Check for updates

ACM DIGITAL LIBRARY | Association for Computing Machinery | acm open

Latest updates: https://dl.acm.org/doi/10.1145/3726301.3728399

RESEARCH-ARTICLE

Improving the Efficiency of Coordinating Timed Events in Distributed Systems

BYEONGGIL JUN, Arizona State University, Tempe, AZ, United States

EDWARD A. LEE, University of California, Berkeley, Berkeley, CA, United States

MARTEN LOHSTROH, University of California, Berkeley, Berkeley, CA, United States

HOKEUN KIM, Arizona State University, Tempe, AZ, United States

Open Access Support provided by:

University of California, Berkeley

Arizona State University

PDF Download
3726301.3728399.pdf
25 February 2026
Total Citations: 0
Total Downloads: 369

Published: 22 June 2025

Citation in BibTeX format

SIGSIM-PADS '25: 39th ACM SIGSIM Conference on Principles of Advanced Discrete Simulation
June 23 - 26, 2025
Santa Fe, USA

Conference Sponsors:
SIGSIM

Improving the Efficiency of Coordinating Timed Events in Distributed Systems

Byeonggil Jun
Arizona State University
Tempe, AZ, USA
byeonggil@asu.edu

Edward A. Lee
UC Berkeley
Berkeley, CA, USA
eal@berkeley.edu

Marten Lohstroh
UC Berkeley
Berkeley, CA, USA
marten@berkeley.edu

Hokeun Kim
Arizona State University
Tempe, AZ, USA
hokeun@asu.edu

Abstract

For distributed cyber-physical systems (CPS), accurate and precise timing control among distributed nodes helps ensure safety and correctness. Discrete-event (DE) systems serve as an effective model of computation for distributed CPS requiring precise timing control. The high-level architecture (HLA) has been widely used for the simulation of distributed DE (DDE) systems and can be extended for the implementation of distributed CPS. We find that HLA imposes excessive network communication overhead, specifically, through the run-time infrastructure (RTI), for the coordination of timed events for some applications. This paper identifies inefficiencies and proposes a novel approach that reduces the number of messages without compromising control over timing. We evaluate our approach using example patterns of distributed CPS.

CCS Concepts

• **Computing methodologies** → **Discrete-event simulation**; • **Theory of computation** → *Distributed computing models*; • **Computer systems organization** → *Real-time system architecture*; *Embedded software.*

Keywords

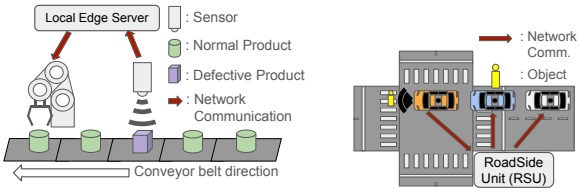
Discrete-event systems, Synchronous-reactive programs, Dstributed systems, Deterministic concurrency

ACM Reference Format:
Byeonggil Jun, Edward A. Lee, Marten Lohstroh, and Hokeun Kim. 2025. Improving the Efficiency of Coordinating Timed Events in Distributed Systems. In *39th ACM SIGSIM Conference on Principles of Advanced Discrete Simulation (SIGSIM-PADS '25), June 23–26, 2025, Santa Fe, NM, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3726301.3728399

1 Introduction

Fine-grained timing control over the network is often required in distributed cyber-physical systems (CPS) consisting of computing components that interact with the physical world and are connected through networks. One of the important challenges in such fined-grained timing control is determinism, which ensures exactly one behavior for the same given initial conditions and inputs.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SIGSIM-PADS '25, Santa Fe, NM, USA
© 2025 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-1591-4/25/06
https://doi.org/10.1145/3726301.3728399



(a) Defect detection. **(b) Intersection.**

Figure 1: Sensing systems that require frequent sensing.

The High-Level Architecture (HLA), an IEEE standard for discrete event (DE) system simulation [20], is one of the ways to support fine-grained timing control with determinism in distributed CPS. HLA can also be used for the execution of DE-based distributed CPS using a run-time infrastructure (RTI) such as CERTI [35], an open-source RTI. HLA-style methods have also been adapted to include features specific to system implementation (vs. simulation), such as asynchronous injection of unanticipated events and zero-delay cycles, which are common in control systems [5].

An HLA-based system is composed of federates, which are individual distributed nodes, and an RTI, which is software that provides services to support executions of federates . Each federate reacts to events in logical time order, and each event is tagged with a timestamp representing a logical time [28]. To ensure that federates execute events in logical time order, federates and the RTI exchange signals including timing information. For example, a federate reports its next event's logical time to the RTI each time the federate tries to advance its logical time. The RTI will grant logical time advancement when it can assure that the federate has seen all input events with earlier timestamps. However, this frequent exchange of signals increases network overhead and can affect the feasibility (ability to meet deadlines) of a system. Jun *et al.* [22] identify certain inefficiencies in the HLA-based system.

Consider sensing systems that require frequent sensing, as illustrated in Figure 1. In a defect detection system shown in Figure 1a, a sensor checks the conditions of products and reports defective products so that a robot arm picks out the defective products. Assume a local edge server controls those components. Suppose we adopt HLA to build this system. The sensor federate must send its next event timestamp to the RTI every time it senses and wait for the RTI to grant logical time advancement. Thus, the round-trip network delay limits the sensing period.

Figure 1b describes another application that requires frequent sensing, connected autonomous vehicles at an intersection. When a vehicle detects an object, it reports the caution to the roadside unit (RSU) [4], a communication node installed within the traffic infrastructure, and the RSU broadcasts the information to others. Similarly, a car needs to be sensing frequently and requires confirmation from the RTI to advance its tag.

This paper introduces methods enabling fine-grained control of timing and reducing the number of signals in distributed systems using HLA-style timing coordination. We set our baseline to an open-source software runtime and coordination language, Lingua Franca (LF) [29] because the time management mechanism of LF's centralized coordinator is roughly based on HLA [5] and provides an extended, working implementation with mechanisms resembling HLA, but also supports asynchronous injection of unanticipated events and other real-time implementation-oriented features. A detailed comparison between traditional HLA/RTI and LF's RTI is developed in Section 3.5. We examine scenarios where inefficiently used signals affect the program's feasibility. Then, we provide solutions to enable precision timing control by eliminating redundant signals while preserving determinism.

In summary, the contributions of this paper are:

- Identifying inefficiencies in state-of-the-art HLA-based timing coordination techniques for distributed systems [5, 22] and their challenges in fine-grained timing control,
- Proposing a methodology to address the identified inefficiencies via in-depth analysis of timing-related signals and eliminating non-essential signals, and
- Case studies of common patterns in distributed CPS followed by evaluation of the proposed methodology through experiments demonstrating the effectiveness of the proposed approach.

2 Related Work

Among algorithms and methods for synchronization mechanisms for distributed discrete event (DDE) simulation [21], HLA [20] has been pervasively used and standardized by the IEEE. Examples using HLA for DDE simulation include continuous simulation by Awais *et al.* [3], using HLA with the functional mock-up interface (FMI) [7], and integrating heterogeneous simulators for complex embedded systems by Brito *et al.* [8].

HLA has also been used for distributed simulation of real-time systems and CPS. For example, Lasnier *et al.* [24] propose a simulation framework for real-time CPS with heterogeneous components using CERTI [35], an open-source RTI for HLA. The PRISE project [11, 12] achieves real-time performance of distributed simulation for systems with hard real-time constraints using CERTI. Ptolemy-HLA [9] gives an HLA implementation for DDE integrated with Ptolemy II [37], an open-source CPS modeling tool supporting a variety of models of computation.

Some prior research has explored various approaches to enhance the efficiency in time management of DDE systems and HLA with RTI. Notable contributions include minimizing the communication in DDE systems through the development of optimal communication sets, as demonstrated by Rudie *et al.* [38]. While Wang and Turner [40] propose optimistic time synchronization with rollback capabilities in HLA and Marotta *et al.* [31] improves the effectiveness of accessing the committed state in speculative parallel DE simulation, such methods are limited to simulation environments and cannot be applied to actual deployment of HLA-based systems where rollbacks are infeasible. The COSSIM simulation framework [39] introduces flexible time synchronization mechanisms for CPS simulation by allowing tradeoffs between time precision and performance through its IEEE HLA-compatible interface. In contrast to these approaches, our approach focuses on achieving improved efficiency in DDE systems while ensuring strict synchronization through enhanced HLA implementations.

Name	Definition
$\mathbb{N}$	A set of non-negative integers represented as a 32-bit unsigned integer
$\mathbb{T}$	A set of non-negative integers represented as a 64-bit signed integer $\cup \{-\infty, \infty\}$
$\mathbb{G}$	A set defined as $\{(t, m) \mid t \in \mathbb{T} \setminus \{-\infty, \infty\},\ m \in \mathbb{N}\} \cup \{(-\infty, 0), (\infty, M_{\max})\}$

Table 1: Set definitions from Jun *et al.* [22] for defining tags in LF. We use $-\infty$ (NEVER) for a time value less than any other, represented as the minimum 64-bit signed integer, and ∞ (FOREVER) for a time value greater than any other, represented as the maximum 64-bit signed integer.

3 Background

3.1 Lingua Franca

Lingua Franca (LF) is an open-source coordination language and runtime designed to implement reactors [27, 30]. Reactors integrate beneficial semantic features from well-established models of computation, such as actors [2], logical execution time [23], synchronous reactive languages [6], and discrete event systems [25] including DEVS [41] and SystemC [26]. In CPS interacting with the physical world in real time, treating time as a purely logical concept is not sufficient. To address the limitation of logical time, LF allows physical and logical timelines to interact deterministically [29].

