

A Distributed Simulation Environment for Cyber-Physical Systems



DISC-IpSC Département Ingénierie Systèmes Complexes,
ISAE-SUPAERO

Supervised by: Janette Cardoso, Pierre Siron

LI Yanxuan

École nationale supérieure de mécanique et d'aérotechnique

A report for *Projet Fin d'étude* submitted for the degree of
Diplôme d'ingénieur & Master de Recherche

September 2015



Acknowledgements

During this internship, I'm so glad to participate in the development of "Ptolemy-HLA" project which is cooperation between University of Berkeley and ISAE-Supaéro. It enriches my professional experiences in various aspects.

I'm really grateful for my two advisors: Janette Cardoso and Pièrre Siron from ISAE-Supaéro. They've responded my questions in time, indicated the errors I've made, clarified my confusions and so on. They did give me a lot of help and guidance for my work. Their patience and rigor deeply impressed me.

Thanks to Christopher Brooks from University of Berkeley who taught me the right way to program and advised me for a better work.

I'd like to thank all my colleagues in my office whom I got along with, and all the people who helped me during my internship.

Thanks very much for my parents, staying by my side, supporting me and giving me a warm.

Abstract

High-Level Architecture is the standard for distribution simulation, which basically provides two mechanisms for time advancing: Next Time Request (NER) and Time Advance Request (TAR). Ptolemy (PTII) is the simulation platform for heterogeneous models. A co-framework called PTII-HLA is constructed, benefiting interoperability and reuse of HLA and heterogeneity of Ptolemy. The main goal of this internship is to improve the PTII-HLA simulation framework by implementing of the TAR mechanism. A particular attention will be done in the case there are simultaneous events which arrive at the same time.

Keywords: distributed simulation, HLA, time advancement, Time Advance Request (TAR)

Contents

1	Introduction	1
1.1	Context	1
1.2	Related Tools	1
1.3	Internship Objective	2
2	PTII-HLA	3
2.1	Overview of Ptolemy	3
2.1.1	Discrete-event Simulation	3
2.1.2	Event Processing	4
2.1.3	Actions for Event Processing	5
2.2	HLA Standard	6
2.2.1	HLA services	6
2.2.2	Time Management	6
2.3	PTII-HLA Co-simulation Framework	7
2.3.1	Ptolemy Components	7
2.3.2	View of Programming	8
2.3.3	Representation of Event Processing	13
3	Basic Analysis and Improvements for Next Event Request	16
3.1	Analysis of NER	16
3.1.1	Mechanism of NER	17
3.1.2	Example of NER	19
3.2	General Improvements	20
3.2.1	Asynchronism problem of PTII start Time	20
3.2.2	Reduction of HLA service calls	20
3.2.3	Rearrangement of HlaManager graphical user interface	21
4	Implementing Time Advance Request in PTII-HLA	23
4.1	Overview of TAR	23
4.1.1	Interests of TAR	23
4.1.2	Principal Mechanism of TAR	23
4.2	Assessing TAR	24
4.2.1	Possible Solutions for TAR Implementation	25

4.2.2	Chosen Approach	29
4.3	Implementation of TAR	30
4.3.1	Primary Results	30
4.3.2	Development	31
4.3.3	Programing	33
4.4	Example of TAR	33
4.5	Validation of TAR	34
4.6	Problem for stopTime	36
5	Conclusion	39
A	HlaManager.java	40
A.1	proposeTime(Time)	40
A.2	_eventsBasedTimeAdvance(Time)	41
A.3	_timeSteppedBasedTimeAdvance(Time)	43
A.4	updateHlaAttribute(HlaPublisher, Token, String)	45
A.5	_putReflectedAttributesOnHlaSubscribers()	47
A.6	Other methods	48

List of Figures

1.1	Co-simulation of Ptolemy and HLA	2
2.1	An example of ptolemy model	3
2.2	Director and actor's work mechanism	4
2.3	Actor's working process in Ptolemy with $t_1 \leq t_2 \leq t_3$	5
2.4	Different time scale with <code>hlaTimeUnitValue= 2</code>	10
2.5	Definition of time axis and related points	11
2.6	Actor only with time parameter	11
2.7	HlaSubscriber with only time parameter	12
2.8	HlaPublisher with only time parameter	13
2.9	A Ptolemy model and its execution	14
3.1	principal concept of NER	16
3.2	NER Time Advancement Diagram, where $f_{RAV}(t) = t$ and $f_{UAV}(t) = t + lah$	17
3.3	NER execution detail	18
3.4	NER Example	19
3.5	Two federates with different Ptolemy start time	20
3.6	HlaManager	22
4.1	principal concept of TAR	24
4.2	Concept of TAR: with UAV delay	26
4.3	Concept of TAR: with RAV delay	28
4.4	Different intervals for TAR	31
4.5	TAR Time Advancement Diagram	32
4.6	Example of TAR	33
4.7	SimpleProducerMultipleConsumerTAR	35
4.8	Results of SimpleProducerMultipleConsumerTAR	35

List of Tables

2.1	Actions for Event Processing	5
2.2	HLA services	6
2.3	PTII-HLA components	8
2.4	Time represent action for HLA and Ptolemy different time axis	15
4.1	Analysis of possible solutions	29

Chapter 1

Introduction

1.1 Context

A **cyber-physical system** (CPS) is an “intelligent” system which combines computing, networking with physical dynamics. With a deep cooperation between embedded computers and networks, they are capable to monitor and control the physical processes; meanwhile feedback loops allow the physical processes to affect the computation. Today, CPS can be found in various domains, such as aerospace, telemedicine, intelligent transportation system, and so on.

The complexity of CPS challenges its design which involves multi-faced subjects and integration difficulty. Generally, CPS is divided into multiple parts and is developed by different engineering teams. As an important means to validate CPS design, simulation requires diverse support for different way of working, even though the integration between these heterogeneous parts and the global evaluation, are not easy tasks. The interoperability is the base of CPS simulation and design.

In our case, we focus our attention on real-time and distributed CPS which is described by heterogeneous models. The simulation tool Ptolemy II is proposed under the condition of HLA standards.

1.2 Related Tools

Ptolemy II (PTII) [4] is an open source tool for designing, modeling, and simulation of concurrent, real-time, embedded, heterogeneous systems which has been developed by the University of California Berkeley (UCB). It's a suitable tool on modeling cyber-physical systems (CPS) which provides different **models of computation** (MoC) such as continuous time for describing physical properties, or discrete events for describing software and control [9].

The IEEE High-Level Architecture (HLA) standard refers to distributed simulation benefiting interoperability and reuse. It makes individual simulators interacting together during the same simulation regardless of their computing platform, such as simulation models, concrete functional codes (in C++, Java, etc.), and hardware equipment [1,9]. An individual simulator who complies

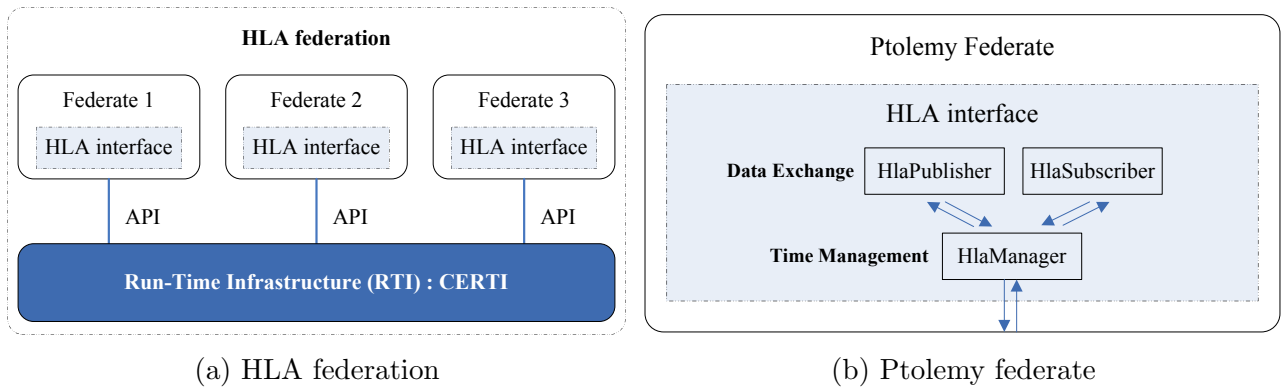


Figure 1.1: Co-simulation of Ptolemy and HLA

with HLA is called a **federate** and the global simulation is called a **federation** (see figure 1.1a). The interaction between these federates is managed by a run-time infrastructure (RTI). In our frame work, an open source RTI named CERTI has been used for a set of services based on HLA specifications.

A collaborative simulation framework called PTII-HLA combining PTII simulations with a distributed simulation provided by a HLA compliant RTI (CERTI) was developed in a joint work between UCB and ISAE [8]. It allows experimenting with heterogeneity of models of computation provided by Ptolemy and interoperability provided by HLA. In PTII-HLA framework, the HLA interface is made up by *HlaManager* which is in charge of time management forging links between Ptolemy and HLA services, and two actors *HlaPublisher* and *HlaSubscriber* which serve the data exchange between federates.

1.3 Internship Objective

HLA basically provides two mechanisms for time advancing: Next Time Request(NER) and Time Advance Request (TAR). The main goal of this internship is to improve the PTII-HLA simulation framework by implementing of the TAR mechanism. A particular attention will be done in the case there are simultaneous events.

The work plan is organized as follows. In chapter 2, a global view of PTII-HLA framework is done to know how they work individually and cooperatively. Chapter 3 shows the working mechanism of NER, meanwhile some improvements has been done after the basic analysis. With the reason of different implementation of TAR mechanism, a discussion is presented in chapter 4 where our approach is detailed. In the end, we make a conclusion in chapter 5. All related java code is presented in appendix A.

Chapter 2

PTII-HLA

2.1 Overview of Ptolemy

In Ptolemy II (PTII) [4], a **model of computation** (MoC) defines the rules for the interaction between modeling components. PTII supports numerous MoC like synchronized dataflow (SDF), dynamic dataflow (DDF), discrete-event (DE), continuous etc, which enables heterogeneous, concurrent modeling and design. Ptolemy models are based on **actor-oriented models**. **Actors** are components that execute concurrently and share data with each other by sending messages via **ports**. A **director** is a component in model to specify its MoC determining how actors communicate with each other. PTII provides a **superdense time** for a model of time expressed by a pair of parameters (t, n) with $t \in \mathbb{R}^+$, $n \in \mathbb{N}^*$, called a **time stamp**, where t is the **model time**, and n is a **microstep** [4]. An simple Ptolemy model with two actors and a discrete-event DE director, is represented in figure 2.1.

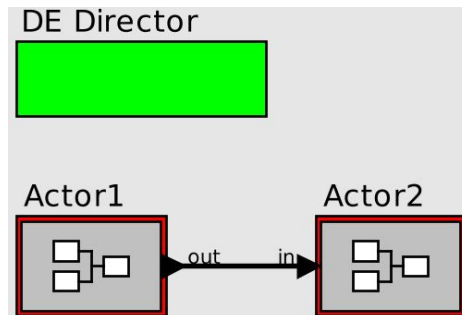


Figure 2.1: An example of ptolemy model

2.1.1 Discrete-event Simulation

The **discrete-event simulation** (DES) models the operation of a system as a discrete sequence of events in time. Each event occurs at a particular instant in time and marks a change of state in the system.

In PTII, each event is represented as a tuple: a time stamp and a value, denoted by (t, n, val) . All events are stored in a queue called `CalendarQueue` as $(t, n, val, actor)$ where “actor” means the destination actor the event is addressed to. Let us consider that $e1(t_1, n_1, val_1, actor_1)$, $e2(t_2,$

$n_2, val_1, actor_1$) are two events in the queue. All events are sorted by time stamp order in the following way:

- If $t_1 < t_2$ or if $t_1 = t_2, n_1 < n_2$, e_1 will be executed before e_2 .
- If $t_1 = t_2, n_1 = n_2$, the director chooses the execution order through actors' ranking, more details in [4].

The rank wipes off the indeterminism of **simultaneous events** which occur at the same time.

The simulation will takes the earliest event e_i in the `CalendarQueue`, advances its model time to t_i , updates its state, adds new events (if any) in the queue. It does this sequential actions until the end of the simulation or when the queue is empty.

2.1.2 Event Processing

A **DE director** is used in Ptolemy to govern the processing of DES simulation. It commands **actors** to handle involving events (see figure 2.2). The execution of DE director consists of three iterate phases: **prefire**, **fire** and **postfire**. To begin with, **prefire** checks whether the director is ready to fire. Then, during **fire** it selects actors according to events in the queue, and asks them to work. Each time, the director takes an earliest event in the queue, finds out the destination actor of this event, and requests it to finish its own tasks. **Postfire** in the end verifies if all actors finish their job.

More precisely, the destination actor under the director's request, will work in three sub-phases (**prefire**, **fire**, **postfire**) for data transmission. **Prefire** verifies preconditions before the actor's execution. During the **fire** phase, the actor reads the data in its input port. After a data processing, the data is sent to the output ports producing a new event. This output event is required to be no earlier in time than the input events that were consumed. This new event will be registered and ordered in the queue. Finally, it's **postfire** to see if the execution can process into next iteration.

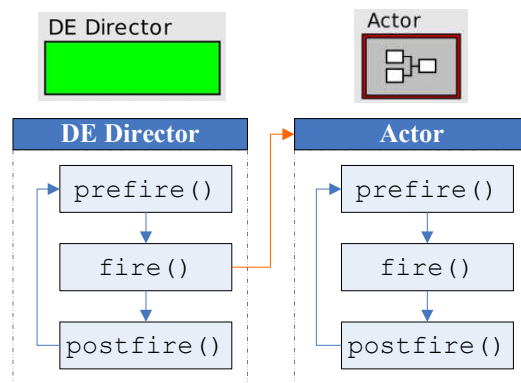


Figure 2.2: Director and actor's work mechanism

2.1.3 Actions for Event Processing

For a better understanding of how to deal with Ptolemy events, we've broken down the event processing into several actions, shown in table 2.1. It's noticed that all actions are taken during **fire** phase.

Table 2.1: Actions for Event Processing

Action	Acted by	Description	Associated Java Method
Get	DE director	Read the earliest event in <code>CalendarQueue</code> .	<code>_eventQueue.get()</code>
Consult	DE director	Propose a valid time to advance to.	<code>_consultTimeRegulators()</code>
Take	DE director	Read the earliest event in <code>CalendarQueue</code> and remove it.	<code>_eventQueue.take()</code>
Set	DE director	Advance the current time setting a new value.	<code>setModelTime()</code>
Put	actor	Generate a new event and put it in the <code>CalendarQueue</code> .	<code>output.sent()</code>

We take an example in figure 2.3 for a further explanation to see how director and actor works with the events.

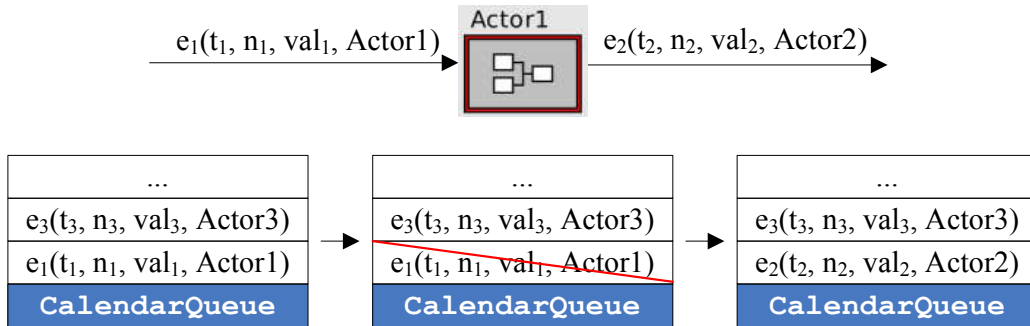


Figure 2.3: Actor's working process in Ptolemy with $t_1 \leq t_2 \leq t_3$

For handling the earliest event e_1 in the queue, the director shall read (**Get**) it at first from the queue to find out what's the destination. Before consuming this event, the director verifies (**Consult**) if it has a right to take action. Once the event is validated, the director processes it immediately (**Take**), advances the current time to t_1 (**Set**). The event then is already removed from the queue and processed by its destination actor *Actor1*. *Actor1* will generate (**Put**) a new event e_2 and stores it in the queue. The whole process is executed in this way:

$$\text{Get } e_1 \rightarrow \text{Consult } t_{e_1} \rightarrow \text{Take } e_1 \rightarrow \text{Set } t_{e_1} \rightarrow \text{Put } e_2$$

2.2 HLA Standard

Aiming at improving the interoperability and reuse of simulators, HLA standard defines an infrastructure to create a global simulation (a **federation**) with a collection of distributed ones (a **federate**) in the following aspects [6]:

- (1) **Rules**: defining the basic duty of global federation and federates.
- (2) **Object model template** (OMT): providing a standard format for information communicated between different federates.
- (3) **Interface specification**: defining a set of services to manage a federation, federates and their interactions.

2.2.1 HLA services

It's the role of the **run-time infrastructure** (RTI) to connect the interaction between federates. It provides a programming library and an application programming interface (API) compliant to HLA specification. In our framework, an open source RTI named CERTI [8] is used, implementing the principle and services of HLA. It's considered as a black box without knowing its detail implementation, but for the useful services.

HLA services are grouped into six management categories to describe a federate's life cycle. For implementing the TAR mechanism, we focus on two of them: **object** management and **time** management. The former is responsible for data exchange, while the latter takes charge of time advance. Their associated services are listed in the following table 2.2. It's noted that the services with * are callbacks from RTI to federates after a `tick()` action, the others are from Federates to RTI. With the help of these services, a federate can deal with messages at proper time.

