



CYBERNET

Cybernet Systems Inc.

Cybernet Systems Inc.  
727 Airport Blvd.  
Ann Arbor, MI 48108

Phone 734 668 2567  
Fax 734 668 8780  
<http://www.cybernet.com>

---

# High Level Architecture Run-Time Infrastructure SDK

---

Faster, Easier to Program, and  
Generate Code With High  
Performance for Massive Multi-player  
Networks

*Copyright, 2000, 2001, 2002 Cybernet Systems Corporation*

---

## Preface

The reader is assumed to be fluent in C++. It will be helpful to refer to documentation available from the DMSO at <http://www.dmsomil/hla>.

Cybernet Systems Corporation has implemented the OpenSkies Massive Multi-Player networking and simulation architecture to improve the real time and massive multi-player capabilities of the DMSO's HLA (<http://www.dmsomil/hla>). The following summarizes the significant difference between Cybernet' OpenSkies implementation and standard HLA:

- HLA is strictly client peer-to-peer. This leads to an  $N^2$  communication bottleneck between simulation and game clients. While acceptable over a local LAN or limited high performance WAN connections, this approach does not scale up well for massive multi-player applications. OpenSkies provides both a peer-to-peer implementation for small-scale multi-player networking and a peer to Lobby Manager/Router Network implementation. The OpenSkies Lobby Manager replaces the DMSO RTIexec. The later implementation allows the network implementer to separate the lobby manager from the federation host and to support a network of multiple federation hosts. Clients connect to the Lobby Manager, which in turn hands them off to a federation host (potentially one of many available to the lobby manager). Federation host media broadcast of client messages to each other, which has the effect of reducing the  $N^2$  communication problem to an  $N$  problem. Federation hosts also use game specific rules to parse messages and route only the ones which are needed to each client, reducing network traffic even further.
- HLA implements both a constructive simulation and a real time simulation mode. OpenSkies only implements the real time simulation mode.
- OpenSkies offers additional API extensions over DMSO HLA, which are described in Chapter 3.

In the Federation Object Model file (.FED), OpenSkies supports extensions over DMSO HLA, which are also documented in Chapter 3.

# Table of Contents

<b><u>PREFACE</u></b> .....	<b><u>II</u></b>
<b><u>TABLE OF CONTENTS</u></b> .....	<b><u>III</u></b>
<b><u>INTRODUCTION</u></b> .....	<b><u>1</u></b>
<b>HIGH LEVEL ARCHITECTURE (HLA)</b> .....	<b>1</b>
HLA RULES .....	1
HLA INTERFACE SPECIFICATION .....	1
OBJECT MODEL TEMPLATE SPECIFICATION(OMT).....	1
<b>RUN-TIME INFRASTRUCTURE(RTI)</b> .....	<b>1</b>
HOW TO INSTALL THE OPENSKIES RTI .....	2
HOW TO START EXECUTION .....	3
<b><u>CYBERNETRTI</u></b> .....	<b><u>4</u></b>
<b>SYSTEM REQUIREMENTS</b> .....	<b>4</b>
<b>C++ HEADER FILES AND LIB FILES</b> .....	<b>4</b>
<b>“HELLOWORLD” : AN EXAMPLE</b> .....	<b>15</b>
THE MAIN BODY OF CODE .....	15
MORE CODE.....	17
<b>INTERFACE CLASS REFERENCE</b> .....	<b>21</b>
HLA_RTI:CPROFILE .....	21
HLA_RTI:CLOBBYMANAGER .....	21
<b><u>CYBERNET’S RTI EXTENSIONS</u></b> .....	<b><u>23</u></b>
<b>LOBBYMANAGER</b> .....	<b>23</b>
ADDITIONAL APIS.....	23
ADDITIONAL RUN-TIME FEATURES WHEN USING PUBLIC SERVER.....	24
<b>FEDERATIONHOST</b> .....	<b>26</b>
<b>THE FEDHOST EXECUTABLE</b> .....	<b>27</b>
<b>CYBERNET FOM LIBRARY</b> .....	<b>30</b>

## Introduction

### What is “High Level Architecture”(HLA) and what is “Run-Time Infrastructure”(RTI)

#### High Level Architecture (HLA)

HLA is a general-purpose architecture for simulation reuse and interoperability. It is defined by the Defense Modeling and Simulation Office (DMSO). It consists of three parts.

- 1) HLA Rules.
- 2) HLA Interface Specification.
- 3) Object Model Template Specification.

#### HLA Rules

HLA rules defines HLA, its components, and the responsibilities of federates and federations. The official document can be found at [High-Level Architecture Rules Version 1.3](#).

