2.1 RTI 1.3-NG Architecture: Components of an Executing HLA Federation

Figure 2.1 shows a high level, logical view of an executing HLA Federation. All of the components shown in the figure are part of a single federation with the exception of the RtiExec. A brief description of each of these components is provided below.

Federate: an HLA-compliant simulation component program, plus a SOM.

Federation: a simulation composed of a set of federates interacting via the RTI services, plus a FOM.

FedExec: manages the federation. It allows federates to join and to resign from the federation, and facilitates data exchange between participating federates.
FDD file: FOM Document Data file, contains information derived from the FOM and used by the RTI at runtime.

RtiExec: a global process that manages the creation and destruction of FedExec's.

RID file: RTI Initialization Data. RTI vendor-specific information needed to run an RTI.

A more detailed description of the FedExec, RtiExec, and the libRTI is provided in Figures 2.2 and 2.3.

FedExec - The Federation Executive

- One running process per executing federation
  - Created by first federate to successfully join federation
- Manages multiple federates joining and leaving the federation execution
  - Assigns unique handles to each federate
- Facilitates data exchange between federates
- Console interface for manual operations

Figure 2.2 The Federation Executive
RtiExec - The RTI Executive

- Manages the creation and destruction of multiple federation executions (with different names)
  - Ensures that each FedExec has a unique name
- Global process executes on one platform
- Listens to a well known port
- Console interface for manual operations

**Figure 2.3 The RTI Executive**

Federates use the libRTI to invoke HLA services as shown in Figure 2.4.

libRTI - The RTI Library

- Makes HLA service methods available to federates
  - Methods communicate with RtiExec, FedExec, and other federates through them
- Written in C++ with interfaces in C++, Java, CORBA IDL, Ada

**Figure 2.4 The RTI Library**
2.2 RTI-NG 1.3v4 Architecture: Components of a Federate

User's code for a federate is linked with *Local RTI Component Code* (LRC) from the C++ library *libRTI* to form a complete federate. These local RTI components provide the services for the federate through communication with the Rtiexec, the FedExec and other federates (*RTI Programmer's Guide*). The components of a single federate are shown in Figure 2.5.

What’s in a Federate?

- The Federate’s code provides internal functionality.
- The Federate’s code must implement the abstract RTI_FederateAmbassador class.
- The LRC provide external functionality as specified by the IFSpec.
- The LRC include the methods for the RTI_RTIambassador class.

**Figure 2.5 What’s in a Federate**

RTIambassador: The federate code requests services of the RTI by calling member functions of the class RTI_RTIambassador, which is contained in the libRTI.

FederateAmbassador: The RTI sends messages and responses to the federate code by calling functions implemented in the federate. In RTI 1.3-NG, these functions are known as “callback” functions and are implemented, in the federate, as a subclass of the class *RTI::FederateAmbassador*. In the IEEE 1516 Federate Interface Specification, this class is called RTI_FederateAmbassador and callback back functions are referred to as “RTI initiated services.”

In RTI 1.3-NG, Class *RTI::FederateAmbassador* is contained in libRTI, and contains pure virtual functions for each possible callback. These routines are simply "place holders" that cannot be called. The federate code must create a derived class from this class that contains the actual implementation for each of
these callback functions.

### 2.3 Division of Responsibilities Between the RTI and Federate Code

Figure 2.6 illustrates the separation of responsibilities for the federate code provided by the user and that provided by libRTI.

![RTI and Federate Ambassadors](image)

**Figure 2-6 Code Responsibilities**

libRTI contains the RTIambassador class, which is used to provide RTI services to the federate. It also contains the abstract class FederateAmbassador which defines all of the interfaces to the callback functions but cannot be used until actual implementations of these functions are provided by the federate.