Table 2.2: HLA services

Category	Services	Description (informal)	Notation
Object	<code>updateAttributeValues()</code>	send and update values	UAV()
	<code>reflectAttributeValues()*</code>	receive updated value	RAV()
Time	<code>timeAdvanceRequest()</code>	ask to advance federate's time	TAR()
	<code>timeAdvanceGrant()*</code>	notify time advance federate's time	TAG()
	<code>nextEventRequest()</code>	ask to advance federate's time	NER()

2.2.2 Time Management

In HLA, incoming messages are categorized as either **receive order** (RO) or **time stamp ordered** (TSO). RO messages conform to the FIFO queuing. TSO messages are assigned a time stamp by federate. The transmission (via **Update Attribute Values**, UAV()) and reception (via **Reflect Attribute Values**, RAV()) of these messages should be ordered by non-decreasing time stamp.

In this way, RTI ensures that no message is delivered “in its past”. We call that a **regulating** federate who generates TSO messages and a **constrained** one who receives TSO messages. Of course, a federate can be both regulating and constrained at a time.

In addition, HLA requires an extra time parameter **lookahead** for regulating federates. It is used to estimate the minimum time stamp of a future event produced by a federate. In other words, there’s a guarantee that no events will be sent if its time stamp is smaller than the current time plus lookahead.

Knowing that in HLA time management, there isn’t any common clock notified by all federates, each federate should manage its local **logical time** and communicate it with the RTI. A request is asked explicitly by a federate to advance to an instant in time, meanwhile its local time is frozen until the authentication of RTI (**Time Advance Granted, TAG()**). By this means, RTI ensures correct coordination of federates.

HLA divides the time management mechanism into two categories:

- **time-stepped**: a federate advances equally in time with a fixed time step. More precisely, the next time value is fixed and the time step can’t change between two calls. **TAR()** is used to make a time advance request.
- **event-driven**: a federate advance to the **time stamp** of the events as they are processed or to a fixed value if there is no event before. **NER()** is applied for a request.

If the lookahead = 0, then the mechanisms are Next Time Request Available (NERA) and Time Advance Request Available (TARA).

2.3 PTII-HLA Co-simulation Framework

The basic idea of how the Ptolemy and HLA-CERTI framework work is: Each Ptolemy model corresponds to a federate in an HLA federation. In other words, a Ptolemy model must deal with distributed events and must advance its logical time according to HLA rules. For the moment, the top level director of a Ptolemy model must be a DE director.

2.3.1 Ptolemy Components

The framework PTII-HLA extends Ptolemy II for distributed simulation using the HLA standard by adding three components: **HlaManager**, **HlaPublisher** and **HlaSubscriber**.

HlaManager is a **decorator** in Ptolemy II, which means it endows elements of the model with parameters. In the case of HLA-CERTI, it *decorates* a time verification process in Ptolemy using HLA time management. Two kind of time requests can be made by each federate : NER() or

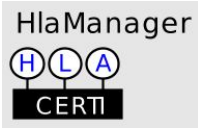


TAR(). Only when the time request is valid through TAG(), the time of model can move forward. The HlaManager interface allows to set up the following time parameters:

- Time Mangement Service: NER() or TAR()
- Time Step in seconds: the equal time increment for a time-stepped federate.
- Lookahead in seconds: the minimum interval in the future that a TSO message should be sent after HLA current time plus Lookahead.
- Hla Time Unit: an extra parameter for the time scaling between Ptolemy model time and HLA logical time, more details in 2.3.2.1 and [2].

HlaPublisher and HlaSubscriber are designed as Ptolemy normal actors, so they still have three sub-phases including prefire, fire and postfire. After prefiring, HlaPublisher will execute a fire action, invoking UAV service of RTI. In this way, it will generate TSO (HLA) messages instead of Ptolemy events comparing with normal actors in Ptolemy. In the side of reception, during director’s firing, all TSO messages shall be received by HlaSubscriber actor. It will take a normal fire action to process the data, generating a new Ptolemy event.

A FOM file is used for the HLA communication, where exchangeable *classes* and their *attributes* should be pointed out in an *object*. HlaPublisher is named as *class.attribute* which indicates one *attribute* in a *class* each time. HlaSubscriber is an actor inside a composite actor named *class*, as the same name in FOM file. The instance of this composite actor is named as *federate.actor* for indicating the transmitting federate and actor. Meanwhile, its outputs are the *attributes* to which the federate subscribes.

Table 2.3: PTII-HLA components

PTII-HLA components	Icon	PTII	HLA
HlaManager		Decorator	Time Advancement: NER(), TAR(), TAG()
HlaPublisher		Actor	Send TSO messages: UAV()
HlaSubscriber		Actor	Receive TSO messages: RAV()

2.3.2 View of Programming

Concerning the implementation of this co-simulation, we need to work with two frameworks in a Java environment:

- Ptolemy II: it acts as the main simulation platform including a set of Java packages. A development version [10] is adopted.
- JCERTI: it is a Java library which is imported in Ptolemy and provides all application programming interfaces (API) for CERTI services. We've installed CERTI [11] version 3.4.3 for use.

In the following sections, we will define precisely the different variables used for time management and event processing.

2.3.2.1 Notion of Time

A simulation of a dynamic system has three notions of time [6]:

- **Physical time:** refers to the time in a physical system which is been modeled by simulation.
- **Wallclock time:** the lapse of time in real world during the simulation execution, which can be measured by a processor clock.
- **Simulation time:** representation of the physical time in simulation. It includes a group of values representing the different instance of time for the modeled system. In a distributed simulation, using HLA standard, there are two types of time:
 - The logical time of a federate, for example Ptolemy model time (section 2.1).
 - The logical time of the federation, for example HLA logical time (section 2.2.2).

In this report, we estimate that a java class or an instance is written in the form of `JavaClass` and a java method is noted like `javaMethod()` or `_javaMethod()`.

a) Time Axis

Ptolemy and HLA-CERTI, as two independent frameworks, they have their own execution rhythm and local time. HLA-CERTI, as a master, governs the advance of local logical time `CertiLogicalTime` for federates in the same federation, but each federate (Ptolemy model) governs its time for `Time` class. So, it's necessary to distinguish the notation of time in Ptolemy and in HLA-CERTI and build their relationships.

In a simulation time axis, there are two scales for each federate: model (or logical) time in Ptolemy and logical time in HLA-CERTI.

- *t*: time scale for Ptolemy `Time`, where `Time` is the `ptolemy.actor.util.Time` Java class in Ptolemy, representing the model time.
- *Lt*: time scale for HLA-CERTI `CertiLogicalTime`, where `CertiLogicalTime` is the `certi.federate.CertiLogicalTime` Java class in HLA-CERTI, representing the logical time.

The federates can have different “time units”. Each federate has a parameter:

- `_hlaTimeUnitValue`: a linear mapping scale between t and Lt . The mapping between the two scales of time is: $Lt = t \times _hlaTimeUnitValue$.

The HLA interface is responsible for the conversion between Ptolemy time and HLA-CERTI logical time by using two methods:

- `_convertToCertiLogicalTime()` : method to convert Ptolemy time to certi logical time.
- `_convertToPtolemyTime()` : method to convert certi logical time to Ptolemy time.

After an initial point synchronization, the initial time for all federates created in HLA-CERTI is set to zero.

An example is indicated in figure 2.4 where `_hlaTimeUnitValue` is equal to 2. It means Lt advances two times more rapidly than t .

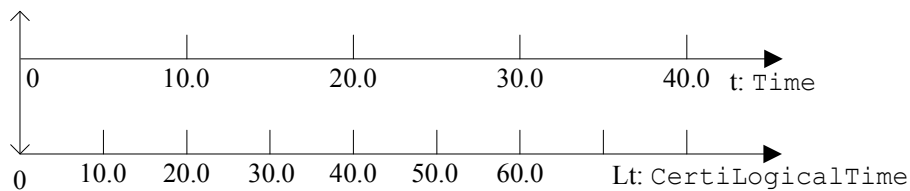


Figure 2.4: Different time scale with `_hlaTimeUnitValue=2`

b) Time Variables

We define the HLA constants:

- *lah* named `_hlaLookahead` which represents the minimum interval in the future that a TSO message will be scheduled.
- T_{s_hla} : hla time step named `_hlaTimeStep` which is fixed at the beginning of the simulation and makes an equal time increment in HLA-CERTI. With this constant parameter, we intend to make a linkage between PTII and HLA in time-stepped mode.

For simplification of nomenclature, we admit that all notations of time we use is under Ptolemy Time scale but with two parallel time axis *ptII* and *hla*.

- t : time scale for Ptolemy Time. It includes two time axis progressing in Ptolemy named t_{ptII} , and in HLA-CERTI named t_{hla} .
- $t_{current}$: point in time axis where the model is right now located. It includes $t_{current_ptII}$ named `currentTime` and $t_{current_hla}$ named `hlaCurrentTime`. The start time in Ptolemy is noted as t_{0_ptII} .

- $t_{nextPointInTime_hla}$: next point in time named `_hlaNextPointInTime` to which time-stepped federates process all events occur up [3]. It's estimated that $t_{nextPointInTime_hla}$ is only known in HLA-CERTI and Ptolemy is not able to retrieve it directly.

$$t_{nextPointInTime_hla} = t_{current_hla} + T_{s_hla}$$

We visualize the associated notations in figure 2.5 where Ptolemy and HLA-CERTI have the same time scale with `_hlaTimeUnitValue = 1`.

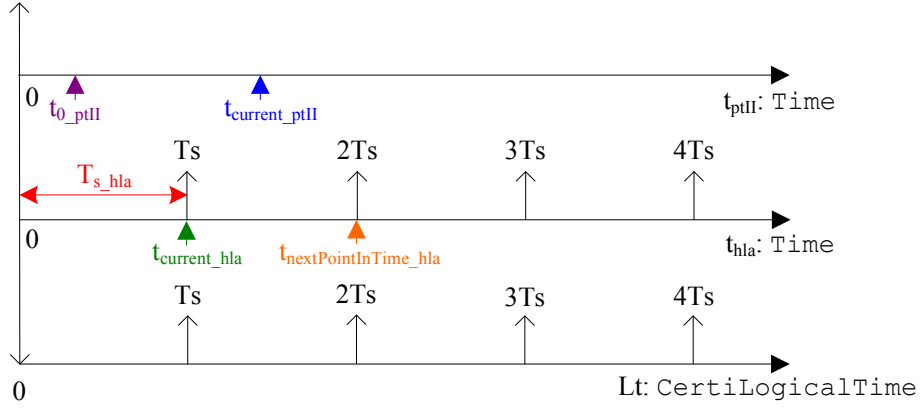


Figure 2.5: Definition of time axis and related points

2.3.2.2 Discrete Event

The time advances in Ptolemy through event processing. In the PTII-HLA framework, the events are generated and consumed by normal actors, besides there are also messages received (RAV)/sent (UAV) from/to the RTI by the new actors HlaSubscriber and HlaPublisher. Let us clarify the notion for different events in the sequel.

- *ptolemy-event* (shown in figure 2.6): a normal **DEEvent** noted as $e(t, n, value, actor)$, which is the registered in `CalendarQueue`, where t is the *model time*, n it the *microstep*, $value$ is the carried value and $actor$ is the destination actor. The pair (t, n) is called *timestamp*. A consummation of an event might generate a following one at present or in the future after `fire()` action.

From now on we call t_e the model time of the *ptolemy-event* e .

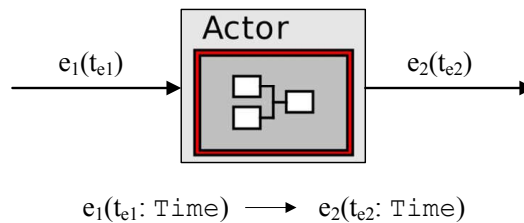


Figure 2.6: Actor only with time parameter

- *rav-event* (shown in figure 2.7): an auxiliary “event” in Ptolemy denoted as $\text{pRAV}(t_{rav}, \text{value})$, which is generated by HLA callback $\text{RAV}(Lt_{rav}, \text{value})$, received by the `HlaSubscriber` actor without registration in the queue. After the call of `_putReflectedAttributesOnSubscribers()` method, this *rav-event* will be put in the queue becoming a *ptolemy-event* named $\text{folRAV}(t_{folRAV}, n, \text{value}, \text{actor})$. After firing of `HlaSubscriber`, this event will be an output event.

So there are three time stamp related to a `HlaSubscriber`:

- Lt_{rav} : time stamp of a TSO message belonging to `CertiLogicalTime` in HLA-CERTI.
- t_{rav} : (auxiliary) time stamp of a *rav-event* in type of `Time` representing for Lt_{rav} .
- t_{folRAV} : time stamp of a *ptolemy-event* belonging to `Time` in `CalendarQueue`.

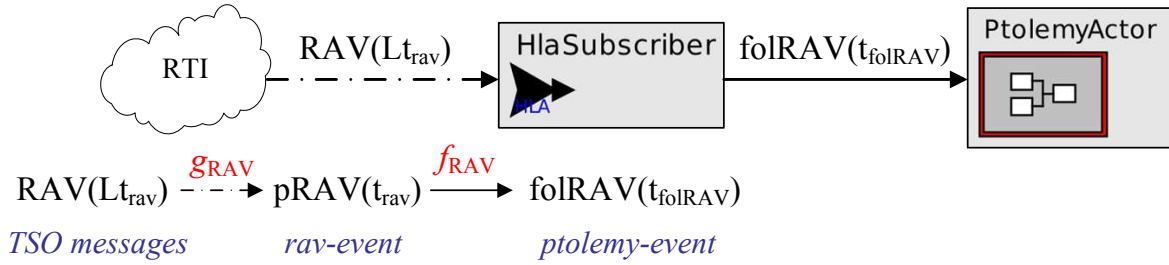


Figure 2.7: `HlaSubscriber` with only time parameter

Besides we define two functions serving to establish the links between these time stamps.

$$g_{RAV} : Lt_{rav} \mapsto t_{rav} = Lt_{rav} / \text{hlaTimeUnitValue} \quad (2.1)$$

$$f_{RAV} : t_{rav} \mapsto t_{folRAV} \quad (2.2)$$

It’s noticed that g_{RAV} is the function `_convertToPtolemyTime()` and f_{RAV} depends on the time management used by the federate (NER or TAR) which will be presented in sections 3.1.1 and 4.2.1 respectively.

- *uav-event* (shown in figure 2.8): an auxiliary “event” in Ptolemy denoted as $\text{pUAV}(t_{uav}, \text{value})$ which is generated by a *ptolemy-event* called $\text{preUAV}(t_{uav}, n, \text{value}, \text{HlaPublisher})$ and received by `HlaPublisher`. During `fire()` phase, `HlaPublisher` will translate it to a TSO message $\text{UAV}(Lt_{uav}, \text{value})$ where Lt_{uav} represents its time stamp belonging to `CertiLogicalTime` and sent it to RTI using `UAV()` service.

There are also three types of time stamps:

- t_{preUAV} : time stamp of a *ptolemy-event* belonging to `Time` in `CalendarQueue`.
- t_{uav} : (auxiliary) time of *uav-event* in type of `Time`, with $t_{uav} = t_{preUAV}$. t_{uav} can be modified explicitly.
- Lt_{uav} : time stamp of a sent TSO message belonging to `CertiLogicalTime` in HLA-CERTI, reminding that $Lt_{uav} = t_{uav} \times \text{hlaTimeUnitValue}$.

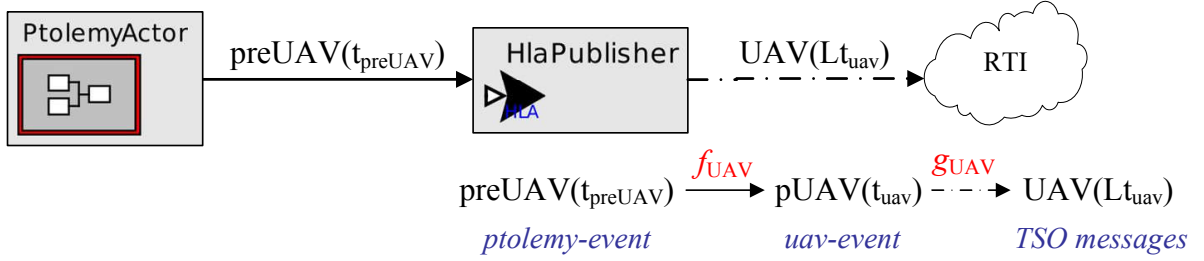


Figure 2.8: HlaPublisher with only time parameter

It's still defined two functions for time management:

$$f_{UAV} : t_{preUAV} \mapsto t_{uav} \quad (2.3)$$

$$g_{UAV} : t_{uav} \mapsto Lt_{uav} = t_{uav} \times _hlaTimeUnitValue \quad (2.4)$$

It's seen that g_{UAV} corresponds the function `_convertToPtolemyTime()` and $g_{UAV} = g_{RAV}^{-1}$. Moreover, f_{UAV} will be detailed in section 3.1.1 and 4.2.1 which depends on the choice of NER or TAR.

