Documentation for the RTI

Marc Moreau

Fairfax, August 25, 1997

Contents

List of Figures

Chapter 1

Introduction

1.1 Rationale

In order to understand the HLA and bring our work into compatibility with it, we have been investigating design and implementation alternatives for a light-weight RTI to be used with distributed virtual simulation (see [PLM97]).

Our RTI prototype grew out of earlier works. The interest and data distribution management part is based on the Dual Mode Multicast (DMMC) scheme (see [PW95] and [FP97]), and the communications portion includes the Selectively Reliable Transmission Protocol (SRTP, see [PL95] and [PL96]).

This document is a brief overview of the implementation of our RTI. It is not an introduction to the HLA concepts or to the DMMC and the SRTP, and a good understanding of the HLA specifications (see [HLA96], [HLA97a], [HLA97b] and [HLA97c]) is necessary to read it.

It also necessary to read and study the source files (more than 12,000 lines!) and to understand the interactions between the components before trying to modify the RTI.

1.2 Which simulation environment for the RTI?

The HLA has been developed in order to facilitate interoperability and reuse of simulation components. It is intended to be applicable to simulations involving pure software representations, man-in-the-loop simulators, and interface to live components. It provides a consistent basis for addressing simulation system design and implementation issues, especially difficult ones like mechanisms for scalability, time management, aggregation and disaggregation.

Our prototype of light-weight RTI has been designed to meet several of these concerns, by choosing a subset of issues and key points.

This RTI is to be used with *virtual simulations* (a simulation involving real people operating simulated systems) or even constructive simulations (simulations that involve simulated people operating simulated systems ; real people only make inputs to such simulations). But since our RTI does not implement the time management services, the simulations need an other mechanism for controlling their advancement along the time axis. We expect, it will be used with *real time* simulations.

Since the goal of this RTI was to explore the *Declaration and Data Distribution Managements*, other difficult issues have been willingly ignored. For example, the retraction mechanism did not make sense because the real time simulations do not permit "undo" mechanism. Aggregation, disaggregation and ownership management were also more related to constructive simulations than man-in-the-loop applications.

This RTI has been designed to support *small to medium size federations*, in relation with the scalability issues, but improvements are still necessary to achieve this goal.

This RTI can be *interfaced to DIS simulations*, by using our DIS to HLA translator. This translator has been designed to work with our RTI and to take benefit of the available DDM services, but can be used with other implementations of the RTI (see the documentation of the translator). Our work in that field was inspired by the Platform Proto-Federation Experiment.

1.3 Overview of the RTI

1.3.1 Software architecture

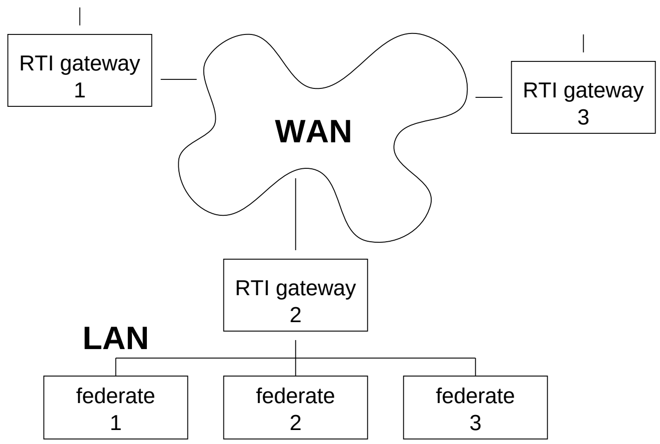

Figure 1.1: Simplified architecture of a federation

Our RTI has a distributed architecture. The different components are connected together using TCP to send and receive the services information (which must be delivered reliably and are rather infrequent) and using SRTP to send and receive simulation data.

As this prototype is based on the DMMC, we use the same hierarchical architecture (see Fig 1.1).

Two kinds of server are in fact running on the gateway of a given LAN.

The Federation Server is a unique process running on the gateway host of the LAN. Its function is to manage the creation and destruction of federations. The

Federation Servers are fully connected together with TCP.

Whenever a new federation is created, the Federation Servers of all involved LANs will start a process called a gateway to provide the RTI services for this federation.

The RTI Gateway provides the RTI services and performs the routing/filtering tasks according to the DMMC approach. All the federates of the LAN are connected via TCP to this process.

Several RTI Gateways may be running on the gateway host if several federations have federates on the corresponding LAN (see LAN B on figure 1.2).

The RTI Ambassador is the library which implements the services of the RTI API. The API is based on the version 1.1 of the specifications, but several modifications have been made in order to fix inconsistencies of the specifications and to make the migration to the version 1.2 easier (see section 3.1).

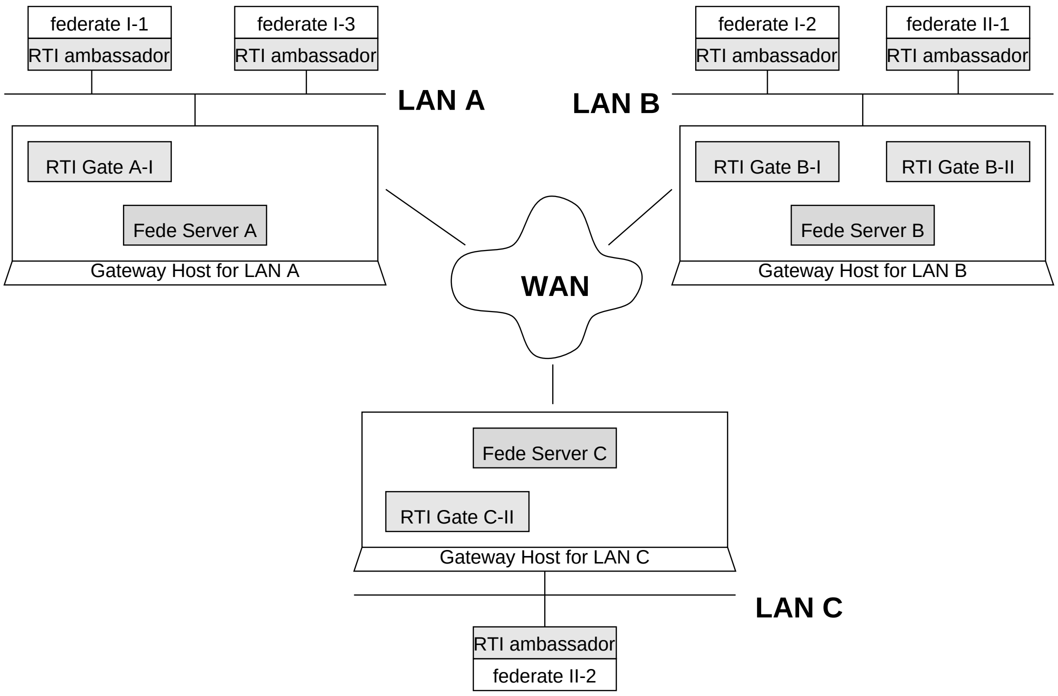

Figure 1.2: RTI Network with 3 LANs (A, B and C) and 2 federations (I and II)

1.3.2 Data flow

Figure 1.3 shows a functional break-out of the RTI. This partitioning does not directly map to software object or methods. It is only intended to underline the main functions and dependences within the RTI.

