

In this chapter, we want to provide you a detailed introduction to High Level Architecture (HLA) to lay the technical background for the rest of the chapters. HLA is both a de facto and de jure standard for distributed simulation. HLA is a simulation systems architecture framework for distributed simulation. Interoperability, along with reusability, is a major design aim of HLA. And it is commonly used in many application domains, specifically in the military realm. In this chapter, various aspects of the standard will be reviewed to furnish user with a comprehensive introduction. This chapter extends the third chapter of (Topçu et al. 2016).

2.1 Prelude

As we provided the historical perspective of distributed simulation in Chap. 1, HLA is the latest de facto and de jure standard for distributed simulations. The efforts began to realize a common technical framework for M&S through (initially) the defense domain, to enable the interoperability of various simulation systems by urging the stakeholders (e.g., the vendors, users, etc.) to a common architecture, which is called as High Level Architecture (DoD 1995). Today, it is commonly used in many application domains, in both military and civilian realm.

The evolution of HLA begins with the U.S. DoD HLA 1.3 specification, then to an IEEE standard in 2000 and then evolving once more in 2010 to its current version, which is called *HLA Evolved*, published in 2010 as IEEE 1516.2010 series (IEEE Std 1516-2010 2010; IEEE Std 1516.1-2010 2010; IEEE Std 1516.2-2010 2010). Unfortunately, the three versions of HLA specifications are not fully compatible with each other both in terms of software and framework, which increases the confusion of users as a result. In this regard, this book is generally based on the HLA Evolved version. Whenever we want to clarify the difference, we explicitly spell the version. For consistency throughout the book, we will refer to the HLA versions as presented in Table 2.1 to eliminate confusion.

Table 2.1 HLA versions

HLA specification	Referrant
The U.S. DoD HLA 1.3 specification	HLA 1.3
IEEE 1516-2000 standard	HLA 1516-2000
IEEE 1516-2010 standard	HLA 1516-2010 or HLA evolved

Today, the studies for the improvement of HLA actively continue spearheaded by Simulation Interoperability Standards Organization (SISO).

The major enhancements to IEEE 1516.2010 series were summarized as modular federation object models (FOMs) and simulation object models (SOMs), Web services communication, improved support for fault tolerance, smart update rate reduction, and dynamic link compatibility (DLC) (Möller et al. 2008).

2.1.1 What Is HLA?

Its standard defines HLA as a simulation systems architecture framework with the aim to facilitate the reuse and the interoperability of simulations (IEEE Std 1516-2010 2010). Cost-effectiveness, quality, and timeliness concerns necessitate reuse of assets not only in simulation domain, but also in all software-intensive domains. Systematic reuse, however, can only be achieved by assets that are designed for reuse. Component-based modularity with loose coupling of components at the heart of HLA is a key enabler for both interoperability and reusability. The interoperability can be defined as the capability of simulations to exchange information in a useful and meaningful way (Yilmaz 2007). The realization of interoperability requires a means of communication between the components.

As pointed out earlier, the motivation of HLA is to provide a common architecture for distributed simulation (Kuhl et al. 1999). Thus, the simulations, whatever their type might be (live, constructive, or virtual), interoperate on a single infrastructure to achieve the simulation objectives (e.g., to train a pilot in an engagement, or to produce data to input to a decision-making process). In a distributed environment, the infrastructure can be based on a direct communication (point-to-point) or an indirect communication (using a mediator) among components. In this regard, HLA adopts the latter approach, especially, to promote the reuse of components by decreasing the coupling of each component with others using a mediator, where *coupling* specifies the degree of the functional interdependency (in terms of communication) between components. This mediator is called *runtime infrastructure* (RTI), and its task is to provide services for the management of distributed simulation, components, and the data communication.

The HLA standard defines an architecture framework as “major functional elements, interfaces, and design rules, pertaining as feasible to all simulation applications and providing a common framework within which specific systems architecture can be defined” (IEEE Std 1516-2010 2010). While functional

elements specify the feature set that will be provided, interfaces define the way that user will consume them. Design rules introduce the practices to be employed to use these functions over the described interfaces to build up a system. This trio, namely, functional elements, interfaces, and rules, constitutes a common framework for developing simulations.

To wrap up, in a broad sense, the standard defines an architectural framework whose aim is to enable component-based, loosely coupled simulation development. The basic assumptions and motivations underneath this effort are summarized as follows (Dahmann et al. 1997).

- Diverse user requirements of today's simulation systems cannot be fulfilled by a single or monolithic structure. Thus, HLA supports decomposing large simulation problems into smaller parts.
- Today's simulations sweep a wide range of domains so that no single group of developers possesses all required knowledge to develop the whole simulation. Thus, HLA supports composing smaller parts into a big simulation system.
- Simulations can be used for more than one application some of which cannot be foreseen during development. Thus, HLA supports reusable simulations that can be composed into various simulation systems that have different requirements.
- Simulations have long life spans so that the technology that uses them is subject to changes. Thus, HLA provides an interface between the simulations and their users that insulates their use from the changing technology such as network protocols, operating systems, and programming languages.

As an historical fact, HLA was originated from the requirements of the defense modeling and simulation community. Early requirements for distributed simulations for collective training, and later, for aggregate simulations for analyzing battlefield situations led to a worldwide accepted simulation standard, HLA. Today, the user community of the standard spreads beyond defense applications. Since the standard was first published, there have been numerous distributed simulation applications in homeland security, space, aeronautics, disaster recovery, air traffic management, transportation systems, and medical domains that take advantage of it.

2.2 Basic Components

HLA is not software, but an architectural framework. It is a set of specifications, which is comprised of three parts:

- *HLA Framework and Rules* specifies the principles of systems design and introduces “a set of rules that must be followed to achieve proper interaction of simulations (federates) in a federation. These describe the responsibilities of simulations and of RTI in HLA federations” (IEEE Std 1516-2010 2010).

- *Interface Specification (IF)*: “The HLA Interface Specification defines the interface between the simulation and the software that will provide the network and simulation management services. RTI is the software that provides these services” (IEEE Std 1516.1-2010 2010). This standard specifies the capabilities of the software infrastructure of HLA, namely *RTI*.
- *Object Model Template (OMT)* presents the mechanism to specify the data model—the information produced and consumed by the elements of the distributed simulation. More formally “the OMT describes a common method for declaring the information that will be produced and communicated by each simulation participating in the distributed exercise” (IEEE Std 1516.2-2010 2010).

Before presenting each major standard in detail, let us begin with introducing some basic terminologies such as federate and federations and runtime infrastructure in this section.

2.2.1 Federate and Federation

A *federate application* is a (simulation) member application that conforms to the HLA standard and implements the interfaces specified in the HLA Federate Interface Specification to participate in a distributed simulation execution (IEEE Std 1516-2010 2010). When we refer to a simulation member as a piece of software, we call it a federate application, and, as an executable component capable of participating in a federation, we call it a *federate*. In order to interact with other federates (through the RTI) in the distributed simulation execution, a federate must join the simulation execution; then it is called as a *joined federate*.¹ A federate application may join the same execution multiple times or may join into multiple executions, creating a new joined federate each time.

It is worth noting that the standard is interested only in the interface of the federate, not how a federate application is structured. The motivation is that a legacy simulation application can be wrapped as a federate, thus, can participate in a federation. Simulations of systems or phenomena, simulation loggers, monitoring applications, gateways, and live entities all can be federate applications. Federate technically can be defined as a single connection to the RTI. So, we can identify it as a unit of reuse (IEEE Std 1516.2-2010 2010) and a member of a federation as depicted in Fig. 2.1. It can be a single process or can contain several processes running on more than one computer. It can be a data consumer, producer, or both. Best practices advocate designing a reusable set of simulation features as a federate. It can represent one platform such as a ship in an aggregate-level simulation or a frigate hydrodynamics model that can be a federate in a full mission training simulator.

¹Throughout the book, application is used as a short form for “federate application”, and federate is used as a short form for “joined federate”, unless otherwise indicated.

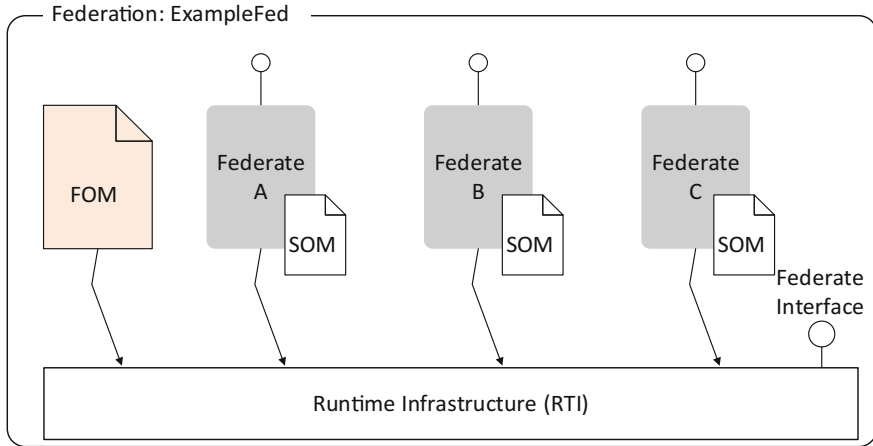


Fig. 2.1 Federation and federates

Federation is a simulation environment with a name, which is composed of the set of federates that share a common specification of data communication that are captured in *federation object model* (FOM) (i.e., *simulation data exchange model* in DS jargon) and interact via the RTI. As presented in Fig. 2.1, federates, whose data communication requirements are documented in their *simulation object models* (SOMs), are composed and interoperate over the runtime infrastructure throughout the federation execution. The information exchange requirements that include class relationships, data representations, parameters, and other relevant data of a federate are documented in the FOM. FOM is composed from the participating federate capabilities (documented as SOM).

Federation execution is a runtime instantiation of a federation; that is an actual simulation execution.

To exemplify the concepts introduced so far, consider a maritime simulation, which is composed of multiple ships, data loggers, and simulation control applications, where all share the same virtual environment and the notion of time. Here, the whole simulation is a federation with a common object model (i.e., FOM) shared by each participant. Each one of the participating members, e.g., a ship, a data logger, or a simulation controller, which is capable to participate in this federation, is a federate with its own object model (i.e., SOM). Assuming this federation is intended for analyzing the traffic management of a strait, then each simulation run is called as a federation execution.

2.2.2 Runtime Infrastructure

Most software architectures rely on infrastructures to enable their promises. HLA also comes with an infrastructure to enable interfederate communication. RTI is the

HLA's underlying software infrastructure, a *middleware*. Federates interact with RTI through the standard services and interfaces to participate in the distributed simulation and exchange data. RTI supports the HLA Rules with the services it provides over the interfaces specified in the Interface Specification (IEEE Std 1516.1-2010 [2010](#)).

The first public available RTI, called RTI 1.3, implements the HLA 1.3 specification and was released in 1998 (Kuhl et al. [1999](#)). Since then, there have been more than 20 RTIs produced as open-source, freeware, or commercial software. Some of the major commercial ones are pRTI™ from Pitch Technologies (Pitch [2013](#)), MÄK RTI ([2013](#)) from VT MÄK. And some popular open-source RTIs are Portico ([2013](#)), OpenRTI ([2011](#)), CERTI ([2002](#)), EODiSP HLA (PNP-Software [2007](#)), and Open HLA ([2016](#)).

2.3 HLA Rules

The principles that a distributed simulation system must adhere to be considered HLA compliant are specified in the standard (IEEE Std 1516-2010, [2010](#)). They are categorized under two headings, namely federation rules and federate rules.

2.3.1 Federation Rules

- Federation shall have an HLA FOM, documented in accordance with the HLA OMT.

The formalization of information exchange is one of the key points of HLA; it thus enables domain-independent interoperability. FOM is a major part of any federation agreement. So, any federation shall have a FOM, in which all the data (object and interaction) exchange that can happen during a federation execution is specified.

- In a federation, all simulation-associated object instance representation shall be in the federates, not in the RTI.

HLA aims to separate federate-specific (domain-specific) functionality from the support for general purpose (simulation) capabilities. So, it is the federates' responsibility to keep the copies of the object instance attribute values they are interested in. RTI does not provide a storage medium for shared data; rather, it provides a medium of transmission.

- During a federation execution, all exchange of FOM data among joined federates shall occur via the RTI.

To permit coherency in data exchange among the participants, federates shall utilize the RTI for data exchange as specified in the FOM. Then, the RTI can manage the execution and data exchange of the federation. Allowing a backdoor for communication would create hidden dependencies among federates, thus, hindering their reusability.

- During federation execution, joined federates shall interact with the RTI in accordance with the HLA Interface Specification.

Two-way interaction between the federate and the RTI shall conform to the federate interface specification. This specification is the base documentation for both the RTI implementers and the federate application developers. A federate uses this standard interface to employ the RTI services.

- During a federation execution, an instance attribute shall be owned by at most one joined federate at any given time.

To promote data integrity, only one federate can own instance of an object attribute at a time. Initially, the creator of an object is the owner of all its attributes. The ownership of an attribute confers the owner the right to update it. Transfer of ownership from one federate to another during execution is mediated by the RTI. Notice that ownership is at the attribute level; the attributes of the same object instance can be shared between different owners, thus allowing implementation of distributed objects.

2.3.2 Federate Rules

- Federates shall have an HLA SOM, documented in accordance with the HLA OMT.

Interoperability and reuse are only possible with an explicit specification of the capabilities and needs of the federates. This is the advertisement part. The object classes, class attributes, and interaction classes with their parameters shall be specified for every federate in its SOM. SOM is mainly used for documentation.

- Federates shall be able to update and/or reflect any instance attributes and send and/or receive interactions, as specified in their SOMs.

Federates can interact with others over updating or reflecting object instance attributes and sending or receiving interactions as specified in their SOMs. Thus, the reuse is enabled. This and the next two rules simply say “No false advertisement!”

- Federates shall be able to transfer and/or accept ownership of instance attributes dynamically during a federation execution, as specified in their SOMs.

As specified in the SOM, federates shall support transferring or accepting the ownership of object instance attributes during execution. This provides flexibility for federation designers in terms of the allocation of responsibility.

