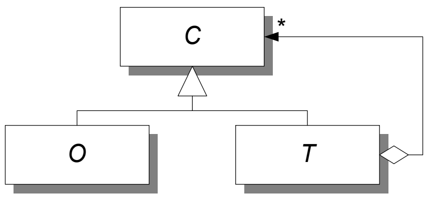
How to become an HLA guru in a short(er) time

COT/6-6-V1.0



Centre for Object Technology

Revision history:	1	980519	Release 1.0

Author(s):	Peter Toft, DMI
Status:	Release
Publication:	Public
Summary:	

LIST OF CONTENTS:

1. Outline of this document

- **Introduction**
- **Obtaining HLA information and programs**
- **Overview of available documents**
- **Simple concepts of HLA** and what documents are relevant for the HLA beginner
- **Running the first HLA program** (helloWorld) running on one machine. What support utilities should be running etc.
- **Designing SOM using the OMDT**
- **Configuration files** The FED file and the RTI.rid
- **Simple example with two classes**
- **Some introduction to three simulators build at DMI**
- **HLA FAQ** - questions and answers. Simple elements and style

2. Introduction

This document tries to provide a lot of information regarding HLA (High Level Architecture) in order to advance the programmer unfamiliar to HLA towards the core of HLA.

HLA, also known as DIS++, is a novel way to build simulator systems. HLA is not a area that you can learn in a day. HLA involves climbing a high learning curve, however it is worthwhile. HLA is an impressive piece of work. HLA features many fine properties, such as dynamic discover and delete functionality of the sub-systems participating in the simulation and dynamic ownership handling. The major aim of HLA is to standardize the interconnection between simulators, hence provide easy ways of connecting simulators running HLA. In December 1997, HLA was accepted as a draft IEEE standard.

RTI versions 1.0.* are government-controlled development. RTI 2.0 will be industry developed, using an open competitive design process. Currently version 1.0.3 is out for the following platforms

- Silicon Graphics (IRIX > 6.2)
- Sun Sparc (Solarix > 2.5+)
- Hewlett Packard (HP-UX > 10.20)
- IBM (AIX > 4.1.5)
- DEC Alpha (OSF1 > V4.0)
- Windows (95/NT 4.0)

and version 1.3 is out for Sun Sparc machines. Version 1.3 should have implemented all announced services, however version 1.0.3 is stable and equipped with functions sufficient for most common simulators.

3. Obtaining HLA information and programs

HLA has it own homepage at **http://www.dmso.mil/hla** from which it is possible to download HLA documentation and programs for RTI version 1.0.3 for free (however it is not Open Source). After an approval you can download the software, note that the development of the HLA software is a military controlled project, hence they do register the sites who get the software.

3.1 Download

Initially choose the item **HLA Software Distribution Center** and register, next (normally after a day) you should receive the password for entering the **Download** item. In that area choose the RTI (HLA) software for the appropriate platform and download. You should also download the installation document and the Programmer’s Guide. Note that DMI has a valid login already (Ole Vilmann is a registered user).

3.2 Installation

From the installation document it is easy to install the RTI. On any Unix or the NT platform the self extracting program package should be executed and it installs under a given path. Note that it will create a **$RTI_HOME** directory named rti under the chosen path containing all program parts.

The environment variable **$RTI_HOME** should be set to the path as well as **$RTI_CONFIG** set to **$RTI_HOME**/config (convert slash to backslash on NT). Furthermore on a Unix machine the **$RTI_HOME**/rti.env should be sourced from the users login files.

For Visual C++ the **%RTI_HOME%**\lang\C++\include and **%RTI_HOME%**\lang\C++\lib should be a part of the search path for include and libraries respectively.

Before running programs it is very sound to obtain information about the basic terminology and some knowledge about HLA in general.

4. Simple concepts of HLA

First some words about HLA terminology

- **High Level Architecture (HLA)**
  HLA is not only a simulator architecture. It contains specifications about
  - how to operate simulators together; how and what data to communicate.
  - specifications about the individual simulators, what services should be offered by a simulator and how the simulator works.
  - An skeleton for designing HLA compliant simulators.
  - A test and verification scheme.
- **Runtime Infrastructure (RTI)**
  The general purpose distributed operating system software which provides the common interface services during the runtime of an HLA simulation. The RTI can be conceived as a post office for all data traffic in the simulator. Note that all information that change during a simulation, that needs to be shared between several simulators needs to pass though the RTI.
- **Object Model Template (OMT)**
  Template used to describe an object. For HLA specification a tool, called OMDT, which can be downloaded from the HLA homepage. The OMDT which enforces specification of objects in the OMT framework. This tool is currently only available for Windows NT.
- **Federation Object Model (FOM)**
  An identification of the essential classes of objects, object attributes, and object interactions. The FOM does not contain information about the actual objects in the simulation (the federates), but only about the possible object classes in the simulation.
- **Simulation Object Model (SOM)**
  A specification of the intrinsic capabilities that a simulator[1] offers to federations. In each SOM a set of attributes (parameters) are listed by type, cardinality, units and specification about how they are updated. The SOM can be compared with a class description, hence several federates can share a given SOM, if they are alike in specification, e.g. all F16 fighter simulators could have the same SOM.
- **Federate**
  A member of a HLA Federation. A federate is an actual simulator with capabilities specified by its SOM. Note that several federates can have the same SOM.
- **Federation**
  A named set of actual interacting federates (all actual simulators), a common federation object model (the FED file), and supporting RTI, that are used as a whole to achieve some specific objective.
- **Class**

[1] Note the HLA uses the term simulator as DMI uses the term subsystem.

Each federate is part of a class, which is the SOM. The C++ concept of a Class map almost perfectly onto the SOM.

- **Attribute**

An attribute is a parameter with one or several values in a SOM that can be changed during a simulation. Other federates may or may not desire to get changes of the attribute depending on the actual simulator settings. The federate who offers the attributes to others will **publish** an attribute and the federate that needs the attribute changes will **subscribe** to the attribute. Examples will be given later regarding how this is done.

- **Interaction**

Ordinarily the fundamental way data is communicated from one simulator to an other is via changes in attributes. The other supported way is via interactions. The intended usage is for orders, i.e. data communication that is only relevant to that very instance in time. The interaction is in principle not stored, for later retrieval. It is possible to use interaction for many things, such as offering a service close to function calls in C++.

The world of HLA contains a rich and consistent terminology, and a more thorough list can be found at http://www.dmso.mil/hla/rti/API.html.

In Figure 1 is shown a typical HLA simulation. It is seen that several federates are running together with two standard support utilities: The **rtiexec** is a daemon program that is started on the server machine. It supports any HLA simulation on the machine including disjoint simulators (e.g. combat and industrial without interconnection). The **fedex** is a process that supports a given federation. It is started automatically by the rtiexec when the first federate joins the federation (a shell will pop up displaying the actual federates).

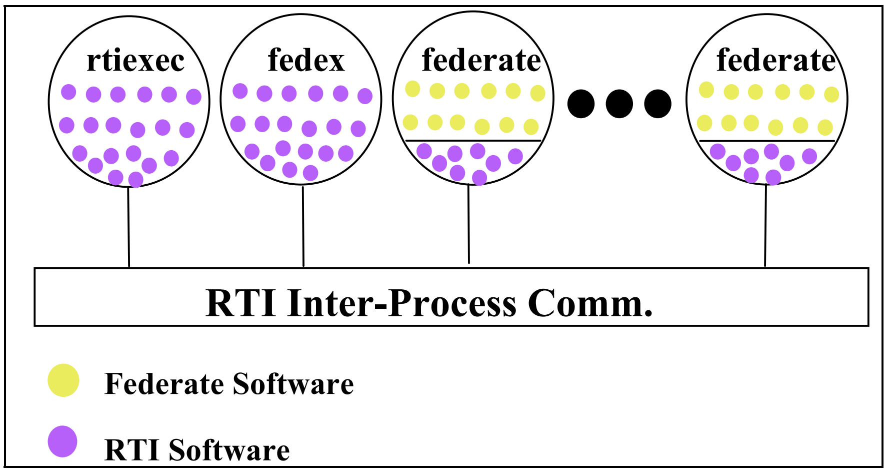


Figure 1

4.1 The Ten Basic HLA Rules

HLA is based on five rules for the federation and five rules for the federates.
In bold are shown official HLA rules, and italic letters for my comments.

4.1.1 Five rules for federations

1. **Federations shall have an HLA federation object model (FOM).** *A FED-file will contain the FOM information necessary.*
2. **In a federation, all simulation-associated object instance representation shall be in the federates, not in the runtime infrastructure (RTI).** *Data cannot be stored in the RTI. The RTI will only route data!*
3. **During a federation execution, all exchange of FOM data among federates shall occur via the RTI.** *It is highly illegal to send data that changes over time directly between simulators. This rule is made in order ensure stability and consistency of the simulation.*
4. **During a federation execution, federates shall interact with the RTI in accordance with the HLA interface specification.** *All interactions must follow the HLA specifications.*
5. **During a federation execution, an attribute of an instance of an object shall be owned by at most one federate at any time.** *Ownership handling is also supported in HLA and an object (a simulator) can at most be owned by one simulator.*

4.1.2 Five rules for federates

1. **Federates shall have an HLA simulation object model (SOM).** *The OMDT will make this documentation showing all attributes, their types and sizes.*
2. **Federates shall be able to update and/or reflect any attributes of objects in their SOMs and send and/or receive SOM interactions externally, as specified in their SOMs.** *The SOM must specify that attributes will be updated given certain conditions, e.g. a defense system may be initiated only if an enemy is within a certain distance.*
3. **Federates shall be able to transfer and/or accept ownership of attributes dynamically during a federation execution, as specified in their SOMs.** *The SOM should document the conditions of ownership handling.*
4. **Federates shall be able to vary the conditions (e.g., thresholds) under which they provide updates of attributes of objects, as specified in their SOMs.** *In version 1.3 of the RTI software it is possible to provide attribute changes if certain logical function is fulfilled. For version 1.0.3 this is yet to come.*
5. **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.** *Time control is of course very important. HLA offers several ways that can be mixed. The important issue is that time in a simulator must be consistent, i.e. have a increasing flow.*

5. HLA infrastructure

The High Level Architecture Run-Time Infrastructure Programmer's Guide mentioned on page 4 is worthwhile to read. It is found a the HLA download web page - currently also stored at **pc0100\\Adm\Adm\Data\Project\98853-COT\Misc\HLA Programmers Guide 1.0.3.doc**

However this document aims to provide an overview that is hard to obtain from the available documents. After reading this document I suggest that the reader start working with actual programs, before reading the Programmers Guide (It is rather complex for the beginner).
One thing should be read early in the Programmers Guide, namely Section 2.4: Configuration and Input Files.