**Note:** Abstract Classes and Pure Virtual Functions: Class RTI_FederateAmbassador is an abstract class, as it contains (in the C++ implementation) pure virtual functions. What is a pure virtual function in C++? A pure virtual function is a function that contains no code in its body, but simply the expression "= 0" in place of the body. Any class containing a pure virtual function is an abstract class, and no objects may be declared of this class. The federate code must create a new class (e.g., HwFederateAmbassador), derived from this class that contains the actual implementations (bodies) for these functions.

Virtual functions in C++ form the basis of object oriented programming in that they allow true polymorphism with dynamic linking. When an implementation for a function is provided in a derived class it replaces the virtual function in the original class. If a pointer p to the base class type actually points to an object of the derived class, a call to
p->f1() will call the function (method) f1 in the derived class, even though p has the
type of a pointer to the base class. This is a change from the "normal" C++ calling rules
where a call to a function referenced by a pointer would always call the function in the
class of the pointer type. The actual function (or method) called is determined by the
type of the object "pointed to," not by the type of the pointer.

The federate code contains the simulation code which defines and creates
various simulation (or federate) objects, and requests services of the RTI by
using the RTI_RTIambassador class. The federate code also creates a class
derived from the RTI_FederateAmbassador class. This derived class contains the
actual implementations for each of the callback functions so that the federate
can receive messages and responses originated by the RTI. Many of the callback
routines are not used in a simple simulation such as the HelloWorld federation.
Implementations for unused callback routines in HelloWorld are simply
statements that print out an error message indicating that this routine should not
have been called.

2.4 Overview of Federations in Execution

Consider a scenario for a system using HLA. This system is running two
federations: Federation 1 and Federation 2. Though one system may run two
federations, the HLA Rules specify that they are independent of each other and
may not exchange any information. This system is illustrated in Figure 2.7, and
has the following components:

There is a single RtiExec and RTI.RID file, shared by both federations

Each Federation contains its own FedExec and FDD file

FedExec1 controls the execution of Federation 1, FedExec2 controls the
execution of Federation 2

Federation1 contains two federates: the White Federate and the Green
Federate

Federation 2 Contains three federates: the Purple, Orange, and Blue
Federates.
2.4 What's in the *HelloWorld* Federate?

Example: The *HelloWorld* federation. The *HelloWorld* federate code has a routine `main()`, some global constants, and class *Country*. The function `main()` contains the code for creating and joining the federation, and for the main simulation loop. The federate code also contains class *Country* which has functions used by `main()` to carry out the simulation activities, as well as class HwFederateAmbassador. class HwFederateAmbassador is a subclass of (derived from) the class RTI_FederateAmbassador, and contains the callback functions which perform the necessary actions in response to messages originating from the RTI.

Quite frequently, the *HelloWorld* federate is simply in a waiting loop waiting for a callback from the RTI to signal that it has received a message from another federate, or that the RTI has granted a request made by the *HelloWorld* federate.
Figure 2.8 demonstrates the steps in the process of starting a federation execution.

1. When a federation is run, the RtiExec is started first.

2. Then a federate, acting as a manager, creates a federation execution by invoking the RTI method \texttt{createFederationExecution} on its RTI Ambassador.

3. The RTIAmbassador then reserves a name with RtiExec, and spawns a FedExec process, and that FedExec registers its communication address with RtiExec. The federation execution is underway.

4. Once a federation execution exists, other federates can join it. That RTIambassador consults RtiExec to get the address of FedExec, and invokes \texttt{joinFederationExecution()} on FedExec. Additional federates can join via the same process.
2.5 Federate Interface Specification

RTI services are divided into six management areas in the RTI specification, as mentioned in Lesson 1. The interface specification provides a description of the functionality of each service and required arguments and pre-conditions necessary for use of the service. Figure 2.9 contains a sample of an interface specification from the draft standard. Figure 2.10 shows the sections contained in an interface specification with a brief explanation of each section. The actual calling sequences are provided for each supported language in an Appendix to the Interface Specification. Prototypes are provided in the RTI 1.3 standard for IDL, C++, Ada 95, and Java.