- Federates shall be able to vary the conditions (e.g., thresholds) under which they provide updates of instance attributes, as specified in their SOMs.

To take part in various federations, or cope with different phases of the same federation execution, a federate must be able to vary its object attribute update rates or interaction send rates, within the limits set forth in its SOM.

- Federates shall be able to manage local time in a way that will allow them to coordinate data exchange with other members of a federation.

Being a simulation on its own, a federate shall be able to manage its own time. Moreover, it must cooperate with the federation so that the RTI can maintain a notion of federation time. HLA supports federates with different time advancement mechanisms, such as time-driven or event-driven, via time management services. HLA also supports different time management strategies, such as conservative and optimistic, within a federation.

2.4 HLA Data Model and Data Communication

In the heart of distributed simulation is the simulation data exchange model which governs the data communication between simulation members. It is essential to understand how data are represented and shared during a federation execution.

In this respect, we introduce the HLA data communication pattern, the design time, and runtime data structures, and we present the HLA object exchange. Here, we answer what an HLA class is and then explain the related concepts, particularly, HLA OMT and the HLA object models. Chapters 4 and 5 demonstrate how to develop an HLA object model using SimGe tool.

2.4.1 HLA Data Communication Pattern

In a communication pattern, it is important to specify the policy of how data are exchanged among components. There are two important aspects of this specification. First, it is important to know what to exchange and how. For the latter question, in general, the participating member applications in a simulation environment can communicate directly in point-to-point manner (Fig. 2.2a) or they can employ a mediator (i.e., broker), part of a middleware, for communicating indirectly (Fig. 2.2b).

For the former question, what to exchange, there are two common approaches for data communication: *message exchange* (Fig. 2.3a) and *object exchange* (Fig. 2.3b).

In case of HLA, HLA specifies data communication via object exchange using a middleware (Fig. 2.4).

Now, we can describe how HLA exchanges objects among federates. The RTI plays the role of a mediator and routes the objects to the related federates. In the point-to-point communication, adopted by Distributed Interactive Simulation (DIS) protocol, the sender must know the receiver. In particular, the sender must know the network Internet Protocol (IP) address of the receiver. But, the middleware architecture model, which HLA adopts, uses a middleware to route the data among federates. Consequently, there is no need, in principle, for a federate to know about other federates. The RTI performs data routing using a *Publish/Subscribe Pattern*. In the following subsections, we will give the details of this pattern.

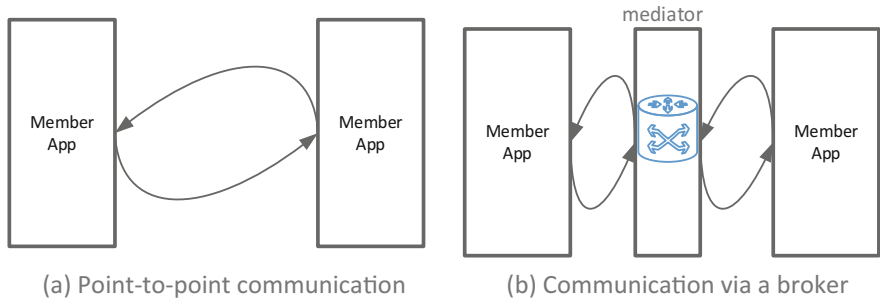


Fig. 2.2 How to exchange? **a** point-to-point **b** using a mediator. We can think of brokerage as a service provided by the middleware (in general terms)

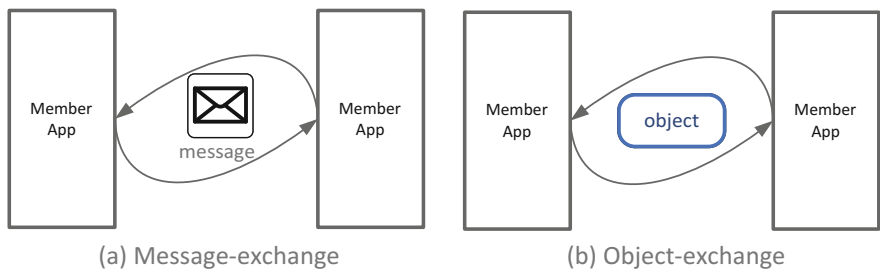


Fig. 2.3 What to exchange? **a** message **b** object

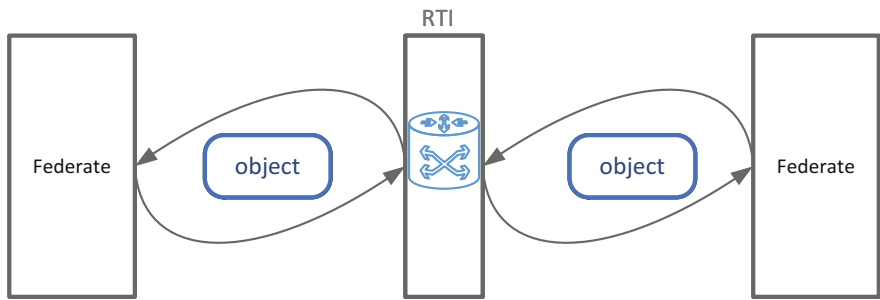


Fig. 2.4 HLA data communication pattern

The answer for what to exchange is that HLA communication model is based on object-exchange technology. In other words, HLA-compliant federates communicate with each other by exchanging objects. Therefore, the technology differs from classical DIS protocols, where data communication is based on message exchange technology, whereas data are exchanged through well-defined messages using

predefined protocol data units (PDUs). The structure of the exchanged data is embedded in DIS protocol. This causes the DIS protocol to be inflexible. For instance, to exchange an entity state, DIS protocol specifies an entity state PDU. Therefore, the simulation engineer can only use those predefined PDUs. You cannot create or define new data structures as all are specified with the standard. In contrast, HLA separates the data and its architecture. In this regard, HLA defines the structure of the data that will be exchanged by the help of a template of what to exchange at design time. This is done by employing the HLA Object Model Template. The simulation engineer can model new data structures, in terms of *HLA classes*, using the HLA OMT specification. The collection of those specified data structures is called an *HLA object model*. In the following subsections, we will expand both OMT and the object models.

2.4.2 Object Exchange: Publish/Subscribe Pattern

In this pattern, the sender and receiver components (i.e., federates) do not know each other. They just declare (to the RTI) what they need and what they can provide to the federation execution. In a federation execution, it is essential to express the relationship between a federate and particular federation objects. Therefore, a crucial federation design activity is to define the *Publish and Subscribe (P/S)* interests of federates with the objects of conceptual model at hand.

The Publish/Subscribe pattern forms the basis of the model of communication used by HLA between federates in terms of objects and interactions. *Publishing* means declaring willingness (and ability) to provide data, which is composed of object classes and their attributes, and interaction classes that the federate is able to update or send. *Subscribing* means declaring interest and the needs in receiving certain data. RTI dynamically routes the data from publishers (producers) to subscribers (consumers).

As shown Fig. 2.5, at runtime, federates can declare to the RTI, which plays the role of an object router, a set of data templates they can provide (i.e., publish), and a set of data templates they are ready to receive (i.e., subscribe) according to the *Federation Execution Details (FED)* for HLA 1.3 federations or *FOM Document Data (FDD)* for HLA 1516 federations, which both are derived from the FOM documented using the OMT specification. Following data declaration, federate can create an object (i.e., register) or can send an interaction, which it published. Afterward, the RTI finds the federates who subscribed to the class of the objects or interactions and then routes the object/interaction to the subscribers. So that, a subscriber federate can receive the interaction or discover the object. Updating the values of the object attributes works the same way. The publisher federate updates the value of an object attribute (i.e., update), and then the update is reflected to the subscriber federates (i.e., reflect).

Now, let us look at closely what an object, class, and an object model mean in HLA.

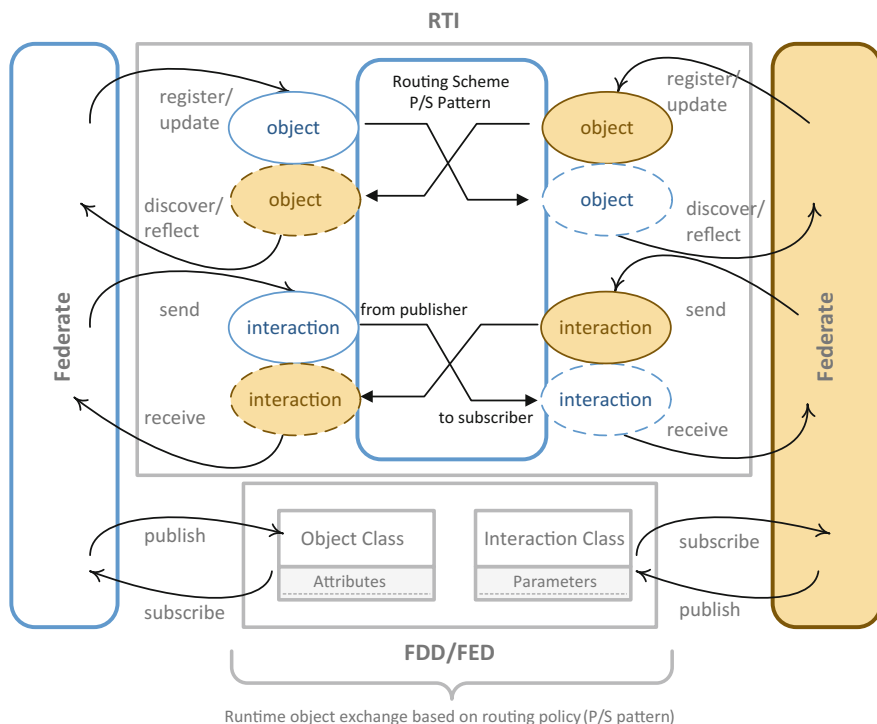


Fig. 2.5 Object exchange based on P/S pattern

2.4.3 Data: Objects, Interactions, and HLA Classes

2.4.3.1 Object Instances and Interactions

The *objects*, also known as *object instance* or *HLA object*, are the primary means of communication in a federation. They can be regarded as the abstractions of simulated entities. A simulated entity may have a lifetime that is as long as the simulation execution time span. And for sure, more than one federate can share an interest over that simulated entity. While one controls the entity, some others may only observe it. Good examples of HLA objects can be platforms or sensors in combat simulations or airplanes in air traffic simulations. Only joined federates can create or delete an object in a federation execution. The lifetime of an object is the duration between its creation and deletion. Only the federate that publishes the object class can create an object that is an instance of that class. Deletion of an object can be done only by the federate, which owns the privilege to delete. This is explained later in the ownership management section.

Object modelers identify the objects to facilitate an organizational scheme. There are *attributes* associated with an object. The values of the attributes determine the object state. The owner federate provides the attribute values by updating them, and

others (that are subscribed to those attributes) receive the values by reflecting those attributes. The position and velocity of a platform object can be examples of attributes.

Interactions, on the other hand, represent an occurrence or an event (analogous to events in the sense of discrete event simulation). So conceptually, they are not durable entities of interest but instantaneous events or occurrences of interest, such as sending of a message or landing of an aircraft. An interaction possesses a collection of data that is related to the occurrence or the event. The members of this data collection are called *parameters*. The parameters of an interaction are analogous to the attributes of an object. The difference is that the parameters of an interaction form a single indivisible group, while the attributes of an object can be grouped in different ways.

2.4.3.2 HLA Classes

Both objects and interactions are unique instantiations of HLA classes (see Fig. 2.6) because each object has a unique handle given by the RTI. *Handles* are the unique identifiers managed by the RTI. In classical sense, a class is “a description of a group of items with similar properties, common behavior, common relationships, and common semantics” (IEEE Std 1516.2-2010 2010). There are two types of HLA classes: *object class* and *interaction class*. An object class is “a template for a set of characteristics that is common to a group of object instances” (IEEE Std 1516.2-2010 2010), where an interaction class is “a template for a set of characteristics that is common to a group of interactions” (IEEE Std 1516.2-2010 2010).

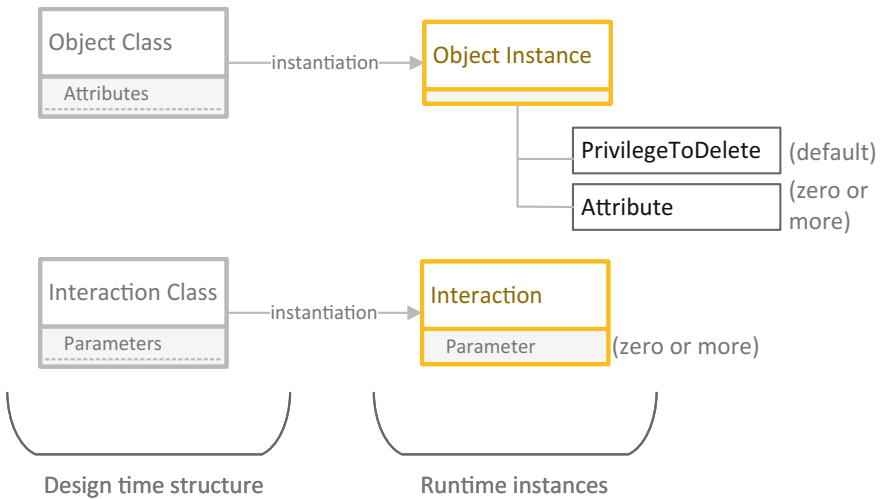


Fig. 2.6 HLA data structure. Each object has a default attribute called as *PrivilegeToDelete*. See Sect. 2.5.4 for the purpose of this attribute

An object class and an interaction class can be thought of as design time structures. They must be documented conforming to the OMT specification before federation execution as a part of the federation object model.

2.4.3.3 Comparison with Object-Oriented Paradigm

Many (computer/software) engineers take a course on object-oriented programming (OOP) in their undergraduate education, so they are tempted to compare and sometimes confuse it with the HLA object/class terminology. In the OOP literature, an object is defined as a software encapsulation of data (state) and behavior (methods) with an identity. In HLA, objects are defined by characteristics (i.e., attribute or parameter) that are exchanged between federates (the visible portion of state) during execution. Behavior is generated by the federates rather than directly by objects. HLA objects do not have methods.