3.2 Tags

In LF, every event is timestamped by logical time and executed in order of logical time. Table 1 demonstrates the definitions of sets $\mathbb{N}$, $\mathbb{T}$, and $\mathbb{G}$, that are used for formally defining logical time in LF. Logical time is represented by a tag $g \in \mathbb{G}$, which is a pair of a *time value* $t \in \mathbb{T}$ and a *microstep* $m \in \mathbb{N}$ where microstep is introduced in the superdense model of time [29]. A sequence of tagged events that occur at the "same time" can be represented with the microstep [14]. In LF, $M_{\max}$, the maximum microstep value, is reserved for the upper bound of the microstep. Ideally, an infinite number of events can happen at the same time, i.e., microstep can be infinity. However, in LF, the microstep is saturated to $M_{\max}$. In this paper, we assume that a tagged event never reaches a maximal microstep, $(t, M_{\max})$ because we will almost never see that 4 billion events occur at the same time in practical applications. The set of tags, $\mathbb{G}$, forms a totally ordered set where, given two tags $g_a = (t_a, m_a)$ and $g_b = (t_b, m_b)$, $g_a < g_b$ if and only if $t_a < t_b$ or $t_a = t_b$ and $m_a < m_b$.

For simplicity, in this paper, we use the elapsed time and microstep since the startup tag (the beginning of the execution) of the program to represent an event's tag, instead of using an *actual* tag with the time elapsed since January 1, 1970. For example, an event can be scheduled at the startup tag, e.g., $(1728963084, 0)$. Instead of using the actual tag, we say the event is scheduled at $(0, 0)$.

3.3 Delays

Figure 2 depicts a diagram of an LF program, *Delays*, with five reactor instances, *a*, *b*, *c*, *d*, and *e*, which are instances of reactor

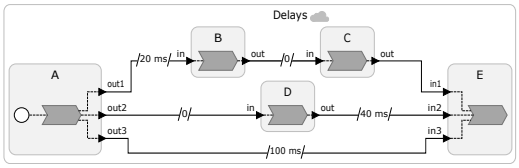


Figure 2: An example LF program with delays.

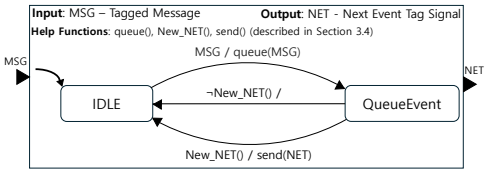


Figure 3: A state machine handling tagged messages.

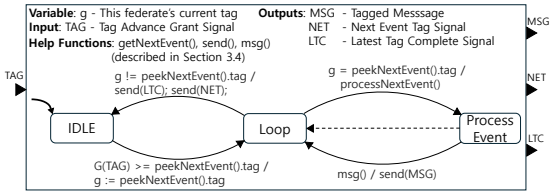


Figure 4: A state machine handling tag advance grant signals.

classes A, B, C, D and E, respectively. The cloud icon next to the program's name indicates this program is federated (distributed), where each top-level reactor is a separate program called a *federate*. Each federate is compiled into an independent executable binary to be deployed on distributed nodes. Dark gray chevrons illustrate *reactions*, procedures that are executed when triggered. For example, the reaction in a is triggered by the startup trigger (denoted by a white circle), causing the reaction to be triggered at the beginning of the execution of the LF program. The body of the reaction can be implemented in C, Python, and Typescript.

LF's syntax allows logical time to elapse on a connection using the concept of *logical delay*. For example, the connection from A to B in Figure 2 has 20 ms of logical delay, meaning that a tag of a message increases by 20 ms while going through the connection. A tagged message originating from the *out1* port of A with a tag $(0,\ 0)$ will have a tag $(20\ ms,\ 0)$ when it arrives at the *in* port of B. LF distinguishes no delay and zero delay; specifically, no delay on a connection means there is no increment to the timestamp or microstep, whereas zero delay increments the microstep by 1. For example, a tagged message sent by A at $(0,\ 0)$ will have a tag $(0,\ 1)$ when D receives that message because of the zero delay.

Because a delay can increment either time or microstep from the originating tag, we define *tag increment* over a connection to compute a message's tag. The function $C\colon \mathbb{T} \longrightarrow \mathbb{G}$ is defined for converting a delay represented by time to a tag [22]:

$$C(d) = (t_d,\ m_d) = \begin{cases} (0,\ 1), & \text{if } d = 0 \\ (d,\ 0), & \text{if } 0 < d < \infty \\ (0,\ 0), & \text{if } d = -\infty \\ (\infty,\ M_{\text{max}}) & \text{if } d = \infty \end{cases} \tag{1}$$

For example, the tag increment between A and B in Figure 2 is $(20\ ms,\ 0)$ and between A and D is $(0,\ 1)$. If there is no delay $(-\infty)$, like between C and E, the tag increment is $(0,\ 0)$, and if there is no connection, the tag increment is represented by $(\infty,\ M_{\text{max}})$.

When there are multiple concatenated connections, a single message can cause a lot more messages downstream. For example, in Figure 2, a message sent from A may eventually cause a message to C. Suppose A sends a message with the tag $(20\ ms,\ 0)$ to B. Federate B may send a message with the tag $(20\ ms,\ 1)$ to C. Thus, we say that there is a tag increment of $(20\ ms,\ 1)$ between A and C.

To calculate tag increments and the tag of tagged messages, LF defines a function A, an operation like tag addition. When a federate sends a message at tag $g_a = (t_a, m_a)$ through a connection with a tag increment $g_b = (t_b, m_b)$, the message's tag is computed by $A(g_a, g_b)$. The key challenge here is handling overflow. Currently, in LF, overflows of the time value and the microstep are treated as errors. Here, we handle overflow in the formalization. Yet, we recognize that useful programs should not encounter overflow since the time value of signed 64-bit int in nanoseconds overflows approximately after 226.4 years, overflowing in the year 2262. The microstep overflows if 4 billion events occur at the same nanosecond. The function $A\colon \mathbb{G}\times\mathbb{G}_d \longrightarrow \mathbb{G}$ is defined as shown below [22]:

$$A(g_a, g_b) = A((t_a, m_a), (t_b, m_b)) =$$

$$\begin{cases} (t_a, m_a + m_b), & \text{if } 0 \le t_a < \infty \wedge t_b = 0 \wedge m_a + m_b < M_{\text{max}} \\ (t_a + t_b, m_b), & \text{if } 0 \le t_a < \infty \wedge t_b > 0 \wedge t_a + t_b < \infty \\ (t_a, M_{\text{max}}), & \text{if } 0 \le t_a < \infty \wedge t_b = 0 \wedge m_a + m_b \ge M_{\text{max}} \\ (\infty,\ M_{\text{max}}), & \text{if } 0 \le t_a < \infty \wedge t_b > 0 \wedge t_a + t_b \ge \infty \text{ or } t_a = \infty \\ (-\infty,\ 0), & \text{if } t_a = -\infty \end{cases} \tag{2}$$

where $\mathbb{G}_d$ is $\mathbb{G}\backslash\{(-\infty,\ 0),\ (\infty,\ M_{\text{max}})\}$. Because the second operand g_b is the tag increment between two federates, g_b cannot be $(-\infty,\ 0)$ because no delay is encoded to $(0,\ 0)$ and g_b of $(\infty,\ M_{\text{max}})$ indicates there is no connection between the two federates and makes no meaningful sense. Note that if t_b is greater than 0, the result $A((t_a, m_a), (t_b, m_b))$ yields a tag that has m_b as the microstep, effectively ignoring m_a. For example, if A produces a message at a tag $(0,\ 3)$ to C, the message will have a tag $(20\ ms,\ 1)$. We do not defend this decision but accept this as the behavior exhibited by LF.

We can also compute the tag increment over multiple concatenated connections by adding delays on each connection. For example, the tag increment between A and C in Figure 2 can be computed by $A((20\ ms,\ 0),\ (0,\ 1))$, which is $(20\ ms,\ 1)$. Thus, an event of A at tag $(t,\ m)$ can cause a message to C at tag $A((t,\ m),\ (20\ ms,\ 1))$.

3.4 Centralized Coordination

LF has its own version of an RTI for coordinating federates. From now on, we use the term *LF-RTI* to distinguish LF's RTI from the traditional HLA RTI. An in-depth discussion of the difference between LF and HLA is given in Section 3.5.

To ensure that a federate executes events in tag order, federates send their current states, and LF-RTI keeps track of each federate's state and updates a tag barrier to allow federates to advance their tags. The first column of Table 2 summarizes the types of messages and signals, and the second and third columns show the variables used for tracking each federate's state. Signals and messages have tags in the payload, and we use a function G to represent the associated tag of a signal or a message. For instance, $G(\text{MSG}_{ij})$ is the tag of the message MSG_{ij}. LTC_j (Latest Tag Complete) is a signal sent from a federate j to LF-RTI to report that j has completed every event at the tag $G(\text{LTC}_j)$. NET_j (Next Event Tag) is a signal sent

Name	Description	Name	Description	Name	Description
MSG_{ij}	Tagged Message from j to i	Q_i	In-transit Message Queue of i	U_i	Immediate Upstream federates of i
LTC_j	Latest Tag Complete from j	L_j	The most recent LTC_j's tag	$\bar{U}_i$	Transitive Upstream Federates of i
NET_j	Next Event Tag from j	N_j	The most recent NET_j's tag	$\bar{D}_i$	Transitive Downstream Federates of i
TAG_i	Tag Advance Grant to i	D_{ij}	Minimum Tag Increment from j to i	B_i	Earliest Incoming Message Tag of i

Table 2: Types of messages, signals, and symbols used in this paper, where i and j are federates.

from federate j to LF-RTI to report the tag of j's earliest unprocessed event. TAG_i (Tag Advance Grant) is a signal sent by LF-RTI to federate i to notify that i has received all messages with a tag that is less than or equal to $G(TAG_i)$.