**IF Specification Format**

<table>
<thead>
<tr>
<th>4.2 Create Federation Execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>The Create Federation Execution service shall create a new federation execution and add it to the set of supported federation executions. Each federation execution created by this service shall be independent of all other federation executions, and there shall be no inter-communication within the RTI between federation executions. The FED designator argument shall identify FED that is required for the federation execution to be created.</td>
</tr>
<tr>
<td>Supplied Arguments</td>
</tr>
<tr>
<td>— Federation execution name</td>
</tr>
<tr>
<td>— FED designator</td>
</tr>
<tr>
<td>Returned Arguments</td>
</tr>
<tr>
<td>— None</td>
</tr>
<tr>
<td>Pre-conditions</td>
</tr>
<tr>
<td>— The federation execution does not exist.</td>
</tr>
<tr>
<td>Post-conditions</td>
</tr>
<tr>
<td>— A federation execution exists with the given name that may be joined by federates.</td>
</tr>
<tr>
<td>Exceptions</td>
</tr>
<tr>
<td>— The federation execution already exists.</td>
</tr>
<tr>
<td>— Could not locate FED information from supplied designator</td>
</tr>
<tr>
<td>— Invalid FED</td>
</tr>
<tr>
<td>— RTI internal error</td>
</tr>
<tr>
<td>Related Services</td>
</tr>
<tr>
<td>— Destroy Federation Execution</td>
</tr>
</tbody>
</table>

Figure 2.9 Sample of Interface Specification Format in RTI 1.3 Standard
(Not a Slide)

Post-conditions specify any changes in the state of the Federation resulting from the call.

Exceptions give all possible exceptions thrown by the service routine. Requests for services should be included in *try-catch* blocks so that appropriate action may be taken for error processing.
Content of an IF Specification

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

Figure 2.10 RTI Interface Specification Content

It is important to note that service interfaces which are callbacks from the RTI to the federate have a dagger (†) following their name. For example, if the Send Interaction service is used by one federate to send interaction (event) data to another federate, the RTI will use the callback function Receive Interaction† to provide the interaction data to the destination federate(s).

2.6 Basic Flow of the HelloWorld Federate

A small subset of the RTI services is used by the HelloWorld federate, and this federate will be used to illustrate several of these service groups. Keep in mind that a federate could potentially simulate many different objects. In this case the HelloWorld federate simulates only a single object of class Country. Some of the services used by the HelloWorld federate could be eliminated for so simple a simulation. However, the HelloWorld federate, as presented, provides a framework for considerably more complex simulations.

Federation Management

Each federate, representing a country, attempts to create the HelloWorld federation, as the order in which the different federates are started is not known in advance (Create Federate Execution). After the federation is successfully
created, further attempts to create it will fail. 
The federate attempts to join the federation (Join Federation Execution). If the 
federation has not yet been fully created, this will fail and must be retried 
repetitively until it succeeds.

Declaration Management

Federate declares data it will send, or publish, to the federation:
   1. Data sent at the end of each timestep (Publish Object Class Attributes).
   2. Data sent at arbitrary times (events) (Publish Interaction Class).

Federate declares data that it is interested in receiving, or subscribing to, from 
other federates:
   1. Data received regularly at the end of other federate's timestep 
      (Subscribe Object Class Attributes).
   2. Data received at arbitrary times (events) (Subscribe Interaction Class).

Time Management

Federate announces to the RTI that it is a time-step type of simulation, and that 
the RTI should coordinate the advancement of time with the other federates to 
assure that data from other federates will be received before time is advanced 
for this federate (Enable Time Regulation, Enable Time Constrained ).

Federate announces to RTI that even though this is a timestep type of 
simulation, it is interested in receiving the infrequent HelloWorld events from 
other federates, whenever they are sent (Enable Asynchronous Delivery).