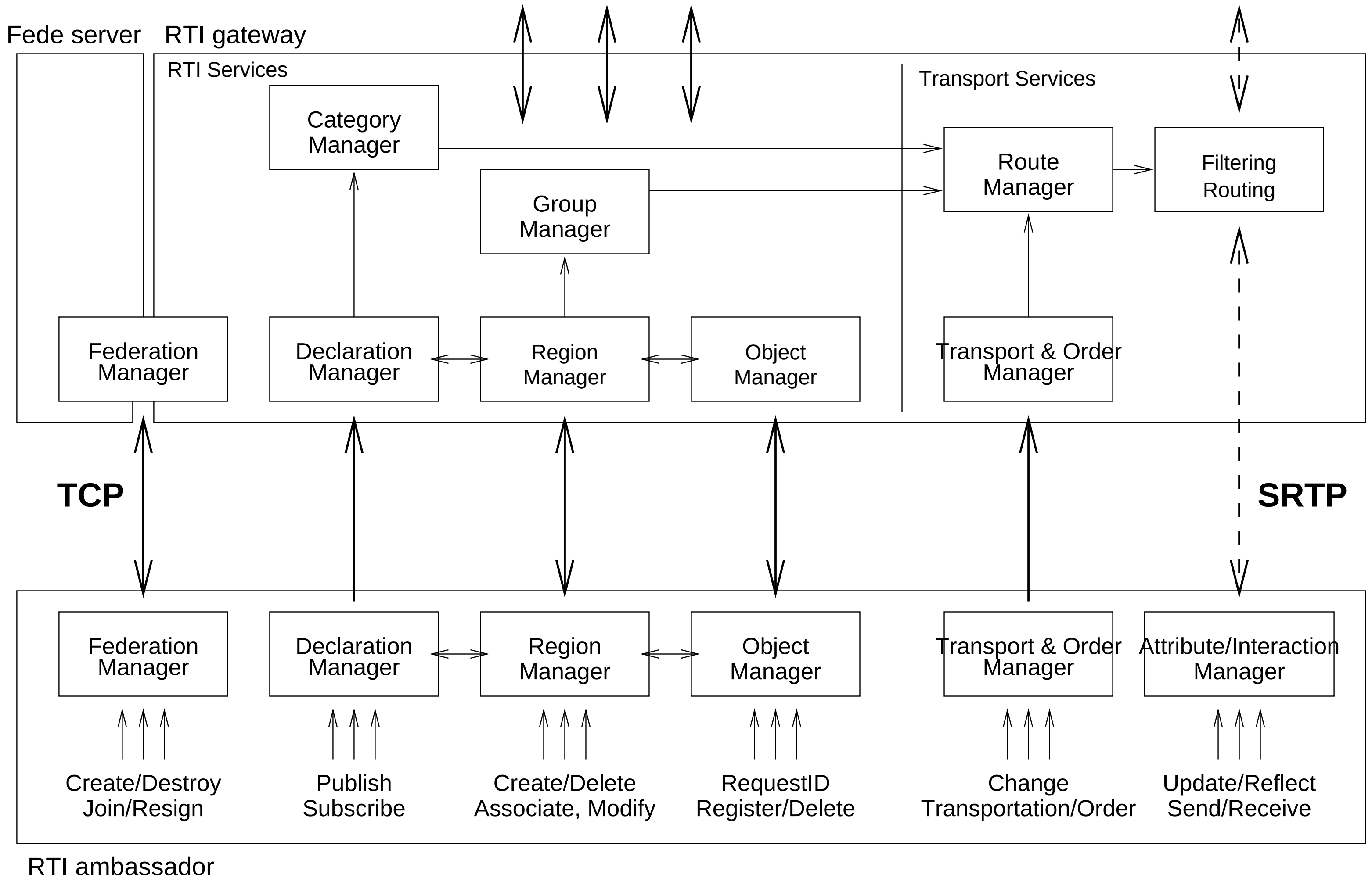

Figure 1.3: Functional break-out and data flow of the RTI

Chapter 2

Object Model

2.1 Short analysis

From the RTI specifications, we can identify the following objects which must be implemented as C++ classes in our program:

- Object Class
- Object
- Attribute
- Interaction Class
- Interaction
- Parameter
- Routing Space
- Region
- Federate

We also need containers to manipulate, store and retrieve large collections of such objects (like the objects simulated by a given federate, the list of subscribed interaction classes...). For efficiency and simplicity reasons, we have chosen to use the *Standard Templates Library* to implement these lists.

The communication implementation requires at least two kinds of object. We need a class which encapsulates the low-level notion of socket, and a message class to represent the data exchanged with TCP and SRTP.

Additional classes or structures are necessary now and there to make things easier, or to implement a given service.

The purpose of this chapter is to give a brief overview of the different classes used within the RTI. It is not a reference document, and many elements have been omitted, especially in the figures. It is necessary to study the source files in order to have a full understanding of the RTI implementation.

2.2 HLA classes

This section describe a set of basic classes which are used by the different components of the RTI. They are defined in the `data-*.hh` files.