The state machines in Figure 3 and Figure 4 illustrate how a federate handles events, signals, and messages. Transitions marked with dotted lines denote default transitions that are enabled when no other transitions are enabled. When federate i receives MSG_{ij}, it creates events corresponding to the message and schedules those events (enqueues to a queue of events), which is done by the function queue(MSG) in Figure 3. If an enqueued event is the new earliest next event, the predicate new_NET() becomes true and the function send(MSG) sends a new NET signal with the tag of the new event.

When a federate i receives a TAG signal, it can process events with tags earlier than or equal to $G(TAG_i)$. Figure 4 describes how a federate handles TAG signals and processes events. The function peekNextEvent() returns the earliest unprocessed event. If $G(TAG)$ is later than or equal to the next earliest event's tag, the federate advances its current tag g to the next event's tag. Then, the federate executes every event scheduled at the current tag g using the function processNextEvent(). New events can be scheduled and new tagged messages can be sent from the federate during processing of the event. If an event produces a new message to other federates (through LF-RTI), the function msg() returns true and the function send(MSG) sends the tagged message to LF-RTI. After processing every event at the current tag, the federate sends LTC with the tag g and NET with the next earliest event's tag.

Now, we discuss the behavior of LF-RTI, which is responsible for guaranteeing that a federate is safe to advance its tag. To predict the behavior of each federate and send proper control signals, LF-RTI maintains variables storing each federate's information and the relationships between federates. The second and third columns of Table 2 summarize these variables. For each federate j, LF-RTI maintains variables N_j to store the most recent $G(NET_j)$, L_j to store the most recent $G(LTC_j)$, and the *in-transit message queue* Q_j, a priority queue sorted by tag, to store tags of in-flight messages sent to j. When LF-RTI forwards MSG_{ij} to j, it enqueues the tag in Q_j, and when LF-RTI receives LTC_j, it dequeues tags earlier than or equal to $G(LTC_j)$ from Q_j.

LF-RTI stores the information on how federates are connected as well as the delay between them. U_i contains the set of immediate upstream federates of i (those with direct connections), $\bar{U}_i$ contains the set of "transitive" upstream federates of i. For example, regarding federate C in Figure 2, U_c has only B but $\bar{U}_c$ has both A and B. $\bar{D}_i$ contains the set of "transitive" downstream federates.

D_{ij} denotes the *minimum* tag increment from federate j to i. When there are multiple paths between two federates, the minimum tag increment is required to predict a federate's next behavior. For example, in Figure 2, a message from A can trigger three messages to E through three different paths: Among those, the path through B and C has the minimum tag increment $(20\ ms,\ 1)$ and thus, D_{AE} is $(20\ ms,\ 1)$. D_{ij} indicates that if a federate j sends a message with a tag g_j, this may cause a message for i with a tag $A(g_j, D_{ij})$, but no earlier. There could be no message at all because any reaction along the way may choose not to send an outgoing message.

Name in HLA	Name in Lingua Franca (LF)
TSO message	Tagged Message
Next Message Request (or Available)	Next Event Tag
Least Incoming TimeStamp	Earliest Incoming Message Tag
Time Advance Grant (or Available)	Tag Advance Grant

Table 3: Naming of LF messages corresponding to HLA.

For a federate i, LF-RTI predicts that i's earliest next event has the tag $\min(N_i, H(Q_i))$. Thus, when $\min(N_i, H(Q_i))$ is changed, LF-RTI decides whether to send TAG_i. First, LF-RTI utilizes LTC signals from the federates in U_i. LF-RTI knows that a federate $j \in U_i$ will not generate a message originated from L_j. However, this doesn't guarantee that j will not send a message with a tag $A(L_j, D_{ij})$, because j may send messages with the tag $A(L_j, D_{ij})$ when the time value of D_{ij} is greater than 0.

For example, suppose a federate j originates messages at $(0,\ 0)$ and $(0,\ 1)$ to i and D_{ij} is $(20\ ms,\ 0)$. Both the messages will have the same tag, $(20\ ms,\ 0)$. Thus, in this situation, LF-RTI can guarantee that j will not send TAG_{ij} with the tag $(19\ ms,\ M_{\max})$, not $(20\ ms,\ 0)$. LF-RTI looks up every immediate upstream federate and finds a tag g, the maximum tag that LF-RTI can allow i to advance using LTC signals. We use $H(Q_i)$ to denote the tag of the message at the head of the queue Q_i. If the tag g is later than or equal to $\min(N_i, H(Q_i))$, LF-RTI grants TAG_i with the tag g.

If $g < \min(N_i, H(Q_i))$, LF-RTI needs to determine B_i, the **earliest (future) incoming message tag** for node i [15], and this will require looking further upstream beyond the federates in U_i. For an immediate upstream federate $j \in U_i$, the tag of the earliest possible future message from j is $A(\min(N_j, H(Q_j), B_j), D_{ij})$, where $H(Q_j)$ denotes the tag of Q_j's head. Thus, we can compute B_i as:

$$B_i = \min_{j \in U_i}(A(\min(N_j, H(Q_j), B_j), D_{ij}))) \tag{3}$$

LF-RTI can send TAG_i with a tag $\min(N_i, H(Q_i))$ only if B_i is *strictly* greater than $\min(N_i, H(Q_i))$. If there is no cycle in the program (paths from i back to itself), then LF-RTI can easily calculate this quantity. If there are only cycles that have at least one after delay, then again, this is easy to calculate. If there is a cycle with no after delays, handling this gets much more subtle, requiring extra coordination [15]. Thus, for the discussion of the rest of this paper, we do not consider the federates within zero-delay cycles.

3.5 Comparison between HLA and LF

Table 3 relates the terminology of HLA with the terminology of Lingua Franca (LF) we use in this paper. Time management in HLA is similar in that each federate sends a signal Next Message Request (NMR) to advance its tag and the RTI sends a signal Time Advance

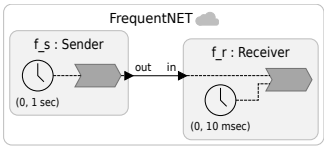


Figure 5: An LF program where the federate Receiver sends NET frequently.

Grant computed by the Least Incoming Timestamp. The latest tag complete (LTC) signals in LF resemble the NULL messages in the algorithm proposed by Chandy and Misra [10]. HLA uses NULL messages to address in-transit messages or to manage logical time by indicating the time advancement of each federate [13, 17].

We point out that HLA assumes two types of federates, time-stepped and event-stepped. HLA provides a Time Advancement Request signal to support time-stepped federates. Because time-stepped federates are special cases of event-stepped federates, LF only provides the signal NET which resembles the signal NMR.

We also note that LF-RTI is a centralized standalone coordinator whereas in HLA, the RTI can be implemented in either a centralized or decentralized manner. For example, in poRTIco [36] v2.1.0, decentralized local RTI components communicate with each other without a centralized server [18]. However, regardless of the underlying implementation, federates must exchange NMR and TAG signals with the RTI to advance their logical time, an inefficiency intrinsic to the HLA protocol itself, as identified in Section 4.

3.6 Downstream Next Event Tag

Jun *et al.* [22] propose to remove some unnecessary signals to address the inefficiency in LF's HLA-based timing coordination. LF-RTI uses NET_i from federate i to compute TAG signals for i's downstream federates and to send a TAG_i to allow i to advance its tag. Thus, a NET_i signal is not necessary if it does not lead to a TAG signal for i's downstream federates and if i itself can advance to the tag $G(\text{NET}_i)$ without further confirmation by LF-RTI.

Specifically, to remove unnecessary NET signals, Jun *et al.* [22] make LF-RTI send a signal called Downstream Next Event Tag (DNET) to notify a federate that the federate only needs to send NET signals with tags later than the payload tag $G(\text{DNET})$. In other words, NET signals with tags earlier than or equal to $G(\text{DNET})$ will not cause any TAG signals downstream.

LF-RTI needs to look up transitive downstream federates in $\tilde{D}_j$ to compute the upper bound tag of unnecessary NET_j signals from a federate j. For a federate $i \in \tilde{D}_j$, if $A(\min(N_j, H(Q_j)), D_{ij})$, the earliest incoming message tag caused by j, is earlier than or equal to $\min(N_i, H(Q_i))$, the tag of i's next event, LF-RTI cannot send TAG_i with the tag $\min(N_i, H(Q_i))$. Let a tag g_i be the tag satisfies the condition $A(g_i, D_{ij}) \leq \min(N_i, H(Q_i))$. Then, NET_j with tags earlier than or equal to g_i is unnecessary for granting a TAG signal to i. For computing g_i, the function $S\colon \mathbb{G}\times\mathbb{G}_d \longrightarrow \mathbb{G}$ is defined [22]:

$$S(g_a, g_b) = S((t_a, m_a), (t_b, m_b)) =$$

$$\begin{cases} (-\infty,\ 0), & \text{if } g_a = -\infty \vee g_a < g_b \\ (t_a - t_b, m_a - m_b), & \text{if } \infty > t_a \geq t_b = 0 \wedge m_a \geq m_b \\ (t_a - t_b, M_{\max}), & \text{if } \infty > t_a \geq t_b > 0 \wedge m_a \geq m_b \\ (t_a - t_b - 1, M_{\max}), & \text{if } \infty > t_a > t_b > 0 \wedge m_a < m_b \\ (\infty,\ M_{\max}) & \text{if } t_a = \infty \end{cases} \tag{4}$$

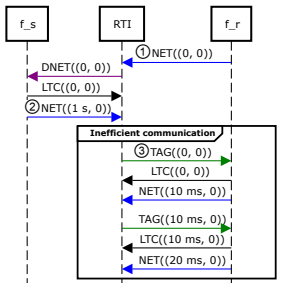


Figure 6: An execution trace of the program in Figure 5 without redundant LTC signals.

where $\mathbb{G}_d$ is $\mathbb{G} \setminus \{(-\infty,\ 0),\ (\infty,\ M_{\max})\}$. If $i \in \tilde{D}_j$, the value D_{ij} cannot be $(\infty,\ M_{\max})$ because D_{ij} of $(\infty,\ M_{\max})$ means there is no path from j to i. Also, D_{ij} cannot be $(-\infty,\ 0)$ because D_{ij} is always greater than or equal to $(0,\ 0)$ ("no delay" is encoded to $(0,\ 0)$). The tag g_i can be computed as $S(\min(N_i, H(Q_i)), D_{ij})$. Finally, we can compute $G(\text{DNET}_j)$ as:

$$\min_{i \in \tilde{D}_j} (S(\min(N_j, H(Q_j)), D_{ij})) \tag{5}$$

4 Challenges and Inefficiencies in Current State-of-the-Art Timing Coordination

This section discusses inefficiencies in up-to-date LF's HLA-based timing coordination and how they can affect a program's feasibility.