Here the **RTI.rid** (found in **$RTI_CONFIG**) is explained. That file is a global file that is used primarily by the **rtiexec** to determine which RTI server machine to use. For the server machine uses the line

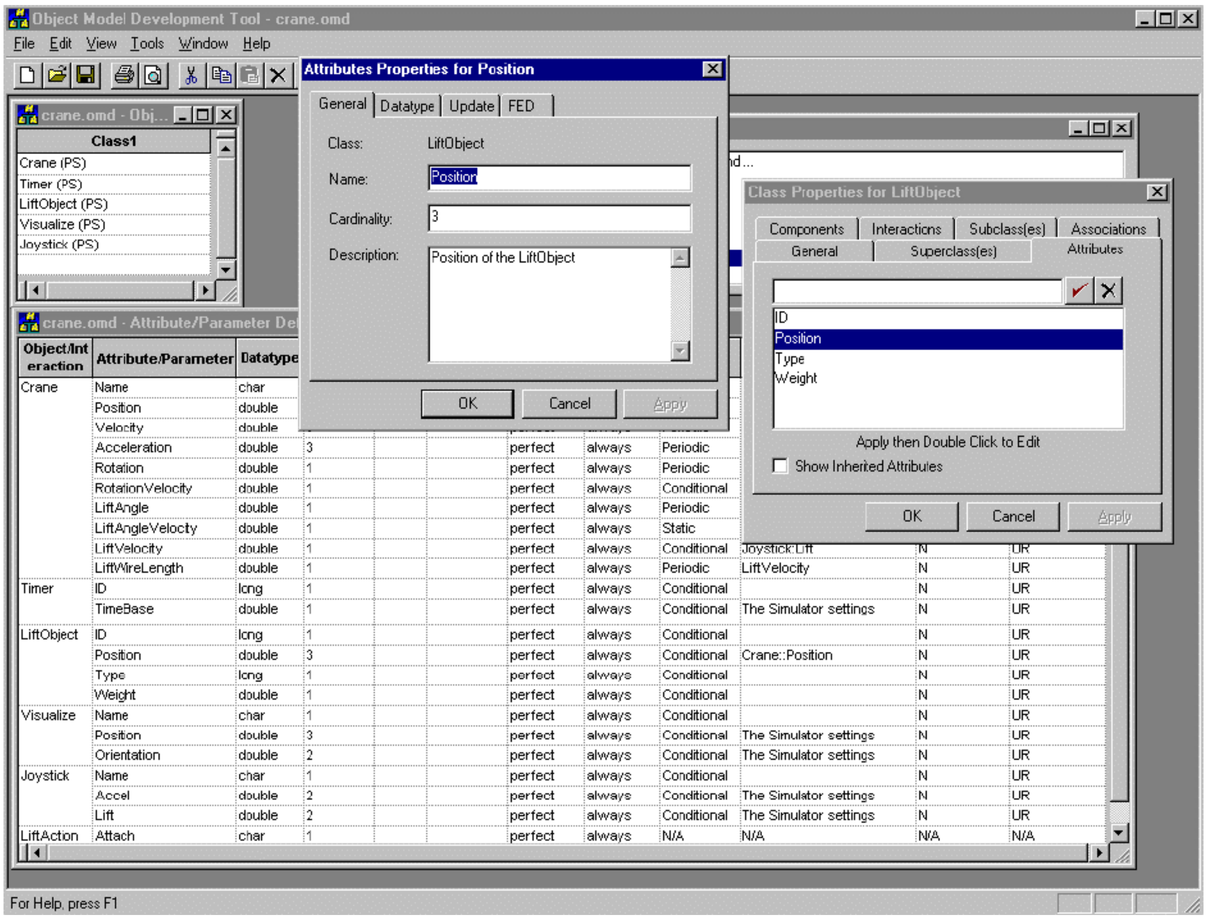
```
RTI_EXEC_HOST           localhost
```

and any client machine should change the line to

```
RTI_EXEC_HOST            my_rti_servermachine
```

where `my_rti_servermachine` should be set to the name of the actual server machine. In the same file, the port used for the RTI is set (`RTI_EXEC_PORT`). The default is 18134. Besides the `RTI_EXEC_HOST` all other parameters are ordinarily left unchanged.

It should be noted that the FED-file, which contains the FOM information, can (should?) be generated by the OMDT tool, found at the HLA download web page. In the next picture is shown a screen dump showing the OMDT tool on a NT machine (only available for NT). It is quite easy to design the classes, the attributes including their types etc., and then the FED-file can be stored directly using a save-as operation.
The OMDT can verify that the model is consistent and well documented, but it cannot generate actual code. However it is a good for defining a common data model and to guide the implementation of the HLA code. The OMDT has rather easy documentation.


Object Model Development Tool - crane.omd
File Edit View Tools Window Help
crane.omd - Obj...
crane.omd - Attribute/Parameter De
Attributes Properties for Position
General Datatype Update FED
OK Cancel Apply
Class Properties for LiftObject
Components Interactions Subclass(es) Associations
General Superclass(es) Attributes
Apply then Double Click to Edit
Show Inherited Attributes
OK Cancel Apply
For Help, press F1

Class1
- Crane (PS)
- Timer (PS)
- LiftObject (PS)
- Visualize (PS)
- Joystick (PS)

Class: LiftObject
Name: Position
Cardinality: 3
Description: Position of the LiftObject

ID
Position
Type
Weight

Object/Interaction	Attribute/Parameter	Datatype									
Crane	Name	char									
	Position	double									
	Velocity	double									
	Acceleration	double	3			perfect	always	Periodic			
	Rotation	double	1			perfect	always	Periodic			
	RotationVelocity	double	1			perfect	always	Conditional			
	LiftAngle	double	1			perfect	always	Periodic			
	LiftAngleVelocity	double	1			perfect	always	Static			
	LiftVelocity	double	1			perfect	always	Conditional	Joystick:Lift	N	UR
	LiftWireLength	double	1			perfect	always	Periodic	LiftVelocity	N	UR
Timer	ID	long	1			perfect	always	Conditional		N	UR
	TimeBase	double	1			perfect	always	Conditional	The Simulator settings	N	UR
LiftObject	ID	long	1			perfect	always	Conditional		N	UR
	Position	double	3			perfect	always	Conditional	Crane::Position	N	UR
	Type	long	1			perfect	always	Conditional		N	UR
	Weight	double	1			perfect	always	Conditional		N	UR
Visualize	Name	char	1			perfect	always	Conditional		N	UR
	Position	double	3			perfect	always	Conditional	The Simulator settings	N	UR
	Orientation	double	2			perfect	always	Conditional	The Simulator settings	N	UR
Joystick	Name	char	1			perfect	always	Conditional		N	UR
	Accel	double	2			perfect	always	Conditional	The Simulator settings	N	UR
	Lift	double	2			perfect	always	Conditional	The Simulator settings	N	UR
LiftAction	Attach	char	1			perfect	always	N/A	N/A	N/A	N/A

5.1 The first example of an HLA simulator

If one was to construct a simple HLA simulator, which should illustrate the core of HLA, one example could be a Driver and Policeman federation. A number of drivers is running at individual speeds and a federate of class Policeman will monitor the speed of the drivers and compare them to a speed limit. If they speed the PoliceMan will send an interaction to the driver to stop. The Driver should then subscribe to the SpeedLimit and maybe also to the Position of the Policeman. The implications are rather obvious.

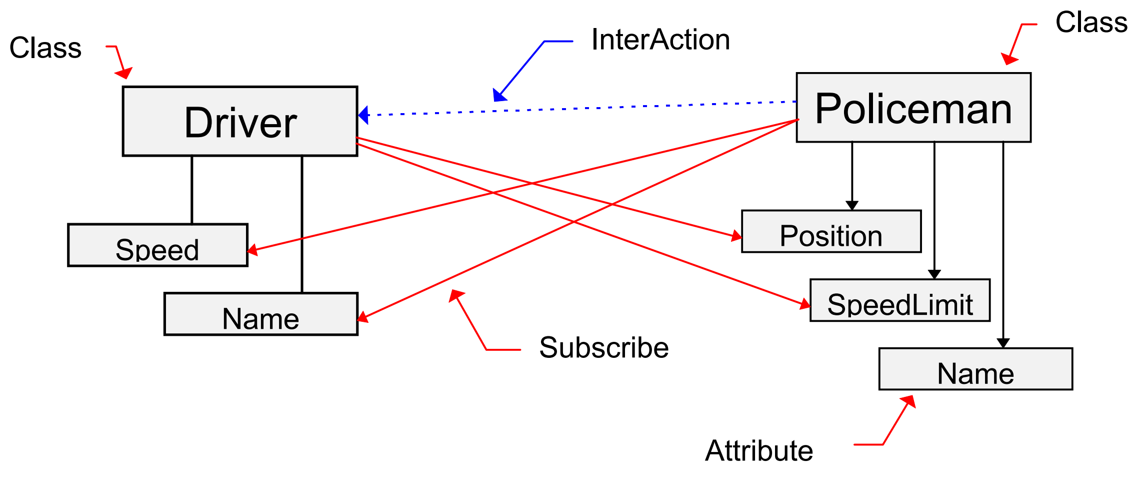
Initially a FED-file must be made. The Programmer's Guide explains the FED-file, primarily used by the **fedex**. For every simulation all federates should agree on the SOM and attribute names, hence the FED-file should be copied to the **$RTI_CONFIG** directory on all participating machines.

For a simple two class example the file, here called **driver.fed** could look like (many lines with standard services have been removed).

```
;; -----------------------------------------------------------
;; Federation Execution Data (FED)
;; -----------------------------------------------------------
(fed
  ;; -----------------------------------------------------------
  ;; object, class, and attribute definitions follow
  ;; -----------------------------------------------------------
  (objects
    (class Driver
      (attribute Name            FED_RELIABLE     FED_RECEIVE)
      (attribute Speed           FED_RELIABLE     FED_RECEIVE)
    )
    (class Policeman
      (attribute  Name           FED_RELIABLE     FED_RECEIVE)
      (attribute  Position       FED_BEST_EFFORT  FED_RECEIVE)
      (attribute  SpeedLimit     FED_RELIABLE     FED_RECEIVE)
   )
   )
  ;; -----------------------------------------------------------
  ;; interactions, class, and parameter definitions follow
  ;; -----------------------------------------------------------
  (interactions
    (class Communication         FED_RELIABLE     FED_TIMESTAMP
      (parameter InterAction)
    )
  )
)
```

The idea is that for each object class (SOM) the name of the Class (Driver and Policeman respectively) and each of the attribute names are listed. The same applies for the interaction classes. Note that the `FED_RELIABLE` means that the attribute changes MUST be send to the federates that desire this this. On the other hand if an attribute is not critical but should most of the time be transmitted, then `FED_BEST_EFFORT` can assist the RTI in the usage of time. If the RTI has to pass a huge amount of data, then these "best effort" attributes can be dumped.

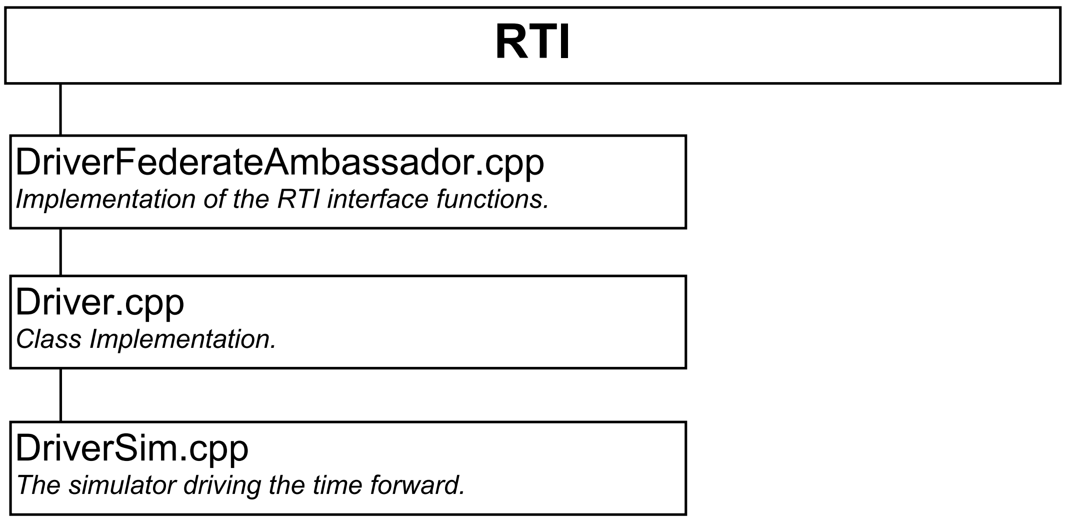
Assume that federate A and B is of class Driver and federate C is of class Policeman. The Policeman federate subscribes to all Speed attributes of the Driver classand he will get the speed of A and B, whenever changed.
If Speed exceeds SpeedLimit then C will send an interaction InterAction of Interaction class Communication to the driver. All federates should subscribe to this interaction class. This is indicated in the following diagram.