#### HLA Interface Specification

This is a language independent specification for the HLA functional interfaces between federates and the runtime infrastructure (RTI). The official document can be found at [Draft Standard for Modeling and Simulation, Federate Interface Specification](#). This document describes the OpenSkies implementation of the RTI.

#### Object Model Template Specification (OMT)

To support its general goals, the HLA requires that federations and individual federates be described by an object model which identifies the data exchanged at runtime in order to achieve federation objectives. The HLA OMT provides a template for documenting HLA-relevant information about classes of simulation or federation objects and their attributes and interactions. This common template facilitates understanding and comparisons of different simulations and federations, and provides the format for a contract between members of a federation on the types of objects and interactions that will be supported across its multiple interoperating simulations. The official document on OMT can be found at [OMT Specification](#).

#### Run-Time Infrastructure (RTI)

While the HLA is an architecture, not software, use of runtime infrastructure (RTI) software is required to support operations of a federation execution. The RTI software provides a set of services, which are used by, federates to coordinate their operations and data exchange during a runtime execution.

## How to Install the OpenSkies RTI

There are two different kinds of installations. One uses a local server, and one uses servers that are provided outside of your LAN by a third party, such as Cybernet Systems Corporation. CybernetRTI can be used under Microsoft Windows 9x, Windows NT, and Windows 2000. LINUX will be supported in the near future.

### Local Installations

For local installations, two code modules need to be installed for RTI: HLA-RTI.DLL and LobbyManager.exe. Besides the two code modules, a file with "FED" extension is also needed. This file can be created and edited with "Object Model Development Tool (OMDT)" from AEGIS Research, or simply with an ASCII text editor. An example of it is "HelloWorld.fed" from DMSO.

HLA-RTI.DLL and LobbyManager.exe can be installed in the same directory as your own application. LobbyManager.exe is a command line application. It maintains a list of running federations. Applications can call LobbyManager.exe via RPC calls to get information such as a complete list of running federations, the host machine for each federation, etc. HLA-RTI.DLL is linked into your application at run time. When hosting a federation, it maintains a list of federates in the federation. The HLA-RTI.DLL itself parses the FED file, and maintains a list of object classes as well as a list of interaction classes. It also keeps track of published and subscribed object and interactions of each federate. When reliable data transmission is required, it distributes data to each federate. It also acts as a client federate for the local computer that is doing the hosting. When acting as a client federate, it connects to the federation host, and provides all RTI interface API's to the application.

The following registry values can be modified to customize the installation. ***They can be modified via functions provided in the SDK.*** You might provide a dialog box in your application to allow the end user to modify them:

Software/"Cybernet Systems Inc.)/CybernetRTI/LobbyManager section:

1. Address. IP address for the machine that LobbyManager runs on. For example, 192.168.0.2
2. Port. Port that LobbyManager uses. The default is "2000". It is a string value.

Software/"Cybernet Systems Inc.)/CybernetRTI/Multicast section:

1. Address. Multicast IP address. The default is 224.9.9.1.
2. Port. Multicast port base value. The default is 22500. It is a DWORD value. This base address is used by LobbyManager to acknowledge its own existence. Each federation will receive a multicast port address from the LobbyManager, which is larger than the base value and smaller than or equal to the maximum port number.
3. MaxPort. Maximum multicast port number. The default is 23500. When all ports between the base port and this port are used up, no more federations can be created.
4. TTL. Multicast TTL.
5. NICAddress. Network interface card IP address. This can be useful when you have multiple NICs in your machine.
6. QueueSizeLimit. Multicast is used for "best effort" communication. Multicast packets are placed in a queue when they arrive. If the queue size has reached this limit, new packets will be abandoned.

Software/"Cybernet Systems Inc.)/CybernetRTI/Fedex section:

1. Address. This is a network interface card IP address that is used when hosting a federation. This can be useful when you have multiple NIC in your machine.
2. Port. This is used for TCP connections while hosting a federation.

Installations that use outside servers

Such installations are mostly the same as local installations, except that you do not install LobbyManager.exe. When hosting a federation, a FederationHost process spawned by LobbyManager on the remote server maintains a list of federates in the federation. The FederationHost process parses the FED file, and maintains a list of object classes as well as a list of interaction classes. It also keeps track of published and subscribed object and interactions of each federate. When reliable data transmission is required, it distributes data to each federate.

In this kind of installations, registry entries for MaxPort and TTL of multicast are not used. The entire Software/"Cybernet Systems Inc.)/CybernetRTI/Fedex section is not used, either.

### How to Start Execution

For local installations, LobbyManager must be started. You can either start it as a stand-alone application or spawn it from your own application. It reads the LobbyManager section in the registry, as described above. Other than setting up the registry, you code does not need to do anything more for LobbyManager. You do NOT need to run "rtiexec" or "fedex".