The HLA standard also mentions this topic. It points out that while class objects are encapsulations for data and the operations, HLA objects are defined by the data that are exchanged between federates during federation execution (IEEE Std 1516-2010 2010). Of course, nothing prevents a federate application from implementing, say, a ship object internally in an object-oriented fashion, perhaps as a Java object, and exposing it to the federation as an HLA object instance via encapsulation. Encapsulation of an (OOP) object into an HLA object is explained in Chap. 7.

Moreover, attributes are treated as first-class structures in HLA, where attribute ownership is subject to change at runtime. A federate other than the one that created an object can update an attribute of the object. Thus, it is possible for an object to be distributed among multiple federates.

2.4.3.4 Class Hierarchy

For a complete object model, it is not sufficient to specify the object and interaction classes separately; there is a need to specify the relationships among them as well. HLA only allows class hierarchy using “is-a” relationship between classes, known as *single inheritance*. In OMT, there is a predefined root class for object classes, called `HLAobjectRoot`, and for interaction classes, called `HLAinteractionRoot`. A class that generalizes a set of properties that may be extended by more specialized classes is called *superclass* (base class in the OOP terminology), and a class extended from it is called a *subclass* (derived class). A subclass inherits all the properties of its superclass. For instance, `Ship` object class is a superclass for `CargoShip`, `RoRo`, and `Tanker` classes as depicted in Fig. 2.7. Here, `Tanker` class inherits the `Name` and `Location` attributes, and in addition, it declares a new attribute, `OilCapacity`.

As we go top-down (from root to leaves), the generalization decreases and specialization increases. Consequently, as the subclasses provide more concrete domain objects, the superclasses provide the classification of classes and reusability.

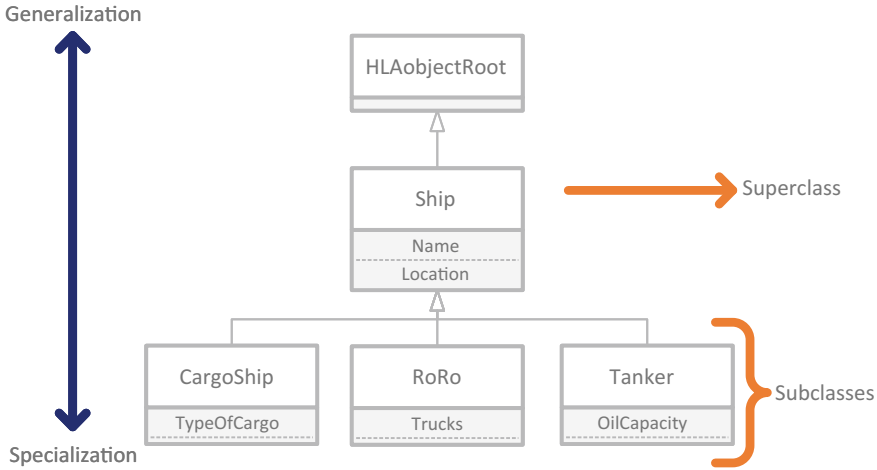


Fig. 2.7 Class hierarchy

2.4.4 Object Model Template

Object Model Template (OMT) is a metamodel (thought of as a template) both for specifying the data exchange and for providing a mechanism for the general coordination as a federation agreement (SISO FEAT 2017) among federates (in the form of FOM). It describes the structure of object models, as well as other relevant information such as synchronization points, via specifying the syntax and the format. Furthermore, the motivation behind developing a template for object models is presented in the standard is to provide an established mechanism for defining capabilities of the participants of the federation over their data exchange specifications (in the form of SOM) and to enable the development of common tool sets for object model development (IEEE Std 1516.2-2010 2010).

OMT is basically represented in tabular format and serialized in OMT data interchange format (DIF). As *federation agreements* involve more than data exchanged, it consists of a number of components in the form of tables, which can be listed as Object Model Identification Table, Object Class Structure Table, Interaction Class Structure Table, Attribute Table, Parameter Table, Dimension Table, Time Representation Table, User Supplied Tag Table, Synchronization Table, Transportation Type Table, Update Rate Table, Switches Table, Datatypes Table, Notes Table, Interface Specification Services Usage Table, and FOM/SOM lexicon. These tables are created for all federations and individual federates. While some require specifications from the designer, certain tables may be left empty depending on the situation.

While the reader is gently advised to go through the standard for the details of all the tables, we would like to introduce some of the important ones. First to mention is the *Object Model Identification Table*. The purpose of this table is to annotate the

object model with the information about how the federate or the federation has been developed. The information provided in this table includes version, modification date, purpose of the object model, its limitations, point of contact, and references. An example Object Model Identification Table, for our running example STMS, is given in Table 2.2.

In the *Object Class Structure Table*, the hierarchical relationship of the classes is specified. This table also indicates if a federate can publish, subscribe, or publish and subscribe these classes. A sample object class hierarchy is depicted in Table 2.3. In this example table, we can easily see the object class hierarchy. The

Table 2.2 Object Model Identification Table example for strait traffic monitoring simulation (STMS) federation


Category	Information
Name	StmsFom
Type	FOM
Version	2.0
Modification data	17/11/2016 12:00 AM
Security classification	Unclassified
Release restriction	NA
Purpose	A sample federation object model for this book
Application domain	HLA general
Description	This object model is provided as a sample project in SimGe object modeling tool
Use limitations	NA
Use history	Topçu et al. (2008, 2016)
<i>Keyword</i>	
Taxonomy	Simulation, maritime traffic management
Keyword	HLA, strait traffic management
<i>POC</i>	
POC Type	Sponsor
POC Name	Okan Topçu
POC organization	Okan Topçu
POC telephone	+1 (111) 111-1111 (fictitious)
POC e-mail	otot.support@outlook.com
<i>References</i>	
Type	Stand-alone
Identification	NA
Other	Created by SimGe at 11/14/2016 19:06:49
Glyph	

Table 2.3 Object Class Structure Table example for STMS federation

HLAobjectRoot (N)	Ship (N)	CargoShip (PS)	
		RoRo (PS)	ConRo (PS)
			RoLo (PS)
		Tanker (N)	GeneralPurposeTanker (PS)
			MediumRangeTanker (PS)
			LongRangeTanker (PS)
			VeryLargeCrudeTanker (PS)

Table 2.4 Parameter Table example for STMS federation

Interaction	Parameter	Data type	Available dimension	Transportation	Order
Radio message	Call sign	String	VHF	Hlabesteffort	Timestamp
	Message	String			

Ship object class, which acts as a superclass for the derived classes: CargoShip, RoRo, and Tanker. Moreover, we see that Ship is marked with N, indicating it is neither publishable or subscribable (which can be thought of an abstract class) while CargoShip is both publishable and subscribable (PS).

The *Interaction Class Structure Table*, likewise, consists of class–subclass relations of interaction classes as well as their publish/subscribe capabilities. The *Attribute Table* is used to specify the characteristics of object classes that are subject to change in the course of federation execution. They are updated by the RTI and made available to the related members of the federation. This table includes data type, update type such as periodical or conditional, if it is conditional, the update condition, ownership policy, publish/subscribe status, its dimensions, its transport method, and its order of delivery. The *Parameter Table* specifies the parameters that characterize the interaction classes (see Table 2.4 for an example). One must note that while the attributes can be published and subscribed on an individual basis, interaction parameters cannot be. So, although the Parameter Table looks like the Attribute Table, one should keep this difference in mind. Thus, a Parameter Table only possesses data type at the parameter level, while having dimensions, transportation, and order at the interaction class level.

All the OMT components are given in detail in Chap. 5.

2.4.5 HLA Object Models

The HLA standard specifies three types of object models. These are simulation object model (SOM), federation object model (FOM), and Management Object Model (MOM).

2.4.5.1 Federation Object Model

The participants of distributed simulation require a common understanding about the communication among themselves. FOM provides a standard way of defining a major part of federation agreements (SISO FEAT 2017). FOM mainly describes the format and the structure of data and events that can be exchanged among federates in a federation execution in form of objects and interactions with their attributes and parameters, respectively. Using FOM, designers can specify the data exchange in their federation in a standard format. There is one FOM per federation. So FOM can be regarded as an *information contract* that enables the interoperability among federates. FOM takes the form of a file at runtime, called FDD/FED file, which is supplied to the RTI in the federation execution. A new FOM can be developed from scratch for each federation as well as an existing reference FOM can be reused. The HLA Evolved standard also supports the modular federation object models (FOMs). The details of the FOM modularity are discussed in the following sections.

The reference FOMs are developed for increasing the interoperability by the simulation communities to agree on a common data model. For instance, real-time platform-level reference FOM (RPR-FOM) is developed to provide a preready reference FOM for real-time platform-level simulations targeting in general the pre-HLA simulations that use the DIS protocol (SISO 2015).

2.4.5.2 Simulation Object Model

With SOM, federates specify their capabilities and data interfaces. Thus, SOM serves as the specification for describing the capabilities of federates to promote reusability. You can determine the suitability of a federate for participation in a federation by examining its SOM. The HLA OMT format is also applicable to define SOMs.

2.4.5.3 Management Object Model

The HLA MOM is used to define the constructs for controlling and monitoring of a federation execution. Federates require insight about the federation execution as well as controlling the execution of individual federates, federation execution, or the RTI. MOM utilizes the Object Model Template format and syntax to define the information to be monitored and the control parameters. Its inclusion is compulsory for all FOMs. This inclusion can be accomplished by consolidating MOM data by HLA Standard *Management and Initialization Module* (MIM). MIM can be defined as a subset of FOM that contains the tables that describe the MOM. All FOMs have a default MIM that is specified by the standard, which can be overridden by a user-supplied MIM.

2.4.6 Object Model Modularity

The HLA Evolved standard brings support for object model modularity, so that SOM and FOM can be composed of one or more modules and one MIM. The previous HLA versions used a monolithic object model. In the HLA Evolved, the

modules are introduced as the partial object models that lay out a modular component to create more flexible and scalable object models (Möller et al. 2008). The major aim is to separate the local (custom) object models from the standardized object models (a.k.a. reference FOMs) such as RPR-FOM. Thus, as a design pattern, one may extend the standardized object model by introducing a partial and custom object model module, which is established upon a base stand-alone module and then inherit all object classes from these base classes. This capability makes the modules smaller and also promotes the reusability of object models for maintainable federations.

There are two types of FOM modules: *stand-alone module* and *dependent module*. A stand-alone module can be used without other FOM modules, so that it serves as a base object model that can be extended by other modules. A dependent module contains some references defined in another FOM module. A reference can point to a superclass (either an object class or an interaction class), a data type, a transportation type, a dimension, or a note defined in another module (Möller et al. 2007). Because of this dependency, the dependent modules cannot be used as a stand-alone FOM.

The stand-alone modules may contain references only to a MOM module. Therefore, a MOM module is always required for FOM modules but optional for SOM modules. One or more stand-alone modules together with a MOM module can be used to build a FOM unless the definitions of concepts do not conflict in the modules. Dependent modules can be built upon one or more stand-alone modules or other dependent modules. But a dependent module cannot be used without a stand-alone module.

The OMT Object Model Identification Table includes a type and identification pair in its references section. So, one may specify the type of FOM/SOM as stand-alone or dependent. If the type is dependency, then the identification field must include all the dependent FOM/SOM module names.

As the RTI uses a subset of data from FOM in the form of an FDD file, the file designators (i.e., the full name of the FDD file) are provided to the RTI in time of creating the federation execution and joining the federation execution. See Chaps. 8 and 11 for implementation details and for how to use multiple FOM modules.

2.5 Interface Specification

The HLA federate interface specification (IEEE Std 1516.1-2010 2010) defines the standard services and interfaces between the RTI and the federate applications to support interfederate communication. In other words, this specification provides a basis for functionally interfacing between a federate application and the RTI component. The functional interface is defined in terms of *RTI services*, which are arranged as seven groups:

- *Federation management* (FM) provides the services to create, control, and terminate a federation execution.
- *Declaration management* (DM) provides the services for federates to declare their intentions on publishing or subscribing object classes and sending and receiving interactions.
- *Object management* (OM) provides the services to register, modify, and delete object instances, and to send and receive interactions.
- *Ownership management* (OwM) provides the services to transfer ownership of attributes of object instances among the federates.
- *Time management* (TM) provides the services and the mechanisms to enable delivering messages in a timely manner.
- *Data distribution management* (DDM) provides the services to refine data requirements at the instance attribute level in terms of values, thus enables reducing unnecessary data traffic.
- *Support services* (SS) includes the utilities for federates such as name-to-handle, handle-to-name transformations, and getting update rate values.

The interface specification provides a description of the functionality of each service and the arguments (both the supplied and returned) and *preconditions* necessary for use of the service. *Post-conditions* specify any changes in the state of the federation execution resulting from the call. *Exceptions* give all exceptions that can be thrown by the service routine. The parts of interface specification are the following:

- Interface name and brief description of service;
- Supplied arguments;
- Returned arguments;
- Preconditions;
- Post-conditions;
- Exceptions;
- Related services.

Let us give an interface specification example for the *confirm synchronization point* service, which is an RTI-initiated service provided under the FM service group. This service is used to indicate the result of a federation synchronization point registration (see following section for a detailed explanation of federation synchronization). Its supplied arguments are a synchronization point label, registration-success indicator, and an optional failure reason. It has no defined returned arguments. The interface specification also defines its preconditions such that the federate needs to be joined to the federation execution and the joined federate has invoked *register federation synchronization point* service for the specified label argument. As a post-condition, it is specified what will be done in case of a positive registration-success indicator. And the standard specifies one exception for this service, which is *federate internal error*.

