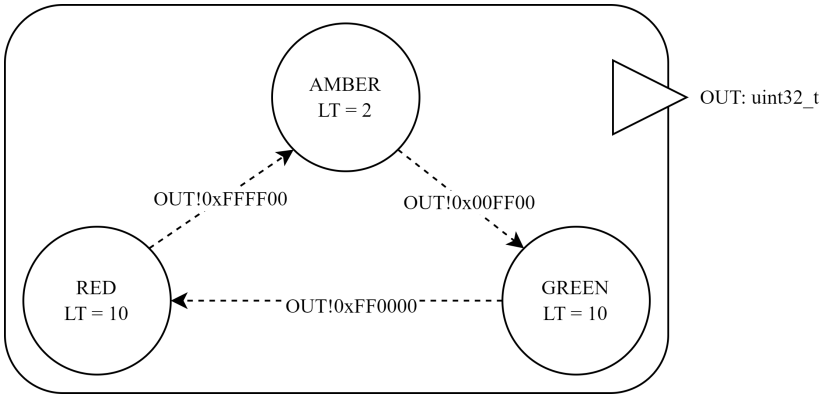


Figure 3: Definition of the Traffic Light atomic model.

Figure 3 portrays the DEVS-graph of the traffic light atomic model. We can observe the three state bubbles Red, Amber, and Green, with the lifetime of each, 10, 2, and 10 respectively, being denoted within the bubbles. We see the dotted lines between the bubble that represent an internal transition and its corresponding output after the expiration of LT. The line also represents the HEX values of the color that would be produced at the output port ‘OUT’ on the right-hand side of the diagram. The OUT port is shown to output data of type uint32_t which is a C++ data type that stores unsigned 32-bit wide integers. The hex color codes are 24-bit wide and hence, a 32-bit wide datatype is the smallest integer datatype that can hold the hex color codes.

This model was implemented in Cadmium and the results of this simulation can be found in later sections. For implementation on a microcontroller, a separate atomic model is created that takes the color codes as

input and produces the appropriate color on a Red-Green-Blue Light Emitting Diode (RGB LED) connected to the microcontroller. This system was then simulated on the ESP32 microcontroller, and the simulation results of this implementation is provided in the upcoming sections.

This traffic light model was then extended using the DEVS-IAC protocol that was previously discussed, to distribute this model among different microcontrollers. The distributed system would constitute two nodes: a transmitter that calculates the current traffic light state and transmits the current color code, and a receiver that receives that color code and outputs onto an RGB LED.

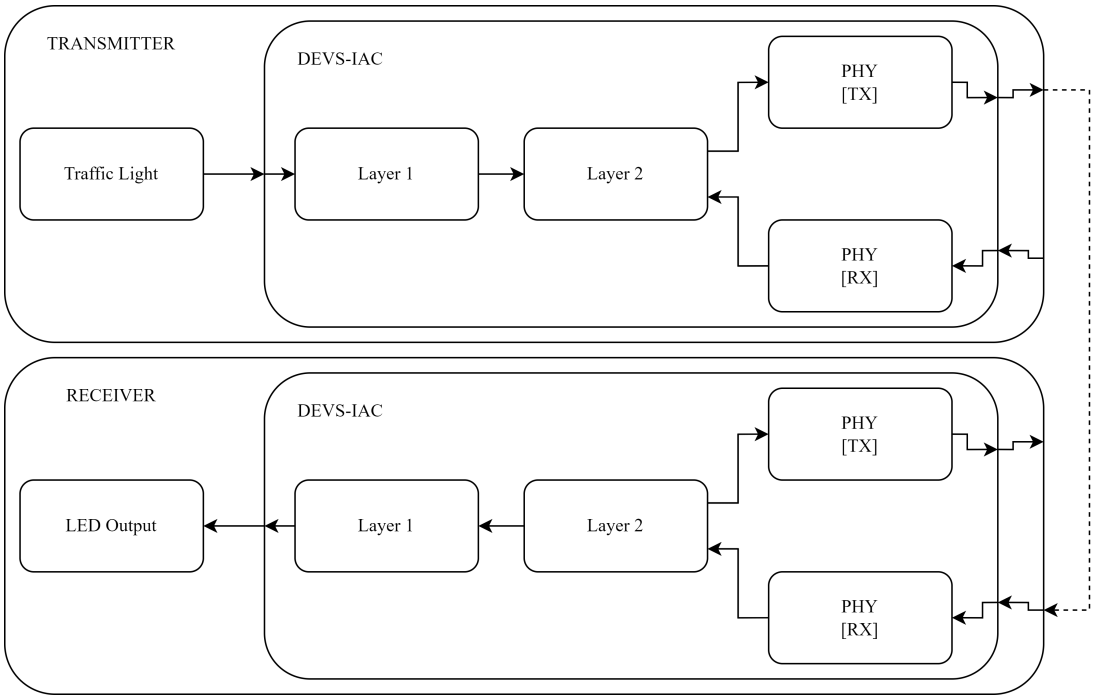


Figure 4: Traffic light in distributed fashion.