For all installations, you should create a HLA\_RTI::CLobbyManager class in you own code. Make sure that this class is present at startup time as well as shutdown time. At startup time, you should call its "Init" member function to initialize it, and at shutdown time, you should call its "Delnit" member function to clean up. The prototype of the Init function is `BOOL Init(DWORD dwUDPPort, BOOL fSearch, DWORD dwTime, BOOL fUseLocalAddress, BOOL fUsePublicServer, BOOL fUseMulticast)`; where "DWORD dwUDPPort" specifies a port for UDP communication. It can be set to 0, and the operating system will pick an available port number. "BOOL fSearch" specifies whether to search for LobbyManager.exe via multicast ping. If you know LobbyManager.exe has already started, or you are going to start it by yourself, you can set fSearch to FALSE. "DWORD dwTime" specifies how long to search for if fSearch is TRUE. If "BOOL fUseLocalAddress" is TRUE, we assume that LobbyManager.exe is running locally. Otherwise we assume that it is running at IP address specified in the registry. For local installations, "BOOL fUsePublicServer" should be set to FALSE. Otherwise a remote server outside of your LAN is assumed. "BOOL fUseMulticast" simply turns multicast on or off. If you use remote server, and your Internet connection does not support multicast, you must set this parameter to FALSE. "Delnit" does not take any parameters.

Between "Init" and "Delnit", you can follow RTI examples provided by DMSO, such as "Hello world". For more information, please read the next chapter.

## CybernetRTI

### Cybernet Systems Corp.'s OpenSkies Implementation of RTI

This SDK is Cybernet Systems Corp.'s OpenSkies implementation of RTI for Microsoft Win32. It can be compiled and linked with your C++ applications. It comes with a setup program. It will install the necessary components for you.

#### System Requirements

CybernetRTI is to be used with Microsoft Visual C++ version 6.0. The code generated with CybernetRTI will run under Microsoft Windows 9x, Windows Me, Windows NT 4.0, and Windows 2000. LINUX version will be available soon.

#### C++ Header files and lib files

The header files that can be included in your applications are RTI.hh, RTITypes.hh, LobbyManager.h, and HLA\_RTIPProfile.h. They are placed in <Installation Directory>\include. The only difference between RTI.hh, RTITypes.hh and the DMSO versions is that static functions use "fastcall" declaration specifications. Other include files can be obtained from the DMSO distribution.

There is just one lib file that should be linked with your application: HLA\_RTI.lib. It is placed in <Installation Directory>\lib.

The definition of [RTI::RTIAmbassador](#), [RTI::FederateAmbassador](#), and [other supporting classes](#) are exactly the same as described in the DMSO document. Certain features such as regions are not supported because they are replaced by high performance CybernetRTI extensions, such as run-time culling.

#### Member functions of RTI::RTIAmbassador that are supported by CybernetRTI

##### FEDERATION MANAGEMENT

1. createFederationExecution;
2. destroyFederationExecution;
3. joinFederationExecution;
4. resignFederationExecution;

##### DECLARATION MANAGEMENT

5. publishInteractionClass;
6. publishObjectClass;

7. subscribeInteractionClass;
8. subscribeObjectClassAttributes;
9. unpublishInteractionClass;
10. unpublishObjectClass;
11. unsubscribeInteractionClass;
12. unsubscribeObjectClass;

## OBJECT MANAGEMENT

13. deleteObjectInstance;
14. registerObjectInstance;
15. requestClassAttributeValueUpdate;
16. sendInteraction;
17. updateAttributeValues;

## TYPES AND ANCILLARY SERVICES

18. getInteractionClassHandle;
19. getParameterHandle;
20. getObjectClassHandle;
21. getAttributeHandle;
22. tick;
23. RTIAmbassador;
24. ~ RTIAmbassador;

### Member functions of RTI::RTIAmbassador that are not supported by CybernetRTI

CybernetRTI does not support functions that are only supported by DMSO RTI 1.0 but not by DMSO RTI 1.3. They are listed below along with other functions that are not supported by CybernetRTI. They fall into 4 categories: Region related, time regulation related, Save and restore related, and ownership related. Some of them, such as the ownership related ones, will be supported by CybernetRTI in the near future. For data recording, Cybernet offers DCAE record/playback. For detail please contact Cybernet Systems Corp. Region related functions are replaced by culling on FedHost servers. The client code will no longer need to deal with regions.