An RTI service defines an interface in a programming language independent way, where an *RTI method* is an implementation of that service using a particular programming language such as C++, Java, and C#. An RTI service can be mapped to more than one methods because of the different argument sets due to the optional arguments in the service specification or because of indicating a success or failure behavior of a service. For example, the “Confirm Synchronization Point” service, discussed above, can be mapped to two methods showing the result as a success or a failure. Below C++ method declarations are given:

```
// 4.12
virtual void synchronizationPointRegistrationSucceeded(
    std::wstring const & label)
    throw (rti1516e::FederateInternalError);

virtual void synchronizationPointRegistrationFailed(
    std::wstring const & label,
    rti1516e::SynchronizationPointFailureReason reason)
    throw (rti1516e::FederateInternalError);
```

In connection with the points mentioned, the RTI methods provided for a programming language constitute an *Application Programming Interface* (API) for interfederate communication. Consequently, each federate must interact with RTI by making method *calls*. The methods, which are provided to user federate applications, constitute the *federate interface*. The methods are grouped into (i) *federate-initiated* methods and (ii) *RTI-initiated* methods, to stress the direction of the communication. The RTI-initiated methods are also called as *callback* methods (see Fig. 2.8) *callback methods*. The Requests for calls should be included in *try-catch* blocks to catch the exceptions thrown so that appropriate action may be taken for error processing.

Typically, we make method calls when we want to instruct the RTI to do something. For example, the federate, `WhiteFdApp`, in Fig. 2.9, calls the method *Request Federation Save* to make the RTI to initiate a federate save. In response, the RTI initiates a federate save by informing each federate with a callback method “Initiate Federate Save.” Figure 2.9 depicts the federate interface and two-way communication represented as a Unified Modeling Language (UML) sequence diagram² (Fowler 2003). Here, all the interfederate communication is done by using this federate interface (i.e., using the methods and callbacks). In some specific RTI distributions (e.g., DMSO RTI 1.3 NG v6), a central process (e.g., `RtiExec`) is required to run RTI software. *RtiExec* (RTI Executive) is a global process, where each federate communicates with `RtiExec` to initialize its RTI components, whereas each `FedExec` manages a federation execution.

²Throughout the book, we will specify the interactions among federates and the RTI as a UML Sequence Diagram. In the diagram representations, we generally tend to use the UML diagrams as a sketch, informal and incomplete, not to lose the focus and simplicity with a strict formalism. So, the user without a deep UML background can follow the diagrams easily. The reader may refer to Fowler (2003), Larman (2004) for UML introduction.

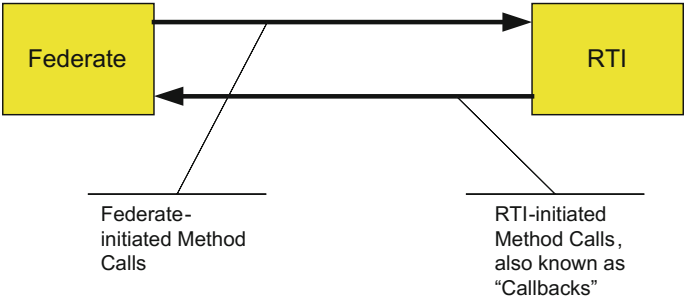


Fig. 2.8 RTI/federate-initiated methods

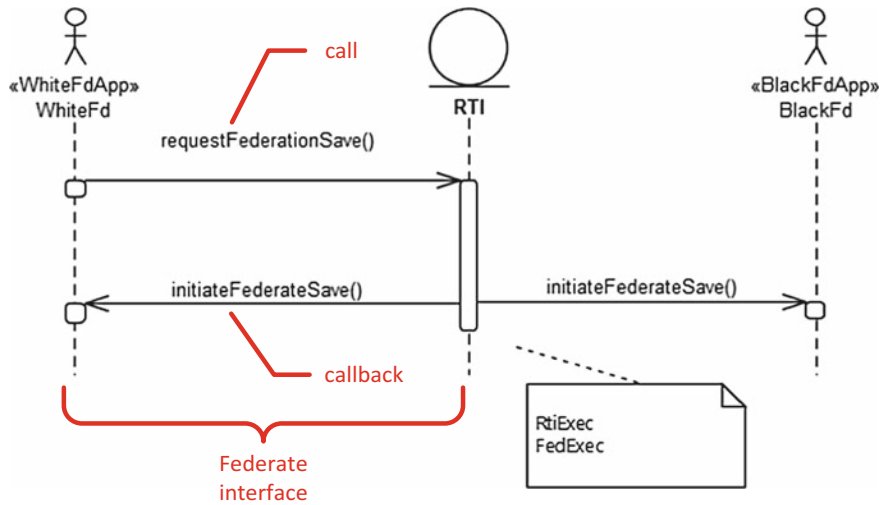
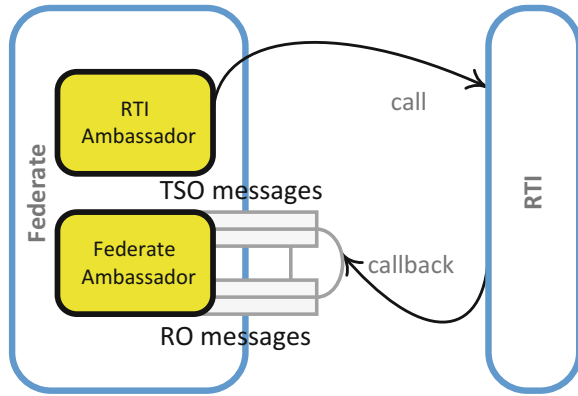


Fig. 2.9 The federate interface

The federate-initiated methods are called via a module, generally known as *RTI Ambassador*, and the callbacks are received by *Federate Ambassador* (see Fig. 2.10). The Federate Ambassador handles two types of incoming messages: *time-stamp-order (TSO) messages* to receive messages delivered in order of timestamp and *receive order (RO) messages* to receive messages delivered in order received. So, the federate must iterate through the queues to process the waiting callbacks. This operation is known as *ticking* or *evoking*.

Fig. 2.10 Ambassadors

2.5.1 Federation Management

Federation management involves the services to initialize, finalize, monitor, and dynamically control (e.g., saving and restoring) a federation execution.

2.5.1.1 Initialization and Finalization

Initialization services are used to connect to the RTI, to create a federation execution, and to join a federation execution. Before any interaction takes place between the un-joined federate and the RTI, a federate application must establish a connection. After a connection is established, federate can interact with the RTI to create a federation execution or to join a federation execution. During connect method, we specify the callback model that the federate prefers. Although the federate interface specification defines two types of callback model, which are *immediate callback model* or *evoked callback model*, a native RTI implementation may implement only one mode or both. In immediate callback model, the RTI shall invoke callbacks immediately. In the evoked mode, the federate must explicitly call a method (i.e., *evoke callback* or *evoke multiple callbacks*) to cause the callbacks to be executed. In HLA 1.3 specification, there is no specification for connect service. In this case, a federate can directly interact with the RTI after instantiating an RTI Ambassador and a Federate Ambassador. In this case, to invoke callbacks, the tick service is used.

Finalization services include resigning from the federation execution, destroying the federation execution, and lastly terminating the connection between the federate and the RTI.

2.5.1.2 Federation Execution Monitoring Services

Monitoring services are generally employed by applications to monitor the federation executions. So, a federate may request a list of federation executions that are currently running. The RTI replies this call with a list of federation execution

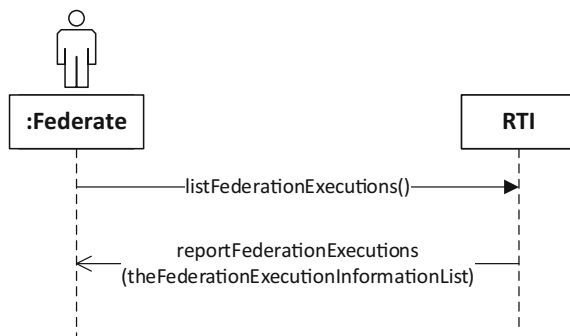


Fig. 2.11 UML sequence diagram for monitoring federation executions. Notice the use of synchronous (method calls by federate) and asynchronous (callbacks from the RTI) messages. Synchronous ones have *solid arrowheads*. The *open arrowheads* are for asynchronous messages

information. Each entry in the list encapsulates a pair of the federation execution name and the logical time implementation name. See Fig. 2.11 for interactions between the federate and the RTI.

2.5.1.3 Synchronization Services

Synchronization mechanism is provided to synchronize activities throughout the federation executions. A *synchronization point* is used to specify a synchronization activity. The synchronization points are declared in the OMT *Synchronization Table*.

Federation synchronization begins with a federate requesting a synchronization point registration (see Fig. 2.12 msg. 1). The RTI informs the requested federate whether registration is successful (msg. 2) or not (msg. 3). The reason for a failed synchronization point registration can be in case where a synchronization point label is not unique. For instance, assume that a federate has registered a synchronization point and as the synchronization process continues, another federate tries to register the same synchronization (i.e., with the same label) and so gets a failure. After a successful synchronization point registration, the RTI announces the synchronization point to the related federates (or all federates) according to the parameter selection on registration (msg. 4). When a synchronization point is announced by the RTI, each federate receiving this callback (msg. 5) replies with a *Synchronization Point Achieved* message after it successfully fulfills the synchronization point requirement. When all the related federates report achievement (either with success or failure), the RTI informs the related federates that the synchronization is completed and reports the failed federates (msg. 6).

See Chap. 5 on how to declare synchronization points in the HLA OMT and Chap. 9 how to implement the federation synchronization.

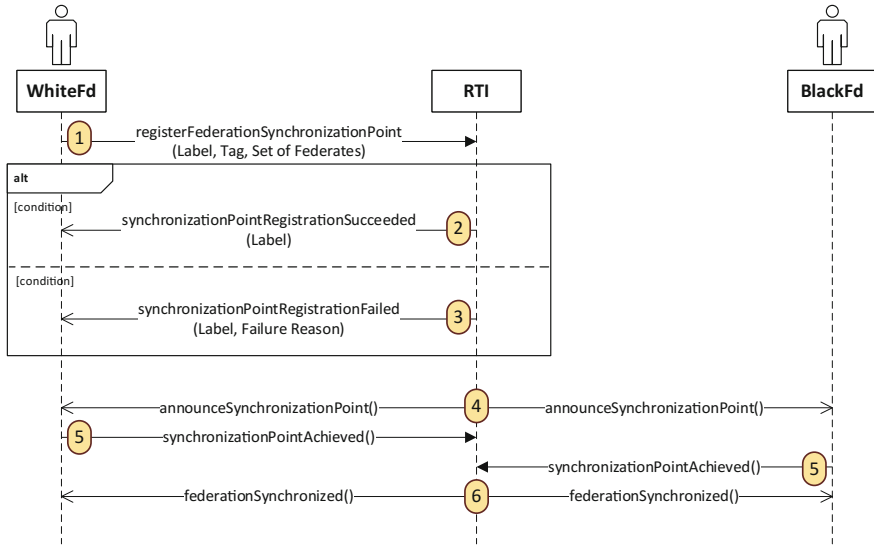


Fig. 2.12 UML sequence diagram for federation synchronization

2.5.1.4 Save/Restore Services

Another important federation management service area is to save a federation execution and then to restore it later to its state when the save is performed.

The interactions between federates and the RTI are depicted in Fig. 2.13. The federation execution save is initiated with a *Request Federation Save* message (Fig. 2.13 msg. 1). Here, the requesting federate provides a federation save label, which will be used in restore operation. The federate may also supply a timestamp to indicate the logical time for federation save. The timestamp can only be provided by a time-regulating federate (see time management section below for time-related concepts). When a timestamp is not present, then the RTI orders all joined federates to save state as soon as possible with a *initiate federate save service* (msg.2). Now, all joined federates are informed about a save operation. When a federate begins a save operation, it informs the RTI with a *federate save begun* call (msg.3). So, the federates may begin to save their application-specific data (whatever they want to keep to restore its state again) in a data store (a file or a database). When the federate completes its save operation, it informs the RTI by issuing a *Federate Save Complete* message (msg.4). Finally, the RTI informs the joined federates either the federation save operation is successfully completed or not by sending a *federation saved* message (msg.5). In case of a successful federation save indication, then the federates may shut down safely. Note that only one federation save can be in progress (IEEE Std 1516.1-2010 2010).

To support the federation save operation, there are some additional services provided by RTI interface (see Fig. 2.14). A federate may abort federation save

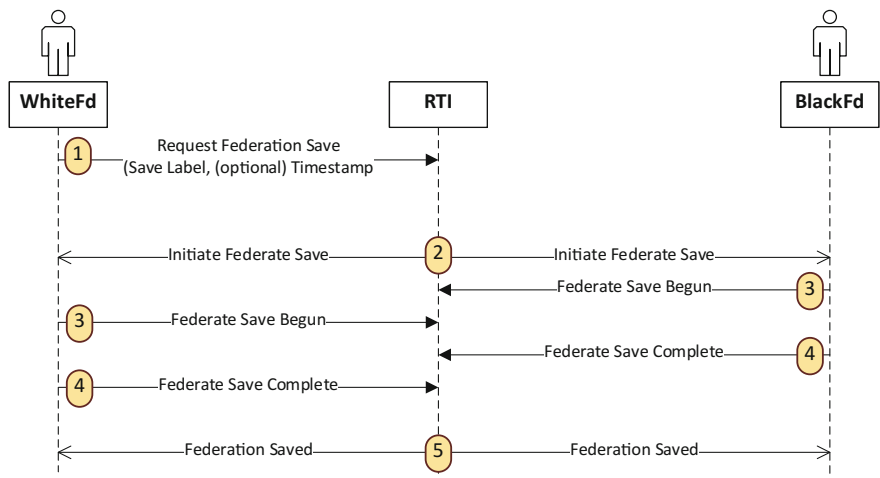


Fig. 2.13 UML sequence diagram for federation save

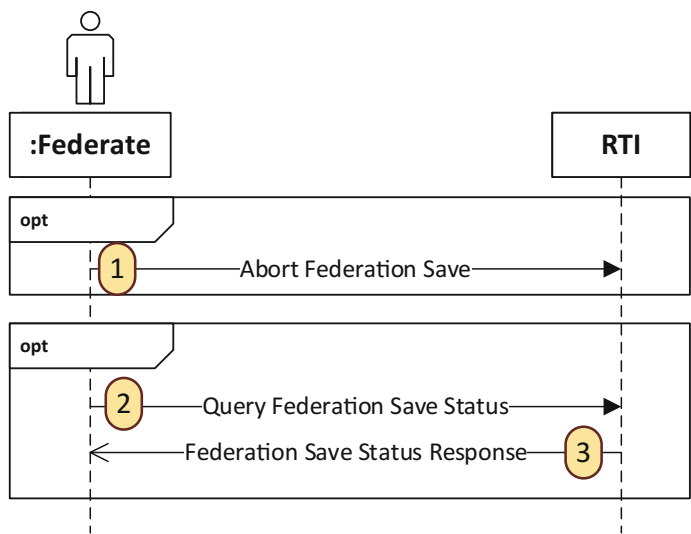


Fig. 2.14 UML sequence diagram presenting the additional services for federation save

(msg.1) or may query the federation save status (msg.2). When a query is requested, the RTI answers with a federation save status callback (msg.3). Note that these additional services are unavailable in HLA 1.3 specification.