5.2 C++ structure of a simulator

In the Driver/Policeman example the implementation is often split in three parts:

- A federate ambassador which contains a fixed set of functions that can be called from the RTI.
- The class implementation of the internal structure that defines the class.
- A small simulator, which primarily initializes the simulator and controls the time advancements.


RTI
DriverFederateAmbassador.cpp
Implementation of the RTI interface functions.
Driver.cpp
Class Implementation.
DriverSim.cpp
The simulator driving the time forward.

5.3 The Federate Ambassador

The federate ambassador contains a fixed set of virtual C++ functions, that are called from the RTI. (Additional ones for ownership handling, and pause handling are also available).

- **discoverObject** When an object is initialized and recognized by the RTI, then the RTI will signal the other subscribing federates, that the object if online.
- **removeObject** Messages that a specific federate now has left the federation.
- **startUpdates** Orders the federate to start delivering attribute values whenever changed. The information is specified at class and attribute level.
- **stopUpdates** Orders the federate to stop delivering attribute values. The information is specified at class and attribute level.
- **reflectAttributeValues** Called by the RTI with a set of new attribute values from other federates, that was subscribed to. This function will not normally call an Update function in the relevant federate class.
- **provideAttributeValueUpdate** Called by the RTI instructing the federate to deliver the value of a given attribute.
- **startInteractionGeneration** Orders the federate that other federates are ready to receive specified interactions.
- **stopInteractionGeneration** Orders the federate that none of the other federates will receive specified interactions.
- **receiveInteraction** Called by the RTI with a set of new interactions.

5.4 The core of the Driver class

The Driver class would in C++ look something like:

```
class Driver
{
  public:
  Driver();
    Driver( RTI::ObjectID  id);
  Driver( int ID);
    ~Driver();
  static Driver* Find( RTI::ObjectID objectId );
    static void    Init( RTI::RTIambassador* rtiAmb );
    void           PublishAndSubscribe();
    void   Update(RTI::FederationTime& newTime );
    void   Update(RTI::AttributeHandleValuePairSet& theAttributes );
    void   Update(RTI::InteractionClassHandle theInteraction,
                     RTI::ParameterHandleValuePairSet& theParameters );
    void     SetSpeed( const double & );
    void     SetName( const char* );

private:
  char * driver_Name;
  double driver_Speed;
};
```

Besides a set of constructors and a destructor and functions to set attributes (SetSpeed and SetName) the three Update functions are the core of the HLA class.

- **`Update(RTI::FederationTime&)`**
  The function is called by the simulator control (DriverSim) when the time is advanced to the specified time. This function should then integrate and update the internal variables to the that time.
- **`Update(RTI::AttributeHandleValuePairSet&)`**
  The function is called by the federate ambassador with a set of new attribute values. The function should then update the internal set of variables according the attribute values. Some of which will be directly copied to internal variables, while others might indirectly imply changes.
- **`Update(RTI::InteractionClassHandle,RTI::ParameterHandleValuePairSet&)`**
  This function is the analog of the previous one, but for interactions. The function will update internal variables. The concept of interactions (compared to attribute changes) intended for messages that only have relevance at the given point in time.

5.5 Subscription to attributes

All communication with the RTI uses type ID’s, which is unique number for each of the class attributes found in the SOM specifications.
This is normally made in the Init function which is a part of the class.
The following piece of code shows that from a set of class names and attribute names (Driver and Speed/Name respectively) a set of RTI ID’s are obtained.
Subsequently the intention to receive changes in the given attributes from other federates and the intention to publish the same set of attributes to other federates are declared. The same is done for the interactions.

```
// First get the RTI ID's for the Driver objects
driver_DriverTypeId  =  driver_rtiAmb->getObjectClassHandle("Driver");
driver_SpeedTypeId = driver_rtiAmb->getAttributeHandle("Speed","Driver");
driver_NameTypeId =  driver_rtiAmb->getAttributeHandle("Name","Driver");

RTI::AttributeHandleSet *DriverAttributes;
DriverAttributes = RTI::AttributeHandleSetFactory::create(2);
DriverAttributes->add(driver_NameTypeId);
DriverAttributes->add(driver_SpeedTypeId);
driver_rtiAmb->publishObjectClass(driver_DriverTypeId, *DriverAttributes );
driver_rtiAmb->subscribeObjectClassAttribute(  driver_DriverTypeId,
                                               *DriverAttributes );
DriverAttributes->empty();
delete DriverAttributes;   // Deallocate the memory

// Get RTI ID’s for Interactions
driver_CommunicationTypeId
 =driver_rtiAmb-getInteractionClassHandle("Communication");

driver_ActionTypeId = driver_rtiAmb->getParameterHandle( "Action",
                                                         "Communication");

driver_rtiAmb->subscribeInteractionClass( driver_CommunicationTypeId );
driver_rtiAmb->publishInteractionClass( driver_CommunicationTypeId );
```

5.6 Update an attribute from others

One of the most central parts of an HLA simulator is the Update function that receives new attribute values. The core of such a function looks the following lines of code. Note that several attribute values might be available in parameters to the function. For interactions a very similar Update function should be made.

```
void Driver::Update(const RTI::AttributeHandleValuePairSet& theAttributes )
{
   RTI::AttributeHandle attrHandle;
   RTI::ULong           valueLength;

   for ( unsigned int i = 0; i < theAttributes.size(); i++ ) {
      attrHandle = theAttributes.getHandle( i );
      if ( attrHandle == driver_NameTypeId ) {
         char name[ 1024 ];
         theAttributes.getValue( i, (char*)name, valueLength);
         name[ valueLength ] = NULL;
         SetName( name );
      }
      if ( attrHandle == driver_SpeedTypeId() ) {
      double* speed = new double[1];
      theAttributes.getValue( i, (char*)speed, valueLength );
      SetSpeed(speed);
      delete speed;
   }
}
```

5.7 Main loop of a simulator

The main time loop of the simulator is controlled by a small loop, that requests for a signal to allow update of the current time with a given time step. During the time the simulator spends in the rtiAmb.tick() function the simulator will have calls from the RTI with updates of attributes and interactions. When the signal to allow the a time increment is obtained then this simulator will merely print the time and the speed of the driver. Subsequently the simulator will call the last of the three Update functions to actually update the state of the federate.

```
while ( counter++ < numberOfTicks ) {
      timeAdvGrant = RTI::RTI_FALSE;
      rtiAmb.timeAdvanceRequest(currentTime + timeStep);
   while (!timeAdvGrant) {
      int eventsToProcess = 1;
      while ( eventsToProcess ) {
            eventsToProcess = rtiAmb.tick(time_min, time_max);
      }
   } //If beyond here then we can advance the time
      cout << " Time = " << currentTime << " Speed = "
            << myDriver->GetSpeed();
   myDriver->Update( grantTime );
   currentTime = grantTime;
}
```

Note that often `eventsToProcess = rtiAmb.tick();` will be used, however this will consume a lot of the CPU power. Using the two parameter version of the tick function is normally far cheaper `eventsToProcess = rtiAmb.tick(time_min, time_max);`

where time_min and time_max are the minimal and maximal time the function spends in that function measured in seconds. I have often used

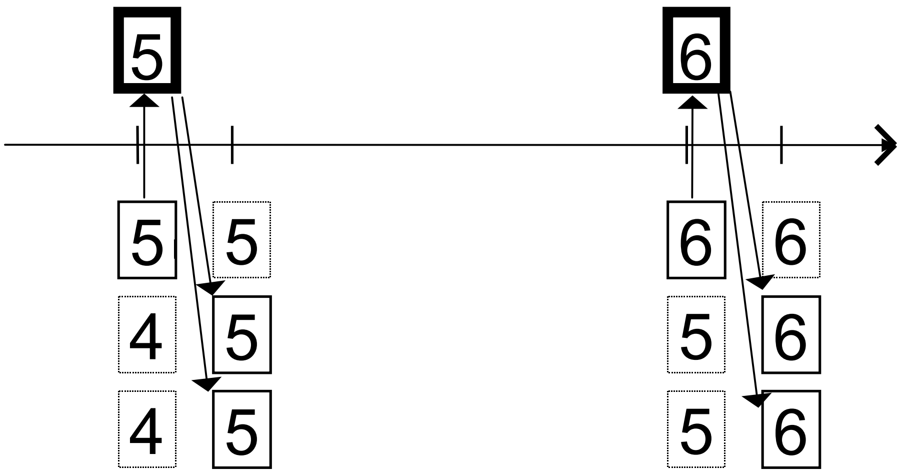
```
eventsToProcess = rtiAmb.tick(0.01, 0.02);
```

which leads to a very low CPU load (2-5 percent).

5.7.1 Simulator time control

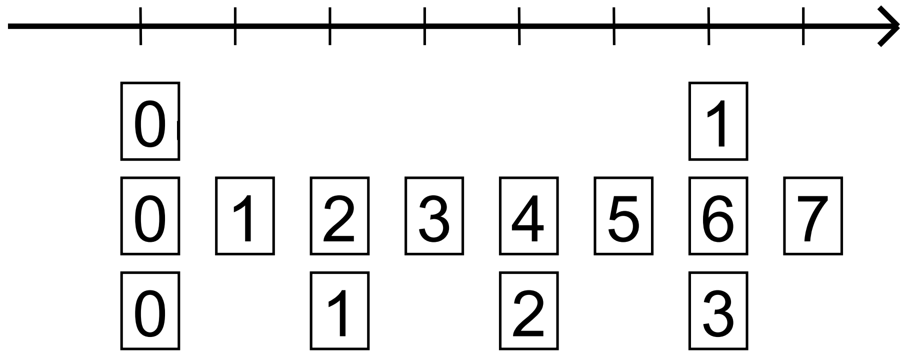
HLA offers several ways to update the time. Any federate can or cannot be **TimeConstrained** and can or cannot be a part of the **TimeRegulation**.
A TimeConstrained federate cannot advance its internal time until RTI allows it. This is used for simulators that should advance time almost simultaneously. Federates that are not TimeConstrained are always allowed to increment time, and changes due to new attributes are as fast as possible. Federates that are a part of the TimeRegulation will block TimeConstrained federates until they too allow time increments to the same time point.