4.1 NET for Requesting TAG

By using the DNET signals described in Section 3.4, current LF can eliminate NET_i signals from federate i that are unnecessary for granting TAG signals to i's downstream federates. Another essential role of NET_i signals is to request a TAG_i signal for itself.

As explained in Section 3.4, LF-RTI sends a TAG_i signal to federate i with a tag $\min(N_i, H(Q_i))$ if $B_i > \min(N_i, H(Q_i))$ when using EIMT for granting a tag. However, this TAG_i proves overly conservative, potentially resulting in redundant NET_i signals if B_i is much later than N_i. Given that B_i represents the earliest tag that i might receive, we can grant any tag earlier than B_i.

Consider the LF program in Figure 5 and its execution trace in Figure 6. We assume that Sender sends a message at the tag $(1\ s,\ 0)$, not at $(0\ ,0)$. Federate f_r sends NET_{f_r} with the tag $(0,\ 0)$ in ① to request a TAG signal. Upon receiving the NET signal, LF-RTI computes the DNET value, which is $S(\min(N_{f_s}, H(Q_{f_s})), D_{f_rf_s})$. In this case, the value is $(0,\ 0)$ because N_{f_s} is $(0,\ 0)$, $H(Q_{f_s})$ is $(\infty, M_{\max})$ as f_s does not have any in-transit messages, and $D_{f_rf_s}$ is $(0,\ 0)$ since there is no delay between the two federates.

After completing the tag $(0, 0)$, federate f_s sends $\text{LTC}_{f_s}((0,\ 0))$ and $\text{NET}_{f_s}((1\ s,\ 0))$ to LF-RTI. By NET_{f_s} in ①, LF-RTI knows that f_r will not later receive any messages with tags earlier than $(1\ s,\ 0)$. Nevertheless, LF-RTI sends $\text{TAG}_{f_r}((0,\ 0))$ to f_r because N_{f_r}, the most recent NET tag from f_r is $(0,\ 0)$. Thus, f_r and LF-RTI exchange NET and TAG signals every 10 ms unnecessarily as shown in the box labeled "Inefficient Communication" in Figure 6.

This also affects the feasibility of the program. Suppose there is a network delay of 15 ms between LF-RTI and federate f_r. Although the period of the timer is 10 ms in terms of the logical time, the period in physical time between the events cannot be less than 15

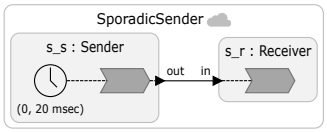


Figure 7: An LF program where the federate Sender sends messages sporadically.

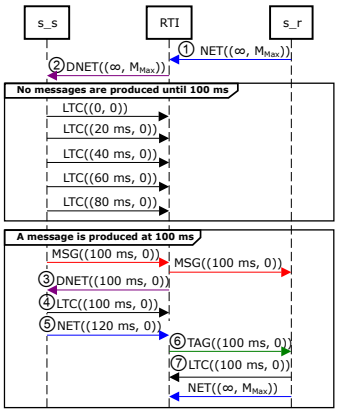


Figure 8: An execution trace of the program in Figure 7.

ms because of the process of sending a NET signal and waiting for the corresponding TAG signal. Section 5 describes a method to exploit EIMT to address this inefficiency.

4.2 Latest Tag Complete

Figure 7 shows a simple LF program with two federates. The reaction in federate s_s is triggered every 20 ms by the timer. Figure 8 illustrates the trace of the program when s_s sends messages every 100 ms starting from the tag $(100\ ms,\ 0)$. The trace only shows the first 100 ms of the execution without including the initialization process for simplicity. Since federate s_r purely reactive, it sends NET_{s_r} with the tag $(\infty,\ M_{\text{max}})$ in ① indicating that it has no events to process. This results in DNET_{s_s} in ② and let s_s skip sending any NET signals until it sends a message. At $(100\ ms,\ 0)$, a message is sent from s_s to s_r via LF-RTI and the new DNET_{s_s} with $(100\ ms,\ 0)$ in ③ is sent to s_s. In response to that DNET signal, s_s sends NET_{s_s} with the tag $(120\ ms,\ 0)$ in ⑤ so that LF-RTI can send TAG_{s_r} in ⑥.

Although DNET signals can eliminate unnecessary NET signals, LTC signals are also sent from the sender federate every time and this is redundant. When LF-RTI receives an LTC_j from a federate j, LF-RTI uses $G(\text{LTC}_j)$ for the following two purposes:

1. To remove tags from the in-transit message queue Q_j. Concretely, LF-RTI removes the tag $G(\text{LTC}_i)$ from Q_i.
2. To compute TAGs for federates i where $j \in U_i$.

For example, in Figure 8, with LTC_{s_s} with the tag $(100\ ms,\ 0)$ in ④, LF-RTI can send the TAG signal ⑥ as explained in Algorithm 1. In the trace, the TAG signal is sent after receiving the NET signal in ⑤ because, according to the behavior of federates in Figure 4, usually a federate sends LTC and NET signals together. The LTC signal in ⑦ is used for removing a tag from Q_{s_r}, the in-transit message queue of s_r. LF-RTI enqueues the tag $(100\ ms,\ 0)$ to Q_{s_r} when forwarding the tagged message to s_r. Upon receiving the LTC signal with $(100\ ms,\ 0)$, LF-RTI dequeues the tag as it knows that the federate receives and processes the message.

So LF-RTI has nothing to do with the other LTC signals with tags earlier than $(100\ ms,\ 0)$ (in the box labeled "No messages are produced until 100 ms" in Figure 8) and those redundant LTC signals incur network overhead since s_s sends them every 20 ms. We show how we can determine whether an LTC signal is essential and remove it in Section 5.2.2.

5 Proposed Approach

This section provides in-depth discussions of our approach to improving the efficiency of coordinating timed events in DDE systems.

5.1 Enabling Fine-grained Timing Control

A federate with any upstream federate must send a NET signal regardless of receiving DNET signals when the federate cannot advance to its next event tag. However, as described in Section 4.1, LF-RTI grants TAG signals too conservatively, which may result in the excessive delay of events. This section discusses a technique that helps federates avoid waiting for TAG signals unnecessarily.

To address this inefficiency, we make LF-RTI compute $G(\text{TAG}_{d_u})$ using B_{d_u}. Let us define a function LE for this computation. For a federate i, LE takes a tag $g = B_i$ as an input and returns the latest tag that LF-RTI can grant to i. For example, if B_i is $(1\ s,\ 1)$, $LE(B_i)$ will be $(1\ s,\ 0)$. Also, if B_i is $(2\ s,\ 0)$, $LE(B_i)$ will be $(2\ s - 1\ ns,\ M_{\text{max}})$. Formally, we define $LE\colon \mathbb{G} \longrightarrow \mathbb{G}$ as:

$$LE(g) = LE((t, m)) = \begin{cases} (-\infty,\ 0), & \text{if } t = 0 \text{ and } m = 0 \\ (t - 1, M_{\text{max}}), & \text{if } 0 < t < \infty \text{ and } m = 0 \\ (t, m - 1), & \text{if } 0 \le t < \infty \text{ and } m \neq 0 \\ (-\infty,\ 0), & \text{if } t = -\infty \\ (\infty,\ M_{\text{max}}) & \text{if } t = \infty \end{cases} \tag{6}$$

In most cases, $LE(g)$ returns the latest tag earlier than g. If g is $(-\infty,\ 0)$, $LE(g)$ must also be $(-\infty,\ 0)$ as there is no tag earlier than g. If g is $(\infty,\ M_{\text{max}})$, $LE(g)$ must be $(\infty,\ M_{\text{max}})$ because $(\infty,\ M_{\text{max}})$ of B_i means that i can advance to any tag. With the function LE, for each federate i, we can compute $G(\text{TAG}_i)$ by $LE(B_i)$.

Algorithm 1 shows the procedure of computing $G(\text{TAG}_i)$. As described in Section 3.4, LF-RTI first calculates $G(\text{TAG}_i)$ using LTC signals. The following Section 5.2.1 covers an in-depth discussion on computing $G(\text{TAG}_i)$ using LTC signals. When LF-RTI cannot grant a TAG signal using LTC signals, LF-RTI calculates B_i, the earliest incoming message tag of i. We improve the baseline we use by making LF-RTI grant $LE(B_i)$ instead of $\min(N_i,\ H(Q_i))$ as shown in Line 10.