The federation restore operation is similar to the federation save operation. The process begins with a *Request Federation Restore* service call (see Fig. 2.15, msg.1). This call must be supplied with a federation save label argument. A valid request is confirmed by the RTI using a confirm federation restoration request sent

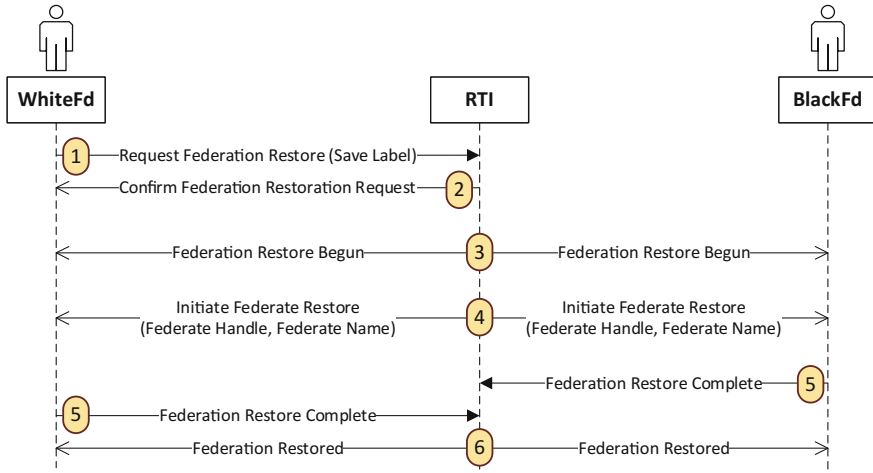


Fig. 2.15 UML sequence diagram for federation restore

to the requesting federate (msg.2). Then, the RTI informs all the federates that a federation restoration is about to happen by issuing a *Federation Restore Begun* callback (msg.3). Thus, the federates stop providing new information (e.g., updating attribute values) to the federation and wait for an initiate order. The RTI instructs the joined federate to initiate a federate restore (msg.4). The federate must return to its previous state indicated by the federation save label. Here, the federate loads its saved application data kept in the federation save and as a result of this service invocation, the federate's handle and name change according to the provided arguments by the RTI from the values taken by the join federation execution service. After federate completes its restoration, it informs the RTI by calling the *Federate Restore Complete* service (msg.5) and waits for the invocation of the federation restored service (msg.6).

Similar to the save operation, additional services exist for aborting and querying a federation restore.

2.5.2 Declaration Management

As pointed in the previous (HLA Data Model) section, HLA uses publish and subscribe mechanism to exchange objects at runtime. In this regard, the *declaration management* services provide all the required means for a federate to declare their capability and interests in a federation execution.

2.5.2.1 Functional Description

Before a federate involves in any data exchange during a federation execution, it must declare its capability (i.e., data it will produce) and its interests (i.e., data it will consume) to the RTI.

To declare capability, the federate uses the *Publish Object Class Attributes* and *Publish Interaction Class* services. Alternatively, a federate can change its declarations by using *Unpublish Object Class Attributes* or *Unpublish Interaction Class* services. To declare interest, the federate uses the corresponding *Subscribe/Unsubscribe* services such as *Subscribe Interaction Class* service.

Furthermore, there are additional services to help the federate fine tune when to start registration or begin sending interactions. Those services are listed below, and how to use them is shown in Chap. 8.

- **Start/Stop Registration For Object Class:** When a federate is interested with the data you can provide, the RTI informs you using this service. In other words, if a federate A subscribes to an object class, which is published by federate B then B is notified by the RTI that it can begin registration of the object instances related to that class. In the same manner, if there is no federate left subscribed to the object class, then the RTI informs the publishing federate to stop registrations. A federate loses its interest when it unsubscribes or leaves the federation execution.
- **Turn Interactions On/Off:** This works the same, but for interactions. When a subscriber is found, then the RTI informs the publishing federate to begin sending interactions.

The use of publish and subscribe services must conform to the FOM. For instance, an object class must be specified as publishable in the FOM. So, a federate can publish this object class.

2.5.2.2 Publish/Subscribe Diagrams

Publish and Subscribe Diagrams are introduced in (Topçu et al. 2003) as design artifacts to focus on the object/interaction interests among the federates. The P/S diagrams present the system in a snapshot view. Initially, all the federate interests are being hard coded at system start-up. During a run, however, a federate can change and re-declare its interests.

The general P/S diagram is a graph of Publish and Subscribe association stereotype elements connected by their various static relationships. It depicts all of the capabilities of member federates in terms of the objects and interactions that they produce or they are interested in.

The rectangular shapes in the diagrams represent the HLA objects, which are exchanged through federation. The oval shapes represent the (joined) federates. The directed arc from a federate to an object means that the federate has the capability to publish the object, and on the other hand a directed arc from an object to a federate means that the federate is subscribed to the object or is interested in the published object and its attributes or possibly subsets thereof.

HLA 1.3 standard allows *routing space* definitions to define the object-exchange regions by utilizing the RTI data distribution management capabilities. In that case, all the exchanges of objects take place in *Default Space* predefined by the RTI.

The diagrams can be expanded to reflect the system in detail by focusing on the different views: The Class-based P/S diagrams and the Federate-based P/S diagrams.

Class-Based P/S Diagrams

The *Class-based P/S diagrams* emphasize the P/S issues between a particular object and a federate. Class-based P/S diagrams show all the common HLA objects and interactions, and their parameters and attributes one by one while depicting the interests of federates over these objects and interactions.

This type of diagram is suitable for federate designers. For example, assume that a federate designer wishes to add a new federate into a federation. The first thing to do is to design the P/S interests with the existing federation objects and interactions.

Notation: As in the stereotype notation, the “P” stands for “publishes,” “S” stands for “subscribes,” and “PS” stands for “both publishes and subscribes.” It is possible for a federate to publish or to subscribe a subset of the available attributes for a given class. The sign “*” means “all attributes.” For interaction classes, it is not possible to specify a subset. Interactions are produced and consumed as a single piece.

Federate-Based P/S Diagrams

The *Federate-based P/S diagrams* depict the capability and interest of a particular federate. It can be used when reviewing existing federates for reuse. From these diagrams, the user can easily check which classes that the federate is capable of creating or needing.

2.5.2.3 Case Study: P/S Diagrams

Here, we provide the publish and subscribe interests of STMS federation. First, we will present the class-based P/S diagrams and then we will give the federate-based diagrams.

Ship object-class P/S diagram: Fig. 2.16 depicts the ship object class and its attributes. The focus of the diagram is a specific object class. The *Ship* object encapsulates the major properties of a ship that entered the strait. As shown Fig. 2.16, only the *ShipFd* can create and update a ship object and its attributes. *StationFd* is informed when a ship object is created or when its attributes are updated to reflect the changes in their traffic display. The *StationFd* only subscribes a subset of the attributes of the ship object. Those are the ship name (Callsign) and its location.

Station object-class P/S diagram: Fig. 2.17 depicts the *Station* object class and its attributes. *Station* object encapsulates the major properties of a traffic station located along the strait. The diagram shows that only the station federate is responsible to create and update a station object. The *ShipFd* is informed about traffic stations around them.

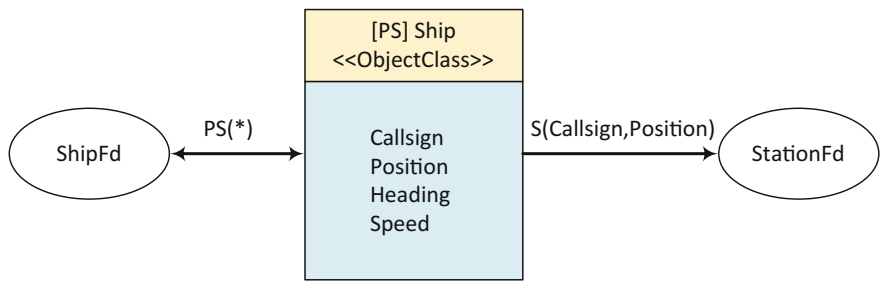


Fig. 2.16 Ship object class P/S diagram

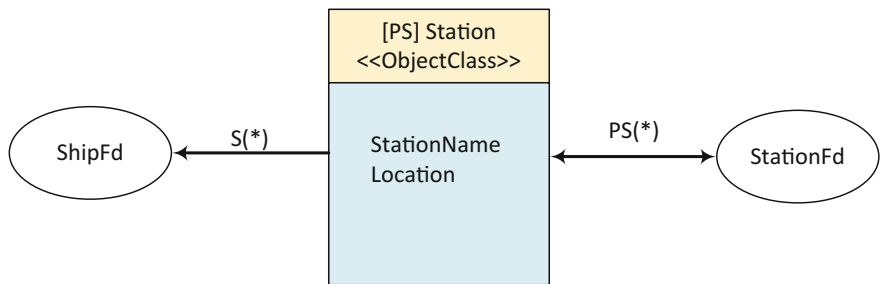
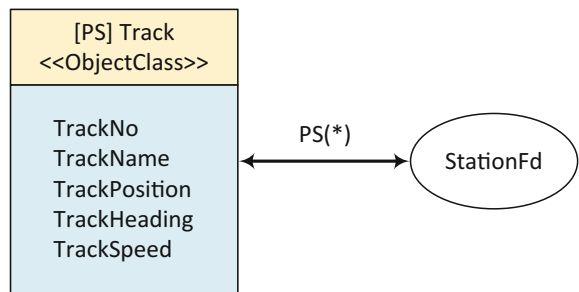


Fig. 2.17 Station object class P/S diagram

Fig. 2.18 Track object class P/S diagram



Track object-class P/S diagram: Fig. 2.18 depicts the Track object class and its attributes. Track object is the encapsulation of a ship which is found in the area of responsibility of a station, and it is used to exchange the track information between traffic stations. Each track is managed only by one station when a ship is within its area of responsibility. When ship passes to another area, which is managed by a different station, then track ownership is also exchanged between stations. Only the StationFd can create and update a track object and its attributes.

Radio message interaction-class P/S diagram: Fig. 2.19 depicts the RadioMessage interaction class and its parameters. Radio message interaction is the encapsulation of a radio transmission between ships and stations. It consists of

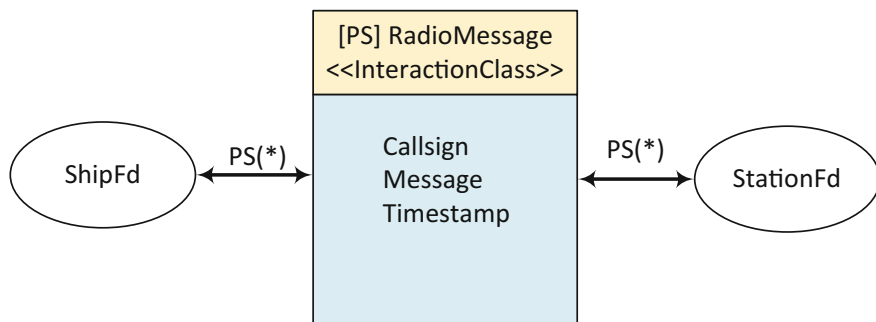
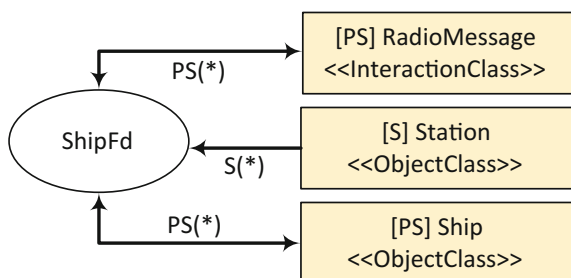


Fig. 2.19 Radio message interaction class P/S diagram

Fig. 2.20 ShipFd federate-based P/S diagram



three parameters: Callsign, Message, and Timestamp for the transmission. Remember that an interaction typically corresponds to an event in the sense of discrete event systems. The primary difference between objects and interactions is persistence: Objects persist, interactions do not.

ShipFd P/S diagram: Fig. 2.20 depicts all the P/S status of the ShipFd. Now, the focus of the diagram is the federate. We can easily deduce that the ShipFd is capable of generating ship objects and of sending radio message interactions while it requires a station object.

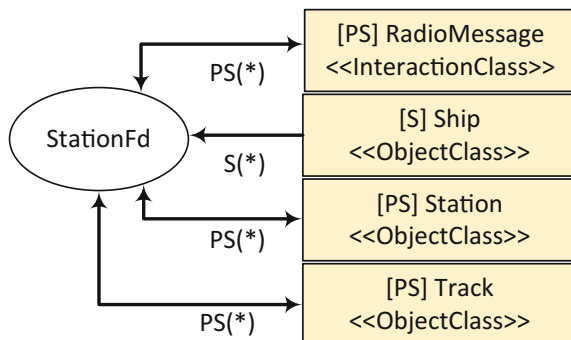
StationFd P/S diagram: Fig. 2.21 depicts all the P/S status of the StationFd. It is capable of generating station and track objects and of sending radio message interactions while it requires a ship object to generate tracks.