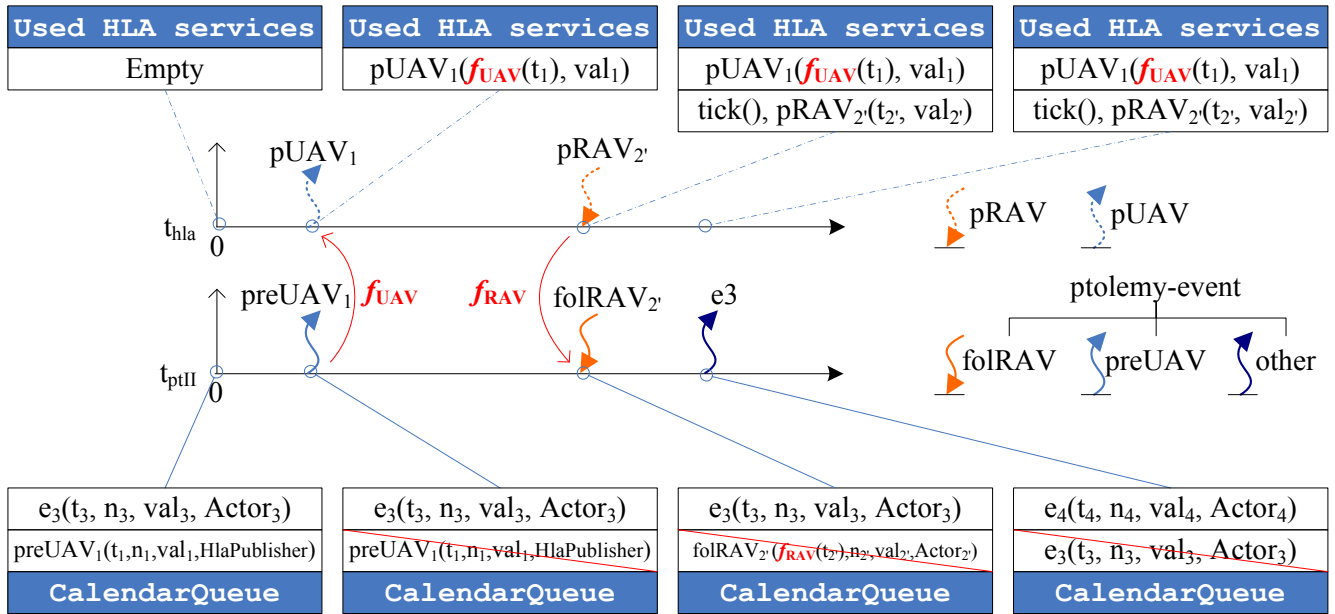
2.3.3 Representation of Event Processing

According to previous notations, we can distinguish three types of *ptolemy-events*: folRAV (output of HlaSubscriber), preUAV (input of HlaPublisher) and others. They can be represented in Ptolemy time axis t_{ptII} . folRAV and preUAV will be *translated* to, respectively, *rav-event* and *uav-event* which are located in HLA time axis t_{hla} . Figure 2.9a depicts the following scenario occurring in 2.9b.

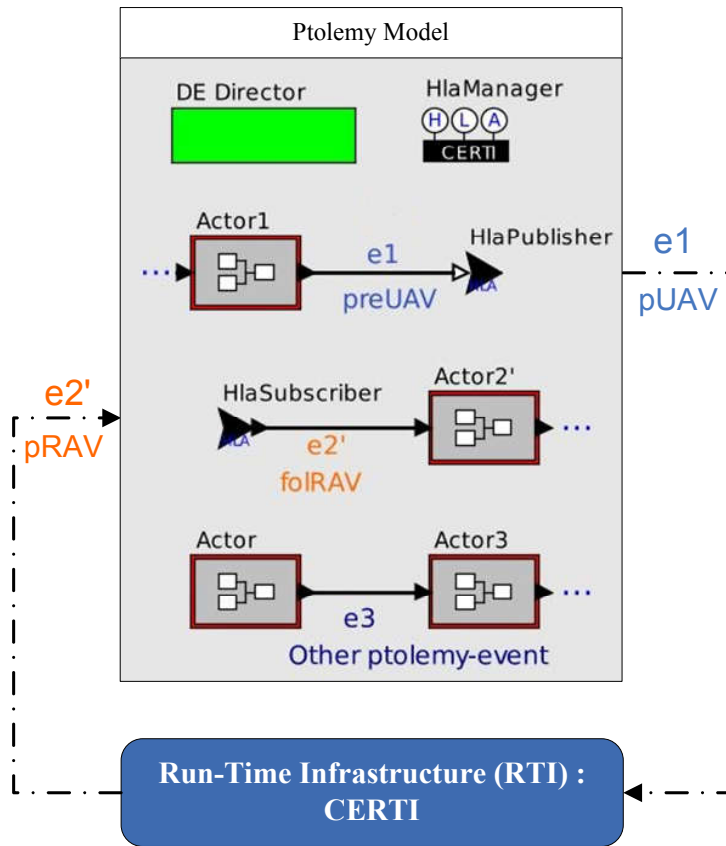
We will present how these events are related in the sequel. It must be pointed out that we will keep the functions f_{UAV} and f_{RAV} without make any choice for TAR or NER time management. The reader must keep in the mind that in figure 2.9a the time stamps in time axis t_{hla} and t_{ptII} are related by these functions, and the time stamps can be different as it will be presented in chapters 3 and 4.

Let us consider that there are following events in the queue at the current time $t_{0,ptII} = 0$: preUAV events $\{preUAV_1\}$ and an other *ptolemy-event* $\{e3\}$. The preUAV will call UAV() service during the `fire()` phase. The *ptolemy-event* $e3$ is a normal event which might generate another *ptolemy-event*.

The event $preUAV_1(t_1, n_1, val_1, HlaPublisher)$ is the first one to be processed, *HlaPublisher* takes `fire()` action, it will generate the event $pUAV_1(f_{UAV}(t_1), val_1)$. The processing of $e3(t_3, n_3, val_3, actor_3)$ by $actor_3$ can generate an other *ptolemy-event* e_4 (after firing) with time stamp $t_4 = t_3$ or $t_4 > t_3$ if $actor_3$ is a TimeDelay actor. For this reason, new generated event e_4 is not presented in the figure. Let us consider that the RTI delivers an *rav-event* $pRAV_{2'}(t_{2'}, val_{2'})$ to the federate, so this event is translated to *ptolemy-event* $folRAV_{2'}(f_{RAV}(t_{2'}), n_{2'}, val_{2'}, Actor_{2'})$. In this way, we can represent dynamically the processing of `CalendarQueue` and make clear the



(a) An example of events' execution in time axis



(b) Ptolemy model

Figure 2.9: A Ptolemy model and its execution

commutation mechanism between Ptolemy and HLA-CERTI.

Let us remind that e , folRAV, preUAV take place only in t_{ptII} and pRAV, pUAV occur only in t_{hla} . In the sequel, subscripts $_{ptII}$ and $_{hla}$ are not indicated anymore except when it is necessary. Even if a *ptolemy-event*, a *rav-event* and an *uav-event* includes more than one parameters, for the sake of simplicity they will be represented by $e(t)$, pRAV(t), folRAV(t), preUAV(t) and pUAV(t), with a unique parameter t in **Time**. If necessary, the complete formula is written to avoid certain ambiguity. With the help of g_{UAV} and g_{RAV} , we can represent HLA services as TAR($g_{UAV}(t)$), NER($g_{UAV}(t)$), TAG($g_{RAV}(Lt)$), UAV($g_{UAV}(t)$), RAV($g_{RAV}(Lt)$). For a simplification, they are respectively noted as pTAR(t), pNER(t), pTAG(t), pUAV(t), pRAV(t) which indicate the HLA services in Ptolemy.

In addition, there is no difference between *rav-event* (pUAV) and pUAV() service because the call of this service generates this so-called pUAV event. It's the same case for pRAV.

Table 2.4: Time represent action for HLA and Ptolemy different time axis

Time axis	Events or services
t in t_{ptII}	$e(t)$, folRAV(t), preUAV(t)
t in t_{hla}	pRAV(t), pUAV(t), pNER(t), pTAR(t), pTAG(t)
Lt	RAV(Lt), UAV(Lt), NER(Lt), TAR(Lt), TAG(Lt)

Chapter 3

Basic Analysis and Improvements for Next Event Request

This chapter presents the existing implementation of NER service as presented in [8] for a good understanding of PTII, HLA frameworks and their cooperation mechanism.

3.1 Analysis of NER

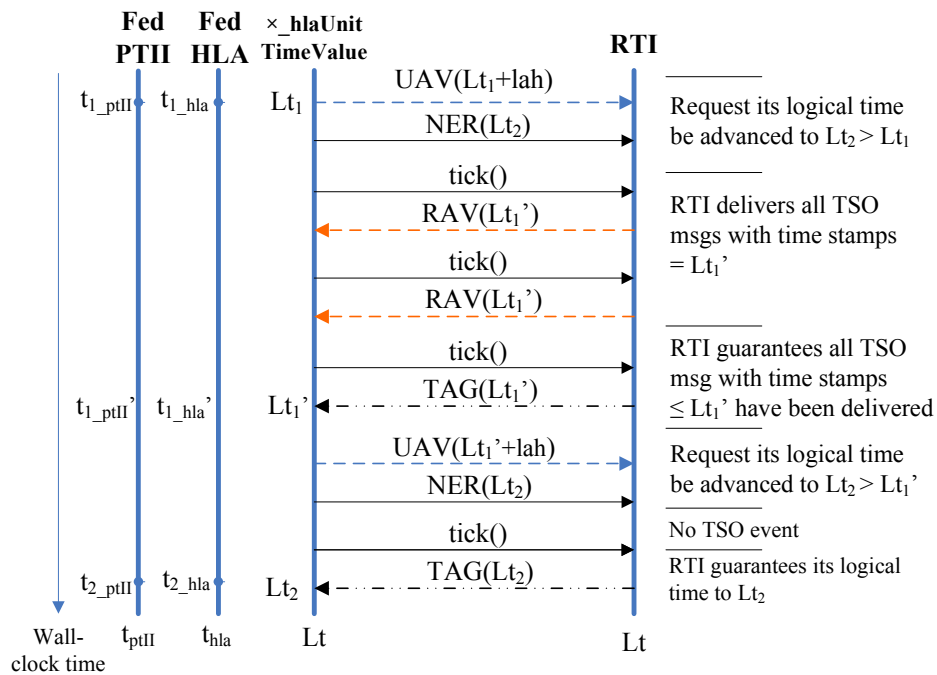


Figure 3.1: principal concept of NER

The main idea of NER mechanism in the HLA-CERTI framework is the following. An event-based federate requests an advance of logical time through NextEventRequest service with time parameter Lt . A typical NER processes as follows (see figure 3.1). At its current logical time Lt_1 , it wants to advance to logical time Lt_2 which is the time stamp of the next TSO message. After a call of $NER(Lt_2)$, the RTI can send (if any) all TSO messages with time stamp Lt_1' (with $Lt_1 < Lt_1' \leq Lt_2$). The reception of those messages is done by a callback $RAV(Lt_1')$. Once the delivery is finished, a $TAG(Lt_1')$ is issued indicating that time request was granted. The logical

time of a federate is advanced to Lt'_1 . Or if there is no TSO messages to be delivered, a TAG(Lt_2) is directly received by the federate that advances its time to Lt_2 .

3.1.1 Mechanism of NER

The analysis of the work was done from [8] and by a step-by-step debug. Figure 3.2 shows a cooperation of PTII and HLA, and figure 3.3 shows the details of their communications.

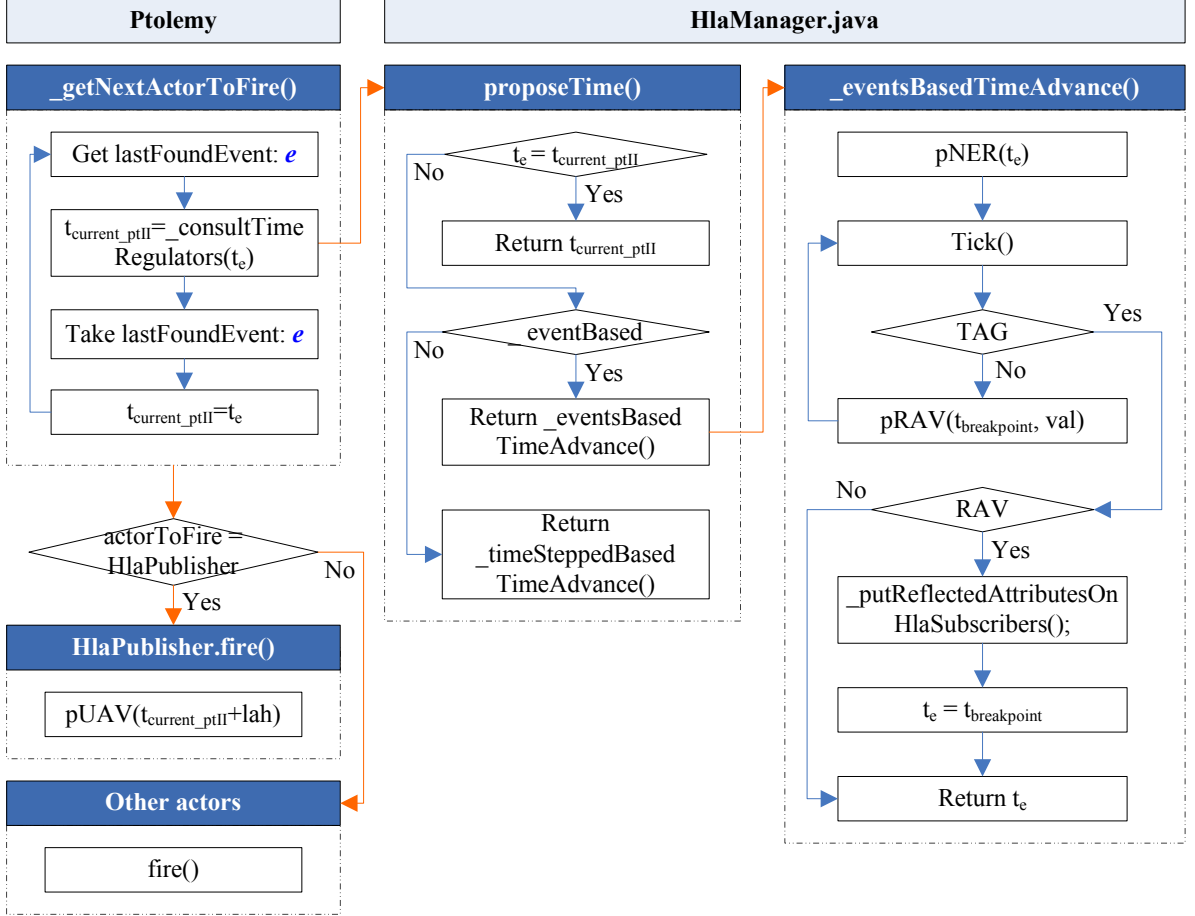


Figure 3.2: NER Time Advancement Diagram, where $f_{RAV}(t) = t$ and $f_{UAV}(t) = t + lah$

On the side of DE director, it gets a next event (i.e. *lastFoundEvent*) which is the most recent *ptolemy-event* in *CalendarQueue* (i.e. the first one sorted by a time stamp order). A time advancement request is made through *_consultTimeRegulators()* method. It commands all *TimeRegulator* classes to find out a valid time to advance to which is the smallest value among all proposed times. It is the role of *proposeTime()* method to check out if this requested time $t_{lastFoundEvent}$ is valid or not.