## FEDERATION MANAGEMENT

1. federateRestoreComplete;
2. federateRestoreNotComplete;
3. federateSaveBegun;
4. federateSaveAchieved;
5. federateSaveComplete;
6. federateSaveNotAchieved;
7. federateSaveNotComplete;
8. pauseAchieved;
9. registerFederationSynchronizationPoint;
10. requestFederationRestore;
11. requestFederationSave;

12. requestPause;
13. requestRestore;
14. requestResume;
15. restoreAchieved;
16. restoreNotAchieved;
17. resumeAchieved;
18. synchronizationPointAchieved;

#### DECLARATION MANAGEMENT

19. subscribeObjectClassAttribute;
20. unsubscribeObjectClassAttribute;

#### OBJECT MANAGEMENT

21. changeAttributeTransportType;
22. changeInteractionTransportType;
23. localDeleteObjectInstance;
24. registerObject;
25. requestID;
26. requestObjectAttributeValueUpdate;

#### OWNERSHIP MANAGEMENT

27. attributeIsOwnedByFederate;
28. attributeOwnershipAcquisition;
29. attributeOwnershipAcquisitionIfAvailable;
30. attributeOwnershipReleaseResponse;
31. cancelAttributeOwnershipAcquisition;
32. cancelNegotiatedAttributeOwnershipDivestiture;
33. isAttributeOwnedByFederate;
34. negotiatedAttributeOwnershipDivestiture;
35. queryAttributeOwnership;
36. requestAttributeOwnershipAcquisition;
37. requestAttributeOwnershipDivestiture;
38. unconditionalAttributeOwnershipDivestiture;

#### TIME MANAGEMENT

39. changeAttributeOrderType;
40. changeInteractionOrderType;
41. disableAsynchronousDelivery;
42. disableTimeConstrained;
43. disableTimeRegulation;
44. enableAsynchronousDelivery;
45. enableTimeConstrained;

46. enableTimeRegulation;
47. flushQueueRequest;
48. modifyLookahead;
49. nextEventRequest;
50. nextEventRequestAvailable;
51. queryFederateTime;
52. queryLBTS;
53. queryLookahead;
54. queryMinNextEventTime;
55. requestFederateTime;
56. requestFederationTime;
57. requestLBTS;
58. requestLookahead;
59. requestMinNextEventTime;
60. retract;
61. setLookahead;
62. setTimeConstrained;
63. timeAdvanceRequest;
64. timeAdvanceRequestAvailable;
65. turnRegulationOff;
66. turnRegulationOn;
67. turnRegulationOnNow;

#### DATA DISTRIBUTION MANAGEMENT

68. associateRegionForUpdates;
69. createRegion;
70. deleteRegion;
71. notifyAboutRegionModification;
72. registerObjectInstanceWithRegion;
73. requestClassAttributeValueUpdateWithRegion;
74. sendInteractionWithRegion;
75. subscribeInteractionClassWithRegion;
76. subscribeObjectClassAttributesWithRegion;
77. unassociateRegionForUpdates;
78. unsubscribeInteractionClassWithRegion;
79. unsubscribeObjectClassWithRegion;

#### TYPES AND ANCILLARY SERVICES

80. dequeueFIFOasynchronously;
81. disableAttributeRelevanceAdvisorySwitch;
82. disableAttributeScopeAdvisorySwitch;
83. disableClassRelevanceAdvisorySwitch;
84. disableInteractionRelevanceAdvisorySwitch;
85. enableAttributeRelevanceAdvisorySwitch;
86. enableAttributeScopeAdvisorySwitch;

87. enableClassRelevanceAdvisorySwitch;
88. enableInteractionRelevanceAdvisorySwitch;
89. getAttributeName;
90. getAttributeRoutingSpaceHandle;
91. getDimensionHandle;
92. getDimensionName;
93. getInteractionClassName;
94. getInteractionRoutingSpaceHandle;
95. getObjectClass;
96. getObjectClassName;
97. getObjectInstanceHandle;
98. getObjectInstanceName;
99. getOrderingHandle;
100. getOrderingName;
101. getParameterName;
102. getRegion;
103. getRegionToken;
104. getRoutingSpaceHandle;
105. getRoutingSpaceName;
106. getTransportationHandle;
107. getTransportationName;

### Additions to RTI::RTIAmbassador in CybernetRTI

In order to use CybernetRTI extensions to RTI::RTIAmbassador, you must construct HLA\_RTI::CRTIAmbassador instead of RTI::RTIAmbassador. The former is derived from RTI::RTIAmbassador. The new functions allow network clients specify the sizes of attributes before actual updates. This reduces the size of network data transmitted later on.

#### DECLARATION MANAGEMENT

1. void publishInteractionClass(RTI::InteractionClassHandle theClass, const CParameterHandleSet& parameterList);
2. void subscribeInteractionClassParameters(RTI::InteractionClassHandle theClass, const CParameterHandleSet& parameterList, RTI::Boolean active = RTI::RTI\_TRUE);

#### OBJECT MANAGEMENT

3. RTI::ObjectHandle registerObjectInstance(RTI::ObjectClassHandle theClass, const char \*theObject, RTI::ObjectHandle CullingHandle); This allows the network client to specify another object to be used for culling. The actual culling rule itself can be specified by application developer on the server.