The top figure in Figure 4 portrays the transmitter end of the distributed system. From left to right, we observe the traffic light model (as *Traffic Light*) itself that was defined previously in Figure 3. Further, we see that the output of *Traffic Light* goes directly to the input of the *DEVS-IAC* coupled model. We observe that the atomic models present within this coupled model, that is, *Layer 1*, *Layer 2* and *PHY [TX]* are the same as those that are present in the DEVS-IAC protocol discussed earlier. The bottom figure in Figure 4 portrays the receiver end of the distributed system. From right to left we see the *DEVS-IAC* coupled model with the *PHY [RX]*, *layer2* and *layer1* atomic models. These models correspond to the layers in the DEVS-IAC described in the previous sections. The output of the *DEVS-IAC* coupled model goes to the *LED Output* model that is an instance of the led_output atomic specification. The transmitter and receiver are implemented on two ESP32S3 microcontrollers. The microcontrollers were then linked using a copper cable. The pins that transmit and receive the signals were defined in the models.

The centralized traffic light simulator was run on the ESP32S3 Microcontroller. The output log of the simulation is shown in Table 1. We can observe that the simulation log has five parameters. The Time in seconds, the Model ID, the Model Name, the Port Name and Data. The time field denotes the time of simulation when the snapshot was taken. In the case of real time simulation, it also gives us a comprehensive understanding of how long the simulation has been running. We see that this simulation was run for 22.001s. The Model IDs and Model Names field displays the model ID and name provided by the model developer. The Port name displays the Port at which the output is present at that instant. This field is only populated when the $\lambda$ of that model has been called. The Data field displays either the data present at the output ports or the current state of the system.

Table 1: The Centralized traffic light model simulation log.

| **Time (s)** | Model ID | Model Name | Port Name | Data |
|---|---|---|---|---|
| 0 | 1 | led_output | | State = 1 |
| 0 | 2 | traffic_light | out | 0x200000 |
| 0 | 2 | traffic_light | | Next State = 0x002000 |
| 0.001 | 1 | led_output | LED | RED |
| 0.001 | 1 | led_output | | State = 0 |
| 10 | 1 | led_output | | State = 1 |
| 10 | 2 | traffic_light | out | 0x002000 |
| 10 | 2 | traffic_light | | Next State = 0x201000 |
| 10.001 | 1 | led_output | LED | GREEN |
| 10.001 | 1 | led_output | | State = 0 |
| 20 | 1 | led_output | | State = 1 |
| 20 | 2 | traffic_light | out | 0x201000 |
| 20 | 2 | traffic_light | | Next State = 0x200000 |
| 20.001 | 1 | led_output | LED | AMBER |
| 20.001 | 1 | led_output | | State = 0 |
| 22 | 1 | led_output | | State = 1 |
| 22 | 2 | traffic_light | out | 0x200000 |
| 22 | 2 | traffic_light | | Next State = 0x002000 |
| 22.001 | 1 | led_output | LED | RED |
| 22.001 | 1 | led_output | | State = 0 |

Observing the log, we see that at the $0^{th}$ second, three events occur: The traffic_light model changes state from 0x200000 to 0x002000, the out port of the model outputs the value 0x200000 and the state of the led_output model changes to 1. This signifies that, at the $0^{th}$ second, the traffic light model presents the value corresponding to red at its output port. The led_output model receives this value at its input port and activates itself (sets state to 1). At the $0.001^{th}$ second (one millisecond later) we see that the led_output model presents RED at its output port and changes its own state to 0, thereby passivating it. These five events successfully set the RGB LED to red. This can be seen.

Per our definition, the traffic light should transition to green 10s after red. This is what we observe. Looking at the 5 successive snapshots from the $10^{th}$ second, we observe that the same 5 events occur, but now sets the color of the RGB LED to green; three events, one to activate the led_output model, one to change the state of the traffic_light model from 0x002000 to 0x201000 and one to output 0x002000 at the $10^{th}$ second and two events 1 milli second later that outputs the color onto the RGB LED and passivates the led_output model.