On the side of the *HlaManager*, as an intermediate layer, it links the communication between Ptolemy and HLA-CERTI implementing *TimeRegulator*. Let us detail the way: time management which is done through the method *proposeTime()* as depicted in figure 3.2. For the NER, a federate requests directly to advance to $t_{lastFoundEvent}$. If it receives any *rav-event* at $t_{breakpoint}$

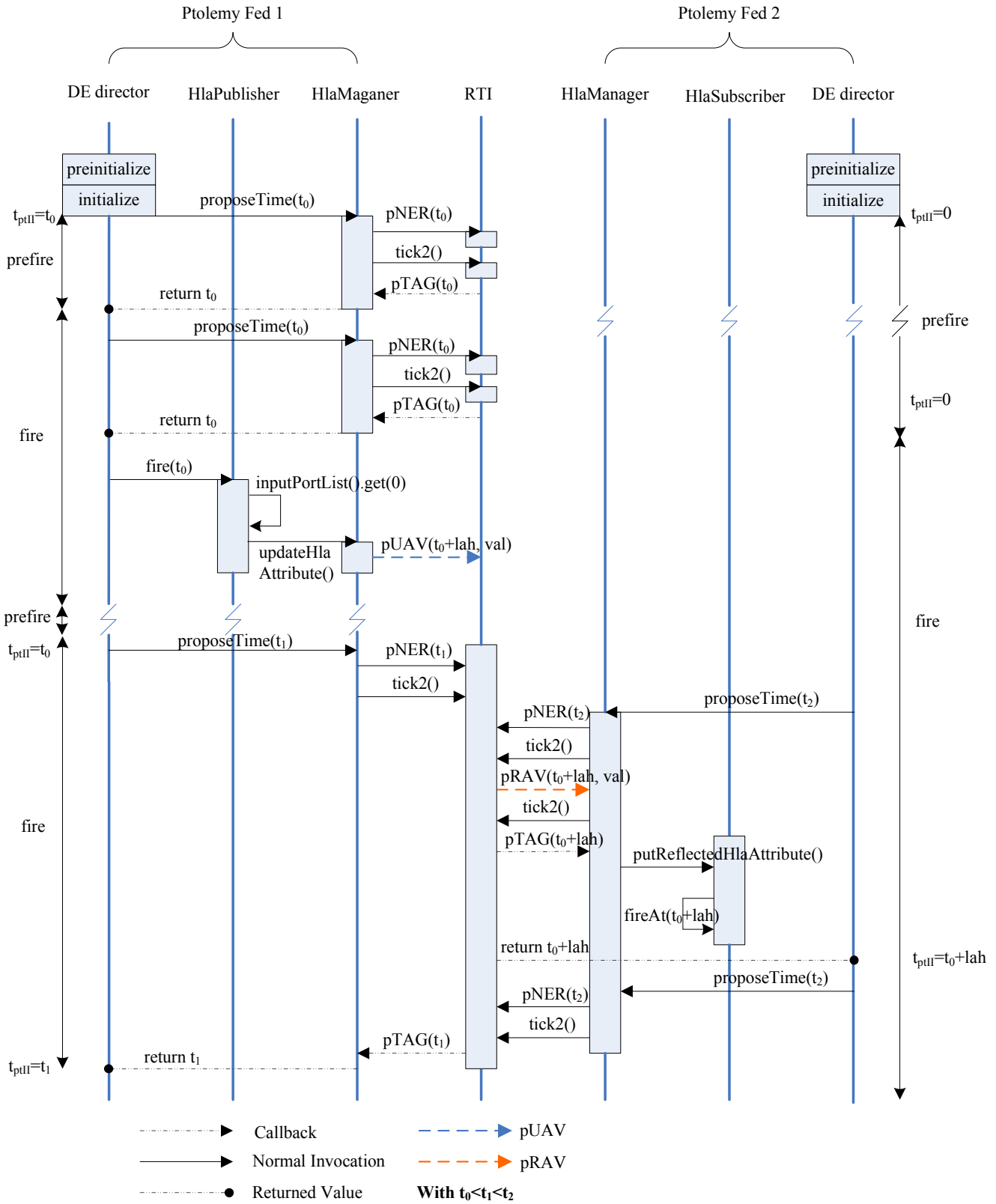


Figure 3.3: NER execution detail

($t_{current_ptII} < t_{breakpoint} \leq t_{lastFoundEvent}$) with a following $pTAG(t_{breakpoint})$, the `CalendarQueue` is refreshed by putting all $pRAV(t_{breakpoint})$ inside. It is the returned time who represents a valid time to advance to. So we have got the right to consume the next event in the queue. This next event is either a new $folRAV(t_{breakpoint})$ or the original event $lastFoundEvent$. PTII finally advances its current time and takes a `fire()` action.

From the event execution, we see that each progression of Ptolemy time corresponds a progression of HLA time. For a `HlaPublisher`, the `fire()` action should invoke `UAV()` service to send TSO messages. Meanwhile, every time we get *rav-event* or not, `HlaSubscriber` will be waken up after the reception of TAG.

Reminding that the previous definition in section 2.3.2.2 for $f_{UAV} : t_{preUAV} \mapsto t_{uav}$ and $f_{RAV} : t_{rav} \mapsto t_{folRAV}$. For NER mechanism we have

$$\forall t \in \mathbb{R}^+, f_{RAV}(t) = t \quad (3.1a)$$

$$\forall t \in \mathbb{R}^+, t_{folRAV} = f_{UAV}(t) = t + lah \quad (3.1b)$$

3.1.2 Example of NER

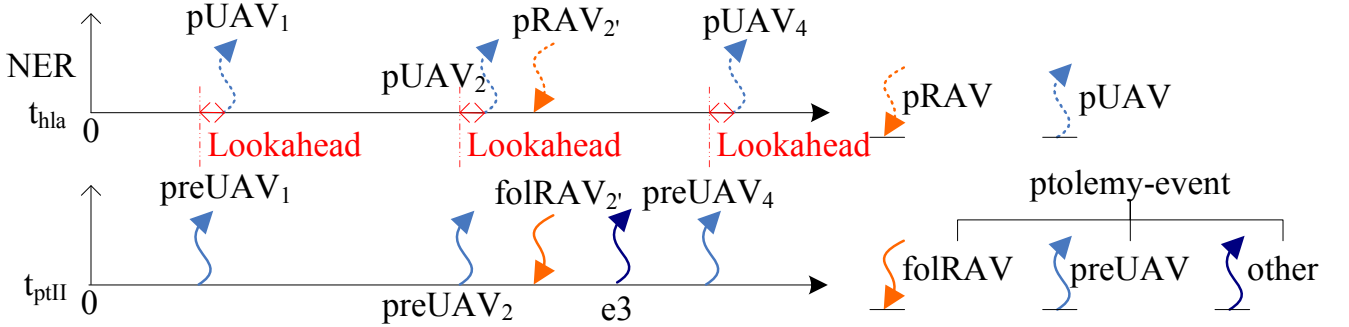


Figure 3.4: NER Example

We'll take an example for better comprehension of NER mechanism (see figure 3.4).

Execution process Time starts from $t_{current_ptII} = 0$, $t_{currentT_hla} = 0$.

- Get $preUAV_1(t_1) \rightarrow pNER(t_1) \rightarrow \{\text{tick}(), pTAG(t_1)\}$, $t_{current_hla} = t_1 \rightarrow$ no RAV \rightarrow Take $preUAV_1 \rightarrow$ Set $t_{current_ptII} = t_1 \rightarrow pUAV_1(t_1 + lah)$
- Get $preUAV_2(t_2) \rightarrow pNER(t_2) \rightarrow \{\text{tick}(), pTAG(t_2)\}$, $t_{current_hla} = t_2 \rightarrow$ no RAV \rightarrow Take $preUAV_2 \rightarrow$ Set $t_{current_hla} = t_2 \rightarrow pUAV(t_2 + lah)$
- Get $e3(t_3) \rightarrow pNER(t_3) \rightarrow \{\text{tick}(), pRAV(t_{2'})\} \rightarrow \{\text{tick}(), pTAG(t_{2'})\}$, $t_{current_hla} = t_{2'} \rightarrow$ Put $pRAV_{2'}(t_{2'}) \rightarrow$ Take $folRAV_{2'}(t_{2'}) \rightarrow$ Set $t_{current_hla} = t_{2'} \rightarrow \text{fire}(t_{2'})$
- Get $e3(t_3) \rightarrow pNER(t_3) \rightarrow \{\text{tick}(), TAG(t_3)\}$, $t_{current_hla} = t_3 \rightarrow$ no RAV \rightarrow Take $e3 \rightarrow$ Set $t_{current_ptII} = t_3 \rightarrow \text{fire}(t_3)$

- Get $\text{preUAV}_4(t_4) \rightarrow \text{pNER}(t_4) \rightarrow \{\text{tick}(), \text{pTAG}(t_4)\}, t_{\text{current_hla}} = t_4 \rightarrow \text{no RAV} \rightarrow \text{Take preUAV}_4 \rightarrow \text{Set } t_{\text{current_ptII}} = t_4 \rightarrow \text{pUAV}(t_4 + \text{lah})$

3.2 General Improvements

3.2.1 Asynchronism problem of PTII start Time

Federates in the previous version of HLA-CERTI framework could have the same stop time. But the execution of the federation went wrong if they had different start time.

Let us consider the following case in figure 3.5 which is a simple producer/consumer model. We've got two federates named **A** and **B**. **A** is the producer within an HlaPublisher. **B** is the consumer within an HlaSubscriber. After initialization, the current time in PTII and HLA will be set. So we have:

- For **A**, $t_{\text{current_ptII}} = t_{0_ptII} = 0, t_{\text{current_hla}} = 0$;
- For **B**, $t_{\text{current_ptII}} = t_{0_ptII} = 5, t_{\text{current_hla}} = 0$.

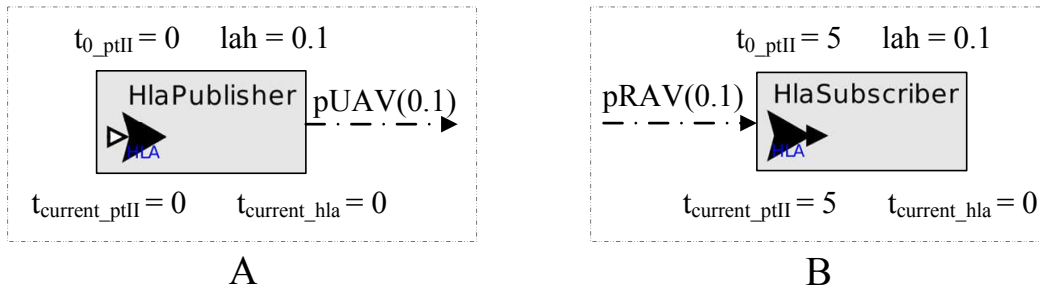


Figure 3.5: Two federates with different Ptolemy start time

Then, **A** has the right to send an *uav-event* denoted $\text{pUAV}(0.1)$. In the reception, **B** gets the associated *rav-event* as $\text{pRAV}(0.1)$, so $\text{folRAV}(0.1)$ is put in the queue. But for **B**, $t_{\text{current_ptII}} = 5$, the simulation displays an error because the director is unable to fire the actor at the requested time in its past. Hence, for avoidance of such a case, an arbitrary choice is made without loss of information: all *rav-event* with time stamp $t_{\text{rav}} < t_{0_ptII}$ are registered at t_{0_ptII} .

$$\forall t \in \mathbb{R}^+, f_{\text{RAV}}(t) = \begin{cases} t_{0_ptII} & \text{If } t < t_{0_ptII} \\ t, & \text{Otherwise} \end{cases} \quad (3.2)$$

The current version allows the use of different start time and the results are coherent with our expectation.

3.2.2 Reduction of HLA service calls

After a step-by-step debug for NER, it is found that a Ptolemy federate calls several times $\text{pNER}()$ at its current time. Obviously, it is unnecessary to make an request to advance to its own current time which should be validated without hesitation. That's why an additional condition is put into use in $\text{proposeTime}()$ to reduce the HLA services as indicated in algorithm 1.

Algorithm 1 Propose a valid time for Ptolemy’s time advancement

Require: an initial *proposedTime* gets the time stamp of *lastFoundEvent*

Ensure: a valid *proposedTime* to advance to

```
1: procedure PROPOSETIME(proposedTime)
2:   if proposedTime =  $t_{current\_ptII}$  then
3:     return proposedTime;
4:   else call HLA time advancement service;
5:   end if
6: end procedure
```

3.2.3 Rearrangement of HlaManager graphical user interface

The parameters’ value are set in HlaManager graphical user interface. Figure 3.6a depicts the interface of the frame work at the beginning of this work. The modifications made in the code of the framework lead to the following changes in the interface:

- Pull-down menu for NER/TAR.

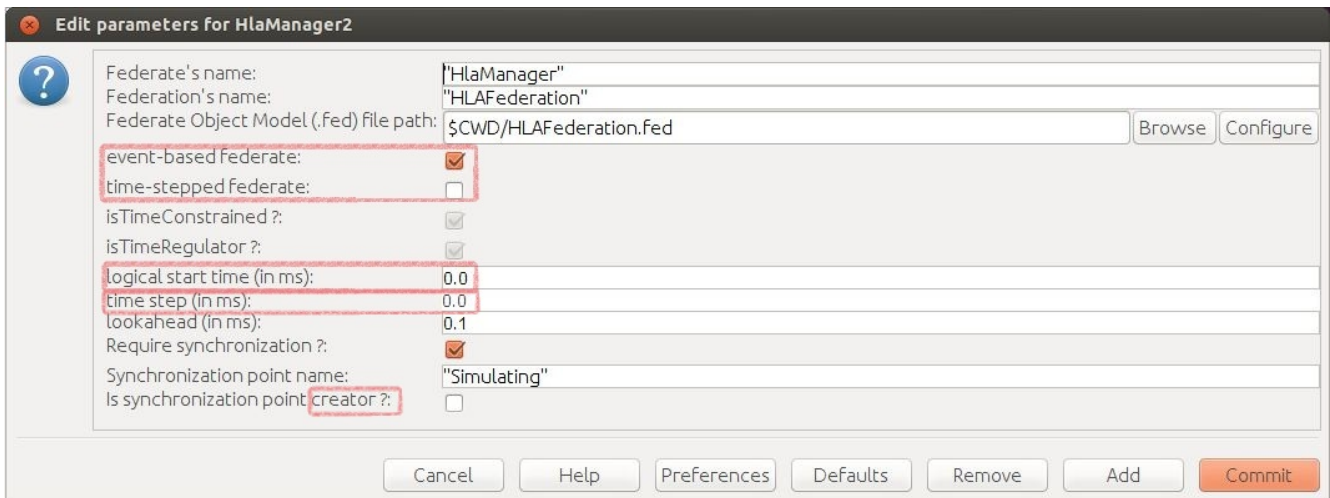
In the previous version, the choice for choosing NER or TAR was implemented by a `Parameter` class. Moreover, the user could choose both mechanism. But in current version, a federate uses either NER or (-exclusive) TAR. The choice has been replaced by a pull-down menu, implemented by a `ChoiceParameter`.

- Remove of logical start time.

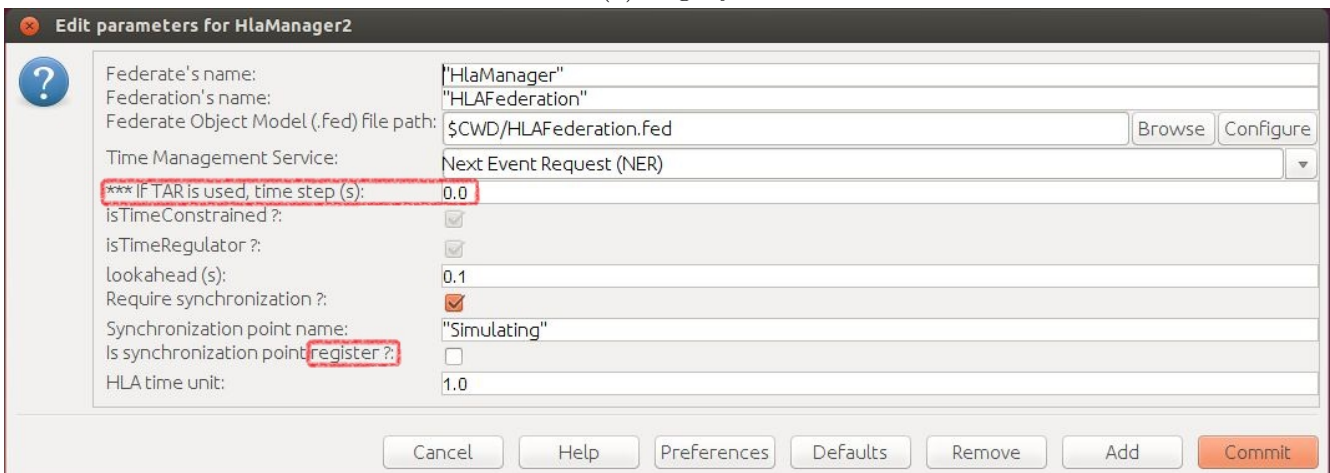
The goal of HLA-CERTI framework is to model and simulate Cyber-Physical systems. For such a simulation, a synchronization point is necessary. After the launch of RTI, a federate waits for jointing of all federates, and keeps stepping until the synchronization point is received. When all federates are ready, the $t_{current_hla}$ is set to zero, and the Ptolemy federates begin to process. That’s why the parameter ”logical star time” was removed.

- Other modifications:

- the term “Synchronization Point *Creator*” was replaced by the correct term “Synchronization Point *Registor*”
- a message recalling that time step is used only if the TAR mechanism is used.
- HLA Time Unit was introduced by [2].



(a) Legacy



(b) New

Figure 3.6: HlaManager

Chapter 4

Implementing Time Advance Request in PTII-HLA

In this part, we'll make clear TAR working mechanism in HLA and try to find out a solution for its implementation under Ptolemy environment.

4.1 Overview of TAR

4.1.1 Interests of TAR

NER mechanism allows to react to each occurrence of an event e , advancing a federate's time to the time stamp of this event. If the federate is at $t_{current_hla}$, it can advance to the time stamp t_e of the event e . TAR mechanism is different: if the federate has a time step T_{s_hla} and is at $t_{current_hla}$, it can only advance to the next point in time ($t_{current_hla} + T_{s_hla}$) if it can receive and handle *raw-events* in the meantime. To an extent, NER is more consistent with the spirit of Ptolemy, whereas it sacrifices the efficiency in some cases. As a result, we are interested in TAR which can be more efficient to process events, if we make a good choice of time step T_{s_hla} and lookahead lah . In this way, RTI spends fewer resource to manage federates.

In reality, each CPS model consists of one or more continuous system. When we build their simulation models, as one of the most important components, the control part is often implemented as a **time-driven** (or sampled) system which acquires the inputs at regular intervals of time and computes its reaction for each sample of its inputs [5]. We often use "synchronous" approach (Lustre, etc) to make its design and analyze the performance. If we want to establish two distributed system for physical system and controller, which means the two are simulated as two federates. Then their communication shall naturally be TAR, the time step may be the sampling period for example.

4.1.2 Principal Mechanism of TAR

With the acquirement in NER analysis, we have known more about PTII and HLA. The key of our work right now is to understand the principle of TAR in HLA, then implement it in Ptolemy

environment.