Member functions of RTI::FederateAmbassador that are called by CybernetRTI

FEDERATION MANAGEMENT

- 25. createFederationExecution;
- 26. destroyFederationExecution;
- 27. joinFederationExecution;
- 28. resignFederationExecution;

DECLARATION MANAGEMENT

- 29. startRegistrationForObjectClass;
- 30. stopRegistrationForObjectClass;
- 31. turnInteractionOn;
- 32. turnInteractionOff;

OBJECT MANAGEMENT

- 33. discoverObjectInstance;
- 34. provideAttributeValueUpdate;
- 35. reflectAttributeValues(RTI::ObjectHandle theObject, const RTI::AttributeHandleValuePairSet & theAttributes, const char \*theTag);
- 36. removeObjectInstance(RTI::ObjectHandle theObject, const char \*theTag);
- 37. turnUpdatesOffForObjectInstance;
- 38. turnUpdatesOnForObjectInstance;

Member functions of RTI::FederateAmbassador that are not called by CybernetRTI

Again, CybernetRTI does not call functions that are only supported by DMSO RTI 1.0 but not by DMSO RTI 1.3. They are listed below along with other functions that are not called by CybernetRTI.

FEDERATION MANAGEMENT

- 39. announceSynchronizationPoint;
- 40. federationNotRestored;
- 41. federationNotSaved;
- 42. federationRestoreBegun;
- 43. federationRestored;
- 44. federationSaved;
- 45. federationSynchronized;
- 46. initiateFederateRestore;
- 47. initiateFederateSave;
- 48. initiatePause;
- 49. initiateRestore;
- 50. initiateResume;
- 51. requestFederationRestoreFailed;

- 52. requestFederationRestoreSucceeded;
- 53. synchronizationPointRegistrationFailed;
- 54. synchronizationPointRegistrationSucceeded;

## DECLARATION MANAGEMENT

- 55. startInteractionGeneration;
- 56. startUpdates;
- 57. stopInteractionGeneration;
- 58. stopUpdates;

## OBJECT MANAGEMENT

- 59. attributesInScope;
- 60. attributesOutOfScope;
- 61. discoverObject;
- 62. void reflectAttributeValues(RTI::ObjectID theObject, const RTI::AttributeHandleValuePairset &theAttributes, RTI FederationTime theTime, const RTI::UserSuppliedTag theTag, RTI::EventRetractionHandle theHandle);
- 63. void reflectAttributeValues(RTI::ObjectHandle theObject, const RTI::AttributeHandleValuePairset &theAttributes, RTI FederationTime theTime, const RTI::UserSuppliedTag theTag, RTI::EventRetractionHandle theHandle);
- 64. reflectRetraction;
- 65. removeObject;
- 66. turnUpdatesOffForObjectInstance;
- 67. turnUpdatesOnForObjectInstance;

## OWNERSHIP MANAGEMENT

- 68. attributeIsNotOwned;
- 69. attributeOwnedByRTI;
- 70. attributeOwnershipAcquisitionNotification;
- 71. attributeOwnershipDivestitureNotification;
- 72. attributeOwnershipUnavailable;
- 73. confirmAttributeOwnershipAcquisitionCancellation;
- 74. informAttributeOwnership;
- 75. requestAttributeOwnershipAssumption;
- 76. requestAttributeOwnershipRelease;

## TIME MANAGEMENT

- 77. requestRetraction;
- 78. timeAdvanceGrant;

79. timeConstrainedEnabled;
80. timeRegulationEnabled;

#### Additions to RTI::FederateAmbassador in CybernetRTI

CybernetRTI has not made any additions to the RTI::FederateAmbassador class.

#### Member functions of RTI::AttributeHandleSet that are supported in CybernetRTI

1. size;
2. add;
3. getHandle;
4. isMember;
5. isEmpty;

#### Member functions of RTI::AttributeHandleSet that are not supported in CybernetRTI

1. empty;
2. remove;

#### Additions to RTI::AttributeHandleSet in CybernetRTI

In order to use CybernetRTI extensions to RTI::AttributeHandleSet, you may cast the RTI::AttributeHandleSet pointer returned by RTI::AttributeHandleSetFactory::create into a HLA\_RTI::CAttributeHandleSet pointer. HLA\_RTI::CAttributeHandleSet is derived from RTI::AttributeHandleSet. The new functions allow network clients to send data in a more compact format.