Support Services

Federate announces to RTI that it is interested in receiving advisories when there 
is/is not some other federate that is interested in receiving (subscribing to) the 
data from this federate (Enable/Disable Attribute Relevance Advisory Switch). 
This will allow the federate to stop sending data when there is no one to receive 
it.
Object Management

Federate creates a country object and registers it with the federation (*Register Object Instance*).

Messages arrive occasionally from the RTI announcing that a new country (object) has joined the federation (*Discover Object Instance*†). *Country* calculates results for this time step and sends new population figure to the other federates (*Update Attribute Values*). The country prints the population for all known countries (including itself) in the console window of the user's terminal.

*Country* occasionally sends "Hello World!" message to the other countries (federates) (*Send Interaction*).

*Country* receives results for this timestep from other federates (*Reflect Attribute Values*†).

*Country* occasionally receives “Hello World!” events (message) from the other countries (*Receive Interaction†*).

*Country* receives notice from the RTI: no other federate exists that is interested in (subscribing to) the population data from this *Country* (object), so it might as well stop sending the data (*Turn Updates Off For Object Instance†*).

*Country* receives notice from the RTI: another federate has indicated that it is interested in receiving (subscribing to) the data from this federate, so begin sending population data (*Turn Updates On For Object Instance†*).

Time Management

*Country* requests for the RTI to advance time by one timestep (*Time Advance Request*) and waits for the request to be granted by making repeated calls to RTI::tick.

*Country* receives notice that the time advance has been granted (callback routine *Time Advance Grant†*).
Federation Management

When the federation has completed its simulation it advances time by one timestep and deletes the country object it created (*Delete Object Instance*). The federate then resigns from the federation (*Resign Federation Execution*).

The federate doesn't know whether it is the last federate to resign, so it attempts to kill the federation itself (*Destroy Federation Execution*). If it is not the last running federate the attempt to kill the federation fails.

### 2.7 Using RTI 1.3 Services in C++ Federates

For example, as shown in Figure 2.11, function *main()* in the *HelloWorld* federate contains a request to advance time by its timestep, which has been stored in a temporary variable *requestTime*. This code is included in a *try* block. A *catch* block immediately follows the *try* block in order to catch the error and print an appropriate error message.

```cpp
try {
    rtiAmb.timeAdvanceRequest(requestTime);
}
catch (RTI::Exception & e) {
    cerr << "FED_HW: ERROR:" << e << endl;
}
```

*Figure 2.11 Sample RTI Service Request – *timeAdvanceRequest(*)*

This code segment catches any RTI exception thrown by the service routine and prints "FED_HW: ERROR:" followed by a textual description of the actual exception "e."
The exceptions portion of the description for this service in the specification shows that the exceptions shown in Figure 2.12 may be thrown by this request (a few are omitted from the list).

### Possible Exceptions Thrown

- The federation time is invalid.
- Federation time already passed.
- The TimeAdvanceRequest is already pending.
- The federate is not a federation execution member.
- Save in progress.
- Restore in progress.
- RTI internal error.

**Figure 2.12 Possible Exceptions Thrown By `timeAdvanceRequest()`**

Any of these exceptions will be caught by the `catch` block. If the author of the federate wanted to be more specific and catch only a selected exception, the type of a particular exception could have been used instead of the catch-all `RTI::Exception` type (e.g., `RTI::FederationTimeAlreadyPassed`). In IEEE 1516, the name of this exception has been changed to `RTI_LogicalTimeAlreadyPassed`.

`rtiAmb` is an object declared at the beginning of `main()` to be of class `RTI::RTIambassador`.

Upon return, even if there are no exceptions, the time advance request has probably not been granted. `main()` simply waits for the grant to take place by the code segment shown in Figure 2.13.