The concept of time Regulation sets whether a federate should be also determine when the time is advanced. When *all* time regulating federates desire to advance the time, then they get the go signal. In the following diagram is shown that the first row below will advance its time hence setting the system time that the remaining federates will subsequently get notification of. One or more federates can be the time controlling systems, and then the slowest federate will determine when the time is incremented. In this scheme interactions and attribute changes have a time stamp. An important note is that the CPU load is highly controlled in the Simulator main program. Read the comments below in subsection



Time management in HLA is a well established. A lot of theory and work has been put into this. A thorough (and difficult) treatment of time management can be found in the document
HLA Time Management Design Document, Version 1.0, 15 August 1996. The document is available at DMI and apparently no longer on the Internet. Furthermore the document named
High Level Architecture Run-Time Infrastructure Programmer's Guide, Version 1.3, 27 March 1998, available from the DMSO HLA homepage does has a lot of theory and examples regarding time management.

Another possible way to control time is that the simulators are allowed to advance their time at will. In that situation the time stamps are not that relevant, and every new attribute or interaction will be treated as fast as possible.



The two time control schemes can be mixed for obtaining maximum performance and flexibility. Note that the variables `timeAdvGrant` and `grantTime` must be made as global variables in the simulator program. Both are used in the federate ambassador functions.

6. How to run HLA programs

It is assumed that the user actually has no prior knowledge regarding how to run HLA programs.

Start locating the first HLA example helloWorld, that is a part of the RTI software distribution. It is found at **$RTI_HOME**\lang\C++\demo.

On Windows NT load the workspace hello and make the executable **hello.exe**. On Unix systems run make to build the program **helloWorld**. Besides the small divergence in naming, where I will adopt the Unix naming convention, it does not matter which version is run.

If the rtiexec is not started type

```
rtiexec 18134
```

which starts the RTI. The port 18134 is a mandatory parameter, but the value can be changed in the **RTI.rid** file found in **$RTI_CONFIG**. Now the global part of the RTI is started.

Locate the **helloWorld** binary and type

```
helloWorld
```

It will complain that arguments are missing

```
usage: helloWorld <Country Name> <Initial Population> [<Number of Ticks>]
```

The idea is very simple. The program will initialize a HLA federate with a chosen country name. In the country federate the population will be exponentially increasing and run a number of ticks (iterations). If several federates are started then they will receive notification of the population of every other federate and once in a federate will send an interaction saying “Hello World” to the others (which explains the name).

Next try to run 10 iterations with the DK country with an initial population of 100 persons.

```
helloWorld DK 100 10
```

This will result in

```
DK 894546149 680317 START
helloWorld: Federate Handle = 1
Country[0] Name: DK Population: 100
Country[0] Name: DK Population: 101
Country[0] Name: DK Population: 102.01
Country[0] Name: DK Population: 103.03
Country[0] Name: DK Population: 104.06
Country[0] Name: DK Population: 105.101
Country[0] Name: DK Population: 106.152
Country[0] Name: DK Population: 107.214
Country[0] Name: DK Population: 108.286
Country[0] Name: DK Population: 109.369
100 894546149 879064 END
Exiting helloWorld.
```

Did you see that a shell window was started with the name fedex ? This is the federation specific part of the RTI which were started. The fedex and the rtiexec together now runs the federate control system, i.e. the RTI. If you type **quit** in the fedex window it will be killed and restarted the next time it is needed, however this is actually not needed.

For each iteration the population will increase with one percent. The present code will run as fast as possible, hence running two federates for 10 iterations will only take a few milliseconds and most likely they will not run in the time slots.

Try instead to locate the file helloWorld.cpp (the main simulator file) and change the line

```
  eventsToProcess = rtiAmb.tick();
```

to

```
  eventsToProcess = rtiAmb.tick(1.0,1.1);
```

which will force the simulator to spend between 1 and 1.1 second per iteration in a loop statement. Recompile and try running two simulators one (USA) with 10 as population as well as the first one. Type

```
helloWorld DK 100 10
```

and

```
helloWorld US 10 10
```

in another shell.

This should produce the following (approximately). Comments in italic letters are shown to the right of the screen output.

```
DK 894546976 110204 START
helloWorld: Federate Handle = 2
Turning Country.Name Updates ON.
Turning Country.Population Updates ON.
Turning Communication Interactions ON.
Discovered object 200001                       ←Here the simulator discovers the other one
Interaction: Hello World!                      ←The other simulator sends an interaction
Country[0] Name: DK Population: 100            ←Note that the simulator prints its own
Country[1] Name: USA Population: 10.1          population and then the population of the
Country[0] Name: DK Population: 101            other simulator.
Country[1] Name: USA Population: 10.201
Country[0] Name: DK Population: 102.01
Country[1] Name: USA Population: 10.303
Country[0] Name: DK Population: 103.03
Country[1] Name: USA Population: 10.406
Country[0] Name: DK Population: 104.06
Country[1] Name: USA Population: 10.5101
Country[0] Name: DK Population: 105.101
Country[1] Name: USA Population: 10.6152
Country[0] Name: DK Population: 106.152
Country[1] Name: USA Population: 10.7214
Country[0] Name: DK Population: 107.214
Country[1] Name: USA Population: 10.8286
Country[0] Name: DK Population: 108.286
Country[1] Name: USA Population: 10.9369
Removed object 200001                          ← Here the other simulator exits
Turning Country.Name Updates OFF.              ← Here we stop sending more Names
Turning Country.Population Updates OFF.        ← Here we stop sending more Population
Turning Communication Interactions OFF.        ← Here we stop sending Interactions
Country[0] Name: DK Population: 109.369
100 894547005 64781 END
Exiting helloWorld.
```

6.1 What happened ?

In the previous example one SOM was in action - the core of the FED-file (HelloWorld.fed) looks like this, i.e. a Name and a Population attribute, which are published and subscribed by all federates. All federates also publish and subscribe to the Communication (an interaction).

```
(fed
  (objects
    (class Country
        (attribute Name          FED_RELIABLE  FED_RECEIVE)
        (attribute Population    FED_RELIABLE  FED_RECEIVE)
    )
  (interactions
    (class Communication         FED_RELIABLE  FED_TIMESTAMP
        (parameter Message)
     )
  )
)
```

Let me explain what happened (You will have to lineprint/see the code now). The program starts the code in helloWorld.cpp, notice the line

```
char* const fedExecName = "HelloWorld"; // Name of the Federation Execution
```

found around line 38, which connects to the RTI using a FED-file named `HelloWorld.fed`. When the program needs to translate RTI Class names and Attribute names such as Country and Population to RTI ID's (unique ID numbers), then FED-file is read and ensures consistent ID's. Note that the name of the FED file is here hardcoded, however this is not mandatory. It can be read from the command line or alike.

In line 58 a federate is initialized

```
myCountry = new Country( argv[1], argv[2] );
```

This makes a call to the initialization function in Country.cpp at line 110. This C++ contructor will make a Class member with a name an initial population.

Next the function will return to helloWorld.cpp where the federate first tries to make a federation (start up the fedex) in line 105. If this is unsuccessful the most likely case is that a federation exists and the federate will try to connect to the federation in line 156.

Before the simulator actually runs the time control is set up in lines 209-210:

```
rtiAmb.setTimeConstrained( RTI::RTI_FALSE );
rtiAmb.turnRegulationOff();
```

The concepts of time control are explained in Section 5.7.1. A more flexible was to construct the time control lines are

```
if (timeconstrained==1)
  rtiAmb.setTimeConstrained( RTI::RTI_TRUE );
else
  rtiAmb.setTimeConstrained( RTI::RTI_FALSE );
if (timeregulation==1)
  rtiAmb.turnRegulationOn();
else
  rtiAmb.turnRegulationOff();
```

The core of the simulator starts in line 281. This simulator runs a preset number of iterations, however this can easily be made dependent on e.g. attributes - in order to implement the "death" of the federate.

The HLA construction in lines 297-324 are very important.
Initially the federate sets that it likes to advance its time with a preset time step called **timeStep**.

```
timeAdvGrant = RTI::RTI_FALSE;
rtiAmb.timeAdvanceRequest(currentTime + timeStep);
```

The actual advancement of time is done in the following lines

```
while (!timeAdvGrant) {
  int eventsToProcess = 1;
  while ( eventsToProcess ) {
    eventsToProcess = rtiAmb.tick();
  }
}
```

Note that these lines should be changed as shown on page 17 (add arguments to the tick-function). While the simulator is not allowed to advance its time, the tick function will stall and start another thread which will allow the update of attributes and interactions from other federates though the RTI.

This will now be reviewed: In the HwFederateAmbassador.cpp the functions **discoverObject** and **removeObject** will be called when a federate joins or exits the federation. The function will get both get both an objectID (a unique identifier for the federate) and the class RTI-ID (identifying the class type) of the federate. The function should then call appropriate (C++) constructors, which initializes the objects. When data (attribute changes or interactions) are transmitted from other federates to this federate the function Federate Ambassador function **reflectAttributeValues** (attributes) and **receiveInteraction** (interactions). The two functions act almost identically by calling an **Update** function - see Section 5.4 - for the class which subsequently stores the data in the class member.

Now back to the simulator. Assume the federate did get the **timeAdvGrant** signal then this program in lines 330-350 will print the status of the class members in the simulation. Finally the federate will actually advance its time by calling the last of the three **Update** functions (the one for advancing the time in Country.cpp). In this function the internal states in the federate will be updated according to the new time stamp (note that state changes due to external attribute changes was handled by the other two **Update** functions). This time **Update** function sets a new value of the population (reformatted from lines 566-570) which equal the current population plus a fraction proportional to the time step.

```
if ( deltaTime > 0 )
   SetPopulation(GetPopulation()*(1+ms_growthRatePerSec*deltaTime));
```

Note that an indirect function embedding is used when setting the Population. It is not necessary in this class function but it is used for provide bounds checking and the function set a variable **hasPopulationchanged**, which is used to check which of the variables should actually be transmitted to other federates.
When the state of the federate has been updated to the new time stamp, then a structure containing all the newly changed attributes will be made by the class function **CreateNVPSet** (also found in Country.cpp). For each of the attributes in the SOM a construction is found like this one:

```
if ( ( hasATTRIBUTEChanged == RTI::RTI_TRUE ) &&
      ( ms_sendATTRIBUTEAttrUpdates == RTI::RTI_TRUE ) )
        containerAttributes->add( RTI_ID,
                                  (char *)POINTER_TO_ATTRIBUTE,
                                  SIZE_OF_DATA_IN_BYTES);
```

where :

- RTI_ID is the RTI attribute ID
- POINTER_TO_ATTRIBUTE is a pointer to the attribute value(s).Type casting to a character array is mandatory.
- SIZE_OF_DATA_IN_BYTES is the length in bytes of the data transmitted (copied from the pointer location).