2.2.1 Object and interaction

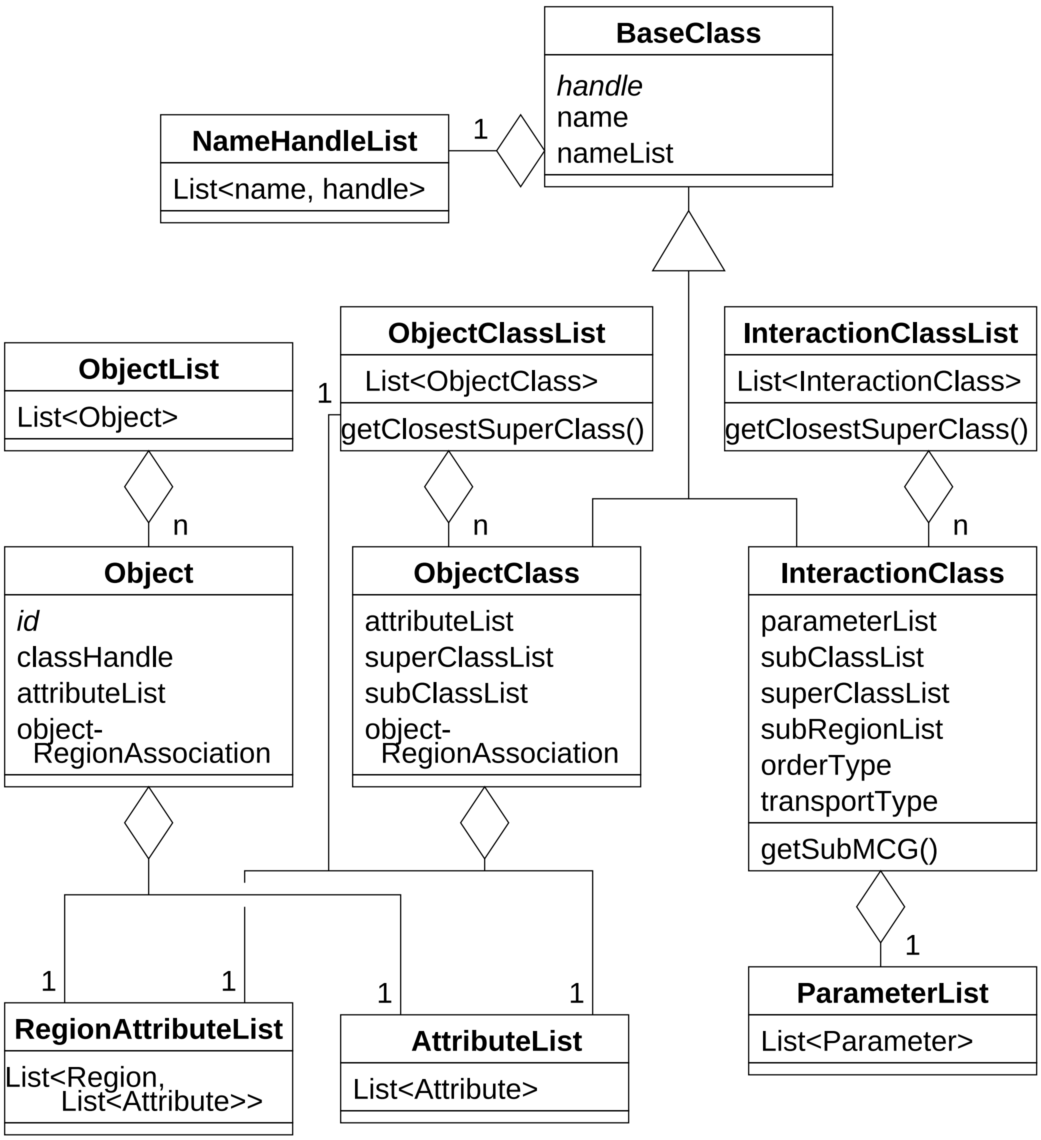

Figure 2.1: Object and interaction related classes (declared in `data-class.hh` and `data-object.hh`)

Object and Interaction classes, defined respectively as `ObjectClass` and `InteractionClass` are subclasses of the `BaseClass` (see figure 2.1). They both inherit a `name`, a `handle` and a `nameList` (see `NameHandleList` below).

Both of them have a `superClassList` and a `subClassList`, to represent the hierarchical relationship described in the Federation Object Model (FOM).

The `ObjectClass` class has an `AttributeList`, whereas the `InteractionClass` class has a `ParameterList`.

There is a strong asymmetry between objects and interactions: an interaction must be manipulated as a whole thing, whereas an object can be divided into independent attributes. The `InteractionClass` has therefore several attributes (list of associated regions, `orderType`, `transportType`...) which only appear in the attribute of the object classes. The `InteractionClass` and `Attribute` also have a `getSubMCG()` method, which returns the set of multicast groups associated to their subscription regions.

The two containers `ObjectClassList` and `InteractionClassList` have a special method `getClosestSuperClass()`: when the class list is a list of subscribed classes, it can be used to return the closest superclass of its argument or `NULL` if the argument class has not been subscribed at all.

An object is described in an `Object` class (see figure 2.1), with an `id`, a `classHandle`, an `attributeList` and an `objectRegionAssociation` (see at the end of this section).

Parameters and attribute are closely related, and the `Attribute` class is in fact a subclass of the `Parameter` class (see figure 2.2). Both of them have a `name`, an `handle`, a `valueLength` attribute and a `buff` attribute to store their value.

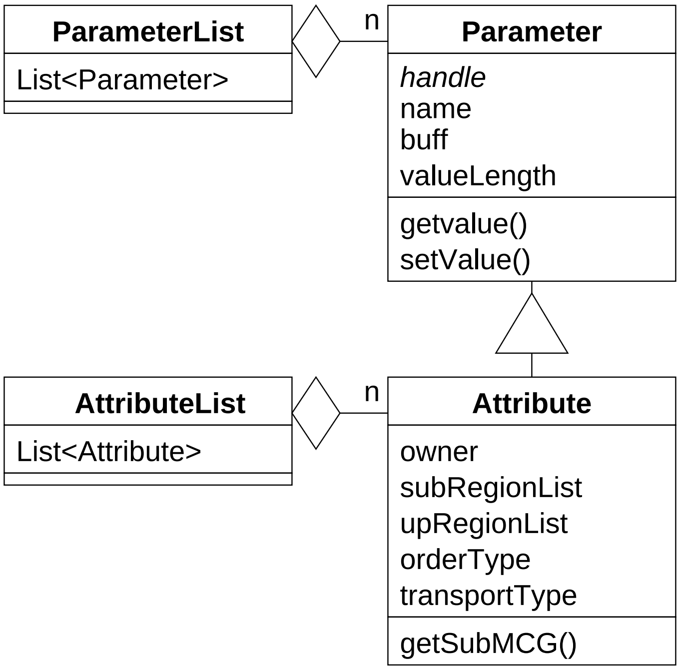

Figure 2.2: Attribute and parameter classes (as declared in `data-object.hh`)

In addition, the `Attribute` class not only has the same extra attributes as the `InteractionClass`, but also an `owner` attribute, saying which federate owns this `Attribute` (see Ownership Management in the RTI specification).

To make the implementation easier, no difference has been made between the `Attribute` of an `ObjectClass` and the `Attribute` of an `Object`. That is why some attributes of the `Attribute` class only make sense for a simulated object (like the value, value length, owner and update regions list) while the `subRegionList` must be considered only when the `Attribute` is part of an `ObjectClass`.

Two additional classes are defined.

`NameHandleList` gives the correspondence between a name and a handle. For example, the `ObjectClass` has a `nameList` attribute which can be used to find the `AttributeHandle` corresponding to a given attribute name (see in the RTI Support Services `getAttributeHandle()`).

`RegionAttributeList` is the implementation of the recently introduced notion of *group of association*. The version 1.2 of the RTI specifications describes how a set of attributes must be associated to a (object, region) pair with the "Associate Region For Update" service or to a (object class, region) pair with the "Subscribe Object Class Attributes With region" service. Therefore, both `Object` and `ObjectClass` have an `objectRegionAssociation` which describes the set of attributes associated to each region.

2.2.2 Routing space and region

The routing space is one of the key notions of the DDM. The `RoutingSpace` is described by its `name`, its `handle`, its number of dimensions `length` and the physical maximum extension of the space `bound`, which is an array of `RTI::Range` of size `length`.

The mapping between the routing space and the multicast group is grid-based. `nbSgnt` is the number of cells along each dimension and `map` describes this mapping. This array is initialized by calling `mapMCG2Space()` with a set of available multicast groups.

`getSgnt()` returns the position of the cell (the segment) corresponding to the given position on the given coordinate.

`getMCG()` returns the multicast group corresponding to the given `index` in `map`.

A `RoutingSpaceList` container is of course available.

A region is implemented as a `BaseRegion` class. It is identified by a unique `id` and described by the `extents` and by a routing space `spacePtr`.

The number of subscription and update references to the region is stored in `nbSub` and `nbUp`. These values can be modified and accessed with `incSub()`, `decSub()`, `getSub()` and `incUp()`, `decUp()`, `getUp()`.

`setMCgroups()` build the set of multicast groups `mcgSet` associated to the region according to its description. It is intended to work in cooperation with the routing space's method `mapMCG2Space()`.

`getThreshold()` returns the threshold for the region with the required accuracy.