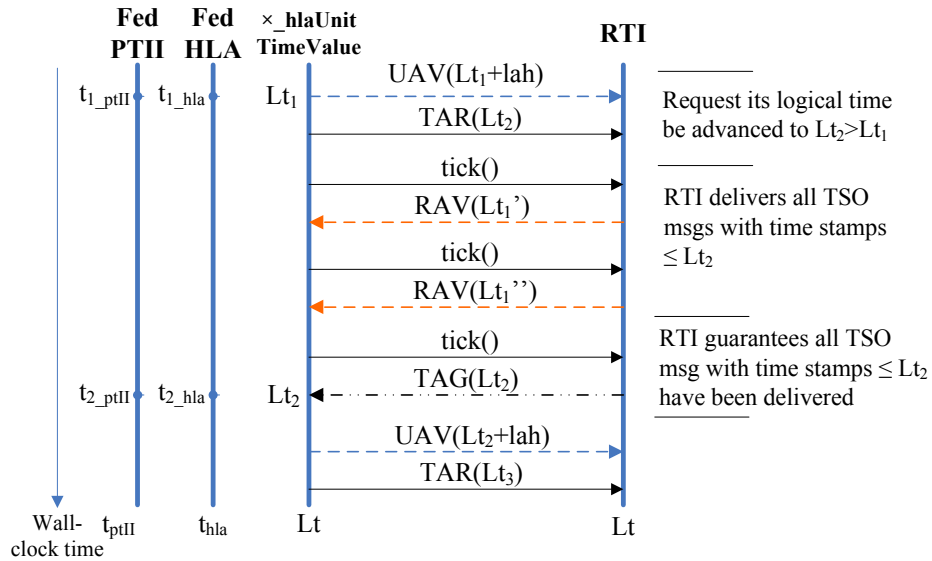


Figure 4.1: principal concept of TAR

In HLA, the time advance primitives provide a protocol for the federate and RTI to jointly control the advancement of logical time [1, 7]. The principal mechanism of TAR is indicated in figure 4.1. At a logical time Lt_1 , a federate requests a time advance to next point in time $TAR(Lt_2)$. Only when the RTI guarantees that all TSO messages with time stamp smaller than or equal to Lt_2 have been successfully delivered. Thus the callback $RAV(Lt_{rav})$ is used for the reception of TSO messages with time stamp Lt_{rav} ($Lt_1 < Lt_{rav} \leq Lt_2$). RTI then grant the logical time advance of federate to Lt_2 through $TAG(Lt_2)$. At the same time, the federate must delay processing any local event until logical time has advanced to the time of that event, otherwise other federates may receive a TSO message in its past. In this way, the synchronization of the execution and the reception of a request are accomplished in the same time.

4.2 Assessing TAR

According to the principle of time advancement request (TAR), a time-stepped federate can set up its logical time scale $hlaTimeUnitValue$ and the time step $T_{s.hla}$.

There are some constraints in the implementation.

- HLA requires:

(R1) For a **HlaPublisher**, an *uav-event* should be sent in the future after $t_{current.hla} + lah$.

$$t_{uav} \geq t_{current.hla} + lah \quad (4.1)$$

- (R2) Or if the federate is not in a time advising phase which means after $\text{pTAR}(t'_{hla})$, we want to call pUAV service without the tick, an *uav-event* should be sent after $t'_{hla} + lah$.

$$t_{uav} \geq t'_{hla} + lah \quad (4.2)$$

It should be watchful because JCERTI accepts this situation without throwing any exceptions. This requirement should be respected explicitly. As a result, we propose that during the advising phase, no actors will be stimulated, no *uav-events* will be generated.

- (R3) For a **HlaSubscriber**, a *rav-event* should be received between $t_{current_hla}$ and t'_{hla} if we make a time advance request at time $\text{pTAR}(t'_{hla})$.

$$t_{rav} \in]t_{current_hla}, t'_{hla}] \quad (4.3)$$

- (R4) With a time advance request at time t'_{hla} , if a time advance grant $\text{pTAG}(t_{hla})$ is received, we must have $t_{hla} = t'_{hla}$, and $t_{current_hla} = t'_{hla}$

- Ptolemy requires:

- (R5) A *ptolemy-event* can only be handled if its time stamp is bigger or equal than its current time which means for folRAV events, we have:

$$t_{folRAV} \geq t_{current_ptII} \quad (4.4)$$

- Federate pattern:

- (R6) A time advance request with time t'_{hla} is made by $\text{pTAR}(t'_{hla})$ if and only if

$$t'_{hla} = k \times T_{s_hla}, \text{ with } k \in \mathbb{N} \quad (4.5)$$

Let us remind that the parameter `hlaNextPointInTime` is defined for the description of its next point in HLA to advance to and its math expression is:

$$t_{nextPointInTime_hla} = t_{current_hla} + T_{s_hla} \quad (4.6)$$

4.2.1 Possible Solutions for TAR Implementation

For the satisfaction of the previous requirements, it comes out two main ideas to implement the TAR algorithm. An example will be taken for further explanation.

At the beginning, we suppose:

$$t_{current_ptII} = t_{current_hla} = 0$$

It's recalled that the previous definitions for $f_{UAV} : t_{preUAV} \mapsto t_{uav}$ and $f_{RAV} : t_{rav} \mapsto t_{folRAV}$.

4.2.1.1 Option 1: with Enforced UAV Delay

Before the execution of a first event in a cycle, a federate does a $\text{pTAR}(t_{\text{nextPointInTime_hla}})$ request at $t_{\text{current_ptII}}$. *Rav-events* can be received. But when a $\text{pTAG}(t_{\text{nextPointInTime_hla}})$ is received, we are sure that no more *rav-events* will arrive in this cycle. At this moment, $t_{\text{current_hla}}$ is set to $t_{\text{nextPointInTime_hla}}$. These *rav-events* should be transferred to *ptolemy-events*. And we are able to execute the original or transferred *ptolemy-events* one by one in this cycle regardless of other *rav-events*. On basis of (R3), (R5) is also valid, so we have the equation 4.7b.

As for the *uav-events*, they should be sent after $t_{\text{current_hla}} + \text{lah}$ to satisfy (R1) shown in equation 4.7a.

$$\forall t \in \mathbb{R}_+, f_{UAV}(t) = \begin{cases} t_{\text{nextPointInTime_hla}} + \text{lah}, & \text{if } t \in]t_{\text{current_hla}}, t_{\text{nextPointInTime_hla}}] \\ t, & \text{if } t > t_{\text{nextPointInTime_hla}} \end{cases} \quad (4.7a)$$

$$\forall t \in \mathbb{R}_+, f_{RAV}(t) = t \quad (4.7b)$$

Example 1 Let us consider the scenario described in figure 4.2, $t_{\text{nextPointInTime_hla}} = T_{s_hla}$. Each step bellow indicates the events in the CalendarQueue and HLA services (if any):

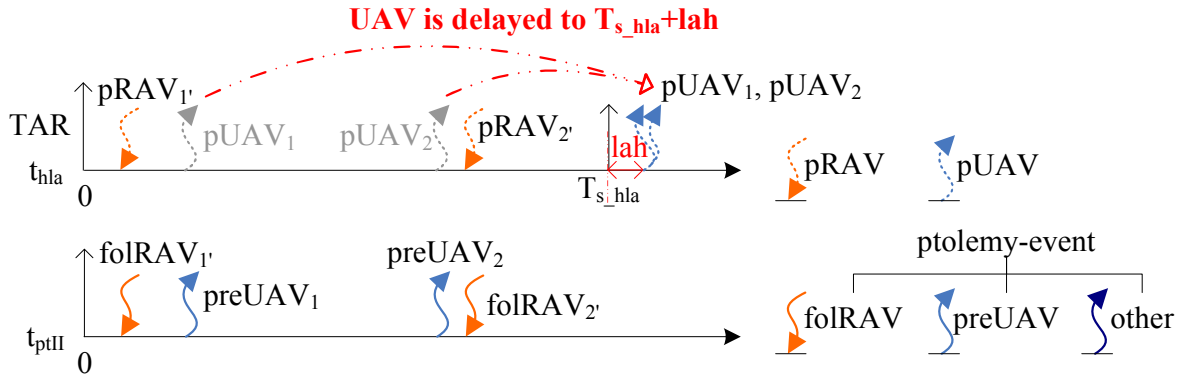


Figure 4.2: Concept of TAR: with UAV delay

Step 1.1 $t_{\text{current_ptII}} = 0, t_{\text{current_hla}} = 0$.

CalendarQueue: $\{\text{preUAV}_1(t_1), \text{preUAV}_2(t_2)\}$.

HLA service: $\text{pTAR}(T_{s_hla}); \{\text{tick}(), \text{pRAV}_{1'}(t_{1'})\}; \{\text{tick}(), \text{pRAV}_{2'}(t_{2'})\}; \{\text{tick}(), \text{pTAG}(T_{s_hla})\}$.

CalendarQueue: $\{\text{folRAV}_{1'}(t_{1'}), \text{preUAV}_1(t_1), \text{preUAV}_2(t_2), \text{folRAV}_{2'}(t_{2'})\}$.

Step 1.2 $t_{\text{current_ptII}} = t_{1'}, t_{\text{current_hla}} = T_{s_hla}$.

CalendarQueue: $\{\text{preUAV}_1(t_1), \text{preUAV}_2(t_2), \text{folRAV}_{2'}(t_{2'})\}$.

Step 1.3 $t_{\text{current_ptII}} = t_1, t_{\text{current_hla}} = T_{s_hla}$.

CalendarQueue: $\{\text{preUAV}_2(t_2), \text{folRAV}_{2'}(t_{2'})\}$.

HLA service: $\text{pUAV}_1(T_{s_hla} + \text{lah})$.

Step 1.4 $t_{\text{current_ptII}} = t_2, t_{\text{current_hla}} = T_{s_hla}$.

CalendarQueue: $\{\text{folRAV}_{2'}(t_{2'})\}$.

HLA service: $\text{pUAV}_2(T_{s_hla} + \text{lah})$.

Step 1.5 $t_{current_ptII} = t_{2'}$, $t_{current_hla} = T_{s_hla}$.
 CalendarQueue: \emptyset .

In a nutshell, both events $preUAV_1$ and $preUAV_2$, occurring at different time t_1 and t_2 , are delayed and sent to the RTI as *rav-events* at time $T_{s_hla} + lah$. As for the pRAV, they are put in the CalendarQueue with their original time stamp.

4.2.1.2 Another implementation for Option 1

The slight different with the previous method is the way we deal with *rav-events*. The main idea of this choice is to make a pTAR() at the beginning of a cycle, deal with the pRAV immediately if we get one, finally try to receive its TAG when all *rav-events* already have been processed. It means that each time we receive a *rav-event*, we process our next event without TAG. In other words, the time interval $[t_{current_ptII}, t_{rav}]$ is valid because no more *rav-events* will arrive earlier than t_{rav} . Thus, pUAV will be handled during an advancing phase, (R2) should be taken into account which validates (R1). It is a must to delay pUAV to $t_{nextPointInTime_hla} + lah$ (see equation 4.7a). The same equations with (4.7) are used in this implementation.

We find out we have the same results comparing with previous idea with a more complicated process. In this way, the previous solution is our preference.

Example 2 We take the previous example to see how this method works.

Step 2.1 $t_{current_ptII} = 0$, $t_{current_hla} = 0$.

CalendarQueue: $\{preUAV_1(t_1, val_1), preUAV_2(t_2, val_2)\}$.

HLA service: $pTAR(T_{s_hla}); \{tick(), pRAV_{1'}(t_{1'})\}$.

CalendarQueue: $\{folRAV_{1'}(t_{1'}), preUAV_1(t_1), preUAV_2(t_2)\}$.

Step 2.2 $t_{current_ptII} = t_{1'}$, $t_{current_hla} = 0$.

CalendarQueue: $\{preUAV_1(t_1), preUAV_2(t_2)\}$.

HLA service: $\{tick(), pRAV_{2'}(t_{2'})\}$.

CalendarQueue: $\{preUAV_1(t_1), preUAV_2(t_2), folRAV_{2'}(t_{2'})\}$.

Step 2.3 $t_{current_ptII} = t_1$, $t_{current_hla} = 0$.

CalendarQueue: $\{preUAV_2(t_2), folRAV_{2'}(t_{2'})\}$.

HLA service: $pUAV_1(T_{s_hla} + lah); \{tick(), pTAG(T_{s_hla})\}$.

Step 2.4 $t_{current_ptII} = t_2$, $t_{current_hla} = T_{s_hla}$.

CalendarQueue: $\{folRAV_{2'}(t_{2'})\}$.

HLA service: $pUAV_2(T_{s_hla} + lah)$.

Step 2.5 $t_{current_ptII} = t_{2'}$, $t_{current_hla} = T_{s_hla}$.

CalendarQueue: \emptyset .

Comparing with Example 1 and 2, we can see that there is no difference dealing with $preUAV_1(t_1)$ and $preUAV_2(t_2)$ (e.g. Step 1.2 to 1.4 or 2.3 to 2.4) but Step 1.2 translates $RAV(t_{e1})$ and $RAV(t_{e2})$ at once, whereas in Example 2, two different steps are needed (Step 2.1 and 2.2).

4.2.1.3 Option 2: with Enforced RAV Delay

We don't intend to deal with the *rav-events* until the accomplishment of all *ptolemy-events*. It means that $\text{pTAR}(t_{\text{nextPointInTime_hla}})$ shall be done after the management of *ptolemy-events* in a cycle. We receive the *rav-events* continuously until $\text{pTAG}(t_{\text{nextPointInTime_hla}})$ is issued by RTI. With (R5), a *rav-event* $\text{pRAV}(t)$ with $t \in]t_{\text{current_hla}}, t_{\text{nextPointInTime_hla}}]$ should be registered in the queue after $t_{\text{current_ptII}}$, so we propose $\text{pRAV}(t_{\text{nextPointInTime_hla}})$ (see equation 4.8b). (R1) is conditionally guaranteed because the time $t_{\text{current_ptII}}$ and $t_{\text{current_hla}}$ are asynchronous. We make sure of (R1) explicitly (see equation 4.8a).

$$\forall t \in \mathbb{R}_+, f_{UAV}(t) = \begin{cases} t_{\text{current_hla}} + lah, & \text{if } t < t_{\text{current_hla}} + lah \\ t, & \text{otherwise} \end{cases} \quad (4.8a)$$

$$\forall t \in \mathbb{R}_+, f_{RAV}(t) = \begin{cases} t_{\text{nextPointInTime_hla}}, & \text{if } t \in]t_{\text{current_hla}}, t_{\text{nextPointInTime_hla}}] \\ t, & \text{if } t > t_{\text{nextPointInTime_hla}}. \end{cases} \quad (4.8b)$$

Example 3 Figure 4.3, $t_{\text{nextPointInTime_hla}} = T_{s_hla}$.

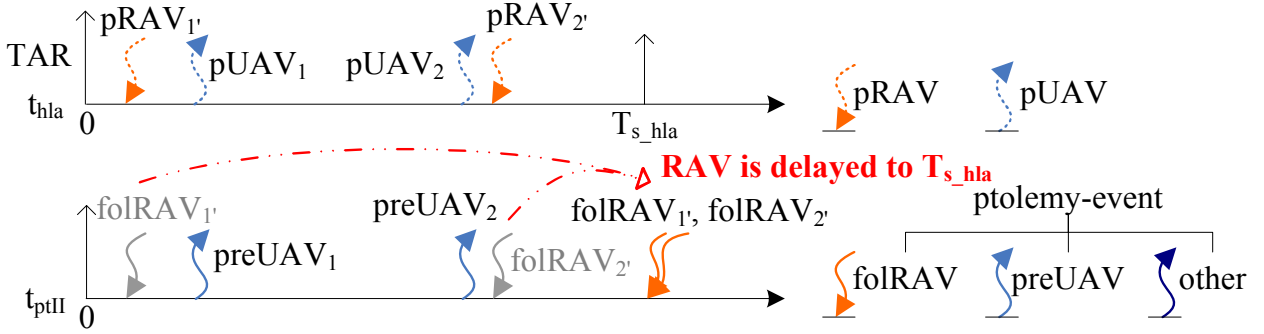


Figure 4.3: Concept of TAR: with RAV delay

Step 3.1 $t_{\text{current_ptII}} = t_1, t_{\text{current_hla}} = 0$.

CalendarQueue: $\{\text{preUAV}_2(t_2)\}$.

HLA service: $\text{pUAV}_1(t_1)$.

Step 3.2 $t_{\text{current_ptII}} = t_2, t_{\text{current_hla}} = 0$.

CalendarQueue: \emptyset .

HLA service: $\text{pUAV}_2(t_2); \text{pTAR}(T_{s_hla}); \{\text{tick}(), \text{pRAV}_{1'}(t_{1'})\}; \{\text{tick}(), \text{pRAV}_{2'}(t_{2'})\}; \{\text{tick}(), \text{pTAG}(T_{s_hla})\}$.

CalendarQueue: $\{\text{folRAV}_{1'}(T_{s_hla}), \text{folRAV}_{2'}(T_{s_hla})\}$.

Step 3.3 $t_{\text{current_ptII}} = T_{s_hla}, t_{\text{current_hla}} = T_{s_hla}$.

CalendarQueue: $\{\text{folRAV}_{2'}(T_{s_hla})\}$.

Step 3.4 $t_{\text{current_ptII}} = T_{s_hla}, t_{\text{current_hla}} = T_{s_hla}$

CalendarQueue: \emptyset .

In a nutshell, the *ptolemy-events* $\text{preUAV}_1(t_1), \text{preUAV}_2(t_2)$ are translated to *uav-events* at their own time stamp t_{e1} and t_{e2} , while *rav-events* are delayed to T_{s_hla} knowing that $t_{\text{current_hla}} = 0$.