Moving ahead, we observe the same set of five events occur at the $20^{th}$ second to change the RGB LED from Green to Amber. Now, on the $22^{nd}$ second, we see that the model changes color from amber to red, following the same five successive event pattern.

Observing these results, we see that the system remained in the red and green state for 10 seconds each and the amber state for 2 seconds. This confirms that our model is indeed operating as per definition. The only slight anomaly we observe is the 1 milli second delay between the change of state in the traffic_light model and the result shown on the LED. This delay is added in to account for the actual time taken to complete the action. When running on a microcontroller, these transactions almost never occur instantaneously, and it is important the developer keeps this in mind lest the model behaves undesirably. Note that the color

values are also different from the ones described in the previous section. The values in this simulation still correspond to red, green and amber.

This behavior can be observed in real time. The is shown in Figure 5.

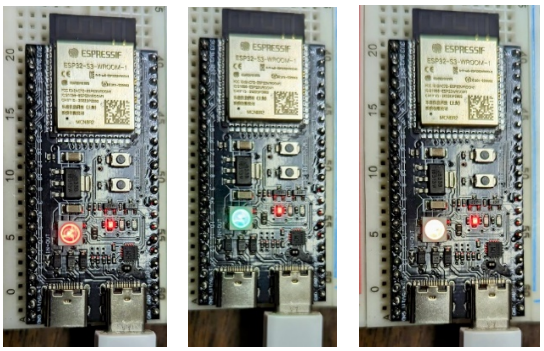

Figure 5: The three colors of the traffic light system on the RGB LED.

Figure 5 Shows the three states of the RGB LED. From left to right, we observe that the RGB LED transitions from RED to GREEN to AMBER. This corresponds to the report generated by the simulator.

Following the centralized simulation, the simulation was run on the distributed system. Herein, two ESP32S3 microcontrollers were wired together as shown in Figure 6, and their log files were collected individually. Table 2 and Table 3 show the simulation logs of the two simulation instances. Since the data in each of these logs are vast, only the output function logs are shown. The internal transition function logs are much the same as those seen in Table 1.

Table 2: Simulation log of the transmitter (only $\lambda$; no $\delta_{int}$).

| Time (s) | Model ID | Model Name | Port Name | Data |
|---|---|---|---|---|
| 0 | 5 | traffic_light | out | 0x200000 |
| 0.1 | 4 | layer1 | downstream_out | {20, 0, 0} |
| 0.2 | 3 | layer2 | downstream_out | {L2_PDU} |
| 0.3 | 2 | phy_tx | out | 5.75998E+13 |
| 10 | 5 | traffic_light | out | 0x002000 |
| 10.1 | 4 | layer1 | downstream_out | {0, 20, 0} |
| 10.2 | 3 | layer2 | downstream_out | {L2_PDU} |
| 10.3 | 2 | phy_tx | out | 5.75998E+13 |
| 20 | 5 | traffic_light | out | 0x201000 |
| 20.1 | 4 | layer1 | downstream_out | {20, 10, 0} |
| 20.2 | 3 | layer2 | downstream_out | {L2_PDU} |
| 20.3 | 2 | phy_tx | out | 6.63959E+13 |
| 22 | 5 | traffic_light | out | 0x200000 |
| 22.1 | 4 | layer1 | downstream_out | {20, 0, 0} |
| 22.2 | 3 | layer2 | downstream_out | {L2_PDU} |
| 22.3 | 2 | phy_tx | out | 5.75998E+13 |

As can be observed from Table 2, the output at the $0^{th}$, $10^{th}$ and $20^{th}$ second marks are much the same as that found in Table 1. At these time instances, the traffic_light atomic model outputs the hex codes for red, green, amber and back to red. However, apart from these similarities, the outputs are quite different from that in Table 1. At the 0.1, 10.1, 20.1 and 22.1 second marks, we see the layer1 present the output as a

stream of byte aligned data points. Taking the $0^{th}$ and $0.1^{th}$ second example, the data 0x200000 was converted to {0x20, 0x00, 0x00}, which are each 8 bits long. At the 0.2, 10.2, 20.2 and 22.2 second marks, we see the layer2 atomic model outputs the {L2_PDU}. The simulation logs show the entire PDU, but for the lack of space, they are not shown in this paper. However, here: *{data: 0x20,datalen_frame_select: True, frame_num: 0x3, total_frames: 0x3, checksum: 0x6 }* is an example of one such PDU. This PDU was produced at the $0.2^{th}$ second of the simulation and shows the data to be transmitted and all the individual values of the various header flags in the PDU. Finally at the 0.3, 10.3, 20.3 and 22.3 second marks, the phy_tx atomic model outputs the bitstream into one of the IO pins of the microcontroller.