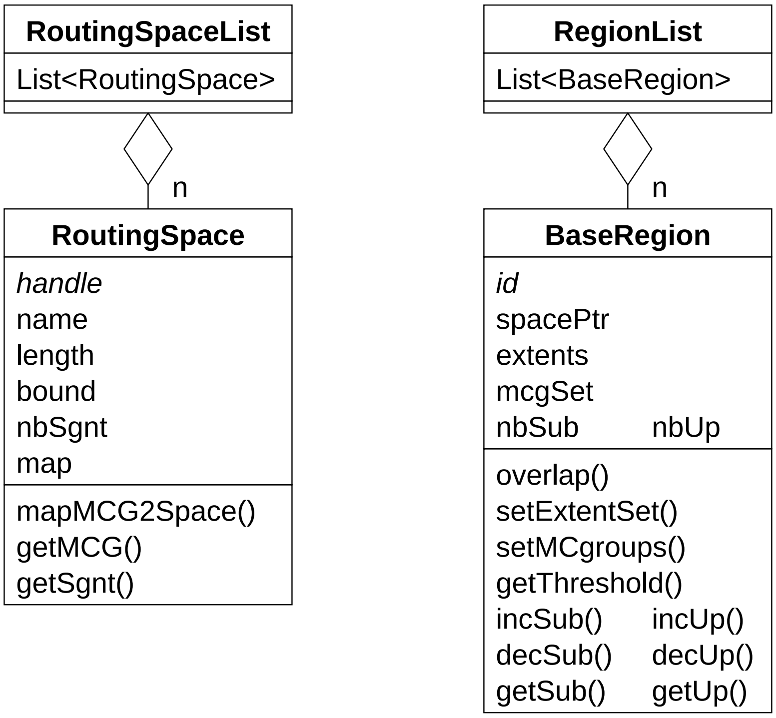

Figure 2.3: Region and routing space (as declared in `data-region.hh`)

2.2.3 Federation Execution Data

The Federation Execution Data (FED) file can be parsed with the `FED` class. The only public method is `readFED()` which reads the file and returns the information in the argument-value parameters. It uses a loop and several stacks (implemented with the `vector` template) to parse the LISP-like syntax of the file.

All the attributes are private. `fileName` and `file` identify the FED file. The others are used internally by the parsing mechanism.

2.2.4 Federate

The RTI gateway and the RTI ambassador need to have a global representation of the associated federates.

The `Federate` class (figure 2.4) contains:

– the `handle` and `name` which identify the federate;

– the `regionList` with all the regions which have been created by this federate (according to the version 1.2 of the specifications, there is no distinction between subscribe and publish regions);

– the subscribed classes `subObjectClassList` and `subInteractionList`;

– the published classes `pubObjectClassList` and `pubInteractionList`;

– the simulated objects `upObjectsList` and the set of all the requested object IDs `idSet`;

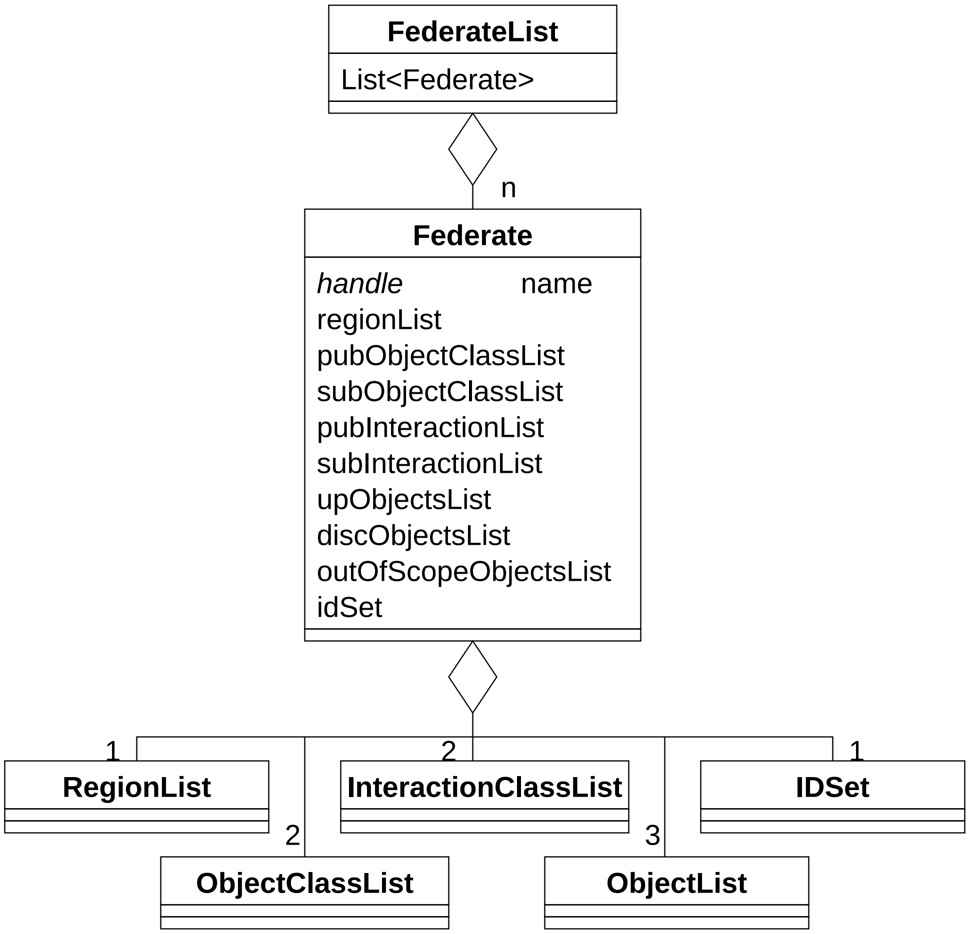

Figure 2.4: Federate (as declared in `data-fede.hh`)

– the discovered objects `discObjectsList` and the previously discovered objects which came out-of-scope `outOfScopeObjectsList`;

The RTI ambassador which is associated to a given federate and the gateway which manages a set of federates do not need to have the same information and the same degree of details about these federates. Therefore, they are not necessarily going to use the same attributes; for example, the RTI gateway does not use the `discObjectsList` and `outOfScopeObjectsList`.

2.3 Communication

2.3.1 Connection

The different pieces of software must send and receive information or messages from the network and from the keyboard concurrently. In order to handle these different streams in a standardized way, we have the `Connection` class. Each connection is identified by its `socketDescriptor`. Five callbacks may be provided by the user to handle different kind of events:

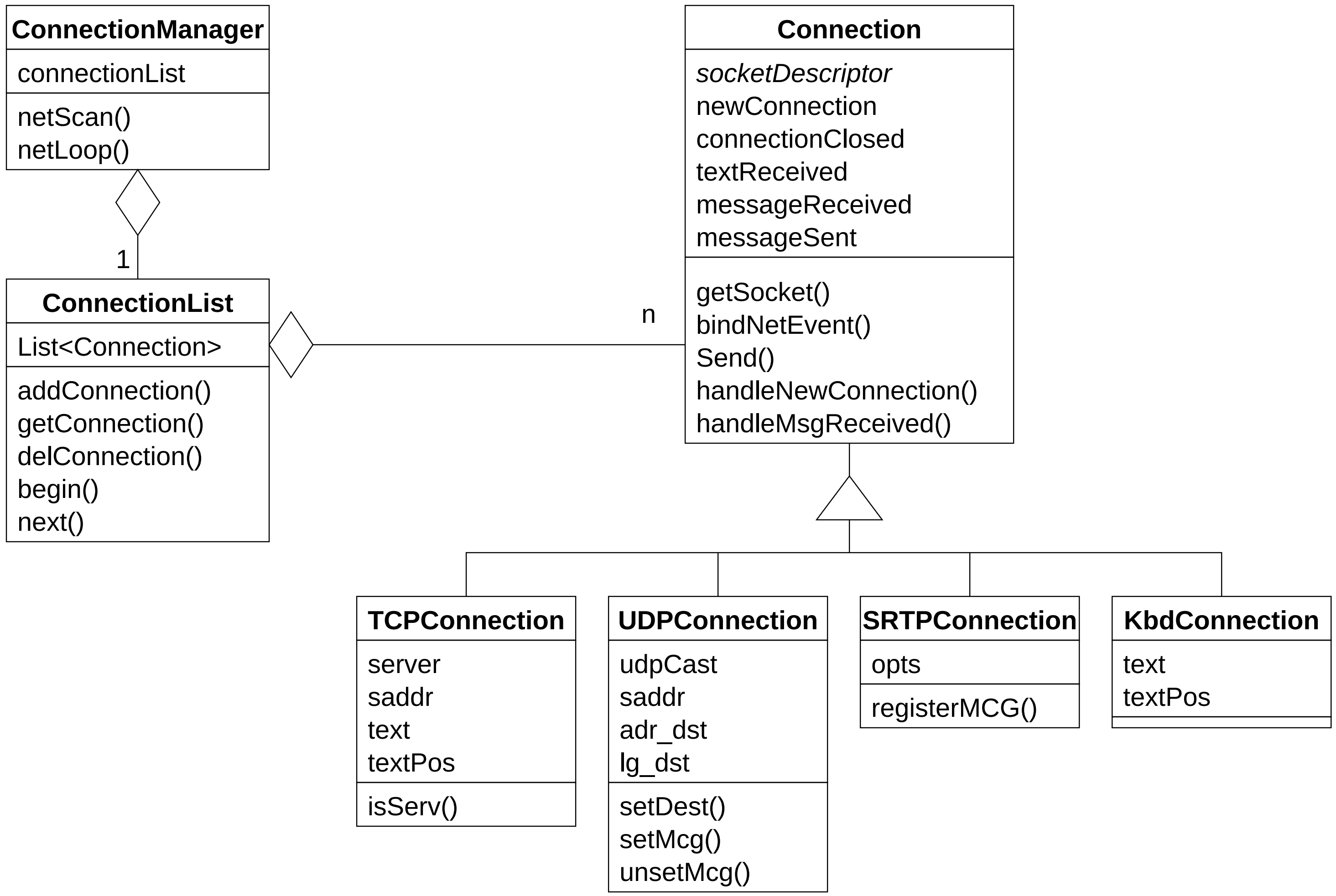

Figure 2.5: The connection related classes (declared in `connect.hh`)

– `newConnection` is called whenever a new client connects to the socket (for TCP only);

– `connectionClosed` is called whenever the connection is broken (used only for connection oriented communication);

– `textReceived` is called whenever new data is received from a stream (TCP or the keyboard);

– `messageReceived` is called whenever a new packet is available (UDP, SRTP);

– `messageSent` can be called whenever data has been sent. (not implemented yet).

The `bindNetEvent()` method binds the user given function to the corresponding event.

Four subclasses are derived from the abstract `Connection` class (see figure 2.5):

– `TCPConnection` has a `sockaddr_in saddr` attribute to describe the connection, the `text` and `textPos` attributes to deal with fragmentation, the `server` and `isServ` attributes to identify the "server" connections.

– `UDPConnection` has a `udpCast` attribute to specify the mode (unicast, broadcast or multicast), a `sockaddr_in saddr` attribute to describe the connection, and an `adr_dst` attribute to describe the destination.

– `SRTPConnection` has an `opts` attribute to specify the SRTP send options.

– `KbdConnection` has just the `text` and `textPos` attributes to deal with fragmentation.

2.3.2 ConnectionManager

The `ConnectionManager` is the heart of the communication system. It has a list of `Connection` (implemented as an hash table) and two methods `netScan()` and `netLoop()`. `netScan()` just scans all the socket descriptors a finite number of times and returns, while `netLoop()` loops indefinitely. Whenever there is a new event, they call the appropriate callback for the corresponding connection.

2.3.3 Handling multicast groups

Due to our choice to implement the RTI using SRTP, we need to manipulate collections of multicast groups.

A given multicast group is represented by a `MCgroup` structure with the name of the multicast group, a flag which is set when the multicast group has been registered, and a reference counter. When the counter value is 0 (in the RTI, when no region is associated to this multicast group) the application may safely unsubscribe the group, using `srtp_unreg_mcg()`.

In order to make the reference count system work properly, it is necessary to have only one instance of each multicast group and to manipulate pointers or references to this multicast group. Otherwise its not possible to keep track of the real number of references among all the copies of the same `MCgroup`.

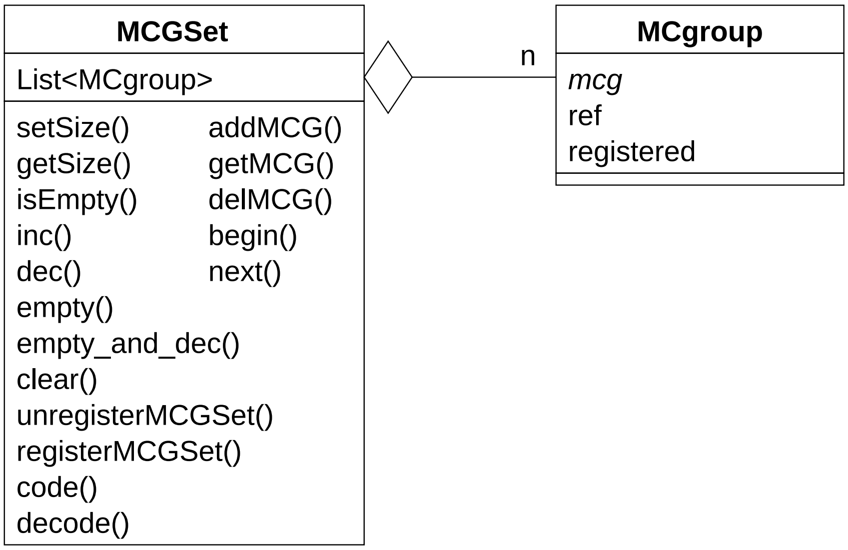

Figure 2.6: Multicast group and multicast group set (declared in `connect.hh`)

A collection of multicast groups is represented by a `MCGSet` class.

This set has of course all the basic methods necessary to add, remove and search for a given multicast group, or to iterate through the set.

But it has also several methods to manipulate the groups of the list:

– `inc()` and `dec()` increase or decrease the number of references of all the multicast groups in the set;

– `empty()` and `empty_and_dec()` empty the set, while `clear()` empties the set *and* deletes the multicast groups.

– `registerMCGSet()` and `unregisterMCGSet()` (un)register the groups of the list, using the SRTP function calls;

– `code()` and `decode()` pack and unpack the list of groups into a ready-to-send array of bytes.