4.2.2 Chosen Approach

Table 4.1: Analysis of possible solutions

	Option 1	Option 2
Implementation	Enforced delay for UAV	Enforced delay for RAV
Time Deviation <i>lah</i> for UAV	Unconditional	Conditional
Processing for simple producer consumer model	<p>Assuming that the execution order is Producer 1, Producer 2, Consumer.</p>	
Feasibility	Not Complicated	
Performance	<p>Not exactly the same behavior with NER, performance depends on the choice of T_{s_hla} and lah. No $pUAV(t)$-$pRAV(t)$ loop in an execution cycle.</p>	

In fact, we could get an another option which combines option 1 and 2 with enforced UAV and RAV delay at the same time with no time deviation. But the accumulation of two times of delay bothers us obviously. We abandon this idea at once. Option 1 and 2 can both be implemented. However, we should pay more attention, in some cases these two options will differentiate the execution order of events. In the example in table 4.1, the federation Consumer would receive $preUAV_1(t_1)$ and $preUAV_2(t_2)$ at $T_{s_hla} + lah$ when using option1 and would receive $preUAV_2(t_2)$, $preUAV_1(t_1)$ at T_{s_hla} . Considering a less time deviation in UAV, option 2 is chosen.

The option 2 allows us to deal with all *ptolemy-events* in a cycle of time then advance the HLA logical time if a permission is granted (the reception of TAG), after a time request TAR made through `proposeTime()`.

In the waiting phase for a TAG, if a *rav-event* arrives with time stamp τ , we don't want to translate it to a *ptolemy-event* with its same time stamp τ . Because the appearance of a new *ptolemy-event* will contradict with our previous work: we have already finished the processing of all *ptolemy-events* in the cycle. So it is proposed to deal with this event at the end of the cycle.

4.3 Implementation of TAR

4.3.1 Primary Results

We will take the option 2 enforcing RAV delay in a Java implementation after a primary analysis. In the following part, the notation for two events a, b :

- $a; b$ means b is executed after a .
- $a \Rightarrow b$ means a is translated to b .

Corresponding to our choice, we have the results as follows.

- 1) All federates are supposed to be regulating and constrained at the same time.
- 2) RTI delivers *rav-events* respecting increasing time stamp order. It means that if two *rav-events* have the same time stamp, they will be executed depending on the order of *uav - events'* generation.

$$\forall \text{pRAV}(t_1, \text{val1}), \text{pRAV}(t_2, \text{val2}) \text{ with } t_1, t_2 \in \mathbb{R}^+$$

$$\begin{cases} \text{pRAV}(t_1, \text{val1}); \text{pRAV}(t_2, \text{val2}), & \text{If } t_1 < t_2, \text{ or } \{t_1 = t_2, \text{pUAV}(t_1, \text{val1}); \text{pUAV}(t_2, \text{val2})\} \\ \text{pRAV}(t_2, \text{val1}); \text{pRAV}(t_1, \text{val2}), & \text{If } t_1 > t_2, \text{ or } \{t_1 = t_2, \text{pUAV}(t_2, \text{val2}); \text{pUAV}(t_1, \text{val1})\} \end{cases}$$

- 3) All *rav-events* in the same cycle will be time stepped to $t_{\text{nextPointInTime_hla}}$. Each *rav-event* will be translated to its associated *ptolemy-event* named folRAV following its reception.

$$\forall t \in \mathbb{R}_+, f_{\text{RAV}}(t) = \begin{cases} t_{\text{nextPointInTime_hla}}, & \text{if } t \in]t_{\text{current_hla}}, t_{\text{nextPointInTime_hla}}] \\ t, & \text{if } t > t_{\text{nextPointInTime_hla}}. \end{cases}$$

- 4) An *uav-event* will be delayed to send if its time stamp is smaller than $t_{\text{current_hla}} + lah$, otherwise it keeps unchangeable.

$$\forall t \in \mathbb{R}_+, f_{\text{UAV}}(t) = \begin{cases} t_{\text{current_hla}} + lah & \text{If } t < t_{\text{current_hla}} + lah \\ t & \text{Otherwise} \end{cases}$$

- 5) Considering the asynchronism problem of t_{0_ptII} (see section 3.2.1), the *rav-events* arriving before t_{0_ptII} , will be all received at the beginning of federate execution.

$$\forall t \in \mathbb{R}, f_{\text{RAV}}(t) = \begin{cases} t_{0_ptII} & \text{If } t \leq t_{0_ptII} \\ t & \text{Otherwise} \end{cases}$$

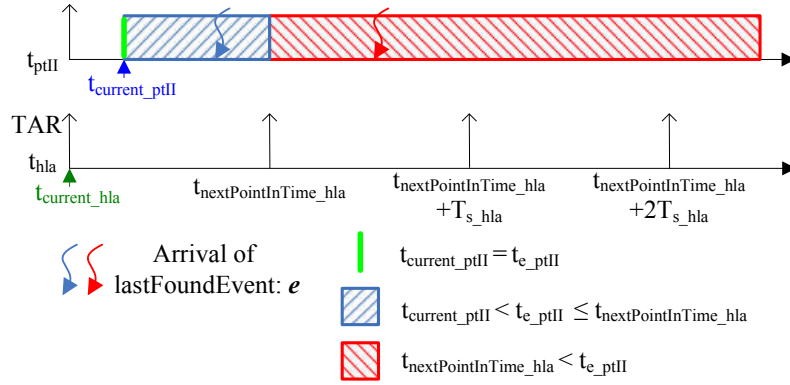


Figure 4.4: Different intervals for TAR

4.3.2 Development

According to the arrival time of the earliest *ptolemy-event*, named *lastFoundEvent*, we can distinguish the following cases showing different behaviors.

- (1) *lastFoundEvent* will arrive at the Ptolemy current time (see figure 4.4 green area) which means

$$t_{lastFoundEvent} = t_{current_ptII}$$

In this case, it is not necessary to call HLA time advancement services (NER and TAR) at its current Ptolemy time. As a result, the performance can be improved by returning its current time without any HLA services.

- (2) *lastFoundEvent* will arrive in the same cycle (see figure 4.4 blue area) which means

$$t_{lastFoundEvent} \in]t_{current_ptII}, t_{nextPointInTime_hla}]$$

With the asynchronism of $t_{current_ptII}$ and $t_{current_hla}$, all *ptolemy-events* in the same cycle will be priorly treated, thus advancing to its current time $t_{current_ptII}$. There's no worry about the arrival of *rav-events* because they are all delayed to their own $t_{nextPointInTime_hla}$.

- (3) *lastFoundEvent* has its time stamp after $t_{nextPointInTime_hla}$ (see figure 4.4 red area).

$$t_{lastFoundEvent} \in]t_{nextPointInTime_hla}, +\infty[$$

There is no *ptolemy-event* in the interval $]t_{current_ptII}, t_{nextPointInTime_hla}]$ during this phase, the federate must do $\text{pTAR}(t_{nextPointInTime_hla})$ then continuously do $\text{tick2}()$ action until its request is granted by a callback $\text{pTAG}(t_{nextPointInTime_hla})$. After then, the $t_{current_hla}$ is advanced to $t_{nextPointInTime_hla}$. If we get any *rav-event*, it must be converted to a *ptolemy-event* with time stamp $t_{current_hla}$ which is equal to $t_{nextPointInTime_hla}$. As a result, the earliest event in `CalendarQueue` is no more the original *lastFoundEvent*, but the recent received event, $t_{lastFoundEvent} = t_{current_hla}$.

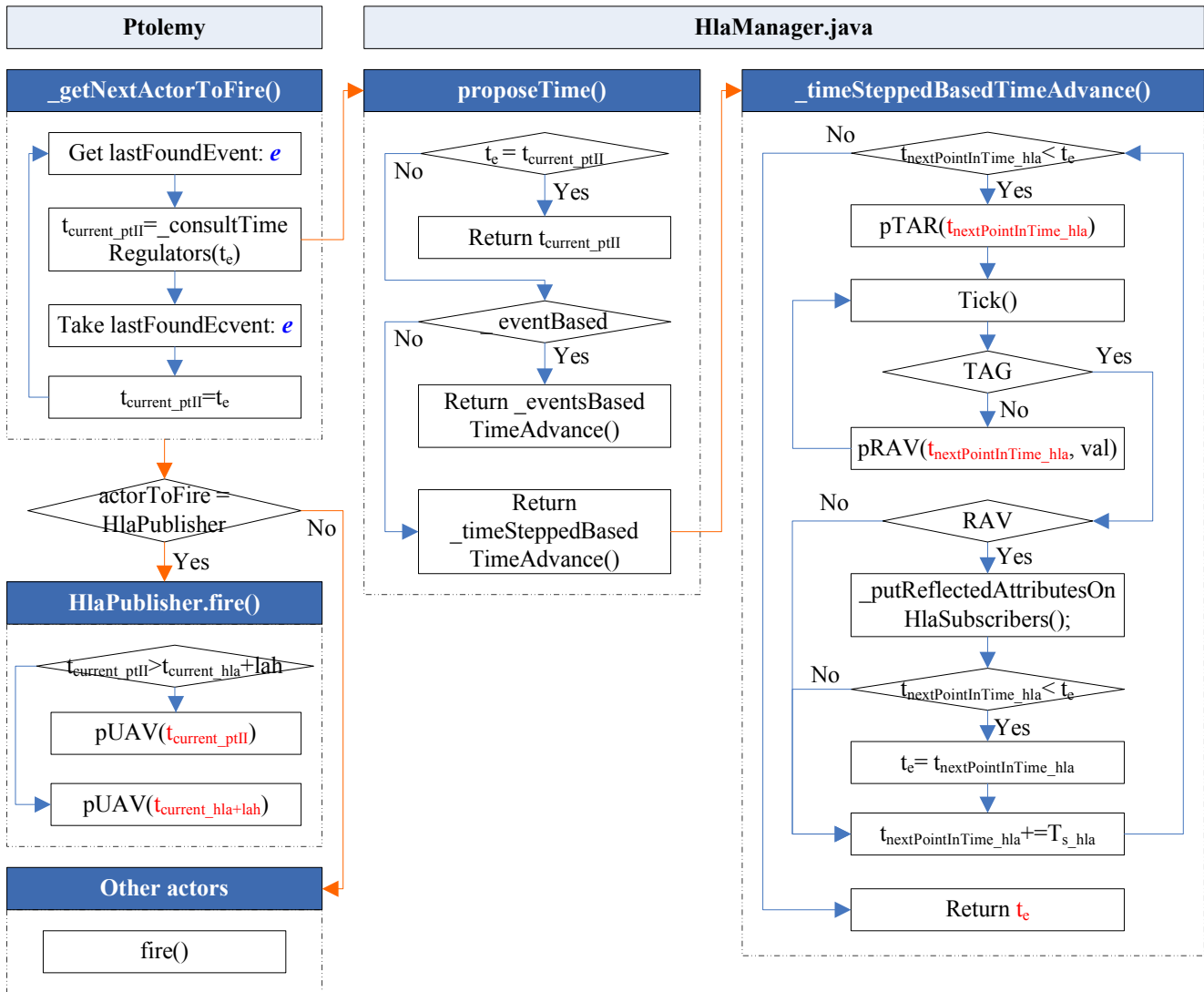


Figure 4.5: TAR Time Advancement Diagram

4.3.3 Programing

We test our condition (1) in section 4.3.2 at the beginning of `proposeTime()` (see algorithm 1 line 2) for reducing HLA services calls, directly return $t_{current_ptII}$ and give a detailed process in `_timeSteppedBasedTimeAdvance()` including above-mentioned case (2), case(3) (see figure 4.5 in section 4.3.2). Its pseudo code is written for a clearer understanding (see algorithm 2).

Algorithm 2 Propose a valid time though TAR

Input: an initial *proposedTime* gets the time stamp of *lastFoundEvent*
Output: a valid *proposedTime* to advance to

```

1: Method: _TIMESTEPPEDBASEDTIMEADVANCE(proposedTime)
2:   /* CASE: LastFoundEvent will arrive after  $t_{nextPointInTime\_hla}$ . */
3:   while proposedTime >  $t_{nextPointInTime\_hla}$  do
4:     pTAR( $t_{nextPointInTime\_hla}$ );
5:     tag ← FALSE;
6:     rav ← FALSE;
7:     while tag = FALSE do
8:       /* Every tick() get a callback pRAV(t) or pTAG(t),
9:        * Once TAG is received,  $t_{current\_hla} = t_{nextPointInTime\_hla}$  */
10:      tick();
11:    end while
12:    if rav = TRUE then
13:      put rav-events in the queue with RAV delay;
14:      if  $t_{nextPointInTime\_hla} < proposedTime$  then
15:        proposedTime ←  $t_{nextPointInTime\_hla}$ ;
16:      end if
17:    end if
18:     $t_{nextPointInTime\_hla} \leftarrow t_{nextPointInTime\_hla} + T_{s\_hla}$ ;
19:  end while

20:  /* CASE: LastFoundEvent will directly or indirectly (via previous
21:   * execution) arrive in the same cycle.*/
22:  return proposedTime;
23: end Method:

```

4.4 Example of TAR

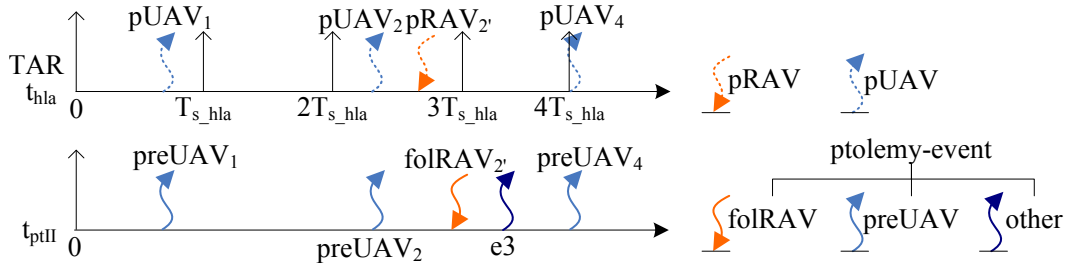


Figure 4.6: Example of TAR

We take our previous example in NER to see how TAR works, shown in figure 4.6.

Execution Process Time starts from $t_{current_ptII} = 0$, $t_{current_hla} = 0$. So $t_{nextPointInTime_hla} = T_{s_hla}$.

- Get preUAV₁(t_1)
 - $t_1 \leq T_{s_hla} \rightarrow$ return $t_1 \rightarrow$ Take preUAV₁ → Set $t_{current_ptII} = t_1$
 - $t_{current_hla} + lah \leq t_{current_ptII} \rightarrow$ pUAV₁($t_{current_ptII}$)
- Get preUAV₂(t_2)
 - $t_2 > T_{s_hla} \rightarrow$ pTAR(T_{s_hla}) → {tick(), pTAG(T_{s_hla})}, $t_{current_hla} = T_{s_hla} \rightarrow$ no RAV, $t_{nextPointInTime_hla} = 2T_{s_hla}$
 - $t_2 > 2T_{s_hla} \rightarrow$ pTAR($2T_{s_hla}$) → {tick(), pTAG($2T_{s_hla}$)}, $t_{current_hla} = 2T_{s_hla} \rightarrow$ no RAV, $t_{nextPointInTime_hla} = 3T_{s_hla}$
 - $t_2 \leq 3T_{s_hla} \rightarrow$ return $t_2 \rightarrow$ Take preUAV₂ → Set $t_{current_ptII} = t_2$
 - $t_{current_hla} + lah < t_{current_ptII} \rightarrow$ pUAV₂($t_{current_ptII}$)
- Get e3(t_3)
 - $t_3 > 3T_{s_hla} \rightarrow$ pTAR($3T_{s_hla}$) → {tick(), pRAV($t_{2'}$)} → {tick(), pTAG($3T_{s_hla}$)}, $t_{current_hla} = 3T_{s_hla} \rightarrow$ with RAV, folRAV_{2'}($3T_{s_hla}$) is put in the queue. → $t_{2'} \leq 3T_{s_hla} \rightarrow t_{2'} = 3T_{s_hla}$, $t_{nextPointInTime_hla} = 4T_{s_hla}$
 - $t_{2'} \leq 4T_{s_hla} \rightarrow$ return $t_{2'} \rightarrow$ Take folRAV_{2'} → Set $t_{current_ptII} = t_{2'}$
 - fire($t_{2'}$)
- Get e3 with $t_3 \leq 4T_{s_hla} \rightarrow$ return $t_3 \rightarrow$ Take e3 → Set $t_{current_ptII} = t_3$
- fire(t_3)
- Get e4 with $t_4 \leq 4T_{s_hla} \rightarrow$ return $t_4 \rightarrow$ Take e4 → Set $t_{current_ptII} = t_4$
- $t_{current_hla} + lah < t_{current_ptII} \rightarrow$ pUAV₄($t_{current_hla}$)

4.5 Validation of TAR

After the implementation of TAR algorithm, we want to see whether it performs in our expectation. A simple producer multiple consumer model has been constructed to verify the consistency with the specifications. Its detail is shown in figure 4.7. Moreover, it is also provided after installation of Ptolemy, situated in \$PTII /org /hacerti /SimpleProducerMultipleConsumerTAR.

Here, we have got a unique producer (prod1), following two parallel consumers (cons1 & cons2). Another two parallel consumers (cons3 & cons4) are connected to cons2 which permit us to distinguish the different behavior of NER and TAR. In this example, there are *ptolemy-events* which are produced by actor DiscreteClock, and RAV messages which are received by HlaSubscriber in prod1 producer1(as depicted in figure 4.8c).

All federates start with Ptolemy start time $t_{0_ptII} = 0$ and have the same lookahead lah **lah=0.1**. All of them have the same time step T_{s_hla} **Ts = 10** except for prod1 which has **Ts=4**.