By granting TAG signals using EIMT (B_{f_r}), as shown in Figure 9, we can eliminate redundant NET and TAG signals shown in Figure 6. In Figure 9, after LF-RTI receives signal ②, LF-RTI can send $\text{TAG}_{f_r}(1\ s - 1\ ns,\ M_{\text{max}})$ instead of signal ③. Then, f_r will not send NET signals unless f_r sends a message to f_s or its next event's tag is later than or equal to $(1\ s,\ 0)$.

5.2 Improving Efficiency of LTC Signals

Among the two purposes of LTC_j explained in Section 4.2 (i.e., 1. to remove tags from the in-transit message queue Q_j and 2. to

Algorithm 1 Computation of $G(\text{TAG}_i)$

```
1:  ▷ Get the possible TAG value using LTC signals.
2:  if TAGbyLTC() ≥ min(N_i, H(Q_i)) then
3:      ▷ Compute the maximum tag to grant using LTC signals.
4:      ▷ If that tag allows i to advance its tag, grant it.
5:      send TAG_i where G(TAG_i) = minLTC
6:  else
7:      ▷ Compute the EIMT of i for granting TAG signals.
8:      B_i ← min_{j∈U_i}(A(min(N_j, H(Q_j), B_j), D_ij))
9:      if B_i > min(N_i, H(Q_i)) then
10:         send TAG_i where G(TAG_i) = LE(B_i)
11:     end if
12: end if
13: ▷ A helper function for the main algorithm above.
14: function TAGbyLTC()
15:     result ← (∞, M_max)
16:     for j ∈ U_i do
17:         ▷ Compute the candidate using L_j and the delay.
18:         cand ← A(L_j, D_ij)
19:         if (0 < D_ij.time < ∞ ∧ L_j.microstep < M_max) ∨
              (D_ij == (0, 1) ∧ L_j.microstep == M_max − 1) then
20:             ▷ Adjust the candidate to avoid a wrong TAG signal.
21:             cand ← LE(cand)
22:         end if
23:         if result > cand then
24:             ▷ Find the minimum among candidates.
25:             result ← cand
26:         end if
27:     end for
28:     return result
29: end function
```

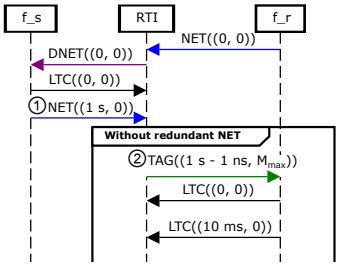


Figure 9: An execution trace of the program in Figure 5 without redundant NET signals.

compute TAGs for federates i where $j \in U_i$), only the former is the essential role of LTC_j . In other words, federate j does not have to send an LTC signal if that signal will not remove a tag from Q_j because LF-RTI can grant TAG signals without those LTC signals.

5.2.1 Compute $G(TAG)$ using LTC signals. Lines 14-29 in Algorithm 1 describe the function to compute the TAG_i value using LTC signals. For each federate $j \in U_i$, LF-RTI finds the tag g that j will not send MSG_{ij} with tags earlier than or equal to the tag g. Notice that if j is the immediate upstream of i, D_{ij} can only be $(0,\ 0)$, $(0,\ 1)$, and $(t,\ 0)$ where $0 < t < \infty$. Naively, given that j completes the tag L_j, LF-RTI can predict that j will not send any MSG_{ij} with tags earlier than equal to $A(L_j,\ D_{ij})$.

However, the federate j may send MSG_{ij} with the tag $A(L_j,\ D_{ij})$ (but not earlier) in the following cases:

- D_{ij} is $(t,\ 0)$ where $t > 0$ and L_j is $(t_j,\ m_j)$ where $0 \le t_j < \infty$ and $m_j < M_{\text{max}}$. Suppose j has an event at $(t_j,\ m_j + 1)$. The tag $A(L_j,\ D_{ij})$ is $(t + t_j,\ 0)$ since the delay greater than 0 ignores the originated microstep. Federate j may send another message at $(t + t_j,\ 0)$ cause $A((t_j,\ m_j + 1),\ D_{ij})$ is also $(t + t_j,\ 0)$
- D_{ij} is $(0,\ 1)$ and L_j is $A(t,\ M_{\text{max}} - 1)$ where $0 \le t < \infty$. The tag $A(L_j,\ D_{ij})$ is $(t_j,\ M_{\text{max}})$. Assume j has an event at $(t,\ M_{\text{max}})$. Federate j may send another message at $(t,\ M_{\text{max}})$ because the microstep is saturated to M_{max}.

Consequently, the tag g must be $LE(A(L_j,\ D_{ij}))$ in these situations as shown in lines **??**-22 of Algorithm 1.

To demonstrate that computing $G(\text{TAG})$ values is not the essential role of LTC signals, we present the following Theorem 1 proving that the potential range of $G(\text{TAG}_i)$ computed by B_i includes the range of possible $G(\text{TAG}_i)$ attained by L_j.

Theorem 1. *The $G(TAG_i)$ value computed by L_j where $j \in U_i$ is always earlier than or equal to the $G(TAG_i)$ value computed by B_i.*

Assumption 1. *When federate j finishes a tag, it sends LTC_j and NET_j together, and LF-RTI receives them almost at the same time.*

Assumption 2. *When LF-RTI receives LTC_j and NET_j, LF-RTI removes the tag $G(LTC_j)$ from Q_j and computes the TAG_j value.*

Lemma 1.1. *When LF-RTI computes a $G(TAG_i)$ value, for every federate $j \in U_i$, L_j is earlier than equal to $LE(\min(N_j, H(Q_j), B_j))$.*

Proof of Lemma 1.1. L_j is the tag that j has already finished and $\min(N_j, H(Q_j), B_j)$ is the earliest tag that j is expected to advance. Thus, L_j must be strictly earlier than $\min(N_j, H(Q_j), B_j)$ and earlier than or equal to $LE\ \min(N_j, H(Q_j), B_j)$. □

Lemma 1.2. *Let g_{LTC_j} be the tag such that LF-RTI guarantees that federate $j \in U_i$ will not send MSG_{ij} with any tags earlier than or equal to the tag g_{LTC_j} using L_j. Similarly, let g_{NET_j} be the tag that LF-RTI assures that federate $j \in U_i$ will not send MSG_{ij} with any tags earlier than or equal to these tags. The tag g_{LTC_j} is always earlier than or equal to the tag g_{NET_j}.*

Proof of Lemma 1.2. Suppose L_j is $(t_l,\ m_l)$ and D_{ij} is $(t_d,\ m_d)$. Moreover, let $g_n = (t_n,\ m_n)$ denote $\min(N_j, H(Q_j), B_j)$. We know that g_n is strictly later than L_j by Lemma 1.1. Using g_n, LF-RTI predicts that the earliest possible tag of MSG_{ik} is $A(g_n, D_{ik})$ and hence, g_{NET} is $LE(A(g_n,\ D_{ik}))$. Consider the following two cases depending on how we compute g_{LTC}.

1. The tag g_{LTC} is computed by $LE(A(L_k, D_{ik}))$ as D_{ij} and L_j satisfies the condition described in Lines **??**-19. We already know that g_{NET_j} is computed by $LE(A(g_n, D_{ij}))$ and $L_j < g_n$. $A(L_k, D_{ik})$ can be identical to $A(g_n, D_{ik})$ if the time values are identical. For example, if L_k is $(0,\ 0)$, g_n is $(0,\ 3)$ and D_{ij} is $(1,\ 0)$, both $A(L_k, D_{ik})$ and $A(g_n, D_{ik})$ are $(1,\ 0)$. However, $A(L_k, D_{ik})$ **cannot** be later than $A(g_n, D_{ik})$,
2. The tag g_{LTC_j} is computed by $A(L_j, D_{ij})$. In this case, D_{ij} is $(0,\ 0)$ or D_{ij} is $(0,\ 1)$ while L_j is *not* $(t,\ M_{\text{max}} - 1)$ where $0 \le t < \infty$. When there is no delay, i.e., $D_{ij} = (0,\ 0)$, g_{LTC_j} is L_j and g_{NET_j} is $LE(A(g_n, D_{ij}))$, which is $LE(g_n)$. By Lemma 1.1, $g_{LTC} = L_k$ is earlier than or equal to $g_{NET} = LE(g_n)$.

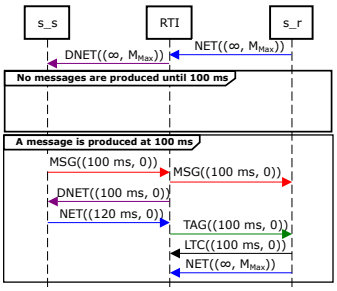


Figure 10: An execution trace of the program in Figure 7 without redundant LTC signals.

When there is a microstep delay, i.e., $D_{ij} = (0,\ 1)$, g_{LTC} would be $A(L_k, (0,\ 1))$; we need to consider two cases, whether an overflow occurs in the microstep or not:

- If L_k is $(t_l,\ M_{\text{max}})$ where $0 \leq t_l < \infty$, the tag g_{LTC}, which is $A(L_j, D_{ij})$, is $(t_l,\ M_{\text{max}})$ cause the microstep is saturated to M_{max}. The earliest possible g_n is $(t_l{+}1,\ 0)$ because g_n must be strictly later than L_j. Thus, g_{NET} must be later than or equal to $LE(A((t_l + 1,\ 0), (0,\ 1)))$, which is $(t_l + 1,\ 0)$. Definetely the tag g_{NET}, $(t_l + 1,\ 0)$, is later than g_{LTC}, $(t_l,\ M_{\text{max}})$.
- If L_k is $(t_l,\ m_l)$ where $0 \leq m_l < M_{\text{max}} - 1$, the tag g_{LTC} is $A((t_l,\ m_l), (0,\ 1))$, which is $(t_l,\ m_l{+}1)$. Similarly, the earliest possible g_n is $(t_l,\ m_l + 1)$ and g_{NET} must be later than or equal to $LE(A(g_n, D_{ik}))$, which is $(t_l,\ m_l + 1)$. So the tag g_{NET}, $(t_l,\ m_l + 1)$, must be later than g_{LTC}, $(t_l,\ m_l + 1)$.