Hence, we observe that the transmitter is working as expected. However, it is to be observed that between the output of the traffic_light atomic and the phy_tx atomic, 300 milliseconds pass. This 300-millisecond delay was artificially induced to give the microcontroller time to execute the transition methods. When stress testing the system, we were able to reliably bring this delay down to a few nanoseconds, but in the final model, we have let this delay remain to make it compatible with extremely quick and complex systems.

However, to illustrate this point, the receiver was designed with no induced delays whatsoever. The output of the receiver simulation log can be seen below:

Table 3: Simulation log of the receiver (only $\lambda$; no $\delta_{int}$).

| Time (s) | Model ID | Model Name | Port Name | Data |
|---|---|---|---|---|
| **0.3** | 3 | phy_rx | out | {L2_PDU} |
| 0.3 | 4 | layer2 | upstream_out | {20, 0, 0} |
| 0.3 | 5 | layer1 | upstream_out | 0x200000 |
| 0.3 | 2 | led_output | LED | RED |
| **10.3** | 3 | phy_rx | out | {L2_PDU} |
| 10.3 | 4 | layer2 | upstream_out | {0, 20, 0} |
| 10.3 | 5 | layer1 | upstream_out | 0x002000 |
| 10.3 | 2 | led_output | LED | GREEN |
| **20.3** | 3 | phy_rx | out | {L2_PDU} |
| 20.3 | 4 | layer2 | upstream_out | {20, 10, 0} |
| 20.3 | 5 | layer1 | upstream_out | 0x201000 |
| 20.3 | 2 | led_output | LED | AMBER |
| **22.3** | 3 | phy_rx | out | {L2_PDU} |
| 22.3 | 4 | layer2 | upstream_out | {20, 0, 0} |
| 22.3 | 5 | layer1 | upstream_out | 0x200000 |
| 22.3 | 2 | led_output | LED | RED |

Table 3 shows the simulation log of the receiver. We observe that at the 0.3, 10.3, 20.3 and the 22.3 second mark highlighted in bold are the instances where the receiver received the transmitted output from the transmitter. Since the simulation was started at the same time, they are synchronized, and the time values can be matched. Hence, we see that, the transmitted time and the received time is the same. The entire transmission takes 13μs, which is instant at the scale of this simulation. In the proceeding time steps, we see the layer2 and layer1 atomic successfully decoding the incoming messages. Finally, we see the led_output atomic output the appropriate color for the RGB-LED.

As previously stated, the receiver was not induced with artificial delay, and hence the time difference between the reception of the packet and the color change of the led is in the range of a few microseconds.

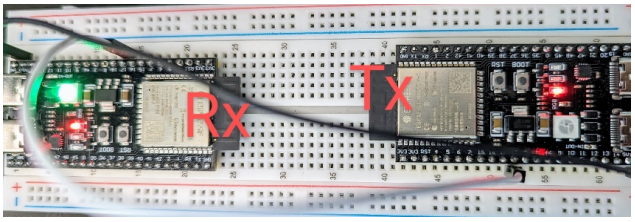


Figure 6: Distributed Simulation System.

Figure 6 shows the two ESP32S3 connected to each other. The ESP32S3 on the right is the transmitter and the one on the left is the receiver. The white jumper wire is the signal line, and the black jumper represents the ground connection between the two.

## 5 CONCLUSION

In this paper we have defined the DEVS-IAC protocol using the DEVS formalism. This protocol takes inspiration from the Ethernet stack, modifying the same to work for the Real Time scenario. Each of the layers of the protocol had a multitude of parameters, each of which were permuted and tested for optimal performance, two of these tests were shown in the previous sections by inducing a delay within the system.

The protocol was then implemented in Cadmium on the ESP32S3 microcontroller, using the traffic light system as a case study. In the case study we simulated the traffic light model, and the simulation logs were observed and verified. When comparing the results of the distributed model against the centralized model, we see that there is no significant time delay between the two implementations. Even when induced with a delay, the protocol was able to maintain the behavior of the system.

This protocol was developed to be hardware agnostic and extremely generic. This protocol as a coupled model can be used to decentralize any DEVS based model across multiple microcontrollers. The protocol was defined to take in any input, regardless of datatype or data length and transmit it across a wired medium. The modularity of this protocol enables any developer to change the various layers to fit the specifications of other protocols. A wireless physical layer can be implemented to enable wireless transfer as opposed to wired.