6.1.1 Byte swapping

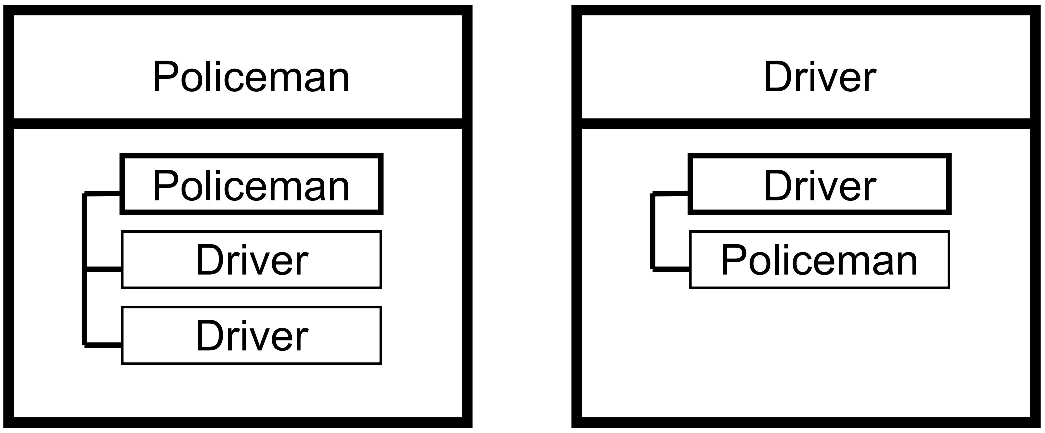
Note also that the **CreateNVPSet** function should include byte swapping as well as the **Update** functions for receiving data from other federates.
Unfortunately HLA does not specify a fixed standard for this. Earlier on Corba was a part of the HLA strategy which directly solves this problem. However Corba was discarded due to unfavorable initial performance evaluations. Hence no official policy exists on this point - see Section 10.6.
Currently it seems that HLA simulators should choose to send and receive data in a big endian format (network format)[2].

6.2 Store attributes in HLA with several SOMs

The Federate Ambassador function **discoverObject** will get both an objectID (a unique identifier for the federate) and the class RTI-ID (identifying the class type) of the federate. However it is very

[2] Processors like Intel store data in the little endian format, which is byte reversed compared to the big endian format used by most Unix workstations.

important to notice that when receiving attributes from other federates that the Federate Ambassador function **reflectAttributeValues** does NOT get the Object class ID. This is a very important!
The HLA design concepts does, I my humble opinion, have a minor flaw in the design (however it can be solved and the reason will be explained in the following). HLA implicitly works with a strategy where each program creates a federate which subscribes to the relevant attributes in the relevant SOMs. The way the HLA designers implicitly chooses to store the information is by generating a class member in the program when the **discoverObject** is called.
In the Driver/Policeman scenario the Policeman program will initially make a Policeman federate and the Driver make its own Driver federate. Next the Policeman and the driver will discover each other and initialize a class instance for the driver and vice versa. The diagram below shows that the Policeman now has discovered two drivers and the Driver subscribed to the Policeman SOM only, hence the Driver here has its own Driver federate and the Policeman registered.



In the helloWorld only one SOM is relevant, however it shows the way attributes ordinarily are stored. In line 323 in HwFederateAmbassador.cpp an additional class instance is created - and apparently now used. It is used, because it calls County.cpp in line 142. In line 150 the linking of all class members are made as indicated in the diagram above:

```
Country::ms_countryExtent[ Country::ms_extentCardinality++ ] = this;
```

The static class member `ms_extentCardinality` counts the number of federates and the static class member `ms_countryExtent` will store a pointer to the class instance. If the “main” federate then needs data from the other federates it will obtain this via the `ms_extentCardinality` array, hence it can use class functions for getting the appropriate attribute values.

This strategy is perfectly viable, and it is not a problem that the **reflectAttributeValues** only has the Object RTI ID and data values (not the Class RTI ID), because the **reflectAttributeValues** function should directly call an appropriate Update function.

However this might not a desirable path to choose - see the discussion in Section 10.4. The strategy does imply that the total memory usage scaled squared with the attribute core memory. If five federates subscribe to an attribute it will also be stored in all the other five simulators (federates). This can be a major problem if the simulators are to run with huge date sizes on the same machine.

6.2.1 The SOMs in the Crane simulator

In the Crane simulator build for COT during the HLA feasibility study another path was chosen. The prehistory is that several classes should interact and use attributes :

- **Timer** - a timer device which controls time advances. It generates a clock (time controlling element) that the remaining time constrained SOMs will use to advance its time.
- **Joystick** - a control element that generates two attributes and an interaction. Two degrees of freedom are used for controlling a horizontal crane force and two degrees of freedom for controlling lift and rotation of the crane. An interaction can be send to signal that the crane should now connect to an object (the LiftObject SOM) and allow movement of the object.

- **Crane** - mathematical driving model and a lot of rules. The Crane has a simplified mathematical movement model driven by the Joystick. It generates an acceleration, velocity, and position attributes.
- **LiftObject** - an object that can be lifted. It receives interactions from the Joystick to be connected to the Crane. If the LiftObject is very close to the Crane (this is checked) the connect interaction will imply that the LfitObject actually sets its position equal to the Crane position. The LiftObject generates one attribute, namely its position.
- **Visualize** - an object that senses and prints all object positions. In this case it senses the position of the Crane and all LiftObjects.

The subscribing and publishing classes are shown to the left of the table below. The attributes are shown in the top. The black color means generation of a parameter and the gray color means that the SOM will subscribe to the attribute.

	Time	Controls	Position	Lift Interaction
Timer	■			
Joystick	▨	■		■
Crane	▨	▨	■	▨
LiftObject	▨		■	▨
Visualize	▨		▨	

In case of the Crane simulator I did not choose that the Crane should initialize a Joystick instance when it is discovered. Instead the attribute changes were directly fed to the Crane class, in order to let the Crane class hold all the parameters that determines the state. This is easier to implement than the strategy outlined above.
The nasty problem that the **reflectAttributeValues** only gets the Object RTI ID and the attribute types and values and not the class RTI ID's and the attribute type RTI ID's are <u>not</u> unique. The header of the function looks like

```
virtual void reflectAttributeValues (
        RTI::ObjectID                     theObject,
  const RTI::AttributeHandleValuePairSet& theAttributes,
        RTI::FederationTime               theTime,
  const RTI::UserSuppliedTag              theTag,
        RTI::EventRetractionHandle        theHandle)
```

The attributes passed in the second parameter is a container with a set of attribute changes from the object identified by its Object RTI ID - the first parameter. The container is decoded in the Update function for new attributes as shown in Section 5.6 on page 14. In the Update function the attribute RTI ID stored in `attrHandle` will match the ranging in the FED-file. For each class they are labeled 1, 2, 3, etc. Two classes will have the same attribute RTI ID's, hence the way I proceeded to be able to separate attributes from several classes was to form an array of matching Object RTI ID's (RTI::ObjectID) and RTI class ID (RTI::ObjectClassHandle). This table is updated when the **discoverObject** is called - the implementation is now part of the Crane simulator. Furthermore the Update function for new attributes was extended from

```
void Update(  const RTI::AttributeHandleValuePairSet& theAttributes);
```

to include the parameter class

```
void Update(  const RTI::AttributeHandleValuePairSet& theAttributes ,
              RTI::ObjectClassHandle theObjectClass);
```
this is perfectly viable and allowed. The problem is further discussed in Section 10.4

7. How to add a new SOM to an existing FOM

This section tries to describe the steps to add a new HLA SOM and how to set up makefiles/project files.

Assume that a simulator consists of four classes: Timer, Joystick, Crane, and Visualize. And I want to add a new SOM called LiftObject. The example is rather general, and the following I try to describe how to copy existing HLA compliant files and modify the files to support a new SOM. Start by defining the class attributes with the OMDT tool. This is the proper fashion to define the SOM/FOM in a standard way, that aids the following implementation. In the OMDT tool, insert a new class. Add the relevant attributes and interactions. Remember to add the descriptions, in order to be able to check the consistency of the SOM. It is very important to describe the cardinality of the attributes (how many entries) and their types. Finally use the feature to save-as an OMD-file which contains classes, attributes, their types and cardinality. This file is important when implementing the classes (SOMs).

For NT Visual C++ add a project where some options should be considered specially.
Select **Project** - **Settings** select **All Configurations** (highlight all projects) and choose
Under C/C++
  Warning Level: **Level 3**
  Debug info: **Program Database**
  Optimizations: **Disable (Debug)**
  Preprocessor: **Additional include directories: ..\..\..\include**
  Code Generation: Use run-time library: **Debug Multithreaded DLL**
Under Link
  General: Add **rti.lib** and **wsock32.lib** to the Object/library modules (change
    rti.lib to rti-r.lib for a release version. The rti.lib contains debug information. rti-r.lib does not)
  Output:
    Stack allocations:
      reserve 1000000
      commit 1000000

Add these files in the relevant Imakefile (for Unix systems).

I plan to add three source cpp-files:

  src/LiftObjectFederateAmbassador.cpp (The RTI support functions)
  src/LiftObject.cpp (The class implementation)
  src/LiftObjectSim.cpp (The actual simulator part with initialization and time advances)

and two include files

  include/LiftObject.hh (The Class functions)
  include/LiftObjectFederateAmbassador.hh (The Class ambassador functions)

Start copying the relevant header files from a template (another class implementation) and get ready to modify. Go to the include directory (the include files)

  copy Crane.hh to LiftObject.hh
  copy CraneFederateAmbassador.hh to LiftObjectFederateAmbassador.hh

Copy relevant source files like it was done for the include files. Go to src (the source files)

  copy Crane.cpp to LiftObject.cpp
  copy CraneSim.cpp to LiftObjectSim.cpp
  copy CraneFederateAmbassador.cpp to LiftObjectFederateAmbassador.cpp

Another class implementation could also be used as the template. Use the OMD-file to change the class attributes to the new ones, and do search and replace. Start by Making the LiftObject.hh right according to the OMDT !! If the class attributes names started with crane then do global search and replace (be careful) to liftobject.
Next add the files to the revision control system; CVS and/or Visual Source safe.

- Furthermore in the LiftObject.hh and LiftObject.cpp, add the relevant TypeID's and type-strings (holding the RTI names of the subscribed classes) that the federate needs to know.
- Modify LiftObjectSim.cpp - which is rather easy. Check the data that should be printed on the screen. Remember that the fedExecName in the LiftObjectSim.cpp should not be changed, which is the name of the FED-file to read when connecting to the RTI.
- In LiftObject.cpp.
  - Start making the constructors. Use a common mask (LiftObject_Init), which should be called by all constructors and then add the relevant minor changes to the individual constructors, in order to limit the redundant.
  - In the PublishAndSubscribe part edit the list of relevant TypeID calls for the classes that the LiftObject need to subscribe to.
  - If you also want to subscribe to other classes, add separate statements in the CreateNVPSet function and remember to update the variable No_of_attributes to the right number.
  - In SetInteractionControl add send<ATTRIBUTE>AttrUpdates sections for each of the attributes that needs to be send to others.
  - In the section Update(newTime) Add all of the dynamics, update the relevant variables and keep the last part of the code. Use the Set<ATTRIBUTE> functions - do not set the variables directly. This will aid the program to enforce parameter control and set the check variable has<ATTRIBUTE>changed from false to true.
  - In each of the functions for setting each of the attributes Set<ATTRIBUTE>, do set (and check) the attributes and remember to set the variable has<ATTRIBUTE>Changed to RTI:RTI_TRUE.
- In the LiftObjectFederateAmbassador.cpp in the section for provideAttributeValueUpdate make a part for each of the attributes relevant.

Try the simulator - good luck!

8. Evaluation of HLA

During the development of the simulator DMI has several times been in contact with the HLA support site `rti_support@msis.dmso.mil,` and the response time and quality of answers have been good.

The simulator has been programmed using a version 1.0.3 C++ API from DMSO, which is very well suited for HLA development. The concepts of SOM's matches the classes in C++ very well. It was found that the programming style used to make the simulators is rather strict, but this definitely also aids rapid development of simulators. The present RTI software is mature and industrial simulators can be build using the software. The RTI version 1.3 **Migration Document** (available from http://www.dmso.mil/cgi-bin/hla_dev/hla_soft.pl) describes the changes from 1.0.3 to 1.3. It seems that only few changes should be made to the 1.0.3 programs in order to run under 1.3.