Consequently, in every situation, the tag g_{NET_j} is later than or equal to the tag g_{LTC_j}. □

Proof of Theorem 1. The TAG_i value computed by LTC signals is $\min_{k \in U_i}(g_{LTC_k})$ and the TAG_i value computed by EIMT is $\min_{k \in U_i}(g_{NET_k})$. Thus, by Lemma 1.2, the potential range of $G(\text{TAG}_i)$ computed by B_i includes the range of possible $G(\text{TAG}_i)$ attained by L_j. □

According to Theorem 1, we know $G(\text{TAG}_i)$ computed by B_i is more efficient than $G(\text{TAG}_i)$ computed by LTC signals, in that the former always allows federate i to advance to a later (or at least the same) tag. However, Theorem 1 is valid only if federates send LTC signals to remove tags from the in-transit message queue correctly. Section 5.2.2 explains how we can assure a federate i sends an LTC_i signal only when that signal removes any tag from Q_i.

5.2.2 Latest Tag Confirmed. When LF-RTI receives LTC_i with a tag greater than or equal to $H(Q_i)$, LF-RTI can remove the tag $H(Q_i)$ from Q_i. Hence, when a federate i completes a tag g_i, the federate i needs to send LTC_i only if g_i is later than or equal to $H(Q_i)$. Now, the signal LTC_i *confirms* that a federate has received tagged messages with tags earlier than or equal to $G(\text{LTC}_i)$ and completed the tag $G(\text{LTC}_i)$, rather than just informing of i's latest completed tag. Thus, we also rename the signal to Latest Tag *Confirmed* (with the same acronym of LTC) to better reflect the role of this signal.

A simple way to send LTC signals considering the in-transit message queue is to let each federate i maintain a queue of tagged messages they have received. Intuitively, this is federate i's copy of LF-RTI's Q_i. Let Q_i' be the queue maintained by federate i. When federate i completes a tag g_i, it checks $H(Q_i')$, the head of Q_i', and sends LTC_i with g_i if $g_i = H(Q_i')$.

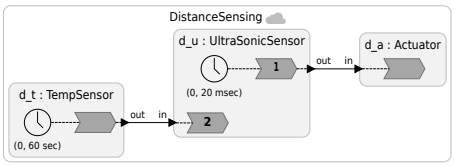


Figure 11: An LF program consists of a temperature sensor, an ultrasonic sensor, and an actuator.

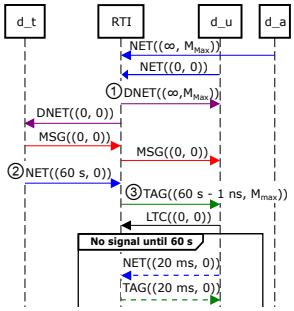


Figure 12: An execution trace of the LF program in Figure 11.

The only downside of this approach is the memory overhead to maintain the copies of LF-RTI's in-transit message queues. In LF, each tag occupies 12 Bytes of memory. To reduce such memory overhead, we leverage LF's network input reaction, a special reaction generated for each input port to receive input network messages. When a federate receives a tagged message with a tag g_i via an input port p, the network input reaction for the port p is scheduled at the tag g_i. That is, if any of the network input reactions is triggered at the tag g_i, the federate knows that it has received a tagged message with the tag g_i. Thus, the federate only needs to maintain one global flag indicating whether a network input reaction has been triggered at the current tag. When the federate completes the current tag and the global flag is 1, the federate sends an LTC with the completed tag and resets the global flag to 0.

Figure 10 shows the trace when we apply the Latest Tag Confirmed signal to an LF program in Figure 7. As s_s has no upstream federate, it does not receive any messages and does not need to send any LTC signals. On the other hand, s_r sends $\text{LTC}_{s_r}((100\ ms,\ 0))$ because it has received the message with the tag $(100\ ms,\ 0)$.

6 Case Studies

In this section, we introduce case studies to show how our approach can be applied to the real-world scenarios of distributed CPS.

6.1 Distance Sensing System

Figure 11 shows an LF program that senses an object using an ultrasonic sensor. Federate d_t measures the ambient temperature and reports it to federate d_u to compensate for the temperature effect on the ultrasonic sensor. Examples of such systems include industrial high-accuracy ultrasonic ranging sensors, such as Siemens SITRANS LU150/LU180 ultrasonic level transmitters[1] used in various applications [32, 34]. In this example, Federate d_u polls the sensor every 20 ms and sends a tagged message to d_a if the distance is smaller than the given threshold (for handling an emergency).

[1]https://www.siemens.com/us/en/products/automation/process-instrumentation/level-measurement/continuous/ultrasonic/sitrans-lu150-lu180.html

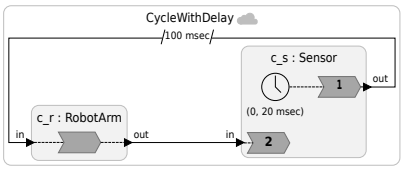


Figure 13: An LF program for defect detection in Figure 1a.

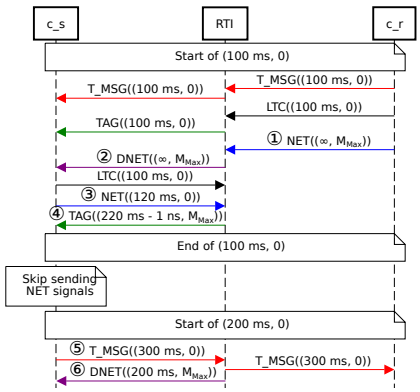


Figure 14: An execution trace of the LF program in Figure 13.

Federate d_t polls the temperature sensor every minute because the temperature sensing does not require a high resolution.

Figure 12 shows the trace of the program in Figure 11. Thanks to $\text{DNET}_{d_u}((\infty,\ M_{\text{max}}))$ in ①, d_u does not need to send NET_{d_u} unless it should request additional TAG_{d_u} signal to advance its tag. LF-RTI sends $\text{TAG}_{d_u}((60\ ns - 1,\ 0))$ in ③ after receiving $\text{NET}_{d_t}((60\ s,\ 0))$ in ② because B_{d_u} is $(60\ ms,\ 0)$ and d_u can advance its tag to any tags earlier than $(60\ s,\ 0)$. Without our approach, the $G(\text{TAG}_{d_u})$ would be $(0,\ 0)$, which is N_{d_u}, and federate d_u and LF-RTI needs to exchange NET and TAG signals every 20 ms. The NET and TAG signal in the box marked *No signal until 60 s* shows the exchange of signals in the situation that the TAG_{d_u} value is decided by NET_{d_u} instead of $LE(B_{d_u})$.

6.2 Cycle with Delay

We can also exploit our solution for federates composing a cycle with delays, like the LF program in Figure 13. The LF program is designed for the defective product detection system in Figure 1a. A sensor scans products and notifies the robot arm if there is a defective product. Because the sensor and the robot are apart by some distance, we add a 100 ms logical delay between federate c_s and c_r. The robot picks out the defective product 100 ms after the sensor detects it. Federate c_r sends a message to federate c_s to confirm the robot's operation. If the robot fails to pick out the target product, the system needs to stop the conveyor belt and invoke an error handling protocol. For simplicity, the program in Figure 13 only shows the robot arm and the sensor.

Figure 14 shows an execution trace of the program. We assume that defective products appear and c_s send messages every 200 ms starting from $(0,\ 0)$. At $(100\ ms,\ 0)$, LF-RTI receives $\text{NET}_{c_r}((\infty,\ M_{\text{max}}))$ in ①, resulting in $\text{DNET}_{c_s}((\infty,\ M_{\text{max}}))$ in ②. Also, c_s sends $\text{NET}_{c_s}((120\ ms,\ 0))$ in ③ after completing $(100\ ms,\ 0)$. Then LF-RTI sends $\text{TAG}_{c_s}((220\ ms - 1\ ns,\ M_{\text{max}}))$, calculated by $LE(B_{c_s}) = LE((220\ ms, 0))$. As a result of signals ② and ④, c_s skips sending NET_{c_s} until $(200\ ms,\ 0)$.

The message and signal ⑤ and ⑥ show how the DNET value is calculated when there is a delay on a connection. After LF-RTI receives the message ⑤, $\min(N_{c_r}, H(Q_{c_r}))$ is $(300\ ms,\ 0)$ and the delay, $D_{c_rc_s}$, is $(100\ ms,\ 0)$. Thus, LF-RTI computes the DNET value as $S((300\ ms,\ 0), (100\ ms,\ 0)) = (200\ ms,\ M_{\text{max}})$.

7 Evaluation

For evaluation, we compare the following approaches:

1. HLA-like: the current implementation of LF
2. SOTA: the implementation with DNET (explained in Section 3.6) applied to the HLA-like implementation
3. Our solution: the approach presented in Section 5

We note that neither HLA-like nor SOTA approaches implement the traditional HLA. However, the HLA-like approach coordinates timing in a manner similar to HLA with minor modifications described in Section 3.5 with the same core timing mechanisms. Thus, we expect the performance differences observed in our experiments to be consistent with those found when comparing against a traditional HLA implementation.