This code segment sets a global Boolean flag `timeAdvGrant` to `false` and waits for it to become `true`. Rather than spinning in a computational loop `tick()` is called. This simply relinquishes control for a time not less than .01 seconds, nor greater than 1.0 seconds, which gives the RTI a chance to run.
Using “Tick”

- Tick used to wait for service completion
  - Allows RTI a chance to execute and respond to request
  - Waits for not more than 1.0 sec’s, nor less than .01 sec’s

- `timeAdvGrant` is a global variable, initialized to false, set `true` in callback routine `TimeAdvanceGrant`†

```c
    timeAdvGrant = RTI::RTI_FALSE;
    while (!timeAdvGrant)
    {
        rtiAmb.tick(0.01, 1.0);
    }
```

Figure 2.13 Using Tick to Wait for `TimeAdvanceGrant`†

When other federates have sent all of the data that is possible to send before the requested time, the RTI will notify the federate that its `timeAdvRequest` has been granted by calling the callback routine `TimeAdvanceGrant`† in the federate during one of the intervals where federate control of the processor has been relinquished by calling `tick()`.

The call-back routine `TimeAdvanceGrant`† in the federate simply sets the flag `timeAdvGrant` to true (RTI::RTI_TRUE) and returns.

When control is returned to the federate, it notices the flag is now set and escapes the while-loop.

**Note:** Using the PC-NT implementation of the RTI 1.3-NG, only a few functions are coded re-entrantly, so it is important that, in general, no routine in the RTI_FederateAmbassador may call a service routine in the RTI!!
2.8 Summary: Basic Responsibilities of Code for *HelloWorld* Federation

Basic types of responsibilities of the *HelloWorld* federate code are shown in Figure 2.14.

![HelloWorld Code Responsibilities](image)

**HelloWorld Code Responsibilities**

- 1. Create and destroy the federation.
- 2. Join and Resign from the federation.
- 3. Declare data to be published and subscribed to by the federation.
- 4. Send/Receive data to/from other federates.

**Figure 2.14 Basic Responsibilities of *HelloWorld* Federate Code**

1. Create and destroy the federation.

All federates try to create the federation when they begin execution, and try to destroy the federation before exiting. Federates should not make assumptions about the order in which they are executed. This would be difficult to control in a simulation with geographically distributed federate executions. Trying to create a federation that already exists will fail. Attempting to destroy a federation that still has other executing federate members will also fail, so no problems arise if the creation or destruction is not really appropriate.

The federation's name is *HelloWorld*.

2. Join and Resign from the federation.

The federate attempts to join the federation. If this is the federate that just created the federation, its creation may not be complete. The *HelloWorld* federate will attempt to join the federation 20 times, sleeping for 2 seconds
between tries.

The federate's name, used in the Join request, is given as the name of the
country specified on the command line when the federate is executed.

3. Declare data to be published and subscribed to by the federation.

The federate must declare any data to be sent or received by the federate by
making the appropriate service calls as described in 2.6. Objects simulated by
this federate must be registered with the federation as described in 2.6. (A more
detailed description of this mechanism will be included in Module 1, Part 5.)

4. Send/Receive data to/from other federates.

5. Advance time. A much more detailed description of time management
services will be given in Module 1, Parts 4 and 6.

Assignment

Use the HLA Course Lab Notes to help you to make a minor change to the
HelloWorld federate, recompile and run the new federate.

Suggested Readings

1. Look up the interface specifications mentioned in this lesson in the IEEE
   P1516.1, Draft Standard for Modeling and Simulation (M&S) High Level
   Architecture (HLA) – Federate Interface Specification.

2. Read Chapters 1 and 2 of the High Level Architecture Run-Time
   Infrastructure Programmer’s Guide 1.3-NG. Also look up the interface
   specifications for the interfaces used in this lesson in the Appendices to
   this Guide. Compare the information to that obtained in [1] above. The
   Guide gives the actual C++ prototypes for the RTI services, for a
   particular implementation of the RTI. Details of operation are provided
   here that are not possible to specify in the implementation-independent
   descriptions of the services provided by the standard.

3. Review the syntax and semantics of pure virtual functions, abstract
classes, and derived classes in a book you own for C++, such as: The
   Annotated C++ Reference Manual by Margaret A. Ellis and Bjarne
   Stroustrup.