2.5.2.4 Smart Update Reduction

Support for *smart update reduction* Update reduction is a new feature in the HLA Evolved. An *update rate* is defined as the rate that attribute values are provided either (i) by the RTI to a subscribing federate or (ii) by the owning federate to the RTI (IEEE 1516.1-2010 2010). The update rates are defined by giving a name and a maximum rate in the FOM of a federation according to the OMT and it is a part of the FDD file used by the RTI at runtime.

The update rate can be specified as a supplied argument (as *update rate designator*) when subscribing to object class attributes. In this case, the RTI provides

Fig. 2.21 StationFd
federate-based P/S diagram



the value updates to the subscribing federate at a maximum update rate specified by the designator. In case an update rate designator is not supplied in the subscription step, then no update reduction takes place.

The update rate is also notified to the owning federate (i.e., the responsible federate, which will update the attribute values) by the RTI with *Turn Updates On for Object Instance* service (see the following section) as a returned argument (i.e., update rate designator). Thus, the owning federate can regulate its update rate with respect to the maximum rate provided by the designator conforming to the federation agreement for update scheme.

Using the RTI support services, a federate can ask the RTI about the actual update value of a specified update rate designator using *Get Update Rate Value* service or the update rate value of a specific attribute using *Get Update Rate Value for Attribute* service.

2.5.3 Object Management

In Sect. 2.4, we explained the object instances and object exchange. Now, let us see which services are used. The object exchange is done by using the *object management* (OM) services. A publishing federate must be capable of:

- Registering (i.e., creating) an object instance;
- Updating the values of the instance attributes;
- Sending an interaction;
- Deleting an object instance.

A subscribing federate must be capable of:

- Discovering an object instance;
- Reflecting the values of the instance attributes;
- Receiving an interaction;
- Removing an object instance (the RTI informs when an object is deleted by the owner).

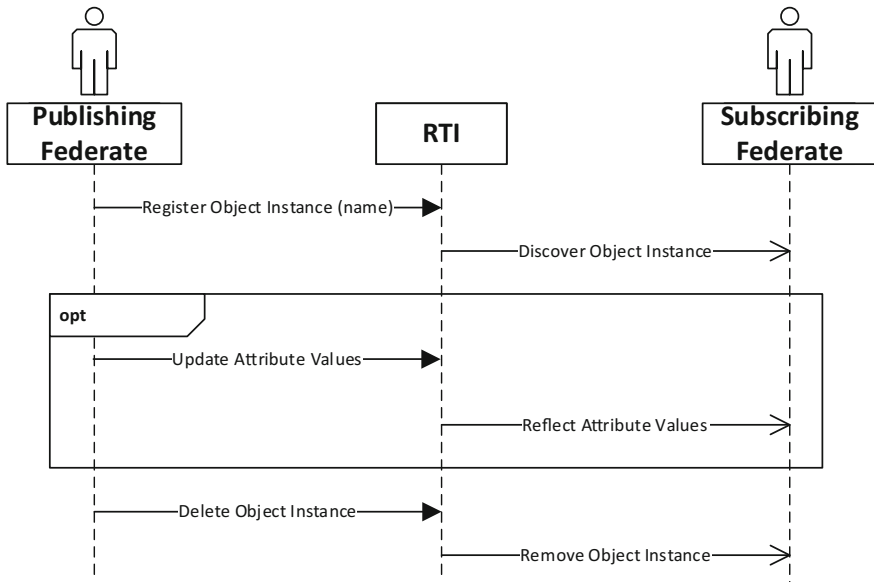


Fig. 2.22 Object management services for an object instance

The services for interaction exchange are simply *Send Interaction* and *Receive Interaction* services. The major object management services used in object-exchange interaction is depicted in Fig. 2.22. When registering an object instance, the federate may supply an *object instance name*. In this case, before invoking the registration service, the name must be reserved by the registering federate using *Reserve Object Instance Name* service.

As a design principle, for the sake of bandwidth utilization, a federation object or interaction is only sent through network when interested federates join the federation execution, instead of periodically sending a heartbeat-like message. For example, when EnviFd joins the federation, it only creates an environment object if there exists an interested federate (i.e., a federate that has subscribed to the environment object class) or not. If it exists, it registers the environment object. Again, the RTI informs the registering federate to turn updates on (or off) for the object instance. So, the federate can begin to update its attribute values, resulting in diminished network load. See Chap. 8 for the details and the implementation of *Start/Stop Registration for Object Instance* and *Attribute Relevance Advisory Switch* services.

2.5.4 Ownership Management

The responsibility of deleting an object instance and updating its attribute values is initially given to the creator federate. By using the *ownership management*

(OwM) services, it is possible to share the responsibility for updating and deleting object instances. To enable this, each attribute of an object instance has an owner. The owning federate has the privilege to update the attribute. Thus, the ownership of attributes of an object instance can be shared by more than one federate. To delete an object, a special attribute, named as `privilegeToDelete` for HLA 1.3 and `HLAprivilegeToDeleteObject` for HLA Evolved, exists for all object instances by default. The federate that owns this attribute for an object instance has the right to delete the object. If the federate owns the privilege-to-delete attribute of an object instance, then it has the delete responsibility for that object. The OwM services enable federates to transfer the ownership of an attribute including the ownership of the privilege-to-delete attribute.

The transferability of the ownership of an attribute is specified in the FOM during design time. Each attribute has a property to indicate if the ownership of an attribute of an object instance can be transferred to other federates or not. Transfer is possible by *divesting* or *acquiring* the ownership. See Chap. 5 on how to specify attribute ownership transferability in a FOM.

We can roughly talk about two strategies to deal with the ownership transfer: the pull and push strategies. In general case, the federates try to negotiate to hand over the ownership in both strategies. The conceptual views (without an RTI) for both strategies are depicted in Fig. 2.23.

In the *pull strategy*, a federate tries to acquire the ownership of some of the attributes of a specific object instance. The specified attributes can be either unowned (*orphaned*) or owned by some federate. In case of unowned attributes, the RTI simply informs the acquiring federate whether the ownership is transferred or not. This method is known as *orphaned-attribute pull*. In case the attributes are owned by a federate, then the willing federate and the owned federate must agree on transfer. The willing federate (acquiring federate) explicitly asks the RTI to acquire the ownership of the attributes of a specific object instance. The owned federate either accepts the transfer of the ownership or does not. This method is known as the *intrusive pull*.

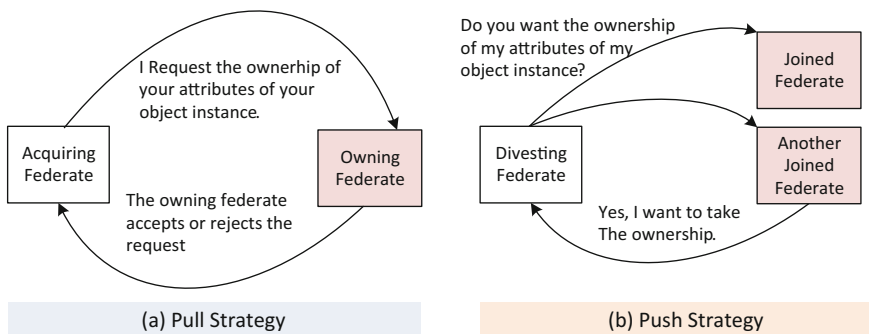


Fig. 2.23 Conceptual views for pull and push strategies

In the *push strategy*, a federate does not want the ownership of some attributes of a specific object instance that it owns. In this strategy, there are two methods to push the ownership. In the first method, the federate may unconditionally give up the ownership of some attributes to immediately relieve its responsibility of updating and/or deleting an object instance. This is called as *unconditional push*. The specified attributes become *unowned*. And the RTI seeks for a willing federate to own those attributes until attributes are owned. The second method is called *negotiated push*. In this case, there is a federate, which is willing to divest the ownership of all or some of its attributes of an object instance. The ownership transfer is completed as the result of negotiation when a willing federate is found. The willing federate follows a pull strategy to take the ownership. See Chap. 9 on how to implement these strategies.

2.5.5 Data Distribution Management

Data distribution management (DDM) services are employed to filter data distribution to reduce the volume of both the transmission and the reception of federation-wide data. Federates may create data filters to refine data transmission and reception in the form of distribution regions.

DDM services are related with other RTI services. Therefore, we can group DDM services into three categories. The first category includes core DDM services, used to manage the regions based on the dimensions defined in the FOM. The other categories involve the services related with the declaration management and object management services to transmit and receive data with regions. The regions are associated with an object class attribute or with an interaction class for subscription.

The core DDM services are the following:

- Create region;
- Commit region modifications, and
- Delete region.

The DDM services related with the declaration management services are as follows:

- Subscribe object class attributes with regions;
- Unsubscribe object class attributes with regions;
- Subscribe interaction class with regions;
- Unsubscribe interaction class with regions.

The DDM services related with the object management services are listed below. The regions are associated with an object instance attribute or with an interaction (in send operation).

- Register object instance with regions;
- Request attribute value update with regions;
- Send interaction with regions;
- Associate regions for updates, and
- Unassociate regions for updates.

The implementation of services is explained in Chap. 9 with some case studies. Here, we will provide a theoretical background for DDM services by giving definitions of important concepts, on which DDM services are based.

- A *dimension* is the fundamental DDM concept, which is defined as “a named interval of nonnegative integers” (IEEE Std 1516.1-2010 2010). The interval begins with zero to a dimension upper bound specified in the FOM (i.e., FDD). A federation object model may specify one or more dimensions so that each attribute or interaction can be associated with a set of available dimensions. See Chap. 5 on how to specify a dimension and how to associate an attribute or an interaction class with a set of dimensions in a FOM.
- A *range* is “a continuous half-open interval on a dimension” (IEEE Std 1516.1-2010 2010). The range has end points denoted by a lower bound and an upper bound. Since the interval is half-open, the lower bound is closed (where it is denoted by a square bracket and the value is included) and the upper bound is open (where it is denoted by a parenthesis and the value is excluded). According to the HLA Evolved specification, the upper bound must be greater than the lower bound and the minimum possible difference between them can be one.
- A *region* can be a one-dimensional region or multi-dimensional region based on the region specification.
- A *region specification* “is a set of ranges. The dimensions contained in a region specification shall be the dimensions of the ranges that are included in the region specification. A region specification shall contain at most one range for any given dimension.” (IEEE Std 1516.1-2010 2010). Each range is a region specification is normalized as zero to dimension’s upper bound. In specification, the upper bounds are not included.
- Note that dimensions are design entities while regions are runtime entities (i.e., they are created at runtime).
- A *region template* is “an incomplete region specification where one or more dimensions have not been assigned ranges.” (IEEE Std 1516.1-2010 2010).
- When a region specification is associated with the related element (i.e., object class attribute or interaction class for subscription, object instance attribute for update, and interaction for sending), then it is defined as a *region realization* (IEEE Std 1516.1-2010 2010).

The RTI provides a *default region* including a range beginning from zero to the dimension’s upper bound for each dimension specified in the FDD.

For a hypothetical example, let us assume that a FOM declares two dimensions called as X and Y. These dimensions define a two-dimensional coordinate system

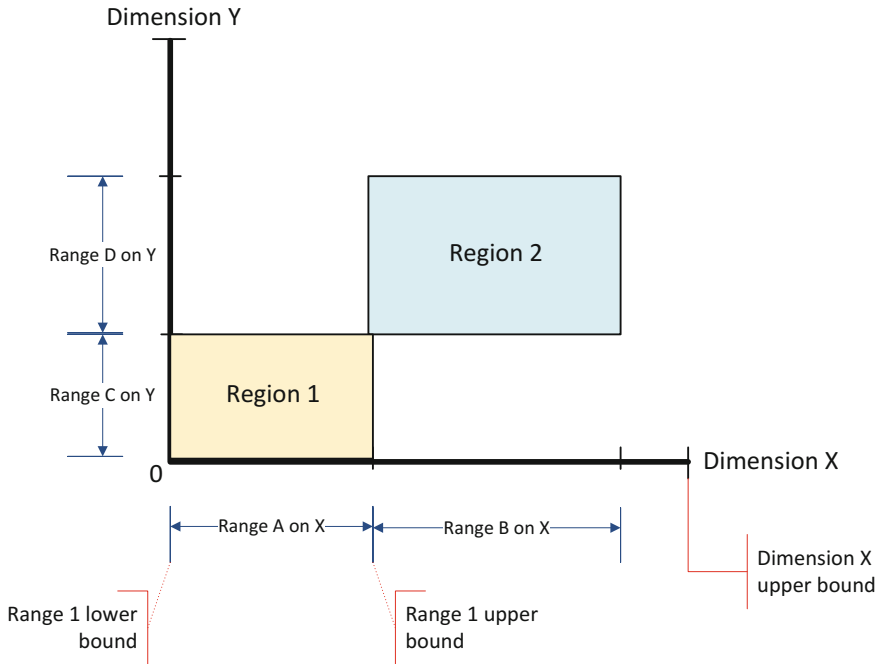


Fig. 2.24 Two-dimensional coordinate system and sample regions

(see Fig. 2.24), for which federates may express an intention to send or receive data by associating its attributes and interactions with the dimensions. To realize the intention, the regions must be created for subscription and update purposes. In Fig. 2.24, two regions are illustrated. Both regions are defined using the ranges specified on the dimensions. As shown in the figure, the ranges are specified by a lower and upper bounds on a specific dimension. For instance, Region 1 is specified by the Range A on X and Range C on Y.

2.5.6 Time Management

The main concern of services and mechanisms provided by *time management* (TM) is to coordinate the advance of (logical) time of each federate and to deliver messages in a consistent order during the federation execution. For this purpose, there are total 23 time management services, four of which are RTI-initiated call-backs (IEEE Std 1516.1-2010 2010).

2.5.6.1 Basics

Before giving the details of time management services, it would be useful reviewing some terms related to time issues in HLA and the RTI. As we introduced time and

change in a general sense in Chap. 1, time in HLA is regarded as a point in the *HLA time axis*, whereas the time will advance in intervals through these points. So, a federate can reference these points on the time axis to associate itself or some of its messages (IEEE Std 1516.1-2010 2010). *Messages* are events or some activities such as sending an interaction. Association of a federate with the HLA time axis in terms of discrete time creates a federate *logical time*. Associating the activities means that the federate assigns a *timestamp* (a designation of logical time) to the message with related to the HLA time axis. A major purpose of TM services is to enable sending and receiving TSO messages ordered with a timestamp.