We select the three examples in Figure 7, Figure 11, and Figure 13 as benchmarks. We simulate the examples and compare the lag, defined as the time difference between the physical time (wall-clock time) and the logical time of reactions responsible for polling a sensor. Specifically, we measure the lag in:

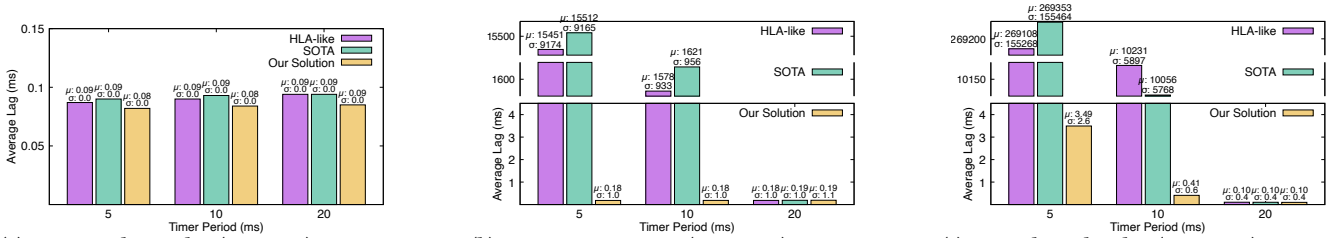
- The reaction of federate s_s in SporadicSender (Figure 7),
- Reaction 1 of federate d_u in DistanceSensing (Figure 11), and
- Reaction 1 of federate c_s in CycleWithDelay (Figure 13).

If the lags in those reactions exceed the timer period, the system will fail to meet the real-time requirements. Also, we evaluate the efficiency of our solution by comparing the number of control signals, NET, LTC, TAG, and DNET signals.

We run the federates and LF-RTI on a Raspberry Pi 4 Model B 4GB, communicating through the loopback interface (127.0.0.1). For realistic network conditions, we emulate artificial network latency using NetEm, Linux's traffic control (tc) tool [19]. As 10 ms of end-to-end latency is required for industrial automation [16], we set a mean delay to 5 ms to achieve a 10 ms end-to-end delay between federates (through LF-RTI). Additionally, we add a jitter of ±1 ms with the tool NetEm [19] for a more realistic simulation.

We note that clock synchronization is critical in real-world distributed CPS. Although there is no clock synchronization error in our experiments because every process runs on the same device, we believe the real-world CPS can also achieve the same behavior with the network time protocol (NTP) [33] over the local network, which provides time synchronization under 1 ms, or the hardware-assisted precision time protocol (PTP) [1], which provides synchronization precision below ten nanoseconds.

In each example, we assume that actual messages are sent every 5 seconds (e.g., whenever the sensor detects events that require communication) while we adjust the sensing intervals (or timer periods). However, in real-world scenarios, actual detection often happens far less frequently. For instance, in a smart factory's defect detection system or a vehicle's emergency braking system, defects or too-close objects ideally do not appear every 5 seconds.



(a) In SporadicSender (Figure 7): Average lag of the reaction of federate *s_s*.

(b) In DistanceSensing (Figure 11): Average lag of Reaction 1 of federate *d_u*.

(c) In CycleWithDelay (Figure 13): Average lag of Reaction 1 of federate *c_s*.

Figure 15: Comparison of the average lags of our solution and two baselines, HLA-like and SOTA for various timer periods.

SporadicSender (Figure 7)

HLA-like						SOTA						Our Solution					
Timer Period	5ms	10ms	20ms	50ms	100ms	Timer Period	5ms	10ms	20ms	50ms	100ms	Timer Period	5ms	10ms	20ms	50ms	100ms
NET	100,159	50,155	25,161	10,152	5,163	NET	654	458	346	261	255	NET	647	450	357	261	271
LTC	100,102	50,102	25,102	10,102	5,102	LTC	100,102	50,102	25,102	10,102	5,102	LTC	100	100	100	100	100
TAG	294	195	102	102	102	TAG	296	197	102	102	102	TAG	291	192	101	101	101
DNET	0	0	0	0	0	DNET	199	200	200	200	199	DNET	200	200	200	200	200
Total	200,555	100,452	50,365	20,356	10,367	Total	101,251	50,957	25,750	10,665	5,658	Total	1,238	942	758	662	672

DistanceSensing (Figure 11)

HLA-like						SOTA						Our Solution					
Timer Period	5ms	10ms	20ms	50ms	100ms	Timer Period	5ms	10ms	20ms	50ms	100ms	Timer Period	5ms	10ms	20ms	50ms	100ms
NET	48,049	48,113	25,825	10,784	5,820	NET	48,080	48,038	25,811	10,787	5,802	NET	1,326	1,306	1,306	1,329	1,300
LTC	100,512	50,512	25,512	10,512	5,512	LTC	100,512	50,512	25,512	10,512	5,512	LTC	509	509	509	509	509
TAG	98,938	50,633	25,510	10,504	5,504	TAG	97,504	50,540	25,511	10,504	5,504	TAG	520	520	520	519	520
DNET	0	0	0	0	0	DNET	714	954	1,010	1,010	1,010	DNET	1,009	1,010	1,010	1,009	1,010
Total	247,499	149,258	76,847	31,800	16,836	Total	246,810	150,044	77,844	32,813	17,828	Total	3,364	3,345	3,345	3,366	3,339

CycleWithDelay (Figure 13)

HLA-like						SOTA						Our Solution					
Timer Period	5ms	10ms	20ms	50ms	100ms	Timer Period	5ms	10ms	20ms	50ms	100ms	Timer Period	5ms	10ms	20ms	50ms	100ms
NET	98,440	49,414	25,150	10,173	5,166	NET	98,425	49,405	25,167	10,167	5,172	NET	5,284	5,290	5,294	5,216	5,159
LTC	100,102	50,102	25,102	10,102	5,102	LTC	100,102	50,102	25,102	10,102	5,102	LTC	198	198	198	198	198
TAG	98,476	49,458	25,503	10,203	5,103	TAG	98,442	49,427	25,503	10,203	5,103	TAG	5,337	5,339	5,338	5,265	5,202
DNET	0	0	0	0	0	DNET	201	201	301	301	301	DNET	301	301	301	301	301
Total	297,018	148,974	75,755	30,478	15,371	Total	297,170	149,135	76,073	30,773	15,678	Total	11,120	11,128	11,131	10,980	10,860

Table 4: Number of exchanged signals during the 500 sec of runtime with timer periods from 5 ms to 100 ms.

Figure 15 illustrates the measured lags when the timer period is 5, 10, and 20 ms. The graphs do not show the results of 50 and 100 ms because they are almost identical to the result of 20 ms. Figure 15a demonstrates the lag of federate *s_s*'s reaction of the program in Figure 7. The average lag is always around 0.09 ms, negligible compared to the timer periods, even with the 5 ms period, because *s_s* can advance its tag without TAG signals.

Our experimental results demonstrate that our solution allows fine-grained timing control in distributed CPS with fine-grained timer periods even in challenging networks. Specifically, Figure 15b and Figure 15c show the average lags measured in reactions of the programs in Figure 11 and Figure 13. Federates *d_s* and *c_s* suffer excessive lags when the timer period is 5 ms or 10 ms because they must receive TAG signals from LF-RTI to advance their tag, rendering the fine-grained timing control infeasible. In the HLA-like and SOTA implementations, a federate waits for one round trip delay because of sending NET and receiving TAG in order to advance its tag each time. Thus, if the sensing period is smaller than the round trip delay, the lag grows unboundly. We note that the system cannot recover from the accumulated lag unless it periodically drops piled up events, which violates LF execution semantics. However, with our solution, a federate can advance its tag without having to wait for LF-RTI's TAG signal every time, thanks to TAG signals computed by EIMT. This enables the system to operate with higher-resolution sensing even under challenging network conditions.

Also, the experimental results show that our solution drastically reduces the communication overhead for timing information. Table 4 shows the number of signals for each signal type. Our solution significantly reduces the total number of signals for each example compared to the HLA-like and SOTA implementations. In SporadicSender (Figure 7), DistanceSensing (Figure 11), and CycleWithDelay (Figure 13), the number of signals is reduced by 8.4 to 162.0 times, 5.0 to 73.6 times, and 1.4 to 26.7 times, respectively. The results show that our approach is more effective as message-sending events become relatively sparser (i.e., the timer period becomes shorter with the constant message-sending period of 5 seconds).

8 Conclusion

This paper proposes an efficient coordination scheme for timed events in DDE systems to enable fine-grained timing control in runtime deployment as well as in simulation. We outline the inefficient use of signals for timing coordination in centralized DDE systems, which incurs excessive cost and, in the worst case, makes it infeasible to build systems requiring high-precision timing control. Our evaluation using the state-of-the-art open-source DDE system and real-world use cases confirms our solution eliminates the inefficiencies by reducing the overhead of network signals and enabling precise timing control even under challenging environments with a delay greater than the sensing period and a moderate level of jitter.

Acknowledgments

This work was supported in part by the NSF I/UCRC for Intelligent, Distributed, Embedded Applications and Systems (IDEAS), NSF award #2231620, NSF award #CNS-2233769 (Consistency vs. Availability in Cyber-Physical Systems), and the iCyPhy Research Center (Industrial Cyber-Physical Systems) at UC Berkeley.

References

[1] 2020. IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems. *IEEE Std 1588-2019 (Revision ofIEEE Std 1588-2008)* (2020), 1–499. doi:10.1109/IEEESTD.2020.9120376

[2] Gul A. Agha. 1997. *Abstracting Interaction Patterns: A Programming Paradigm for Open Distributed Systems.* Springer US, Boston, MA, 135–153. doi:10.1007/978-0-387-35082-0_10

[3] Muhammad Usman Awais, Wolfgang Mueller, Atiyah Elsheikh, Peter Palensky, and Edmund Widl. 2013. Using the HLA for distributed continuous simulations. In *2013 8th EUROSIM Congress on Modelling and Simulation*. IEEE, 544–549.