2.3.4 Messages

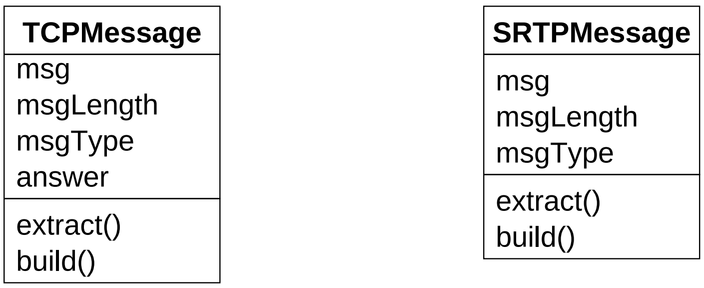

Figure 2.7: Two classes of message (declared in `data-msg.hh`)

Two classes represent the two kinds of messages we want to send on the network. `TCPMessage` and `SRTPMessage` are very similar (see figure `message.eps`). They both have:

– `msg` and `msgLength` attributes to represent the message;

– `msgType` is an enumerated value, describing the type of the message (for example `REGISTER_OBJECT` or `MODIFY_REGION` for a TCP message, and `UPDATE` or `SEND` for an SRTP message);

– the `build()` and `extract()` methods to pack and unpack all these elements into a ready-to-send array of bytes.

The `TCPMessage` has an additional `answer` attribute to make the difference between the requests and answers to a given service.

Additional structures are defined in `data-msg.hh`. They represent different standard headers for the messages we have to send and are intended to make the manipulation of messages easier and quicker.

2.4 The RTI ambassador

2.4.1 Specific classes

Three classes and their container are declared in `data-trans.hh`. They correspond to specific problems of the RTI ambassador.

The received messages are represented by the `MsgRcvd` structure, and stored in a `MsgRcvdList` container. The `MsgRcvd` structure just contains the serial number `sn` and timestamp `ft` of the received and processed messages. The `MsgRcvdList` class is a normal container, except that the `addMsgRcvd()` method uses the private method `clean()` to remove old messages and free some memory.

This list is used to *detect message duplication*, which happens whenever a given message is sent on different multicast groups.

"Reliable" communication is obtained by sending empty mode 0 messages to trigger the loss detection for the corresponding significant mode 1 packet. The

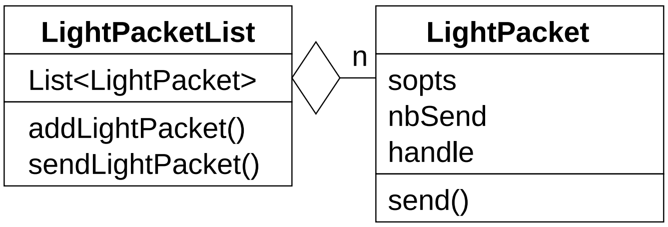

Figure 2.8: Light packets for reliability (declared in `data-trans.hh`)

`LightPacket` class only contains the send options `sopts` (which are the send options of the mode 1 packet), the `handle` of the federate (the content of the mode 0 message) and the number of mode 0 messages to send `nbSend`. This counter is decremented automatically whenever the `send()` method is called.

The corresponding container `LightPacketList` has only two methods:

– `addLightPacket()` adds a packet to the list;

– `sendLightPacket()` sends one exemplar of each packet and removes light packets with a `nbSend` of 0.

The data distribution mechanism must deliver all the attribute/value pairs in an Update Attribute Values service invocation, which have identical transportation and message ordering type, in one single Reflect Attribute Values service invocation. Of course the data distribution mechanism must also deliver the attributes on the appropriate multicast groups.

The `updateAttributeValues()` method therefore needs the `McgAtt` structure, which associates to a given multicast group a set of attributes to be sent on this multicast group.

The corresponding `McgAttList` behaves like a normal container, except for one method; `getMcgAtt (MCgroupPtr mcgp, RTI::ULong s)` returns the `McgAtt` structure with multicast group `mcgp` or *creates* it, with a set of attribute/value pairs of size `s`.

Sorting the different attributes and building the `McgAttList` for each multicast group and transport service is the main reason for the poor runtime performances of the RTI.

2.4.2 The RTI ambassador

The public interface of the RTI ambassador is described in the specification documents. These methods implemented in the `RTIamb*.cc` files.

The private data and private methods are declared in `RTIambPrivate.hh`.

The methods are:

– `readRID()` used to read and parse the `.local` initialization file (see 3.3);

– `handleTCP()` which is the callback used by `theConnectionManager` for the TCP messages;

– `handleSRTP()` which is the callback used by `theConnectionManager` for the SRTP messages;

The callbacks must be declared static, because they are called as functions, independently of a given object instance. The consequence is that several attributes which are manipulated by these callbacks must be declared as static too. However, it is not a problem because there is only one `RTIambassador` instance.

2.5 The RTI gateway

2.5.1 Specific classes

The state of the other gateways is represented internally with a `Gate` class (see figure 2.9). A gateway is identified by the `name` of its host. The attribute `s` is the corresponding socket descriptor for TCP connection. The other attributes are lists which provide an aggregated view of the interests of the corresponding LAN:

– `subInteractionList` and `subObjectClassList` (the subscribed classes) and `regionList` (the defined regions) are used by the DM and DDM services.

– the `upObjectsList` contains all the updated objects. This is used by the `requestObjectAttributeValueUpdate()` service.

– an additional attribute `upObjectClassList` would be necessary in order to implement the `requestClassAttributeValueUpdate()` service.

The `GateList` container is also defined.

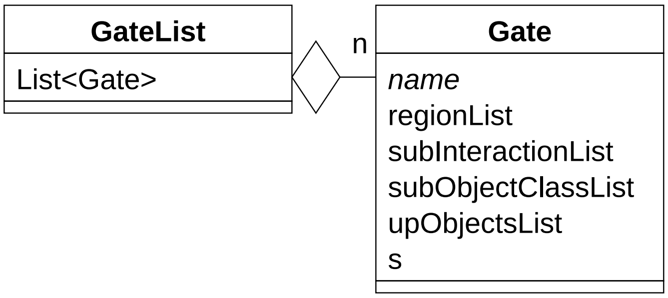

Figure 2.9: The `Gate` class and its container (declared in `data-gate.hh`)

Fast access to certain type of information is required by the gateway. Several hash tables are available:

– `FedeSockList` is based on the `hash_map` template. The federate handle is used as a key to access the `Connection` to the federate.

– `ObjectSockList` is based on the `hash_multimap` template. The object ID gives access to the connections of the gateways or federates updating attributes of this object: a `requestObjectAttributeValueUpdate()` call can be forwarded quickly to the appropriate process.

– `ClassSockList` is available and could be used in the same way for the implementation of `requestClassAttributeValueUpdate()`.

– `RegionObjectList` is available but is not used at all!

2.5.2 The RTI gateway

The code and objects of the RTI gateway are defined in `rti-gate.cc`.

The main functions are the following.

– `init()` initializes the data structure, reads the FED, connects the gateway to the gateway network...

– `handleWANMsg()`, `handleWANConnection()` and `handleWANClosing()` are the callbacks associated to the WAN TCP connection;

– `handleLANMsg()`, `handleLANConnection()` and `handleLANClosing()` are the callbacks associated to the LAN TCP connection;

– `handleKbd()` executes the commands from the user's input line.