In the future, we plan on developing a wide range of layers for the DEVS-IAC protocol to not only allow communication between microcontrollers, but also to enable data transmission amongst various cadmium simulation instances running on the same or even different systems.

## REFERENCES

[1] P. Neumann, "Communication in Industrial Automation-What is Going On?", *Control Engineering Practie*, vol. 37, no. 4, pp. 63-71, 2004.

[2] IEEE Standard for Information Technology-Telecommunications and information exchange between systems-Local and metropolitan area networks-Specific requirements Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications, IEEE standard 802.11, 2007.

[3] P. Pedreiras, L. Almeida and J. A. Fonseca. “The Quest for Real-Time Behavior in Ethernet” in *The Industrial Information Technology Handbook*, R. Zurawski, Ed. : CRC-Press, 2005 pp. 1-14.

[4] H. Saadawi and G. Wainer, "Verification of real-time DEVS models," in *Proc. 2009 Spring Simulation Multiconference*, San Diego, CA, 2009, pp. 1-8.

[5] B. P. Zeigler, H. Praehofer, and T. G. Kim, *Theory of Modeling and Simulation*, 2nd ed. San Diego: Academic Press, 2000.

[6] G. A. Wainer, "Applying modelling and simulation for development embedded systems," in *MECO*, 2013, pp. 1-2.

[7] S. M. Govind, J. S. R. Alex, and G. A. Wainer, "Adapting the DEVS kernel 'RT-CADMIUM' to the ESP32 embedded platform," *arXiv preprint arXiv:2304.07961*, 2023.
[8] B. Earle, K. Bjornson, C. Ruiz-Martin, and G. Wainer, "Development of a real-time DEVS kernel: RT-Cadmium," in *Spring Simulation Conference (SpringSim)*, Fairfax, VA, 2020, pp. 1-12.
[9] B. P. Zeigler, S. Mittal, and M. K. Traore, "MBSE with/out Simulation: State of the Art and Way Forward," *Systems*, vol. 6, no. 4, p. 40, 2018.
[10] S. A. Stein, "The ISO connectionless transport standards," *ACM SIGCOMM Computer Communication Review*, vol. 14, no. 4, pp. 18-20, 1984.
[11] M. M. Alani, "OSI model," in *Guide to OSI and TCP/IP Models*, M.M. Alavi, Ed., 2014, pp. 5-17.
[12] S. Eidson, B. Gaines, and P. Wolf, "30.2: HDMI: High-Definition Multimedia Interface," in *SID Symposium Digest of Technical Papers*, vol. 34, no. 1, pp. 1024-1027, 2003.
[13] M. R. Ahmed, M. D. Haque, and M. A. Z.-S. I. Khan, "An Efficient Digital Encoding Technique to Improve the Power Spectral Density Based on Manchester Encoding Technique," in *International Conference on Electronics, Computer and Communication (ICECC 2008)*, Rajshahi, Bangladesh.
[14] A. Falcone and A. Garro, "Distributed co-simulation of complex engineered systems by combining the high level architecture and functional mock-up interface," *Simulation Modelling Practice and Theory*, vol. 97, p. 101967, 2019.
[15] M. Mirz, S. Vogel, B. Schäfer, and A. Monti, "Distributed real-time co-simulation as a service," in *2018 IEEE International Conference on Industrial Electronics for Sustainable Energy Systems (IESES)*, 2018, pp. 534-539.
[16] M. U. Awais, "Distributed hybrid co-simulation," Ph.D. dissertation, TU Wien, Vienna, Austria 2015.
[17] G. Wainer and M. Moallemi, "Designing real-time systems using imprecise discrete-event system specifications," *Software: Practice and Experience*, vol. 50, no. 8, pp. 1327-1344, 2020.
[18] R. Cárdenas and G. Wainer, "Asymmetric Cell-DEVS models with the Cadmium simulator," *Simulation Modelling Practice and Theory*, vol. 121, p. 102649, 2022.

## APPENDICES

The appendices are available here.

## AUTHOR BIOGRAPHIES

**SASISEKHAR GOVIND** is an M.A.Sc student at Carleton University. He holds a bachelor's degree in Electronics and Communications Engineering from VIT, India. His research interests lie in embedded systems and distributed simulations. His email address is sasisekharmangalamgo@cmail.carleton.ca.

**GABRIEL WAINER** is a Professor at the Department of Systems and Computer Engineering at Carleton University. He received his M.Sc. (1993) from the University of Buenos Aires, Argentina, and his Ph.D. (1998, highest honors) from UBA/ Université Aix-Marseille-III, France. He is a fellow of SCS. His email address is gwainer@sce.carleton.ca.