1. CBitFlags &GetFlags(void); This can be used to set HLA\_RTI::CAttributeHandleSet::fSorted and/or HLA\_RTI::CAttributeHandleSet::fFullList flags. For example, if we have a HLA\_RTI::CAttributeHandleSet pointer p, we can call p->GetFlags().Set(HLA\_RTI::CAttributeHandleSet::fSorted) to set the sorted flag.
2. void setFirstHandle(unsigned long Handle); If the attribute handle set is a sorted contiguous sequence of integers, you will need to call this function to specify the handle of the first attribute.
3. void add(unsigned long dwType, unsigned long dwSize); This allows network client to specify the size of attributes in a sorted handle set, so that when the network client sends network data later, it will not need to specify sizes for these attributes ever again.

#### Member functions of RTI::AttributeHandleValuePairSet that are supported in CybernetRTI

1. size;
2. add;

3. getHandle;
4. getValue;
5. getValueLength;
6. getValuePointer;

### Member functions of RTI::AttributeHandleValuePairSet that are not supported in CybernetRTI

1. getTransportType;
2. getOrderType;
3. getRegion;
4. remove;
5. moveFrom;
6. empty;
7. start;
8. valid;
9. next;

### Additions to RTI::AttributeHandleValuePairSet in CybernetRTI

In order to use CybernetRTI extensions to RTI::AttributeHandleValuePairSet, you may cast the RTI::AttributeHandleValuePairSet pointer returned by RTI::AttributeSetFactory::create into a HLA\_RTI::CAttributeVertexArray pointer. HLA\_RTI::CAttributeVertexArray is derived from RTI::AttributeHandleValuePairSet. The new functions allow network clients to send data in a more compact format.

1. CBitFlags &GetFlags(void); This can be used to set HLA\_RTI::CAttributeVertexArray::fSorted and/or HLA\_RTI::CAttributeVertexArray::fFullList flags. For example, if we have a HLA\_RTI::CAttributeVertexArray pointer p, we can call p->GetFlags().Set(HLA\_RTI::CAttributeVertexArray::fSorted) to set the sorted flag.
2. void setFirstHandle(unsigned long Handle); If the attribute handles form a sorted contiguous sequence, you will need to call this function to specify the handle of the first attribute.
3. void add(const char \*pValue, DWORD Size, bool fIncludeLength); This allows network client to specify the value of an attribute in a sorted array. If fIncludeLength is false, the length of the value buffer will not be sent onto the network. This function must be called in the same order as the handle values. For example, you must add the value of attribute 0 before all other attributes.
4. DWORD GetBufferSize(void) const. This returns the size of the raw data buffer.
5. const BYTE \*GetBuffer(void) const. This returns the actual data buffer. If the fSorted is on, i.e., if we have a HLA\_RTI::CAttributeVertexArray pointer p, and p->GetFlags().Check(HLA\_RTI::CAttributeVertexArray::fSorted) returns a non-zero value, we can read incoming data using GetBufferSize and GetBuffer functions. For example, if we send a packed structure {double d; float f; short s;}, and GetBuffer returns pBuffer, we can cast pBuffer into double \* to get the value of d, increment pBuffer by sizeof(double), cast it into float \* to get the value of f, and then increment pBuffer by sizeof(float), cast it into short \* to get the value of s. GetBufferSize should return no less than sizeof(double) + sizeof(float) + sizeof(short);

Member functions of RTI::ParameterHandleValuePairSet that are supported in CybernetRTI

1. size;
2. add;
3. getHandle;
4. getValue;
5. getValueLength;
6. getValuePointer;

Member functions of RTI::ParameterHandleValuePairSet that are not supported in CybernetRTI

1. getTransportType;
2. getOrderType;
3. getRegion;
4. remove;
5. moveFrom;
6. empty;
7. start;
8. valid;
9. next;

Additions to RTI::ParameterHandleValuePairSet in CybernetRTI

In order to use CybernetRTI extensions to RTI::ParameterHandleValuePairSet, you may cast the RTI::ParameterHandleValuePairSet pointer returned by RTI::ParameterSetFactory::create into a HLA\_RTI::CParameterValueSet pointer. HLA\_RTI::CParameterValueSet is derived from RTI::ParameterHandleValuePairSet. The new functions allow network clients to send data in a more compact format.