It was also found that HLA includes climbing a high learning curve. The documentation to the 1.0.3 version is huge and spread over many documents that combined exceeds thousand pages. The documentation for the very new release 1.3 seems to be far better, but is quite comprehensive.

The prototype crane simulator has been tested on both an 180 MHz R5000 SGI O2 with 96 MB RAM and an 300 MHz Pentium II PC running Windows NT 4.0 with 64 MB RAM. The HLA federation is running on both platforms and the overall performance is better on the SGI machine presumably due to a significantly higher internal bandwidth on the SGI machine and the higher performance of IRIX compared to NT. However for the simple HLA performance tests, see below, the NT performed very fast, presumably due to the use of NT-pipes for communication.
The HLA documentation also state limitations in the Windows version when ran using Windows95. Using several federates the simulator system is simply unstable.

The programs can be easily ported from one platform to the other (SGI and NT), if GUI implementations are avoided and the macro expansions used in Visual C++ are avoided.
The debugging environment in Visual C++ is very good, however running 3 federates and on in the debugger at the same time, is very demanding for Windows NT. A fact that has not been observed on the SGI machine, probably due to the better scheduling algorithm on UNIX systems. It should also be noted that HLA does not specify how to handle the endian problems (PC systems and UNIX workstation). Using mixed big endian systems implies that special endian decoding should be included specially in the C++ code. A nasty problem that has been observed using Visual C++ is that using the optimization flags made the simulators fail. When running without optimization and full debug level implies that the observed problems vanished. Using the standard compiler CC on SGI problems like that was not observed.

Furthermore it should be noted that the memory demands of the NT version are roughly as follows:

- 1.5 MB for the rtiexec (global RTI process)
- 2.5 MB per fedex (global federation process)
- 7.7 MB per federate for RTI library

On the Unix version (the SGI version) the memory usage is very different

- negligible memory for the rtiexec (global RTI process)
- negligible memory per fedex (global federation process)
- 10-15 MB per federate for RTI library

This indicates that the programming style should try to limit the number of federates below 10-20 per machine based on the current average memory. Furthermore the issue of simulator speed might also limit the number of federates per machine. Using time regulation so that all simulators are time constrained by the timer federate showed that 10 Hz was no problem using five different federates on the same machine (SGI). Furthermore several simulations has been made using non-time constrained operation (where the federate should run as fast as possible and update states whenever external attribute changes are available.

From fruitful discussions with the HLA support site several style issues have been settled. Only non-static parameters should be allowed to be an attribute. If the federates, e.g., need a scene height in a given position (mathematically $z=h(x,y)$), and the scene height function does not change, then this should be handled externally using simple function calls not passing through the RTI. Currently the 1.0.3 version of the HLA software does not provide a very good solution here. However in the upcoming 1.3 version an option to obtain regions of attributes. The region should then be centered around the actual federate position. Further discussion is found in Sections 10.8 and 10.12.

8.1 The HLA crane simulator

At DMI a simplified crane simulator has been developed in HLA to investigate the feasibility of using HLA as the basis for the OSS decision support system as well as the MTC crane training simulator. The crane simulator was build starting at a small scale design with a crane and a joystick that controlled the to crane operation. Later three other simulators were added; a timer, a visualisation simulator, and a lift object (an object to be lifted).
The crane simulator has many nice features, which are directly offered or supported by HLA.

- The simulators will dynamically discover relevant objects and modify the internal structure after this, e.g., if the Timer federate is not started the other federates will stall until it is online.
- Furthermore the visualisation federate will dynamically discover federates of the Crane and LiftObject and show the position of those objects. If a federate is removed from the federation this information will be signaled to the relevant federates, e.g. the Visualize object which subsequently stops showing the position of the federate.
- The C++ API suggests that class functions are made for setting private variables. This aids enforcing range changing and logical tests to support robust simulators.
- A lot of code reuse is possible from one simulator to the next.

9. Documentation for the three Simulators

Next a short introduction is made for three simulator systems made at DMI. The source code can be found at **PC0100\\Adm\Adm\Data\Project\98853-COT\Misc\HLA Programs**.

9.1 A Simple two-Class Example

After studying the helloWorld example I decided to make a two class example based on the helloWorld example. My intention was to have two classes that made interactions for communicating between the classes. I followed my own recipe from Chapter 6 starting with the existing Country class and making a WarPower class. The helloWorld example is a simulator system with a population that increases exponentially, hence I wanted something that could modify this growth, hence the war class, which can either send an text interaction with contents "You are my friend", which will make the Country class federate(s) happy and their population will instantaneously double. The WarPower federate can also send an text interaction with contents “You will die!", which will remove half of the Country population. This is virtually the simplest two class example and the die/friend idea could as well have been replaced by an economical stock trading system.

The program package **two_class_example.tar.gz** has a README file which explains the basis usage. Unpack with gunzip (GNU unzip) and tar (Unix tar).

9.2 The HLA performance system

Internally at DMI we have wished to test the speed of HLA and compare it to similar data for the 3MS simulators. The system design has been made on a single class system similar to the helloWorld example. The general idea is a master/slave constellation where both the master and slave allocates an attribute array (named Container) holding a number of bytes (default 4000 bytes) specified by the user. Every 5. Iteration (user specified value) the master tell the RTI the its container has been changed. The container will be send to the slave federate which subscribes to the Container attribute. When received in the slave federate, the slave will then set a flag signaling that its own container should be send to the master federate (actually the contents of the master data container will be copied to the slave one). Finally, when the master federate receives data from the slave it will compute the total round tour time and average the round trip time measurements.

The program package **HLAperformance.tar.gz** has a README file which explains the basis usage. Unpack with gunzip (GNU unzip) and tar (Unix tar).

Run in seperate shells

```
rtiexec 18134
mastersim -m Performance.ini
mastersim -s Performance.ini
```

The -m switch means run as master and -s for run as slave

The slave will not print data to the screen and the master will print the round trip time measurements. A lot of the simulator settings are determined in the initialization file Performance.ini:

`timeconstrained=0`	*Binary value. Simulator is not time constrained by the other one*
`timeregulation=0`	*Binary value. Simulator is not time regulating*
`NumberOfCounts=300000`	*Number of iterations*
`Name=Performance`	*Register the simulator under the name*
`ContainerSize=4000`	*Size in bytes of the data transmitted*
`DebugLevel=Normal`	*Debuglevel : Debug, Normal, or Silent*
`SleepMin=0.001`	*In the main data receive loop minimum time to use in seconds*
`SleepMax=0.002`	*In the main data receive loop maximum time to use in seconds*

`SleepIteration=0`	*In the main data loop delay of the simulator in seconds*
`Repeat=5`	*Send data every Repeat iteration (cannot be less than two)*
Average=10	*Average round trip times over Average number of trails*

9.3 The Crane simulator

The crane simulator is far more complex than the two other simulators. The outline of the SOMs have been shown in subsection 6.2.1.

10. HLA FAQ

When designing and programming HLA compliant programs several questions will pop up. Luckily it is possible to get HLA help for free. Two email addresses are available, and both will normally respond quickly.

- **hla@msis.dmso.mil** This help site is devoted to major issues of HLA. Where/which/how in the HLA world. Very helpful answers.
- **rti_support@msis.dmso.mil** This is the primary help site to use for the programmer. Both actual programming problems as well as style issues can be discussed here. The quality of the answers are normally very high.

Below is listed a lot of emails from and to the rti_support site. Notice that the emails have been edited and merged in order to make a more clear presentation. The rti_support site has not validated the following as such, and the responsibility lies solely at the author.

10.1 Basic question about subscribe

QUESTION:

Assume two federates A and B are in play (with different SOM's / different object classes).
Assume that A is a driver of class AA and it has an attribute DrivingSpeed (for sake of argument).
Assume that B is a "policeman" of class BB who will stop any federate if attribute DrivingSpeed>some_limit.
How can I make a policeoperation in HLA? Since B is not the same class as A he will like to subscribe to the attribute DrivingSpeed of Class A.

Solution A: Does class BB need to include (speaking C++ language) the AA-class (seems to be too much)
Solution B: Should the A class make a field matching the decided one.
Solution C: Don't subscribe directly and use Interaction classes. I know that this works.

ANSWER:

In this case, Federate B would issue a "Subscribe Object Class Attributes" service call to the RTI with class name "AA" and attribute name "DrivingSpeed". Most likely, the "DrivingSpeed" attribute would be published by the "AA" class at some periodic rate. In order to hold down network traffic, Federate B might want to set up a routing space and subscription region for the "DrivingSpeed" attribute so that he will only receive updates when the instance of the "AA" class is within a certain range. As Federate B receives updates for this attribute via the "Reflect Attribute Values" service, the new value of this attribute can be compared to "some_limit" to see if a ticket is warranted.

As you say, interactions can also be used, but this seems awkward for regular "heartbeat" type updates to characteristics of domain-specific objects. Would suggest above approach.

10.2 Time regulation

Federate A of class AA Federate B of class BB Federate C of class CC

Federate A is very slow and he makes changes that B and C should react on. B subscribes to attributes in class AA, (and C to attributes in class BB). This works fine, however it is in this situation foolish that B and C should run no time constrains. My intend would be to have federate A run as fast as possible (it is rather slow) and let B and C run with time constrained and regulation on.

You probably want to have federate "A" be time-regulating (but not time-constrained) and federates "B" and "C" be time-constrained (but not time-regulating.) This sounds like what you're proposing.

In the documentation to the helloWorld example it is made by

rtiAmb.setTimeConstrained(RTI::RTI_TRUE);
rtiAmb.turnRegulationOn();

My problem is how to do with federate A ?? I want to use

rtiAmb.turnRegulationOn()

to have B and C wait to A, but then I get errors when I send interactions.

Error:InvalidFederationTime : Federation time already passed 1300

If I print the rtiAmb.requestFederateTime() is matches the time in my Update function (like in helloWorld). What am I not doing right here ?? And where do the number 1300 come from ??

When a federate is time-regulating, it is only allowed to send out events with timestamps greater than or equal to the current federate time PLUS the lookahead. If you add a small amount to the timestamps of the outgoing events then this problem should clear up.

10.3 General time problems

Class A: A very slow system - it is time regulating system (It is a Timer). Class B,C and D: Fast systems - all of them with no time regulation. It is apparently running fine, however the slow simulator of class A can after some time do an abort which I can see on both and an SGI O2 and an NT system:

Are federates B, C, and D time-constrained? Time-regulation only has an effect on federates that are time-constrained. (This is probably not related to the problem -- just a general suggestion.)

If B,C, and D are time-constrained by A, then it runs perfectly. I have a small switch to turn for that. If B,C, and are NOR time-constrained then the problem can be seen.

However it can make an error and exit with a core dump, and I have a hard time locating the problem. It seems that it might have something to do with the very high bandwidth used to pass changed attributes between B, C and D - which kills A (A does not subscribe to any of B,C or D attributes).

With RTI 1.0, all updates are sent to all federates regardless of what they've subscribed to. Unnecessary information is discarded by the RTI at the receive-side and not delivered through the federate ambassador. This means that if you're using reliable communications, it's conceivable that federate A could have its TCP buffers overrun if it were extremely slow. You might want to test this by using best-effort communications (if you're not already.)