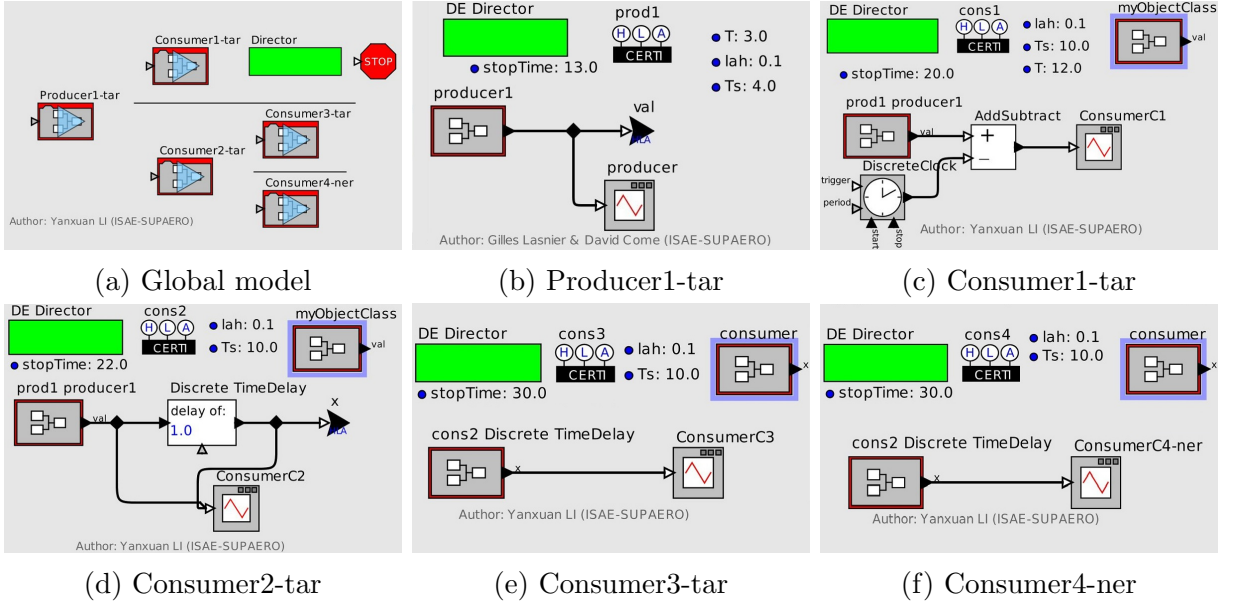


Figure 4.7: SimpleProducerMultipleConsumerTAR

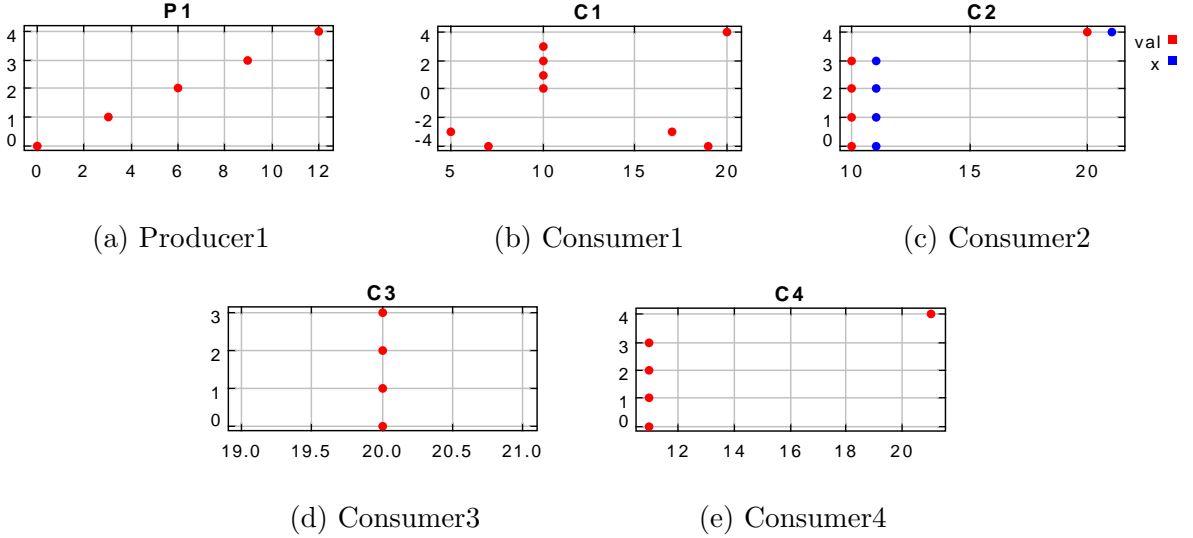


Figure 4.8: Results of SimpleProducerMultipleConsumerTAR

T represents the period of a **DiscreteClock** that generates the periodic events.

Their execution results are shown in figure 4.8. Ignoring *microstep* and *actor*, an event is noted as $e(t, val)$. `prod1`, at the beginning, generates the ramp value every three seconds, denoted as $e_1(0, 0)$, $e_2(3, 1)$, $e_3(6, 2)$, $e_4(9, 3)$, $e_5(12, 4)$. `cons1` and `cons2` receive all these values at the same time. Knowing that all *rav-events* with time stamp smaller than $t_{nextPointInTime_hla}$ shall be delayed to $t_{nextPointInTime_hla}$. Four values corresponding to e_1, e_2, e_3, e_4 are received at time 10 by `cons1` & `cons2`, and the rest one corresponding to e_5 is received at time 20. Moreover, for `cons1`, the actor **DiscreteClock** produces the “internal” events at time 5 and 7 with a period 12. Meanwhile, the events in `cons2` are delayed explicitly with time 1.0.

Federates `cons3` and `cons4` work at the same time for reception. There’s no doubt that `cons3`

shall receive the first four values at its time step 20 with a TAR application (see figure 4.8d). Then cons4 reacts immediately to obtain the sent values thanks to NER, as shown in figure 4.8e. These results are the expected ones according to the analysis done in section 4.2.

4.6 Problem for stopTime

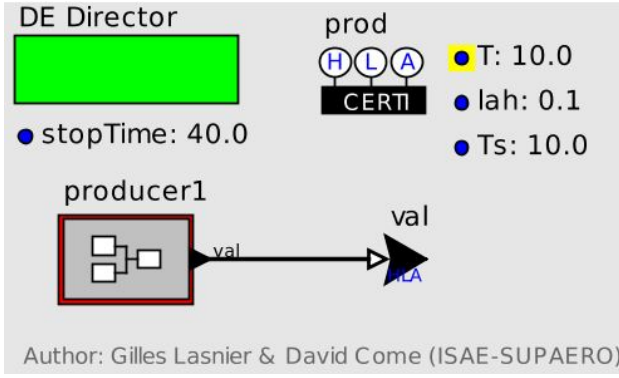
Let us note t_{stop} for the stop time of each federate. If we analyze the results depicted on figure 4.8 in detail, we can observe that:

1. The event $e_5(12, 4)$ sent by federate prod1 is *delayed* by federate cons2 to timestamp $t=20$. During the execution of cons2, an additional delay is affected by a **TimeDelay** actor (with delay of 1 time unit). The associated value is sent through the output \mathbf{x} with timestamp $t = 21$ and is displayed in figure 4.8c. Note that the stop time of this federate is $t_{stop} = 22 > 20$.
2. The same event $e_5(12, 4)$ sent by federate prod1 is also *delayed* by federate cons1 to timestamp $t=20$. Note that the stop time of federate cons1 is $t_{stop} = 20$ and this event is displayed in figure 4.8b.
3. The attribute \mathbf{x} is subscribed by federate cons3, with $t_{stop} = 30$. Those events with timestamp $t=11$ (at cons2) are delayed by cons3 at timestamp $t=20$. We expect that the event with timestamp $t = 21$ (at cons2) would have been received by cons3 at timestamp $t = 30$ and displayed. But only events with $t = 20$ (at cons3) are displayed but the one with $t=30$ is not shown in figure 4.8d.

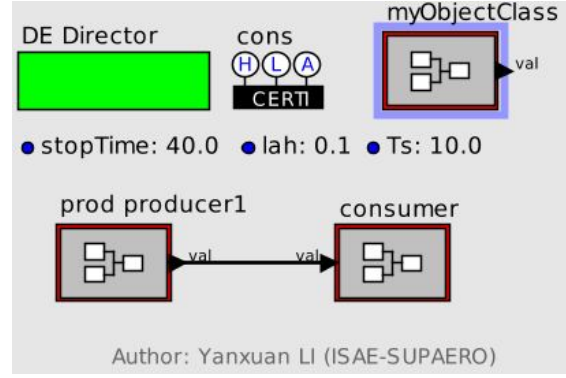
What is the difference between federate cons1 and cons3? In both cases, $t_{stop} = k \times T_{s_hla}$.

Let us remind that a federate will stop for two conditions:

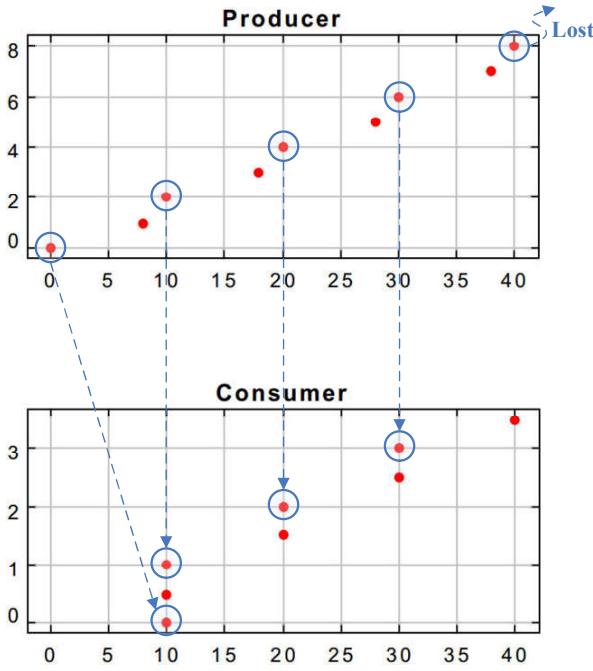
- (1) **CalendarQueue** is empty, so there is no events to treat.
After dealing with all rav-events with time stamp < 20 (at this time, $t_{current_hla} = 10$), there's no events in the queue of cons3 so it terminates its work, and the federate will lost the data in its last cycle of execution.
- (2) **CalendarQueue** still has events inside but the next event i.e. `lastFoundEvent` will arrive later than `stopTime` which means $t_{lastFoundEvent} > t_{stop}$.
Federate Cons1 ends up under this condition because **DiscreteClock** regenerates a new event at time $t = 5 + 2 * T = 29$ with a value val and stock it in the queue. In this way, DE director will get this event, make a time advance request TAR(20) to HlaManager through `proposedTime()`. After the authority of RTI, $t_{current_hla}$ is advanced to 20, and the $t_{current_ptII}$ will be advanced to 29. At the end, it finds out this proposal time 29 is later than the t_{stop} for cons1 which stops the work of federate.



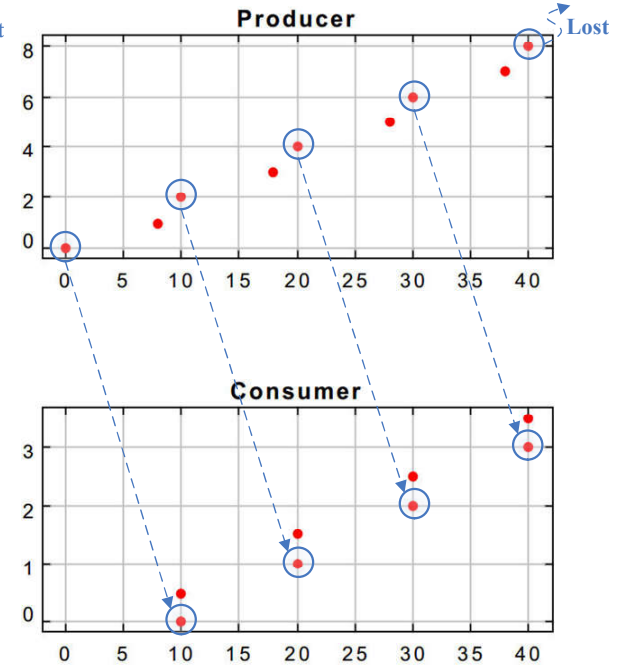
(a) Producer



(b) Consumer



(a) Result for solution (a)



(b) Result for solution (b)

Proposal Solutions This two solutions are not yet coded in the `HlaManager.java`. We suppose the stop time $t_{stop} = nT_{s_hla}$, with $n \in \mathbb{N}$. A simple producer/consumer model (see figure ??) is used for explaining the difference of two results.

- (a) Process the *ptolemy-events* with time stamp kT_{s_hla} ($0 < k < n$) before the time advance request $TAR(kT_{s_hla})$ except for the events with time stamp 0 and nT_{s_hla} . In this case, $t_{current_hla}$ maintain at time kT_{s_hla} when we process the $preUAV((k+1)T_{s_hla})$, with $k \in]0, n[$.

$$\forall k \in]0, n[, \text{ if we have } preUAV(kT_{s_hla}) \Rightarrow UAV(kT_{s_hla})$$

$$\text{Then, } RAV(kT_{s_hla}) \Rightarrow folRAV(kT_{s_hla})$$

$$\text{If } k = 0, n, \text{ } preUAV(kT_{s_hla}) \Rightarrow UAV(kT_{s_hla} + lah)$$

$$RAV(kT_{s_hla} + lah) \Rightarrow folRAV((k+1)T_{s_hla} + lah)$$

Conclusion: $\forall k \in]0, n[, preUAV(kT_{s_hla})$ will be received at time kT_{s_hla} , but for $k = 0, n$, $preUAV(kT_{s_hla})$ will be received at $(k+1)T_{s_hla}$.

(b) Process the *ptolemy-events* with time stamp kT_{s_hla} ($0 \leq k \leq n$) after the time advance request $TAR(kT_{s_hla})$. In this way, $t_{current_hla}$ advance to kT_{s_hla} , $t_{current_hla} = k * Ts$ earlier than the *ptolemy-events* processing. It means that:

$$\forall k \in [0, n], \text{ if we have } \text{preUAV}(kT_{s_hla}) \Rightarrow \text{UAV}(kT_{s_hla} + lah)$$

$$\text{Then, } \text{RAV}(kT_{s_hla} + lah) \Rightarrow \text{folRAV}((k + 1)T_{s_hla})$$

Conclusion: $\forall k \in [0, n]$, $\text{preUAV}(kT_{s_hla})$ will be received at time $(k + 1)T_{s_hla}$.

Chapter 5

Conclusion

This internship provides an opportunity to know distributed simulation works. Attention was focused to improve the PTII-HLA simulation framework by implementing the of the TAR mechanism. As a standard for distributed simulation, HLA aims to offer the interoperability and reuse of federates. While PTII takes charge of heterogeneous simulation. We benefit well with their co-simulation framework.

With the existing implementation of NER work mechanism, its analysis was done at the beginning which provides us an opportunity to get more acknowledges. We have found out some shortcomings in NER implementation, various slight modifications were naturally done to improve the NER. Meanwhile, being familiar with the PTII-HLA co-simulation framework, we intended to think out our algorithm for TAR implementation. Even though HLA standards concerning TAR are well specified, we have made some choices for the implementation under certain constraints of Ptolemy. All assumptions of our work were detailed in this report. The explanation of algorithm is always not easy, that's why we tried to find a formal way to let the algorithm make sense. For a validation of TAR, we made several *corn cases* models to see its performance which conforms our expectations.

This work has several perspectives: implement the special cases for NER and TAR where lookahead is equal to zero, named next event request available (NERA) and time advance request available (TARA). Now that TAR is validated for quite simple models, a more complicated application could be implemented.

During the whole internship, I became familiar with Unix system because all the work are done inside, working with the Ptolemy project headed by Christopher Brooks at University of Berkeley allowed me to learn how to work with other programmers for developing a software together, how to properly program, what is the pattern design. The tool svn is so good to make a version control. This internship give me a chance to learn a lot of new things and put them in use.