1. CBitFlags &GetFlags(void); This can be used to set HLA\_RTI::CParameterValueSet::fSorted and/or HLA\_RTI::CParameterValueSet::fFullList flags. For example, if we have a HLA\_RTI::CParameterValueSet pointer p, we can call p->GetFlags().Set(HLA\_RTI::CParameterValueSet::fSorted) to set the sorted flag.
2. void setFirstHandle(unsigned long Handle); If the attribute handles form a sorted contiguous sequence, you will need to call this function to specify the handle of the first attribute.
3. void add(const char \*pValue, DWORD Size, bool fIncludeLength); This allows network client to specify the value of an attribute in a sorted array. If fIncludeLength is false, the length of the value buffer will not be sent onto the network. This function must be called in the same order as the handle values. For example, you must add the value of attribute 0 before all other attributes.
4. void setCullingHandle(DWORD dwObjectHandle). This allow network client to turn on culling for this interaction transmission based on the specified object.
5. DWORD GetBufferSize(void) const. This returns the size of the raw data buffer.
6. const BYTE \*GetBuffer(void) const. This returns the actual data buffer. If the fSorted is on, i.e., if we have a HLA\_RTI::CAttributeValueArray pointer p, and p->

GetFlags().Check(HLA\_RTI::CAttributeValueArray::fSorted) returns a non-zero value, we can read incoming data using GetBufferSize and GetBuffer functions. For example, if we send a packed structure {double d; float f; short s;}, and GetBuffer returns pBuffer, we can cast pBuffer into double \* to get the value of d, increment pBuffer by sizeof(double), cast it into float \* to get the value of f, and then increment pBuffer by sizeof(float), cast it into short \* to get the value of s. GetBufferSize should return no less than sizeof(double) + sizeof(float) + sizeof(short);

### Factory classes that are supported in CybernetRTI

1. RTI::AttributeSetFactory;
2. RTI::AttributeHandleSetFactory;
3. RTI::ParameterSetFactory;

### Factory class that are not supported in CybernetRTI

There is only one factory class that is not supported by CybernetRTI. That is RTI::FederateHandleSetFactory. CybernetRTI supports direct queries to LobbyManager for federation information. There is no need to join any federation or publish and subscribe to anything for federation management.

### Additional factory class in CybernetRTI

There is only one new factory class: HLA\_RTI::CParameterHandleSetFactory. It creates a HLA\_RTI::CParameterHandleSet class. HLA\_RTI::CParameterHandleSet is used for publishing interaction classes whose parameter handles are sorted and contiguous. One can specify the known sizes of parameters using HLA\_RTI::CParameterHandleSet in when publishing such interactions.

### Exceptions

Applications should be prepared to catch exceptions as indicated in the definition of DMSO RTI classes. CybernetRTI does not add any new exceptions. Exceptions that can be thrown by CybernetRTI extensions are listed in the following prototypes:

1. static HLA\_RTI::CParameterHandleSet\* \_\_fastcall  
HLA\_RTI::CParameterHandleSetFactory::create(DWORD count)  
throw(RTI::MemoryExhausted, RTI::ValueCountExceeded);
2. void HLA\_RTI::CRTIAmbassador::publishInteractionClass (RTI::InteractionClassHandle  
theClass, const CParameterHandleSet& parameterList) throw  
(RTI::InteractionClassNotDefined, RTI::OwnershipAcquisitionPending,  
RTI::FederateNotExecutionMember, RTI::ConcurrentAccessAttempted,  
RTI::SaveInProgress, RTI::RestoreInProgress, RTI::RTIinternalError);
3. RTI::ObjectHandle  
HLA\_RTI::CRTIAmbassador::registerObjectInstance(RTI::ObjectClassHandle theClass,  
const char \*theObject, RTI::ObjectHandle CullingHandle) throw(  
RTI::ObjectClassNotDefined, RTI::ObjectClassNotPublished,  
RTI::ObjectAlreadyRegistered, RTI::FederateNotExecutionMember,

```

        RTI::ConcurrentAccessAttempted, RTI::SaveInProgress, RTI::RestoreInProgress,
        RTI::RTIinternalError);
4. void
   HLA_RTI::CRTIAmbassador::subscribeInteractionClassParameters(RTI::InteractionClassH
   andle theClass, const CParameterHandleSet& parameterList, RTI::Boolean active =
   RTI::RTI_TRUE) throw( RTI::InteractionClassNotDefined,
   RTI::FederateNotExecutionMember, RTI::ConcurrentAccessAttempted,
   RTI::FederateLoggingServiceCalls, RTI::SaveInProgress, RTI::RestoreInProgress,
   RTI::RTIinternalError);

```

### Other CybernetRTI-only classes

Besides the new classes HLA\_RTI::CParameterHandleSetFactory and HLA\_RTI::CParameterHandleSet, CybernetRTI's main additions are the LobbyManager class and the profile class. We will discuss them in more detail in the next chapter.