[4] Javier Barrachina, Piedad Garrido, Manuel Fogue, Francisco J Martinez, Juan-Carlos Cano, Carlos T Calafate, and Pietro Manzoni. 2013. Road side unit deployment: A density-based approach. *IEEE Intelligent Transportation Systems Magazine* 5, 3 (2013), 30–39.

[5] Soroush Bateni, Marten Lohstroh, Hou Seng Wong, Hokeun Kim, Shaokai Lin, Christian Menard, and Edward A. Lee. 2023. Risk and Mitigation of Nondeterminism in Distributed Cyber-Physical Systems. In *ACM-IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE)*. doi:10.1145/3610579.3613219

[6] Albert Benveniste, Paul Caspi, Stephen A Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert De Simone. 2003. The synchronous languages 12 years later. *Proc. IEEE* 91, 1 (2003), 64–83.

[7] Torsten Blochwitz, Martin Otter, Martin Arnold, Constanze Bausch, Christoph Clauß, Hilding Elmqvist, Andreas Junghanns, Jakob Mauss, Manuel Monteiro, Thomas Neidhold, et al. 2011. The functional mockup interface for tool independent exchange of simulation models. In *Proceedings of the 8th international Modelica conference*. Linköping University Press, 105–114.

[8] Alisson V Brito, Harald Bucher, Helder Oliveira, Luis Feliphe S Costa, Oliver Sander, Elmar UK Melcher, and Juergen Becker. 2015. A distributed simulation platform using HLA for complex embedded systems design. In *2015 IEEE/ACM 19th International Symposium on Distributed Simulation and Real Time Applications (DS-RT)*. IEEE, 195–202.

[9] Janette Cardoso and Pierre Siron. 2018. Ptolemy-HLA: A cyber-physical system distributed simulation framework. *Principles of Modeling: Essays Dedicated to Edward A. Lee on the Occasion of His 60th Birthday* (2018), 122–142.

[10] K.M. Chandy and J. Misra. 1979. Distributed Simulation: A Case Study in Design and Verification of Distributed Programs. *IEEE Transactions on Software Engineering* SE-5, 5 (1979), 440–452. doi:10.1109/TSE.1979.230182

[11] Jean-Baptiste Chaudron, Martin Adelantado, Eric Noulard, and Pierre Siron. 2011. HLA high performance and real-time simulation studies with CERTI. (2011).

[12] Jean-Baptiste Chaudron, David Saussié, Pierre Siron, and Martin Adelantado. 2014. Real-time distributed simulations in an HLA framework: Application to aircraft simulation. *Simulation* 90, 6 (2014), 627–643.

[13] Jean-Baptiste Chaudron, Pierre Siron, and Martin Adelantado. 2018. Analysis and Optimization of time-management services in CERTI 4.0. (2018).

[14] Fabio Cremona, Marten Lohstroh, David Broman, Edward A Lee, Michael Masin, and Stavros Tripakis. 2019. Hybrid co-simulation: it's about time. *Software & Systems Modeling* 18 (2019), 1655–1679.

[15] Peter Donovan, Erling Jellum, Byeonggil Jun, Hokeun Kim, Edward A Lee, Shaokai Lin, Marten Lohstroh, and Anirudh Rengarajan. 2024. Strongly-Consistent Distributed Discrete-event Systems. *arXiv preprint arXiv:2405.12117* (2024).

[16] Daquan Feng, Lifeng Lai, Jingjing Luo, Yi Zhong, Canjian Zheng, and Kai Ying. 2021. Ultra-reliable and low-latency communications: applications, opportunities and challenges. *Science China Information Sciences* 64, 2 (20 Jan 2021), 120301. doi:10.1007/s11432-020-2852-1

[17] Richard M. Fujimoto. 1998. Time Management in The High Level Architecture. *SIMULATION* 71, 6 (1998), 388–400. doi:10.1177/003754979807100604 arXiv:https://doi.org/10.1177/003754979807100604

[18] Moritz Gütlein, Wojciech Baron, Christopher Renner, and Anatoli Djanatliev. 2020. Performance Evaluation of HLA RTI Implementations. In *2020 IEEE/ACM 24th International Symposium on Distributed Simulation and Real Time Applications (DS-RT)*. 1–8. doi:10.1109/DS-RT50469.2020.9213641

[19] Stephen Hemminger et al. 2005. Network emulation with NetEm. In *Linux conf au*, Vol. 5. 2005.

[20] IEEE. 2010. IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA)– Framework and Rules. *IEEE Std 1516-2010 (Revision of IEEE Std 1516-2000) - Redline* (2010), 1–38. doi:10.1109/IEEESTD.2010.5953411

[21] Shafagh Jafer, Qi Liu, and Gabriel Wainer. 2013. Synchronization methods in parallel and distributed discrete-event simulation. *Simulation Modelling Practice and Theory* 30 (2013), 54–73.

[22] Byeonggil Jun, Edward A. Lee, Marten Lohstroh, and Hokeun Kim. 2024. Efficient Coordination for Distributed Discrete-Event Systems. In *ACM-IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE)*.

[23] Christoph M. Kirsch and Ana Sokolova. 2012. *The Logical Execution Time Paradigm.* Springer Berlin Heidelberg, Berlin, Heidelberg, 103–120. doi:10.1007/978-3-642-24349-3_5

[24] Gilles Lasnier, Janette Cardoso, Pierre Siron, Claire Pagetti, and Patricia Derler. 2013. Distributed simulation of heterogeneous and real-time systems. In *2013 IEEE/ACM 17th International Symposium on Distributed Simulation and Real Time Applications*. IEEE, 55–62.

[25] Edward A Lee, Jie Liu, Lukito Muliadi, Haiyang Zheng, and Cladius Ptolemaeus. 2014. Discrete-event models. *System Design, Modeling, and Simulation using Ptolemy II* (2014).

[26] Stan Liao, Steve Tjiang, and Rajesh Gupta. 1997. An efficient implementation of reactivity for modeling hardware in the Scenic design environment. In *Proceedings of the 34th annual Design Automation Conference*. 70–75.

[27] Marten Lohstroh. 2020. *Reactors: A Deterministic Model of Concurrent Computation for Reactive Systems.* Ph. D. Dissertation. EECS Department, University of California, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2020/EECS-2020-235.html

[28] Marten Lohstroh, Edward A. Lee, Stephen Edwards, and David Broman. 2023. Logical Time for Reactive Software. In *Workshop on Timing-Centric Reactive Software (TCRS), in Cyber-Physical Systems and Internet of Things Week (CPSIoT)*. ACM. doi:10.1145/3576914.3587494

[29] Marten Lohstroh, Christian Menard, Soroush Bateni, and Edward A. Lee. 2021. Toward a Lingua Franca for Deterministic Concurrent Systems. *ACM Trans. Embed. Comput. Syst.* 20, 4, Article 36 (May 2021), 27 pages. doi:10.1145/3448128

[30] Marten Lohstroh, Íñigo Íncer Romeo, Andrés Goens, Patricia Derler, Jeronimo Castrillon, Edward A. Lee, and Alberto Sangiovanni-Vincentelli. 2020. Reactors: A Deterministic Model for Composable Reactive Systems. In *Cyber Physical Systems. Model-Based Design*. Springer International Publishing, Cham, 59–85.

[31] Romolo Marotta, Federica Montesano, and Francesco Quaglia. 2023. Effective Access to the Committed Global State in Speculative Parallel Discrete Event Simulation on Multi-core Machines. In *Proceedings of the 2023 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation* (Orlando, FL, USA) *(SIGSIM-PADS '23)*. Association for Computing Machinery, New York, NY, USA, 107–117. doi:10.1145/3573900.3591117

[32] Stephen Milligan, Henry Vandelinde, and Michael Cavanagh. 2013. *Understanding ultrasonic level measurement.* Momentum Press.

[33] D.L. Mills. 1991. Internet time synchronization: the network time protocol. *IEEE Transactions on Communications* 39, 10 (1991), 1482–1493. doi:10.1109/26.103043

[34] B Nalini, V Nandhini, E Kavitha, and R Chandralekha. 2014. Implementation of Temperature Compensation Technique with Ultrasonic Ranging for Obstacle Identification. *International Journal for Research and Development in Engineering (IJRDE)* (2014), 230–234.

[35] Eric Noulard, Jean-Yves Rousselot, and Pierre Siron. 2009. CERTI, an Open Source RTI, why and how. (2009).

[36] Portico Project. 2016. Portico RTI: Open Source HLA Run-Time Infrastructure, Version 2.1.0. https://github.com/openlvc/portico. Accessed: April 13, 2025.

[37] Claudius Ptolemaeus. 2014. *System design, modeling, and simulation: using Ptolemy II.* Vol. 1. Ptolemy. org Berkeley.

[38] Karen Rudie, Stéphane Lafortune, and Feng Lin. 2003. Minimal communication in a distributed discrete-event system. *IEEE transactions on automatic control* 48, 6 (2003), 957–975.

[39] Nikolaos Tampouratzis, Ioannis Papaefstathiou, Antonios Nikitakis, Andreas Brokalakis, Stamatis Andrianakis, Apostolos Dollas, Marco Marcon, and Emanuele Plebani. 2020. A novel, highly integrated simulator for parallel and distributed systems. *ACM Transactions on Architecture and Code Optimization (TACO)* 17, 1 (2020), 1–28.

[40] Xiaoguang Wang, Stephen John Turner, Malcolm Yoke Hean Low, and Boon Ping Gan. 2004. Optimistic synchronization in HLA based distributed simulation. In *Proceedings of the eighteenth workshop on Parallel and distributed simulation*. 123–130.

[41] Bernard P Zeigler, Yoonkeon Moon, Doohwan Kim, and George Ball. 1997. The DEVS environment for high-performance modeling and simulation. *IEEE Computational Science and Engineering* 4, 3 (1997), 61–71.