– `handleSRTP()` performs the filtering/forwarding functions for all the data coming from the LAN or from the WAN.

Several additional functions performs specific tasks and are described in the code itself.

2.6 The federation server

The federation server is a small program intended to create and destroy the federations. It maintains internally a list of federations and a list of remote servers.

The most important functions are:

– `init()` ;

– `handleWANMsg()`, `handleWANConnection()` and `handleWANClosing()` are the callbacks associated to the WAN TCP connection;

– `handleLANMsg()`, `handleLANConnection()` and are the callbacks associated to the LAN TCP connection;

– `handleKbd()` executes the commands from the input line;

– `launchFede()` and `killFede()` creates and destroys the federations.

Chapter 3

Implementation: Things you should know

3.1 The Application Programming Interface of the RTI

This implementation is based on the version 1.1 of the RTI specifications, but several modifications have been made to the Declaration Management and to the Data Distribution Management in order to prepare the migration to the version 1.2 and to fix inconsistencies.

These modifications have two kinds of consequences, on the API itself and on the behavior of several services.

3.1.1 What's new with version 1.2?

The version 1.2 should provide better definitions and a more consistent description of the RTI.

In-scope and out-of-scope objects or interactions are clearly defined in term of class subscription, class publication and overlapping regions.

The distinction between update and subscribe regions no longer exists. Once a region has been defined, it can be used as a publish and as a subscription region.

The Associate region service is described in terms of group of associations. An object/region pair has a set of attributes linked to it.

The role of the subclasses and superclasses in the publish, subscribe and discover mechanisms is now explained in greater detail.

3.1.2 Differences between this implementation and version 1.1

Exceptions thrown by the `unpublishObjectClass()` service no longer include `FederateOwnsAttributes`. Instead, the federate will lose ownership of all attributes associated with the specified object class, as requested by version 1.2.

The exception `ObjectClassNotPublished`, which is obviously missing, has been added.

The parameters of `subscribeObjectClassAttribute()` to subscribe an object class with a region now are a class handle, a region identifier and a set of attributes instead of a single attribute. This makes the call easier to use and allows us to mimic the behavior of the future *Subscribe Object Class Attributes with Region* service.

The RTI uses internally the `RegionAttributeList` class to implement the requested associations.

A new function `sendInteractionWithRegion()` which is missing in version 1.1, but will be added in version 1.2, is already available.

A new function `createRegion()` is implemented. It is strongly advised to use it, because the `createUpdateRegion()` and `createSubscriptionRegion()` will no longer exist in version 1.2.

Right now, the latest two functions just call `createRegion()` and no distinction is made internally between subscription and update regions. They can be used for both update and subscription.

A new exception `RegionIsUsed` can be thrown by `deleteRegion()` if the given region is still used for subscribing or updating.

The RTI types declaration file has been modified in several places.

In order to prepare the migration toward version 1.2, a new enumerated type `ObjectDiscoveryReason` is defined.

The different kinds of set used to manipulate attributes or parameters have been changed and they all look like the `AttributeHandleSet` class, without any pure virtual method, and with three extra methods: `encodedLength()`, `code()` and `decode()`. They still provide the same interface to the federate (the coding functions are intended to be use only by the RTI components who need to exchange information over the network), but are designed in a more standardized way.

The coding and decoding functions are also available for the `ThresholdSet`, `Extents`, `ExtentSet` and `EventRetractionHandle` classes.

3.2 Predefined constants

The `data-rti.hh` file declares several constant values.

Some of them can be described as "soft" limits. They can be changed, and even exceeded during the execution, the hash-tables being automatically resized as necessary.

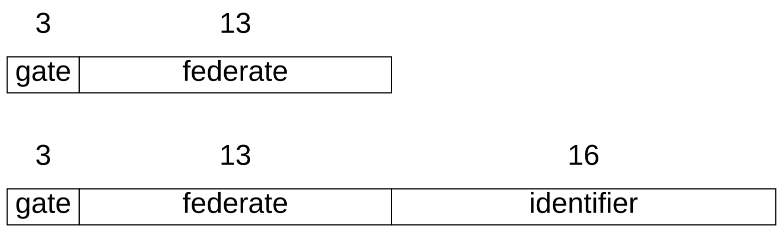

Figure 3.1: Federate handle and identifiers structure

Five constants, however, shouldn't be modified without a full understanding of the identifiers and handles attribution mechanism.

In order to guarantee that all federate handles are unique in a given federation, the first three bits correspond to the gateway number (according to the `.gate` file, see 3.3.2), and the lower thirteen bits correspond to the registration number. The maximum number of gateways `RTI_MAX_GATEWAY` is therefore $2^3 = 8$.

All the objects and regions defined by a federate have a unique identifier, whose sixteen first bits correspond to the federate handle. The lower sixteen bits correspond to a serial number. Therefore, a federate can not define more than `RTI_MAX_ID` objects or regions, with `RTI_MAX_ID` equal to $2^{16} - 1 = 65535$.

The message serial numbers and the event retraction handles are build in the same way, but the old values can be reused, because they are used along with the federation time: this guaranty the ID/time association to be unique.

If the sixteen first bits of an identifier are null, the identifier is global. This is the case for the default region identifier `ALL_REGION`.

SRTP provides 255 different categories. Categories 1 to 127 are used for object classes and categories 129 to 255 are used for interaction classes. The first bit shows whether it is an object or an interaction, and the maximum numbers of classes in both case is 127. `RTI_HANDLE_MASK` is used to extract the HLA class from the SRTP category.

3.3 Initialization files

All the initialization files must be in the directory specified by the `RTI_CONFIG` environment variable, or in the current directory if this variable is not set.

3.3.1 Federation server initialization

The list of Federation servers is described in a file used as initialization data by all the servers. This file must be the same for all of them. The default file is called `server.lst` and is in the `RTI_CONFIG` directory, but another file can be given as an argument to the server.

Each line contains the name of the server host (as returned by `gethostname()`), the port number for the WAN connection and the port number for the LAN connection. The first port number is used by the servers to connect to each other. The second one is used to receive the local creation and destruction requests.

The order of the lines is important: each server is going to connect to all the servers of the list in the mentioned order until it reaches its own line. That way, a fully connected network is defined with a precedence order.

3.3.2 Federation gateway initialization

For a given federation, two initialization files are necessary:

– `<federationName>.gate` is the list of gateways and has the same format at the list of servers. The set of hosts is a subset of the set of hosts in the server file. The precedence order is used to assign the gateway number.

– `<federationName>.local` describes all the networking parameters for the federation. It is specific to a given LAN.

The following is an example of a `.local` file:

```
# GATE host wan_server_port lan_server_port wan_gate_port lan_gate_port
GATE descartes 10000 20000 14000 24000

# DEFAULT_MCG wan_default_mcg lan_default_mcg
DEFAULT_MCG 227.1.1.10 227.1.1.11

# MCG list_of_mcg
MCG 227.1.1.12 227.1.1.13 227.1.1.14 227.1.1.15 227.1.1.16 227.1.1.17
MCG 227.1.1.18 227.1.1.19 227.1.1.20 227.1.1.21 227.1.1.22 227.1.1.23
MCG 227.1.1.24 227.1.1.25 227.1.1.26 227.1.1.27

# Time To Live
WAN_TTL 32
LAN_TTL 1

# Number of mode 0 sent to ensure reliable communications
NB_MODE0 2
```

The WAN and LAN port number must be the same as in the server and gateway initialization files. The TTL fields are optional. The multicast group list can be split on several lines, and may be omitted if the federates are not using the Data Distribution Management services.