All of my attributes are in principle important, hence in my fed-file I have specified

```
 (class MYCLASS
  (attribute MYATTRIBUTE      FED_RELIABLE    FED_RECEIVE)
 )
```

My bet is that I get the TCP buffer overrun.

Yes, sounds like the culprit.

(RTI 1.3, scheduled for release on March 27, does more sophisticated routing and can often avoid sending events to federates that aren't interested.)

What is federate "A" doing when it's not advancing time? If you have a "Sleep()" (or whatever) in federate A, consider using a timed-tick (i.e. the "tick" version with two arguments) instead. This will have the effect of consuming the desired amount of time, but the RTI will continue to drain the queues and process incoming events.

I understand your point made here. I am using sleep on the Unix box and Sleep(1000) on the NT box. However the sleep command is not inserted to actually stop the simulator but rather to emulate the time spend by doing some very nasty calculations.*

After (before) the sleep I have the standard wait construction

```
     try {
         timeAdvGrant = RTI::RTI_FALSE;
         rtiAmb.timeAdvanceRequest(currentTime + timeStep);
       }
     catch ( RTI::Exception& e ) {
         cerr << "Error:" << &e << endl;
       }

     while (!timeAdvGrant) {
         int eventsToProcess = 1;

         while ( eventsToProcess ) {
             eventsToProcess = rtiAmb.tick();
         }
       }
```

You can achieve the sleep effect by doing a "tick(1.0, 1.0)". This will consume a second of time, but it has the added advantage that the RTI will continue servicing incoming traffic. I believe this is what's needed to keep up with the federation and keep your TCP buffers from failing. If there isn't any incoming traffic, the RTI will block and wait, so there shouldn't be any additional CPU overhead introduced by doing things this way.

I have a suggestion to a newer RTI. The tick(min,max) is a fine function, however the time base of the tick function is a second, as far as I know. In many simulators that I would like to build the time base would be smaller, at least 1/10 of a second, hence it could be desirable to have a microtick function as well with a time base in e.g., milliseconds.

You can use arguments less than 1 to tick for less than a second in the current implementation. Your resolution is only limited by the resolution on "gettimeofday" on your system, which is generally very good.

10.4 RTI Communication ID

I have a problem of understanding the RTI-ID's, that causes problems for me. I am building a simulator with several classes, and a similar (higher) number of federates. The federates subscribe to attributes from the other classes in a specific way.

Assume
Class A
 Attribute AA
Class B
 Attribute BA

When I request RTI type id's, I will get something like....

Class handle 1 for A
 Attribute handle 1 for AA (AA_rtiID)
Class handle 2 for B
 Attribute handle 1 for BA (BA_rtiID)

Assume the all B federates subscribe to BA, and AA attributes. Then in my federate ambassador, I will have something like:

```
void HwFederateAmbassador::reflectAttributeValues
        ( RTI::ObjectID                          theObject,
          const RTI::AttributeHandleValuePairSet& theAttributes,
          RTI::FederationTime                    ,
          const RTI::UserSuppliedTag             ,
          RTI::EventRetractionHandle               )
     throw (RTI::ObjectNotKnown,
            RTI::AttributeNotKnown,
            RTI::InvalidFederationTime,
            RTI::FederateInternalError)
{
   Country *pCountry = Country::Find( theObject );

   if ( pCountry )
   {
      //-----------------------------------------------------------------
      // Set the new attribute values in this country instance.
      //-----------------------------------------------------------------
      pCountry->Update( theAttributes );
   }
}
```

And the Update function will track the theAttributes.Gethandle(i) and compare to AB_rtiID, and AA_rtiID

```
void Country::Update( const RTI::AttributeHandleValuePairSet& theAttributes )
{
  RTI::AttributeHandle attrHandle;
  RTI::ULong           valueLength;

  for ( unsigned int i = 0; i < theAttributes.size(); i++ )
  {
    attrHandle = theAttributes.getHandle( i );
    if ( attrHandle == Country::AA_rtiID )      <-----------DANG - they are the same \
    {
      decode and set stuff
    }
    else if ( attrHandle == Country::AB_rtiID ) <-----------DANG - they are the same /
    {
      decode and set stuff
    }
  }
}
```

but in the Update function I cannot tell the sender classes from each other. AFAIK I cannot the the sender class handle in theAttributes, and I cannot tell the two attributes from each other......

What am I missing here ??????

Should I turn to the FederateAmbassador and decode theObject into a class part and a ID - with a modulus 100000 function ??????

I'm not sure I follow exactly what your question is and I haven't read the details of your questions/answers with Rob but here goes a try. I think you are wondering how you handle multiple classes of objects that you wish
to reflect since the attribute handles are non-unique. I am also assuming that your example was not showing inheritance (class B does not inherit from class A).

Your example points out the importance of the discoverObject service. The discoverObject service is the only time your federate will be presented with the class of an object. Based on the class of the object you will need to execute the source code specific to the class. The HelloWorld federate does not show this very well since it only deals with the Country class but here is a better example:

In the reflectAttributeValues service I need to be able to determine which class an object was discovered as. I can do that a couple of ways:

1) create a map from objectID to object class handle in the discoverObject service

2) look in all of my collections (arrays) of objects in the reflectAttributeValues service and which ever one returns a non-null pointer is the type of object
3) have a base class that all my internal objects derive from and have one collection that holds all of them. The base class would have a data member that stores the object class handle.

After we can figure out which class an object ID was discovered as we know what attributes are valid for the class (we know their names, handles, and types) and can process them appropriately. Here is an example of reflect using the #2 approach (this obviously is sub-optimal but I didn't have to write code in the discoverObject function :) :

```
void HwFederateAmbassador::reflectAttributeValues(...){

   Country *pCountry = NULL;
   Person  *pPerson  = NULL;
   State   *pState   = NULL;

   if ( (pCountry = Country::Find( theObject )) != NULL )
   {
      //-------------------------------------------------------------
      // Set the new attribute values in this country instance.
      //-------------------------------------------------------------
      pCountry->Update( theAttributes );
   }
   else if ( (pPerson = Person::Find( theObject )) != NULL )
   {
      //-------------------------------------------------------------
      // Set the new attribute values in this Person instance.
      //-------------------------------------------------------------
      pPerson->Update( theAttributes );
   }
   else if ( (pState = State::Find( theObject )) != NULL )
   {
      //-------------------------------------------------------------
      // Set the new attribute values in this State instance.
      //-------------------------------------------------------------
      pState->Update( theAttributes );
   }
   else
   {
      cerr << "Error: Unknown object type!" << endl;
   }
}
```

I think that item 1) is a manageable way for me. Regarding Item 2) I understand your code, but in my case federate B needs to see the new attributes of class A and update a derived parameter, hence I don't want to make an additional A object in the B process. (I guess that this is why you dont fully get me problem).

Regarding Item 3) I understand the idea, but would like to avoid that one.

I just found yet another way, which I think I will take. I will take your solution 1) + extend the reflectAttributeValues function with

```
( RTI::ObjectID                               theObject,
  const RTI::AttributeHandleValuePairSet& theAttributes,
  RTI::FederationTime                         theTime,
  const RTI::UserSuppliedTag                  theTag,
  RTI::EventRetractionHandle                  theHandle )
```

and then use the theHandle.sendingFederate which I can decode to a class handle and append it to the Update parameter list.

Another email followed up here

When I request RTI type id's, I will get something like....

Class handle 1 for A Attribute handle 1 for AA (AA_rtiID)
Class handle 2 for B Attribute handle 1 for BA (BA_rtiID)

but in the Update function I cannot tell the sender classes from each other. AFAIK I cannot the sender class handle in theAttributes, and I cannot tell the two attributes from each other......

The issue here is that attribute handles are not meaningful in and of themselves -- they are only meaningful when considered in the context of an object-class handle (in which case they denote class-attributes) or an object-instance handle (in which case they denote instance-attributes of the instance.) When used in conjunction with an object-instance handle (as in "reflectAttributeValues"), the attribute handles refer to the object class by which the object instance is known to the federate.

This means that you'll have to include logic that "remembers" the object class by which an object instance was discovered in order to provide a context for the attribute handles in "reflectAttributeValues". Perhaps the easiest and most elegant way to do this is do have an abstract base class such as:

```
class DiscoveredObject {
public:
  DiscoveredObject(RTI::ObjectID);

  void UpdateAttributes(RTI::AttributeHandleValuePairSet&
         // possibly pass time, tag, etc., if necessary
         ) = 0;
}
```

You could then subclass this to construct C++ classes representing object classes "A" and "B" in your federation. When you discover an object, you'd simply instantiate an instance of the appropriate concrete object class -- "A" or "B". When you receive a reflection, you'd simply locate the class instance associated with the reflected object instance and invoke the "UpdateAttributes" method. The "UpdateAttributes" method would then know the correct context for the attribute handles by virtue of the concrete subclass to which it belongs.

Should I turn to the FederateAmbassador and decode theObject into a class part and a ID - with a modulus 100000 function ??????

Interesting idea, but I would recommend against it. RTI 1.0 allows the federate a fair amount of flexibility when it comes to choosing the object ID associated with an object instance. This makes it possible to do tricks involving encoding some sort of information into the object IDs. However, the forthcoming RTI 1.3 (and the new HLA Interface Spec) changes the way object IDs are assigned to object instances -- the RTI is now in complete control of assigning IDs.

You could, however, use the "user supplied tag" to communicate the class of the object that's being updated. I would recommend against this on the grounds that it's inelegant, but it's an option.

Yet another email followed up here

I took direction shown in item 1), and it seems to work fine. Now I can avoid making new objects in all the other federates and still communicate the data required. (I try to avoid this, because it leads to a N^2 memory usage, where N is the number of federates).

In the discoverObject function I create a map of ObjectID's and ObjectClassID.
In the removeObject I likewise remove ObjectID's.

My Class is something like

```
class FederateControl
```

```
{
public:
   FederateControl();
   ~FederateControl();

  //--------------------------------------------------------------
  // Methods acting on the RTI
  //--------------------------------------------------------------
  RTI::ObjectClassHandle GetClass(RTI::ObjectID objectId);
  int GetNumberObjects(void);
  int InsertObject( RTI::ObjectID objectId,
        RTI::ObjectClassHandle theObjectClass );
  int RemoveObject( RTI::ObjectID objectId );

  //--------------------------------------------------------------
  // Static Members
  //--------------------------------------------------------------


private:
  RTI::ObjectID objectIdar[ MAX_FEDERATE + 1 ];
  RTI::ObjectClassHandle objectClassar[ MAX_FEDERATE + 1 ];
  int     no_handles;
};
```

Then I can extend my Update(theAttibutes) to Update(theAttibutes, theObjectClass) where theObjectClass = FederateControl::GetClass(theObject) in my reflectAttributeValues federateambassador-function

In my Update function now I am in full control over ClassID and AttributeID. Home clean!!!

10.5 Unexpected death of federates

I am now in a new situation looking at the way a federate dies and how this influences the other federates. Please correct me now. When a new federate comes up I can get notification from the discoverObject (fine!!).

When a federate dies (for some reason) the FederateAmbassador::stopUpdates will be called if not more updates are required, but it seems as if the removeObject function is not called if the federate for some unexpected reason dies (unclean exit).