Appendix A

HlaManager.java

A.1 proposeTime(Time)

```
/** Propose a time to advance to. This method is the one implementing the
 * TimeRegulator interface and using the HLA/CERTI Time Management services
 * (if required). Following HLA and CERTI recommendations, if the Time
 * Management is required then we have the following behavior:
 * Case 1: If lookahead = 0
 * -a) if time-stepped Federate, then the timeAdvanceRequestAvailable()
 *     (TARA) service is used;
 * -b) if event-based Federate, then the nextEventRequestAvailable()
 *     (NERA) service is used
 * Case 2: If lookahead > 0
 * -c) if time-stepped Federate, then timeAdvanceRequest() (TAR) is used;
 * -d) if event-based Federate, then the nextEventRequest() (NER) is used;
 * Otherwise the proposedTime is returned.
 * NOTE: For the Ptolemy II - HLA/CERTI cooperation the default (and correct)
 * behavior is the case 1 and CERTI has to be compiled with the option
 * "CERTI_USE_NULL_PRIME_MESSAGE_PROTOCOL".
 * @param proposedTime The proposed time.
 * @return The proposed time or a smaller time.
 * @exception IllegalActionException If this attribute is not
 * contained by an Actor.
 */
@Override
public Time proposeTime(Time proposedTime) throws IllegalActionException {

    Time currentTime = _director.getModelTime();
    ...

    // If the proposedTime is equal to current time
    // so it has no need to ask for the HLA service
    // then return the currentTime.

    if (currentTime.equals(proposedTime)) {
        // Even if we avoid the multiple calls of the HLA Time management
        // service for optimization, it could be possible to have events
        // from the Federation in the Federate's priority timestamp queue,
        // so we tick() to get these events (if they exist).
        if (_debugging) {
```

```

        _debug(this.getDisplayName() + " proposeTime() - SKIP RTI"
              + " with current Time is equal to proposed Time ("
              + currentTime.getDoubleValue() + ")");
    }
    try {
        _rtia.tick();
    } catch (ConcurrentAccessAttempted e) {
        throw new IllegalArgumentException(this, e,
            "ConcurrentAccessAttempted ");
    } catch (RTIinternalError e) {
        throw new IllegalArgumentException(this, e, "RTIinternalError ");
    }

    return currentTime;
}

// If the HLA Time Management is required, ask to the HLA/CERTI
// Federation (the RTI) the authorization to advance its time.
if (_isTimeRegulator && _isTimeConstrained) {
    synchronized (this) {
        // Call the corresponding HLA Time Management service.
        try {
            if (_eventBased) {
                return _eventsBasedTimeAdvance(proposedTime);
            } else {
                return _timeSteppedBasedTimeAdvance(proposedTime);
            }
        } catch (InvalidFederationTime e) {
            ...
        }
    }
}

//_lastProposedTime = breakpoint;
return null;
}

```

A.2 _eventsBasedTimeAdvance(Time)

```

/**
 * RTI service for event-based federate (NER or NERA)
 * is used for proposing a time to advance to.
 * @param proposedTime time stamp of lastFoundEvent
 * @return a valid time
 */
private Time _eventsBasedTimeAdvance(Time proposedTime)
    throws IllegalArgumentException, InvalidFederationTime,
    FederationTimeAlreadyPassed, TimeAdvanceAlreadyInProgress,
    FederateNotExecutionMember, SaveInProgress,
    EnableTimeRegulationPending, EnableTimeConstrainedPending,
    RestoreInProgress, RTIinternalError, ConcurrentAccessAttempted,
    SpecifiedSaveLabelDoesNotExist {

```



```

CertiLogicalTime certiProposedTime = _convertToCertiLogicalTime(proposedTime);

if (_hlaLookAhead > 0) {
    // Event-based + lookahead > 0 => NER.
    if (_debugging) {
        _debug(this.getDisplayName()
            + " proposeTime() - current status - " + "t_ptII = "
            + _director.getModelTime().getDoubleValue()
            + "; t_certi = " + _federateAmbassador.logicalTimeHLA
            + " - call CERTI NER -" + " nextEventRequest("
            + certiProposedTime.getTime() + ") with model at "
            + proposedTime.getDoubleValue());
    }
    _rtia.nextEventRequest(certiProposedTime);
} else {
    // Event-based + lookahead = 0 => NERA + NER.
    // Start the time advancement loop with one NERA call.
    ...
}

// Wait the time grant from the HLA/CERTI Federation (from the RTI).
_federateAmbassador.timeAdvanceGrant = false;
int cntTick = 0;
while (!(_federateAmbassador.timeAdvanceGrant)) {
    if (_debugging) {
        _debug(this.getDisplayName() + " proposeTime() -"
            + " wait CERTI TAG - " + "timeAdvanceGrant("
            + certiProposedTime.getTime() + ") by calling tick2()");
    }
    _rtia.tick2();
    cntTick++;
}

// If we get any rav-event
if (cntTick != 1){
    // Store reflected attributes RAV as events on HLASubscriber actors.
    _putReflectedAttributesOnHlaSubscribers();

    // At this step we are sure that the HLA logical time of the
    // Federate has been updated (by the reception of the TAG callback
    // (timeAdvanceGrant()) and its value is the proposedTime or
    // less, so we have a breakpoint time.
    try {
        CertiLogicalTime hlaTimeGranted = (CertiLogicalTime)
            _federateAmbassador.logicalTimeHLA;
        Time breakpoint = _convertToPtolemyTime(hlaTimeGranted);
        if (_debugging) {
            _debug("TAG for " + hlaTimeGranted + "model moves to"
                + breakpoint.getDoubleValue());
        }
    }
    // So we'd like to propose the breakpoint time instead.
    proposedTime = breakpoint;
}

```

```

    } catch (IllegalActionException e) {
        throw new IllegalActionException(this, e,
            "The breakpoint time is not a valid Ptolemy time");
    }
}

return proposedTime;
}

```

A.3 `_timeSteppedBasedTimeAdvance(Time)`

```

/**
 * RTI service for time-stepped federate (TAR or TARA)
 * is used for proposing a time to advance to.
 * @param proposedTime time stamp of lastFoundEvent
 * @return a valid time to advance to
 */
private Time _timeSteppedBasedTimeAdvance(Time proposedTime)
    throws IllegalActionException, InvalidFederationTime,
    FederationTimeAlreadyPassed, TimeAdvanceAlreadyInProgress,
    FederateNotExecutionMember, SaveInProgress,
    EnableTimeRegulationPending, EnableTimeConstrainedPending,
    RestoreInProgress, RTIInternalError, ConcurrentAccessAttempted {

    Time currentTime = _director.getModelTime();

    if (_hlaTimeStep > 0) {
        // Calculate the next point in time for making a TAR(hlaNextPointInTime)
        // or TARA(hlaNextPointInTime)
        Time hlaNextPointInTime = _getHlaNextPointInTime();
        CertiLogicalTime certiNextPointInTime =
            _convertToCertiLogicalTime(hlaNextPointInTime);

        // There are two types of events in a federate model:
        // - rav et uav events via RTI
        // - all events in ptolemy eventQueue

        if (_hlaLookAhead > 0) {
            // Time-stepped + lookahead > 0 => TAR.
            // LastFoundEvent is the earliest event in calendar queue,
            // We'd like to see if it is still the earliest one after
            // a registration of a rav-event.

            // There are two cases:
            // case 1:
            // WHILE time stamp of LastFoundEvent(proposedTime) >
            //     hlaNextPointInTime,
            // THEN TAR(hlaNextPointInTime) service is called.
            //     Tick() doesn't stop Until it receives a TAG(hlaNextPointInTime),
            //     At the end of tick,
            //     IF we get any RAV(t)
            //         all rav should be put in the queue

```

```

//          with a time stamp delay to hlaNextPointInTime;
//          IF hlaNextPointInTime < proposedTime which means
LastFoundEvent
//          is no more our earlist event;
//          THEN proposedTime should be raplaced by
hlaNextPointInTime.
//          END IF
//      END IF
// END WHILE
// case 2: LastFoundEvent is directly or indirectly(via case 1)
//          in (hlacurrentTime, hlaNextPointInTime]
//          we don't want to advance the certi time.
//          so return proposedTime, this event is valid to execute.

// case 1:
while (proposedTime.compareTo(hlaNextPointInTime) > 0) {
    if (_debugging) {
        _debug(this.getDisplayName()
            + " proposeTime() - current status "
            + "t_ptII = " + currentTime + "; t_certi = "
            + _federateAmbassador.logicalTimeHLA
            + " - call CERTI TAR - "
            + " timeAdvanceRequest("
            + certiNextPointInTime.getTime() + ")");
    }
    _rtia.timeAdvanceRequest(certiNextPointInTime);

    // Wait the time grant from the HLA/CERTI Federation (from the RTI).
    _federateAmbassador.timeAdvanceGrant = false;
    int cntTick = 0;
    while (!(_federateAmbassador.timeAdvanceGrant)) {
        if (_debugging) {
            _debug(this.getDisplayName() + " proposeTime() -"
                + " wait CERTI TAG - "
                + "timeAdvanceGrant("
                + certiNextPointInTime.getTime()
                + ") by calling tick2()");
        }

        try {
            _rtia.tick2();
            cntTick++;
        } catch (RTIException e) {
            throw new IllegalArgumentException(this, e,
                e.getMessage());
        }
    }

    // If we get any rav-event
    if (cntTick != 1) {
        // Store reflected attributes as events on HLASubscriber actors.
        _putReflectedAttributesOnHlaSubscribers();
        // If the new rav-event will arrive before our lastFoundEvent,

```

```

        if (hlaNextPointInTime.compareTo(proposedTime) < 0)
            proposedTime = hlaNextPointInTime;
    }
    // Advance the hlaNextPointInTime
    hlaNextPointInTime = _getHlaNextPointInTime();
    certiNextPointInTime =
        _convertToCertiLogicalTime(hlaNextPointInTime);
}

// case 2:
if (_debugging) {
    _debug(this.getDisplayName()
        + " proposeTime() - current status " + "t_ptII = "
        + currentTime + "; t_certi = "
        + _federateAmbassador.logicalTimeHLA
        + " - an event with time stamp = " + proposedTime
        + " will be taken.");
}
return proposedTime;

} else {
    // Time-stepped + lookahead = 0 => TARA + TAR.
    // Start the loop with one TARA call.
    ...
}
}
return null;
}
}

```

A.4 updateHlaAttribute(HlaPublisher, Token, String)

```

void updateHlaAttribute(HlaPublisher hp, Token in, String senderName)
    throws IllegalArgumentException {
    Time currentTime = _director.getModelTime();

    // The following operations build the different arguments required
    // to use the updateAttributeValues() (UAV) service provided by HLA/CERTI.

    // Retrieve information of the HLA attribute to publish.
    Object[] tObj = _hlaAttributesToPublish.get(hp.getName());

    // Encode the value to be sent to the CERTI.
    byte[] valAttribute = MessageProcessing.encodeHlaValue(hp, in);

    if (_debugging) {
        if (hp.useCertiMessageBuffer()) {
            _debug(this.getDisplayName()
                + " - A HLA value from ptolemy has been"
                + " encoded as CERTI MessageBuffer" + " , currentTime="
                + _director.getModelTime().getDoubleValue());
        }
    }
}
}

```

```

SuppliedAttributes suppAttributes = null;
try {
    suppAttributes = RtiFactoryFactory.getRtiFactory()
        .createSuppliedAttributes();
} catch (RTIinternalError e) {
    throw new IllegalArgumentException(this, e, "RTIinternalError ");
}
suppAttributes.add(_getAttributeHandleFromTab(tObj), valAttribute);

byte[] tag = EncodingHelpers.encodeString(_getPortFromTab(tObj)
    .getContainer().getName());

// Create a representation of uav-event timestamp for CERTI.
// HLA implies to send event in the future when using NER or TAR services with
// lookahead > 0.
// To avoid CERTI exception when calling UAV service
// with condition: uav(tau) tau >= hlaCurrentTime + lookahead.
Time uavTimeStamp = null;
if (_eventBased) {
    // for NER, we add the lookahead value to the uav event's timestamp.
    uavTimeStamp = currentTime.add(_hlaLookAhead);
} else {
    // For TAR, all uav-events with timestamp tau
    // that must be published after (hla current Time + lookahead)
    // We've made a choice for implementation of uav(tau):
    // option 1: (all or certain) tau is delayed to hlaNextPointInTime,
    // option 2: (all or certain)tau is delayed to currentTime+lookahead,
    // option 3: others may exist.

    // Here we take option 2: if tau <= hlaCurrentTime + lookahead.
    // tau is delayed to currentTime + lookahead.
    CertiLogicalTime certiCurrentTime = (CertiLogicalTime)
        _federateAmbassador.logicalTimeHLA;
    Time hlaCurrentTime = _convertToPtolemyTime(certiCurrentTime);
    if (hlaCurrentTime.add(_hlaLookAhead).compareTo(currentTime) > 0)
        uavTimeStamp = currentTime.add(_hlaLookAhead);
    else
        uavTimeStamp = currentTime;
}
CertiLogicalTime ct = _convertToCertiLogicalTime(uavTimeStamp);
if (_debugging) {
    _debug(this.getDisplayName() + " publish() -"
        + " send (UAV) updateAttributeValues "
        + " current Ptolemy Time=" + currentTime.getDoubleValue()
        + " HLA attribute \""
        + _getPortFromTab(tObj).getContainer().getName()
        + "\" (timestamp=" + ct.getTime() + ", value="
        + in.toString() + ")");
}
try {
    int id = _registeredObject.get(_federateName + " " + senderName);
    _rtia.updateAttributeValues(id, suppAttributes, tag, ct);
} catch (Exception e) {

```

```

        throw new IllegalArgumentException(this, e, e.getMessage());
    }
}

```

A.5 _putReflectedAttributesOnHlaSubscribers()

```

private void _putReflectedAttributesOnHlaSubscribers()
    throws IllegalArgumentException {
    // Reflected HLA attributes, e.g. updated values of HLA attributes
    // received by callbacks (from the RTI) from the whole HLA/CERTI
    // Federation, are store in the _subscribedValues queue (see
    // reflectAttributeValues() in PtolemyFederateAmbassadorInner class).

    Iterator<Entry<String, LinkedList<TimedEvent>>> it = _fromFederationEvents
        .entrySet().iterator();

    while (it.hasNext()) {
        Map.Entry<String, LinkedList<TimedEvent>> elt = it.next();

        //multiple events can occur at the same time
        LinkedList<TimedEvent> events = elt.getValue();
        while (events.size() > 0) {

            TimedEvent ravevent = events.get(0);

            // All rav-events received by HlaSubscriber actors, RAV(tau) with tau <
            // hlaCurrentTime
            // are put in the event queue with timestamp hlaCurrentTime
            if (_timeStepped) {
                ravevent.timeStamp = _getHlaCurrentTime();
            }

            // If any rav-event received by HlaSubscriber actors, RAV(tau) with tau
            // < ptolemy startTime
            // are put in the event queue with timestamp startTime
            //FIXME: Or should it be an exception because there is something wrong
            // with
            //the overall simulation ??
            if (ravevent.timeStamp.compareTo(_director.getModelStartTime()) < 0) {
                ravevent.timeStamp = _director.getModelStartTime();
            }

            // Get the HLA subscriber actor to which the event is destined to.
            String identity = elt.getKey();
            TypedIOPort tiop = _getPortFromTab(_hlaAttributesSubscribedTo
                .get(identity));
            HlaSubscriber hs = (HlaSubscriber) tiop.getContainer();

            hs.putReflectedHlaAttribute(ravevent);
            if (_debugging) {
                _debug(this.getDisplayName()
                    + " _putReflectedAttributesOnHlaSubscribers() - "

```

```

        + " put Event: " + ravevent.toString() + " in "
        + hs.getDisplayName());
    }
    events.removeFirst();
}
}
}

```

A.6 Other methods

```

/**
 * make a conversion from ptolemy time to certi logical time
 * @param pt ptolemy time
 * @return certi logical time
 */
private CertiLogicalTime _convertToCertiLogicalTime(Time pt) {
    return new CertiLogicalTime(pt.getDoubleValue() * _hlaTimeUnitValue);
}

/**
 * make a conversion from certi logical time to ptolemy time
 * @param ct certi logical time
 * @return ptolemy time
 * @throws IllegalArgumentException
 */
private Time _convertToPtolemyTime(CertiLogicalTime ct)
    throws IllegalArgumentException {
    return new Time(_director, ct.getTime() / _hlaTimeUnitValue);
}

/**
 * Get hlaNextPointInTime in HLA to advance to when TAR is used.
 * hlaNextPointInTime = hlaCurrentTime + Ts.
 * @return next point in time to advance to.
 * @throws IllegalArgumentException if hlaTimeStep is NULL.
 */
private Time _getHlaNextPointInTime() throws IllegalArgumentException {
    return _getHlaCurrentTime().add(_hlaTimeStep);
}

/**
 * Get the current time in HLA which is advanced after a TAG callback.
 * @return hla current time
 */
private Time _getHlaCurrentTime() throws IllegalArgumentException {
    CertiLogicalTime certiCurrentTime = (CertiLogicalTime)
        _federateAmbassador.logicalTimeHLA;
    return _convertToPtolemyTime(certiCurrentTime);
}

```

Bibliography

- [1] B. Bréholée. *Interconnexion de Simulations Distribuées HLA*. PhD thesis, École Nationale Supérieure de L'Aéronautique et de l'Espace, Mars 2005.
- [2] D. Côme. Improving hla-ptolemy cosimulation framework.
- [3] Defense Modeling and Simulation Office, 1901 N. Beauregard Street, Suite 504 Alexandria, VA 22311. *High Level Architecture Run-Time Infrastructure (RTI 1.3-Next Generation Programmer's Guide Version 3.2)*, September 2000.
- [4] C. P. Editor. *System Design, Modeling, and Simulation using Ptolemy II*. Ptolemy.org, 2014.
- [5] J. FORGET. *Architecture de Simulation Distribuée Temps Réel*. PhD thesis, Université de Toulouse, November 2009.
- [6] R. M. Fujimoto. Time management in the high level architecture.
- [7] R. M. Fujimoto and R. M. Weatherly. Time management in the dod high level architecture. 26:60–67.
- [8] G. Lasnier. Toward a distributed and deterministic framework to design cyber-physicalsystems.
- [9] G. Lasnier, J. Cardoso, P. Siron, C. Pagetti, and P. Derler. Distributed simulation of heterogeneous and real-time systems. In *Distributed Simulation and Real Time Applications (DS-RT)*, pages 55–62. IEEE.
- [10] U. of Berkeley. The ptolemy project. <http://ptolemy.eecs.berkeley.edu/ptolemyII/index.htm>.
- [11] ONERA. Certi. http://www.nongnu.org/certi/certi_doc/Install/html/intro.html.