Each federate is assigned a logical time in joining a federation execution, and then the joined federates may advance along the time axis during the federation execution. To advance time in the federation and to advance federates along the HLA time axis, we need *time advancement strategies*, also referred as *time management mechanisms*. Since the specific strategy used for time management is driven by the purpose of the simulation, it is important to control the advancement of federates along the time axis in light of differing internal time management mechanism implemented by the federate applications. So, another major purpose of the time management services is to control the advancements of federates. In general, a federate may advance its time using the following strategies (Fujimoto 1998):

- Time step advancement. Federate advances its time in fixed (time) steps.
- Event-based advancement. Federate advances its time to the timestamp of the next TSO message (event).
- Optimistic advancement. Federate is free to advance its time, but in case it receives a message with a timestamp less than its current logical time, then it roll backs its time.

The first two strategies are known as the *conservative time management* and the third strategy is known as the *optimistic time management* (Fujimoto 2000). The logical time advancement of a federate may be constrained by another federate. In general, there are four main federate types in relation to time as depicted in Table 2.5.

By default, all federates start out as “Neither.” During federation execution, federates can change their states dynamically. See Chap. 9 on how to implement the federate types related to time management.

All regulating federates (meaning both TR and TC&TR federates) are responsible to regulate advancement of (TC) federates along the time axis. To guarantee that a regulating federate would not send any TSO message during a time period, TR federates have a *lookahead* value. The lookahead value represents a duration of time (i.e., a time interval), whereas this value also can be modified during the execution. Only TR federates have the capability to send a TSO message with a timestamp. “Semantic differences exist between the way time is represented for the purpose of depicting timestamps versus calculating lookahead. When depicting timestamps, time can be considered to be an absolute value on the HLA time axis,

Table 2.5 Federate types per time management

Types	Explanation
Time-constrained (TC)	A constrained federate can receive TSO messages, but cannot send. Besides, their time advance is constrained by regulating federates
Time-regulating (TR)	A regulating federate can send a TSO message. It can still receive a TSO message but as an RO message. In addition, they can progress any time without any constraint. So, they regulate the federation time flow
Both TC and TR (TC and TR)	Federates that are both constrained and regulating have the union of the constrained and regulating federates' characteristics. In other words, they can send and receive TSO messages (receive with time information), can regulate federation time flow, and are constrained by other regulating federates
Neither TC and TR	Federates that are neither constrained nor regulating do not participate in federation time management. So, they have complete freedom in managing their own time advance. They cannot send TSO messages, but they can receive TSO messages without time information

and thus, time comparisons can be done to determine if one timestamp is greater than another. Lookahead, in contrast, represents a duration of time, which can be added to timestamps but is generally not used for comparison purposes.” (IEEE Std 1516.2-2010 [2010](#)).

TC federates can receive TSO messages, and they are volunteered to be regulated by a TR federate for its time advancement. The RTI guarantees that no additional TSO message with a timestamp less than the TC federate's current logical time will be delivered to the TC federate. To insure this, a bound is associated with the federate by the RTI, so the federate cannot advance beyond its bound. This bound is called as the *Greatest Available Logical Time* (GALT), which specifies the greatest logical time to which the RTI insures it can grant an advance without having to wait for other joined federates to advance. So, a constrained federate cannot advance beyond its GALT. In general, GALT is important for the TC federates to advance their logical time, but actually all federates have a GALT value, because they may want to switch into constrained and to set a new logical time. GALT is calculated by the RTI with regard to some parameters such as the logical time, lookahead, and requests made by TR federates to advance time. If there is no TR federate, then GALT is undefined, whereas a federate may advance its logical time to any point in the time axis.

Moreover, each federate has a *Least Incoming Time Stamp* (LITS), which specifies the smallest timestamp a federate will receive in the future.

For further details, the reader is gently advised to go through (IEEE Std 1516.1-2010 [2010](#); IEEE Std 1516.2-2010 [2010](#)) for HLA 1516-2010 and (DMSO [2002](#); Fujimoto [1998](#)) for HLA 1.3 standard.

2.5.6.2 HLA Services Related with Time

The messages that can be associated with time are summarized below. The messages correspond the activities in terms of RTI services.

- OM services
 - Update attribute value;
 - Reflect attribute values;
 - Send interaction;
 - Receive interaction;
 - Delete object instance;
 - Remove object instance.
- DDM services
 - Send interaction with regions

The message order type (i.e., either TSO message or RO message) for delivery must match the *preferred order type* of data specified in the FOM for each object class attribute and interaction parameter (see Chap. 5 on how to specify the message order type). A federate may change the preferred order type of an interaction or an attribute at runtime by using the *Change Interaction Order Type* and *Change Attribute Order Type* services, respectively.

The exact order type of a sent message and a received message is somewhat complex, and it is dependent on some conditions such as the preferred order type of data declared, whether a timestamp is supplied or not, and whether the federate is regulating or constrained. The details are presented in (IEEE Std 1516.1-2010 2010). To summarize:

- The order type of a sent message is TSO if and only if preferred order type is TSO, the sending federate is regulating, and a timestamp is supplied with the message.
- The order type of a received message is TSO if and only if order type of sent message is TSO, and the receiving federate is time-constrained.

2.5.6.3 Advancing Time

Each federate is assigned a logical time upon joining a federation execution and then the joined federates may only advance along the time axis during the execution with *Time Advance Grant* after invoking the TM services such as *Time Advance Request* or *Next Message Request* (the full list is displayed in Fig. 2.25) according to their time advancement strategies. See Chap. 9 on how to achieve advancing time.

2.5.6.4 HLA 1.3

In HLA 1.3 specification, the TM jargon differs. See Table 2.6 for a map. The keywords change in service naming. For example, *Query “Logical” Time* service corresponds to *Query “Federate” Time* service in HLA 1.3 or *Next “Message” Request Available* service corresponds to *Next “Event” Request Available* service in HLA 1.3. For further details of time management in HLA 1.3, see (DMSO 2002).

Fig. 2.25 UML sequence diagram for advancing time

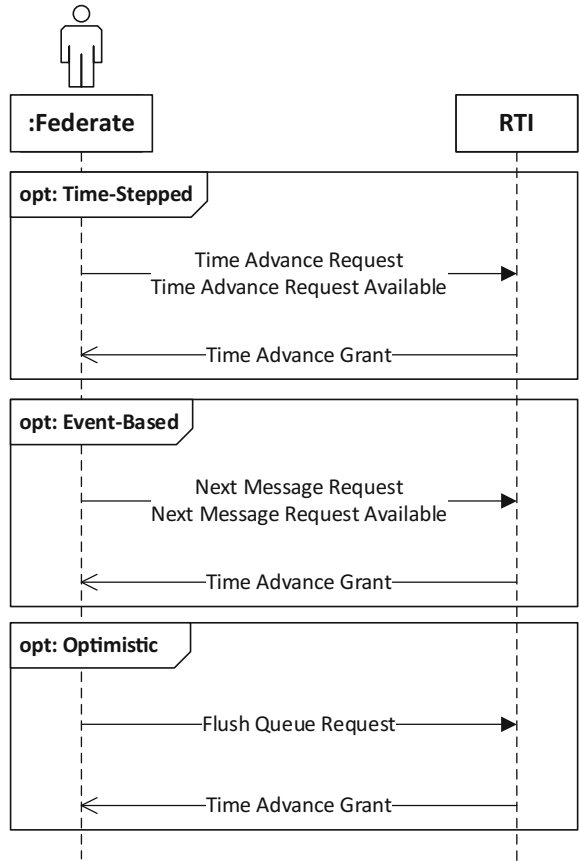


Table 2.6 HLA1.3 and IEEE 1516-2010 TM Jargon

IEEE 1516-2010 keywords	HLA 1.3 keywords
Logical time	Federate time
Message	Event
GALT	LBTS (lower bound timestamp)
LITS	Min. next event time

2.6 Full Life Cycle of a Federation Execution

For introduction, here we describe the full life cycle of a federation from the perspective of a federate. So, we will know what kinds of services a federate and a federation needs. A typical federation execution life cycle starts with the connection to the RTI as depicted in Fig. 2.26, and then:

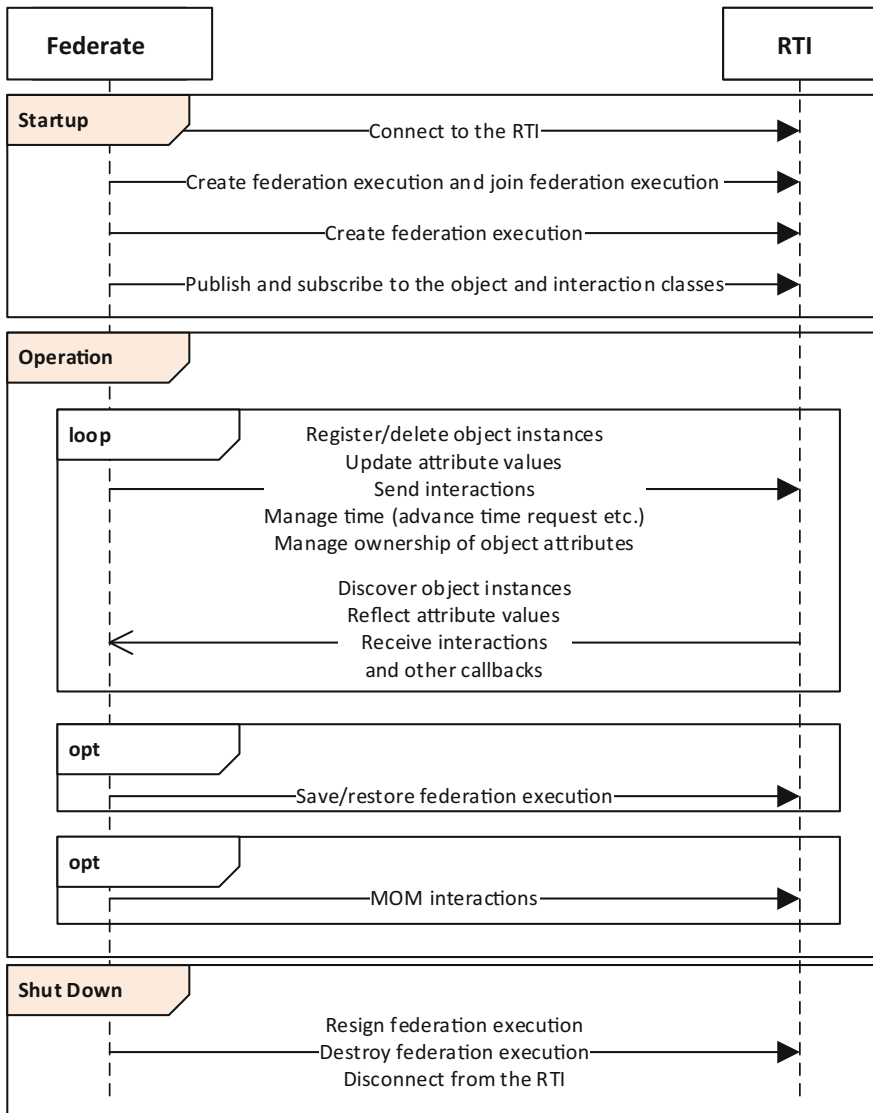


Fig. 2.26 Typical federation execution from the perspective of a federate

- Federate application connects to the RTI to use federate services and interact with the RTI.
- Federate first tries to create the federation execution if not created till then and then joins the federation execution. After joining the federation execution, a joined federate instance in the RTI is created that represents the federate in the federation execution.

- Federate should inform the RTI about its capabilities and interests by publishing and subscribing the object classes and interaction classes. Thus, it establishes its initial data requirements.
- Federate registers (creates) objects that it will provide to other federates.
- Federate may register new objects or update the values of the instance attributes that it registered; may discover new objects, that are created by other federates; may receive updates for the subscribed attributes; and may send and receive interactions.
- Federate deletes objects, which it holds the privilege to delete (generally the objects that created by the federate itself) before leaving.
- Federate manages its time according using RTI time management services (e.g., Time Advance Request), if it specifies a time management policy (e.g., time-regulating federate).
- Federate manages ownership of attributes, if necessary.
- Federate resigns and tries to destroy the federation execution and succeeds if it happens to be the last federate.
- Federate disconnects from RTI.

The order is important. For example, you cannot register an object instance before publishing the related object class. Or, you cannot join the federation execution before connecting to the RTI.

This typical federation execution life cycle affects the design of federates. With no surprise, basic program flow of federates is divided into three phases: system initialization (start-up), main application loop (operation), and system termination (shut down) (Fig. 2.27).

System initialization and termination phases include the RTI *initialization* and *termination* phases, which involve some federation-wide principles. Generally, there are two federation management models: *centralized* and *non-centralized* models. In the centralized model, a specific federate is responsible for the initialization and termination of the federation execution. In non-centralized models, each federate has the equal responsibility for initialization and termination. Initialization and termination phases also include the initialization and termination activities for the scenario play-out, respectively.



Fig. 2.27 Basic program flow of a typical federate

2.6.1 Initialization

HLA does not mandate the creation of a federation execution to the privilege of a particular federate. This policy provides flexibility and non-centralization. One may design that the first job of any one of the federate applications is to try to create a federation execution. The first federate succeeds to create the federation execution if the specified federation execution does not exist, while subsequent federates receive an exception, which indicates that the federation execution does already exist and then they directly join the federation.

In some RTI releases, if joining the federation execution is attempted immediately after the creation of the federation execution, the federation execution may not yet be initialized to communicate with the federate (e.g., the `Fedexec` process is not forked and initialized in case of HLA 1.3). Beforehand, we cannot assume which federate is the first, so the join logic will loop until the join is successful or until some predetermined number of join attempts are exhausted.

The creation of a federation execution requires a *federation name*. It designates a unique federation execution, and the participating federates use it to join into the specified federation execution. All member federates should agree on the unique federation execution name. Therefore, the federation execution name either should be distributed by hand to all participants at start-up or the federation execution name should be hard coded in federates.