3.3.3 Federation Execution Data file

Concerning interaction and object classes, the FED file has the LISP-like format described in the official documentation.

As nothing has been specified so far about the description of the routing space in the initialization file, I had to choose a syntax for this implementation of the RTI. However, it should not be very different from the still-to-be-published official one.

The following is an example of a FED file for this RTI:

```
( fed

  ; list of objects, with the attributes and the subclasses
  (objects
   ( class <name>
     ; the attributes must come first...
     (attribute <name> <transportation> <ordering>)
     ; ... then the subclasses
     ( class <name>
       (attribute <name> <transportation> <ordering>)
     )
   )
   ( class <name>
     (attribute <name> <transportation> <ordering>)
   )
  ) ; end of the objects list

  ; list of interactions, with the parameters and the subclasses
  (interactions
   ( class <name> <transportation> <ordering>
     ; the parameters must come first...
     (parameter <name>)
     ; ... then the subclasses
     ( class <name><transportation> <ordering>
       (parameter <name>)
     )
   )
   ( class <name>
     (parameter <name>)
   )
  ) ; end of the interactions list

  ; here is the description of the routing spaces
  ( routing_spaces
    ( space <name> <number of dimensions>
      ; for each dimension, the range which is going to be used
      (<first> <last>)
      (<first> <last>)
    )
    ( space <name> <number of dimensions>
      (<first> <last>)
    )
  ) ; end of the routing space list

) ; end of the fed

; any further information will be ignored
```

Each routing space is described with its name, the number of its dimension, and the range of meaningful values on each dimension.

It seems reasonable to assume that only a given interval of the available values on each dimension is going to make sense for the simulation. That is why the description of the routing space includes a range for each dimension.

However, it is not really satisfactory to have these numerical values in the FED file, since they are not general, but rather related to a given federation execution. But I did not find any other way to provide these values to the RTI. This part will have to be changed as soon as the new version of the RTI specifications is available.

Chapter 4

Future Work

4.1 Completing and improving the actual implementation

The major flaw of this RTI is the poor runtime performances (at most 80 attribute updates per second against several hundreds updates per second for the RTI F.0). This is moslty due to the fact that we have to handle four different kinds of transport services and the DDM: the RTI must sort out the attributes and build different sets, one per transport type and multicast group (see 2.4.1 on page 15). The SRTP, which is slower than UDP, also increases the overhead.

Improving the performances of the RTI must be the main priority in the future, if we want to maintain and develop this implementation of the RTI.

To increase the robustness of the RTI gateway, it would be necessary to complete the `removeFederate()` function in the rti-gate. If the federates exits normally, the actual version is sufficient, but if a federate crashes, it is necessary to "clean" everything, delete its objects and regions, unsubscribe and unpublish the classes it has subscribed and published.

The minimum rate transport service is not available yet, because we need to send the update with the actual time simulation. The best solution is of course to use the Time Management service, but it has not been implemented yet.

A short-term solution for real-time simulation, if we assume that the federation time unit is second, is to use the computer's clock and the `lastMinimumRate` field of the object. If the last update occured at T with a federation time ft, we can send an update at $T + \delta t$ with federation time $ft + \delta t$. Since we only use an offset and no absolute time, no synchronization mechanism like NTP is required.

4.2 Adapting the RTI to the new specifications

It is an obvious and necessary evolution to adapt the RTI to the latest specifications. This evolution mainly concerns the Declaration and Data Distribution Management services

Improving the interoperability between federates which use and do not use the DDM is an important task.

Actually, the default region, whose ID is `ALL_REGION`, is associated to the default multicast group. If a federate subscribes to a class without using the DDM, it will subscribe to the information send on this multicast group only. If another federate is using the DDM and sending data on specific multicast groups, the first federate will never receive it, altough it has subscribed to it.

The management of the default region must be therefore modified. It will be appropriate to do this while adapting the RTI to a new feature of the specifications: the attributes and interactions will be associated in the FOM to a given routing space.

It will be interesting to have several default (region, multicast group) pairs, one for each routing space. A subset of multicast groups would be associated to each routing space (actually, the same groups are used for the different spaces).

The default region should be managed differently, beeing associated to the default multicast group for the update and to all the multicast groups associated to the routing space for the subscription. This must be done carefully, in order to keep the reference counts consistent (see 2.2.2 and 2.3.3).

The Incorporation of a Management Object Model is not necessary for DM and DDM, but may be a usefull improvement. It is expected anyway that the MOM will be part of the RTI sooner or later.

4.3 Adding new services

An easy to implement service is `requestClassAttributeValueUpdate()`. The necessary data structures is available, and the implementation is just an adaptation of the already existing `requestObjectAttributeValueUpdate()` service.

The Time Management and Ownership Management have not been implemented, since it was not our primary objective, but could be a valuable extension to the actual software.

Bibliography

[FP97] Kenneth E. Frosch and J. Mark Pullen.
"Design and Prototype of a Dual Mode Multicast Application Gateway".
In *1997 Spring Simulation Interoperability Workshop*. Simulation Interoperability Standards Organization, March 1997.

[HLA96] "High Level Architecture Rules Version 1.0".
DMSO, August 1996.
http://www.dmso.mil/projects/hla/tech/rules/RULES1-0.PDF.

[HLA97a] "High Level Architecture Interface Specification Version 1.1".
DMSO, February 1997.
http://www.dmso.mil/projects/hla/tech/ifspec/ifsp11.pdf.

[HLA97b] "High Level Architecture Object Model Template Version 1.1".
DMSO, February 1997.
http://www.dmso.mil/projects/hla/tech/omtspec/omtv11.pdf.

[HLA97c] "HLA Data Distribution Management: Design Document Version 0.5".
DMSO, February 1997.
http://www.dmso.mil/projects/hla/tech/ifspec/ddm_05.pdf.

[PL95] J. Mark Pullen and Vincent P. Laviano.
"A Selectively Reliable Transport Protocol for Distributed Interactive Simulation".
In *13th Workshop on Standards for the Interoperability of Distributed Simulations*, volume 1, pages 651–654, paper number 95-13-102. Institute for Simulation and Training, University of Central Florida, September 1995.

[PL96] J. Mark Pullen and Vincent P. Laviano.
"Prototyping the Selectively Reliable Transport Protocol".
In *14th Workshop on Standards for the Interoperability of Distributed Simulations*, volume 1, pages 585–589, paper number 96-14-089. Institute for Simulation and Training, University of Central Florida, March 1996.

[PLM97] J. Mark Pullen, Vincent P. Laviano, and Marc O. Moreau.
"Creating a light-weight RTI using selectively reliable transmission as an evolution of dual-mode multicast".
In *1997 Fall Simulation Interoperability Workshop*, paper number 97F-SIW-149. Simulation Interoperability Standards Organization, September 1997.

[PW95] J. Mark Pullen and Elizabeth L. White.
"Dual mode multicast for DIS".
In *12th Workshop on Standards for the Interoperability of Distributed Simulations*, volume 1, pages 505–510, paper number 95-12-087. Institute for Simulation and Training, University of Central Florida, March 1995.

Index