I can see the problem if I start the helloWorld example (once again...), in two different xterms. With arguments US and Denmark respectively.

```
Note: Federation execution already exists.FederationExecutionAlreadyExists :
RTIexecImpl::reserveFederationExecution 12100
Denmark 890144940 290319 START
helloWorld: Federate Handle = 1
Country[0] Name: Denmark Population: 1
Country[0] Name: Denmark Population: 1.01
Turning Country.Name Updates ON.
Turning Country.Population Updates ON.
Turning Communication Interactions ON.
Country[0] Name: Denmark Population: 1.0201
Country[0] Name: Denmark Population: 1.0303
Discovered object 2400001
Interaction: Hello World!
Country[0] Name: Denmark Population: 1.0406
Country[1] Name: US Population: 1.0201
Country[0] Name: Denmark Population: 1.05101
Country[1] Name: US Population: 1.0303
Country[0] Name: Denmark Population: 1.06152
Country[1] Name: US Population: 1.0303
Country[0] Name: Denmark Population: 1.07214
Country[1] Name: US Population: 1.05101

**** Here I hit ctrl-c to force a horrible death for the US federate ****

Turning Country.Name Updates OFF.
Turning Country.Population Updates OFF.
Turning Communication Interactions OFF.
Country[0] Name: Denmark Population: 1.08286
Country[1] Name: US Population: 1.06152
Country[0] Name: Denmark Population: 1.09369
Country[1] Name: US Population: 1.06152
```

Note that the removeObject for US is NOT called, hence Country[0] (Denmark) still displays it.

I know that if the federate exits nicely, then everything is fine, but can I get force the RTI to call the removeObject feature in the other federates in case of a federates disappearing (the fedex does see this and closes the connection) ??

No, sorry. In addition to the technical complexity of implementing this (it would become complicated to try to track the "privilegeToDelete" tokens if ownership management services were being used), it's not clear that this would be the "correct" thing to do for all federations. So HLA leaves it up to the federation developer to detect "zombie" objects and deal with them accordingly. Two relatively easy solutions come to mind:

1) keep track of which federates register which objects (communicated using the user-specified tag or the ERH), then use the MOM objects to detect when federates have died and remove the corresponding object instances

2) periodically remove all object instances that haven't been updated in some period of time

10.6 HLA and the curse of different Endians

On different machine types the endian "curse" is a well known problem. At some point I truly hope that the HLA specification will also contain a specification of the byte ordering and data type sizes. One very nice way to handle this could be to have the RTI do the conversion (to e.g. big endian), but I guess this is not feasible as it is implemented right now, but it would allow range checks and the endian conversion code could be hidden.

I am inclined to agree with you, but I believe it's the current intention of the powers-that-be in the HLA world that data representation is entirely the responsibility of the federation. You'd have to take this one up with "hla@msis.dmso.mil". At the very least, it would be nice of the OMT contained a provision for specifying big- vs. little-endian representation, even if the enforcement of such is left to the federation.

10.7 Uniqueness

In the highlander movies the phrase "there can only be one" is often heard.

In most simulators some of the federates are only allowed exactly once. An example could be a scene controller (a master player who knows all goals, additional federates are controlled by him etc.) Can you guide me to an HLA appropriate way to ensure that such a federate is only started once.

Unfortunately there is no good way to do this in RTI 1.0. A possible solution is to have a "manager" federate that monitors the federates currently participating in the federation (using MOM objects and interactions) and arbitrates the joining of new federates.

RTI 1.3 helps to address this problem by allowing global named objects. The RTI won't allow multiple federates to register an object with the name; this probably achieves the effect you're looking for in not having multiple instances of the same federate, correct?

10.8 HLA and Huge data arrays

Next question regards how to include a world scene with height curves etc. and a set of ground vehicles. Should I have vehicles ask an environment federate about the structure often environment (height and e.g. the curvature of the ground) ? And how about transferring the position parameters of the vehicles that is needed to return the height ? Code examples is not needed, but rather general remarks regarding the style to choose.

There are a variety of possible approaches. I'd advise against making everything an RTI object unless there's a good reason to do so. That is, if the environment information consists of data which is entirely static (or static within a given run of a federation) it is probably simpler to have it hard-wired into your federates or in some sort of configuration file that is read in by each federate. If the environment data is going to dynamically change over the lifetime of the federation, it should be transmitted over the RTI (indeed, the HLA Rules state that all such data _must_ be transmitted over the RTI.)

Assuming you're going to be transmitting this data over the RTI, the most natural way would be to model the environment as one or more RTI objects and have joining federates do a "requestClassAttributeValueUpdate" to solicit values for these objects. The environment data will then be maintained locally by each federate. Drawbacks to this approach are:

(1) logic for how to perform computations based on environment data must be built in to each individual federate, and
(2) it may be complicated and expensive (in terms of memory consumption) to have this data stored locally at each federate. The RTI 1.3 can mitigate the transmission/storage requirements in that DDM services allow a federate to specify that it only wishes to receive information about a certain logical region. (There are some RTI 1.0 techniques you could use to attain this effect, but it'd be considerably more involved.)

Alternatively, you could have a "client/server" approach in which knowledge of the environment is maintained centrally by an "environment" federate, and the federation uses interactions to submit computations involving environment data and receive responses. This is a slower and more complicated design (because all environment-related computations must be performed asynchronously by a remote federate and then transmitted via interactions), but it obviates the need for additional logic and storage requirements at each individual federate.

Yes, you're right that in 1.0.3 there is no "silver bullet" that will make this sort of thing easy. Probably the best approach to an interaction-based system would be to divide your grid up into "regions" and have each request to the environment federate return the data for the whole region containing the specified point. This information could be cached by the local federates to reduce the amount of "environment" interactions (assuming your entities aren't going to be rapidly moving from region to region) while still limiting storage requirements for the simulations to a reasonable amount. Kind of a poor man's DDM, if you will. :-)

10.9 Memory usage

This time I consider the memory usage.

As far as I know the memory demands of the NT version are roughly as follows:

- *1.5 MB for the rtiexec (global RTI process)*
- *2.5 MB per fedex (global federation process)*
- *7.7 MB per federate for RTI library*

On the Unix version the memory usage is very different

- *negligible memory for the rtiexec (global RTI process)*
- *negligible memory per fedex (global federation process)*
- *10-15 MB per federate for RTI library*

In my humble opinion this is a problem. With everyday computers the simulators I will build could consist of around 50 federates, hence I need RAM !!!! Even though most of the federates are mostly "sleeping", however have attributes that change once in a while. The simulators should run on a single machine.

One strategy I can choose - however don't want to use - is to bundle the simulators - i.e. fewer federates with an enlarged set of attributes.

It seems rather awesome to use roughly 10 MB per federate. Do you have information whether this will be less in the RTI 1.3 version ? And whether it is possible to make tricks to make the net memory usage go down, again all federates on a single machine.

Unfortunately, the current release of RTI 1.3 uses a great deal of memory. More than 10 megs, even. This is certainly an issue that we'd like to address for future releases of RTI 1.3.

In RTI 1.0, about the only thing you can do is to decrease "MAX_HANDLE_VALUE_PAIRS". In RTI 1.0.3, this value is set to 1000, which is probably almost as low as you can make it without running into problems with too many allocated HV-pairs.

Sorry I don't really have any information that can help you with this situation. The common usage of the RTI is just to have a single federate per machine, or a small number of federates per machine at most. Hence, not very much effort to date has gone into optimizing the RTI for support of a large number of federates per machine.

10.10 Time/Speed issues

In the helloWorld example in the main loop of the bulk of time will be spend in the receive loop, which has a time advance request part

```
  try {
      timeAdvGrant = RTI::RTI_FALSE;
      rtiAmb.timeAdvanceRequest(currentTime + timeStep);
    }
  catch ( RTI::Exception& e ) {
      cerr << "Error:" << &e << endl;
    }
```

and a loop construction:

```
  while (!timeAdvGrant) {
      int eventsToProcess = 1;
      while ( eventsToProcess ) {
    eventsToProcess = rtiAmb.tick();
   }
    }
```

My question regards the fact that this construction eats up CPU while in hangs in the last loop construction. Can I redo the last construction so it does not eat the CPU at full load while still having the same function (maybe somewhat slower). Assuming that the advance frequency is around 1 Hz and the computations per iteration only requires a very small fraction hereof.

Yes, you can use the "timed" variant of "tick". (The RTI 1.3 implementation of "hello world" has been changed to do this.) The timed variant allows you to specify a minimum and maximum time that will be consumed by the "tick". If there isn't any work to be done, the RTI will block waiting for an event rather than waiting in a busy-loop. Your loop might look something like this:

```
  while (!timeAdvGrant)
    rtiAmb.tick(0.1, 0.2);
```

This will block for intervals of 0.1 to 0.2 seconds until a time-advance is made.

10.11 Communication Channels

If I run on several machines all communication uses TCP protocols. If all federates on the same machine can I then use faster stream communication ?

What kind of streams are you referring to? Named pipes? Shared memory? There's currently not any support for this in RTI 1.0 or RTI 1.3 (recall the previous e-mail about the current RTI's not being very optimized with respect to running a large number of federates on a single machine.) However, the default TCP/IP interface used by the RTI should be smart enough to use the loopback interface for local traffic rather than sending packets out over the wire.

And on a WAN could I use secured communication (such as Secure Shell/Secure Socket Layer stuff)?

I'm not sure how SSL works. The RTI 1.0 always uses the default TCP/IP interface, so there's probably no way to run it over SSL. The RTI 1.3 allows you to specify device names for use with TCP and UDP communication, so if SSL works by providing an alternate device (i.e. a different named access point in "/dev/"), you'll be able to do it. I suspect that this isn't how it works, though.

10.12 Maximum Size of Attributes

If I try to add attributes with high cardinality e.g. 100000 the RTI will complain that

Error:ValueLengthExceeded :
RTIPairSet::add ValueLengthExceeded 14605

How can I increase the limit and does this change influence other things?

Unfortunately there's no way to change this limit, which is given by "MAX_BYTES_PER_VALUE" in "RTItypes.hh". One possible technique to work around this limit is to add multiple values for the same handle to the same AHVP-set (allowed by RTI 1.0), then defragment the large value at the receiver side. Be careful, because the values don't show up "on the other side" in the order in which they were added. In particular, I think the order is 1,n,n-1,n-2,n-3,...,2 where the numbers correspond to the sequence in which the handles were added and "n" is the total number of handles added.
RTI 1.3 does not place any (arbitrary) limit on the size of values, so you shouldn't have this problem with 1.3. Hopefully future RTI's will continue this trend.

10.13 Using the OMD file as basis instead of the FED file

I often battle with two problem in current HLA - which are of a general nature, hence it should be addressed by the people who design future HLA standards.

I would suggest that you make the OMD file (or alike) the basis of the federation instead of the FED file. The reason is that the OMD file then could be read when the program sends and receives attribute/interaction data. Currently it is too easy to make an error in this part of an HLA program, e.g., sending float (4 bytes) and receiving in double values (8 bytes). The support structure is rather weak here, however with the OMD file as a base then it would be feasible to enforce so much better control on the data type and cardinality checking etc.

This problem has not got an answer.

10.14 Endian handling in the Future

The problem with multiple endian simulators is very nasty. Currently I think that no firm specification have been made of whether big or little endian formatting is used. I find that it would be better if a specification of endian handling could be a part of the OMD specification, hence defining whether a given SOM (or the FOM) uses little or big endian formatting.
In this way the endian choice would be documented for all federates, hence reducing yet another practical problem. If so, a set of well-tested endian converting functions could be made a part of the RTI programs (and in case of single endian simulation the converting function simply should return its inputs directly for maximum speed).

This problem has not got an answer.