2.6.2 Operation

The operation phase generally includes the *main simulation loop* and an alternative behavior path. The main simulation loop specifies the behavior of the federate for the normal federate execution, which includes the object management, time management, and the ownership management services, while the alternative behavior path is used for abnormal situations such as when save and restore is requested in the federation execution or when MOM interactions are required. The main simulation loop is elaborated in Chap. 8, and, furthermore, it is extended for a graphics-intensive federate application in Chap. 11.

2.6.3 Termination

The shutdown/termination of federation execution is accomplished by the federate that resigns from the federation execution last. In a non-centralized simulation, the same rule applies here; all federates, while resigning, attempt to terminate the federation execution. The last one succeeds while others receive an exception because the federation still has members and resign from the federation without terminating it.

The termination phase consists of three stages (Fig. 2.28): RTI termination, local model termination, and graphics termination. At *RTI termination* stage, the created

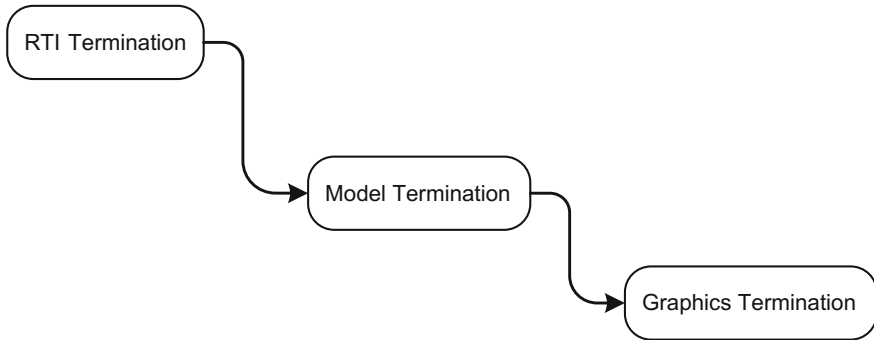


Fig. 2.28 Termination of a federation (Topçu et al. 2016)

objects are deleted and other federates are informed, and then the federate resigns and tries to destroy the federation. At *model termination* stage, the local objects that represent the simulation entities are deleted to free up the application memory, and finally at *graphics termination* stage, the graphics subsystem is shut down.

2.7 Example Federation Deployment

UML deployment diagrams are used to plan and design the execution environment in software-intensive systems by depicting the hardware components and software components. In this context, the specialized and extended form of deployment diagrams can be employed to capture the execution details of federation requirements such as *node* (i.e., hardware component) information (e.g., physical location, IP address, port number, operating system, etc.), *network information* (e.g., network type, bandwidth, etc.), and which federates (i.e., software component) hosted by which nodes (Topçu et al. 2003; Topçu and Oğuztüzün 2005).

An example UML deployment diagram for an HLA federation execution is presented in Fig. 2.29. Here, we can see that there are five hosts, where four of them are distributed in a TCP/IP network and one (Node 4) is a Web client connecting from Internet. The diagram shows us which federate is executed on which host and gives some information about the host (e.g., the host's operating system).

2.8 Federation and Federate States

During a federation run, federates and the federation execution can be found in specific states from the viewpoint of the RTI. Those states are useful to define the context of federates and the federation execution during simulation run.

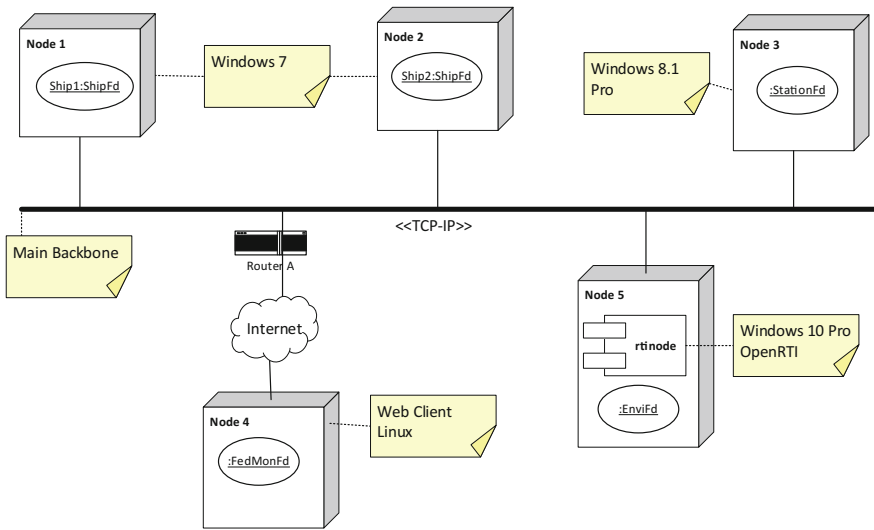


Fig. 2.29 Typical federation execution deployment. The federate applications can be deployed to execute on heterogeneous environments such as Linux/Windows or various versions of Windows

2.8.1 Federation Execution States

Federation execution states are depicted in Fig. 2.30. Initially, the federation execution is in Federation Execution Does Not Exist state when no federation execution exists (either it is not created yet or is destroyed). The directed links show the events that trigger the transition from one event to another event. After the federation execution is created and running, the federation execution state transits to Federation Execution Exists state. This state is a *composite state* that encapsulates two substates: No Joined Federates and Supporting Joined Federates. The substates are not essential from the viewpoint of federate developer. Therefore, we do not go into the details further. The RTI implementers may refer to (IEEE Std 1516.1-2010 2010) for details.

2.8.2 Federate States

A federate is either connected or not connected according to its connection with RTI. Figure 2.31 depicts the basic state diagram from a federation management perspective. The Connected state is a composite state including the states where a federate is joined the federation execution (Joined state) or not (Not Joined state). The joined federate state also includes some substates such as active federate state, federate save in progress state, and federate restore in progress state from the perspective of the RTI (IEEE Std 1516.1-2010 2010).

consists of three main components: the HLA Rules, Interface Specification, and the Object Model Template. The rules specify some policy that federates and federation executions must conform. The interface specification presents a standard interface for the interaction between a federate and the RTI. The OMT specifies the standard for the HLA object models, namely FOM, SOM, and MIM. HLA uses an object-exchange mechanism employing the RTI as the mediator between publishers and subscribers. The RTI library also provides an API with programming language bindings such as Java and C++ for federation developers exposing all the federate service areas; federation management, declaration management, object management, data distribution management, time management, and ownership management. A typical life cycle of a federation execution from the perspective of a federate involves initialization, operation, and termination phases. In the initialization phase, a federate joins the federation execution and then declares its capability and interests to the RTI. During the operation phase, which involves the main simulation loop, the federate registers its objects, updates the attribute values, and sends interactions as it can discover new objects, receive interactions, and reflect the attribute value updates. The federate may also transfer the ownership of some of attributes of its objects to another federates. Of course, time is an important aspect of simulation. Therefore, a federate can interact with the RTI to manage its time policy. In the termination phase, the federate resigns from the federation execution.

As it became a widely accepted standard in the area of distributed modeling and simulation over the last decade, and we see that many new distributed simulation applications in both the civilian and, to a larger extent military realm are being built to be HLA compliant. In this regard, this chapter lays the technical background to develop HLA-compliant distributed simulations.

2.10 Questions for Review

1. Is it a requirement that all the federates use the time management services?
2. Explain the FOM Modularity in HLA Evolved. Show the extension and union techniques in sample FOM modules.
3. What is WSDL? Explain the relation of it with HLA.
4. What is DLC API?
5. Explain the “Connect and List Services” property in HLA Evolved.
6. Discuss the differences of HLA federation management services specified in HLA 1.3 and HLA Evolved standards.
7. Discuss the differences of HLA time management services specified in HLA 1.3 and HLA Evolved standards.

Table 2.7 Time data for federation execution

Federate	Time management scheme	Federation time t_{current}	Lookahead $t_{\text{lookahead}}$	Time step
Fd-1	R	17	10	5
Fd-2	RC	N/A (late joining)	–	–
Fd-3	RC	16	4	4
Fd-4	C	18	–	4
Fd-5	RC	16	5	5
Fd-6	N	0	–	6

8. Discuss the differences of HLA data distribution management services specified in HLA 1.3 and HLA Evolved standards.
9. Discuss the pros and cons of a wrapper API around RTI. In this respect, discuss RACoN API.
10. Explain the class-based filtering and value-based filtering in HLA by giving examples.
11. What are the major components of HLA? Explain each shortly.
12. Table 2.7 presents the time information of federates in a federation execution. In this regard, draw a time diagram for each federate and then:
 - a. Show LBTS value for each federate and state whether the federate will proceed or not.
 - b. Assume that the federate Fd-2 joins the federation execution at time $t = 20$. Calculate the LBTS value for Fd-2.
13. Answer the below questions regarding to the two-dimensional space depicted in Fig. 2.24.
 - a. Draw the region specified by the range A on the dimension X and the range D on the dimension Y.
 - b. Draw the region specified by the range B on the dimension X and the range C on the dimension D.
 - c. If we want to cover all the distribution space (in other words, if we want to create a single region equal to the default region), what should we do?
14. Why is it desirable to have larger lookahead values? Consider time-driven and event-driven federates separately.
15. It is often remarked that HLA compliance, by itself, does not guarantee interoperability of simulations. What else is required on the part of developers?

References

- CERTI. (2002). *CERTI*. <http://savannah.nongnu.org/projects/certi>. Accessed April 08, 2017.
- Dahmann, J. S., Fujimoto, R. M., & Weatherly, R. M. (1997). *The department of defense high level architecture* (pp. 142–149). Atlanta, GA: IEEE.
- DMSO. (2002). *High level architecture run-time infrastructure RTI 1.3-next generation programmer's guide version 6*, s.l. Department of Defense Defense Modeling and Simulation Office.
- DoD. (1995). *Modeling and simulation (M&S) master plan*. Alexandria: DoD.
- Fowler, M. (2003). *UML distilled a brief guide to the standard object modeling language* (3rd ed.). Boston: Addison-Wesley.
- Fujimoto, R. M. (1998). Time management in the high level architecture. *Simulation*, 71(6), 388–400.
- Fujimoto, R. M. (2000). *Parallel and distributed simulation systems* (1st ed.). New York, NY: Wiley.
- IEEE 1516.1-2010. (2010). *Standard for modeling and simulation (M&S) high level architecture (HLA)—Federate interface specification*. New Jersey, NJ: IEEE.
- IEEE Std 1516-2010. (2010). *IEEE standard for modeling and simulation (M&S) high level architecture (HLA)—Framework and rule*. New York: IEEE.
- IEEE Std 1516.1-2010. (2010). *Standard for modeling and simulation (M&S) high level architecture (HLA)—Federate interface specification*. New Jersey, NJ: IEEE.
- IEEE Std 1516.2-2010. (2010). *Standard for modeling and simulation (M&S) high level architecture (HLA)—Object model template specification*. New Jersey, NJ: IEEE.
- Kuhl, F., Weatherly, R., & Dahmann, J. (1999). *Creating computer simulations: An introduction to the high level architecture*. Upper Saddle River, NJ: Prentice Hall PTR.
- Larman, C. (2004). *Applying UML and patterns: An introduction to object-oriented analysis and design and iterative development* (3rd ed.). Upper Saddle River, NJ: Prentice Hall.
- Mäk, V. (2013). *HLA RTI—Run time infrastructure: MÄK RTI*. Retrieved from <http://www.mak.com/products/link-simulation-interoperability/hla-rti-run-time-infrastructure.html>
- Möller, B., Löfstrand, B., & Karlsson, M. (2007). *An overview of the HLA evolved modular FOMs*. Orlando: SISO.
- Möller, B., et al. (2008). *HLA evolved—A summary of major technical improvements*. SISO: Orlando, FL.
- Open HLA. (2016). *Open HLA*. Retrieved from <https://sourceforge.net/projects/ohla/>. Accessed April 08, 2017.
- OpenRTI. (2011). *OpenRTI project website*. Retrieved from <https://sourceforge.net/projects/openrti/files/>. Accessed October 11, 2016.
- Pitch. (2013). *Pitch pRTI™—Connect your systems*. Retrieved from <http://www.pitch.se/products/prti>
- PNP-Software. (2007). <https://www.pnp-software.com/eodisp/home.html>. Retrieved from <https://www.pnp-software.com/eodisp/home.html>. Accessed April 08, 2017.
- Portico. (2013). *The portico project*. <http://www.porticoproject.org/>. Accessed August 30, 2015.
- SISO FEAT. (2017). *FEAT user's guide reference for the federation engineering agreements template*. Orlando, FL: Simulation Interoperability Standards Organization.
- SISO, & Siso, F. E. A. T. (2015). *Standard for real-time platform reference federation object model v2.0*. Orlando, FL: Simulation Interoperability Standards Organization (SISO).
- Topçu, O., Adak, M., & Oğuztüzün, H. (2008). A metamodel for federation architectures. *Transactions on Modeling and Computer Simulation (TOMACS)*, 18(3), 10.
- Topçu, O., Durak, U., Oğuztüzün, H., & Yılmaz, L. (2016). *Distributed simulation: A model driven engineering approach* (1st ed.). Cham (Zug): Springer International Publishing.
- Topçu, O., & Oğuztüzün, H. (2005). Developing an HLA based naval maneuvering simulation. *Naval Engineers Journal*, Winter, 117(1), 23–40.

- Topçu, O., & Oğuztüzün, H. (2010). Scenario management practices in HLA-based distributed simulation. *Journal of Naval Science and Engineering*, 6(2), 1–33.
- Topçu, O., Oğuztüzün, H., & Hazen, M. (2003). *Towards a UML profile for HLA federation design. Part II* (pp. 874–879). Montreal, Canada: SCS.
- Yilmaz, L. (2007). *Using meta-level ontology relations to measure conceptual alignment and interoperability of simulation models* (pp. 1090–1099). Washington, DC: IEEE.

Guide to Distributed Simulation with HLA

Topçu, O.; Oğuztüzün, H.

2017, XXV, 307 p. 203 illus., 183 illus. in color.,

Hardcover

ISBN: 978-3-319-61266-9