## "HelloWorld" : An Example

### The main body of code

In an application that uses RTI, the first few things it should do is to declare a HLA\_RTI::CLobbyManager class, a CFederateAmbassador class (explained below), and a text string char \*szFederateName that uniquely identifies this federate:

```

HLA_RTI::CLobbyManager LobbyManager;           // A global variable
CFederateAmbassador FedAmb;                   // Another global variable
... ..                                        // Explained later
const char *szFederateName;                   // Initialized elsewhere

// The first thing to do is to initialize LobbyManager:
    if(!LobbyManager.Init(0, TRUE, 10, FALSE, FALSE, TRUE)) // Search for
    {                                                         // LobbyManager for
        /* Error handling code here */                       // 10 seconds. fUseLocalAddress
        return FALSE;                                       // is not used when fSearch is
    }                                                         // TRUE

// From here on most of the code will be standard RTI code, and can be
// found in DMSO RTI distribution.
// Create an RTI ambassador. Most RTI calls will be done through this pointer.
RTI::RTIambassador *prtiAmb;
try
{
    prtiAmb = new RTI::RTIambassador;
}
catch ( RTI::Exception& e )
{
    LobbyManager.DeInit();
    /* More error handling code here */
    return FALSE;
}

```

```

// Create a new federation called "HelloWorld"
const char szFedName[]="HelloWorld";
try
{
    prtIAmb->createFederationExecution(szFedName, "HelloWorld.fed");
}
catch (RTI::FederationExecutionAlreadyExists)
{
    // This OK. Another "HelloWorld already started". Do nothing.
}
catch ( RTI::Exception& e )
{
    delete prtIAmb;
    LobbyManager.DeInit();
    /* More error handling code here */
    return FALSE;
}

// Now join the federation
RTI::FederateHandle FederateID;
try
{
    // char *szFederateName should be a unique string name for each federate
    // It should be initialized prior to starting RTI code
    FederateID = prtIAmb->joinFederationExecution(szFederateName,
                                                (char *)szFedName, &FedAmb);
}
catch(RTI::FederateAlreadyExecutionMember &e)
{
    // A federate with our name is present
    prtIAmb->destroyFederationExecution(szFedName);
    delete prtIAmb;
    LobbyManager.DeInit();
    /* More error handling code here */
    return FALSE;
}
catch(RTI::CouldNotOpenFED &e)
{
    prtIAmb->destroyFederationExecution(szFedName);
    delete prtIAmb;
    LobbyManager.DeInit();
    /* More error handling code here */
    return FALSE;
}
catch(RTI::ErrorReadingFED &e)
{
    prtIAmb->destroyFederationExecution(szFedName);
    delete prtIAmb;
    LobbyManager.DeInit();
    /* More error handling code here */
    return FALSE;
}

// To be explained below
if(!PublishAndSubscribetoObjects(prtIAmb))

```

```

{
    prtIAmb->resignFederationExecution(
        RTI::DELETE_OBJECTS_AND_RELEASE_ATTRIBUTES );
    prtIAmb->destroyFederationExecution(szFedName);
    delete prtIAmb;
    LobbyManager.DeInit();
    /* More error handling code here */
    return FALSE;
}

// Now loop until an exit signal is received
while(!CheckExitSignal())
{
    prtIAmb->tick(0.0, 5.0);
}

// Now resign from the federation, and then destroy it.
try
{
    prtIAmb->resignFederationExecution(
        RTI::DELETE_OBJECTS_AND_RELEASE_ATTRIBUTES );
    prtIAmb->destroyFederationExecution(szFedName);
    }
    catch(...)
    {
    delete prtIAmb;
    LobbyManager.DeInit();
    /* More error handling code here */
    return FALSE;
    }

delete prtIAmb;
LobbyManager.DeInit();
return TRUE;

```

### More Code

The above is the main code body that uses RTI. It makes use of the following.

CFederateAmbassador FedAmb

This class must be derived from RTI::FederateAmbassador, and it must overload some of the RTI::FederateAmbassador member functions. These overloaded functions are callback functions. When something happens on the network, one of these callback functions will be called. Some of the most useful ones are listed below:

```

void CFederateAmbassador::startRegistrationForObjectClass(
    RTI::ObjectClassHandle theObjectClass)
    throw(RTI::ObjectClassNotPublished, RTI::FederateInternalError);

```

This function is called when someone on the network is now interested in objects in this class you have published. You should register objects in this class that you published.

```

void CFederateAmbassador::stopRegistrationForObjectClass(
    RTI::ObjectClassHandle theObjectClass)

```

```
throw(RTI::ObjectClassNotPublished, RTI::FederateInternalError);
```

This function is called when no one on the network is interested in objects in this class you have published anymore. You can unregister objects in this class that you published.

```
void CFederateAmbassador::turnInteractionsOn(
    RTI::InteractionClassHandle theInteraction)
    throw(RTI::InteractionClassNotPublished, RTI::FederateInternalError);
```

This function is called when someone on the network is now interested in the interaction you have published. You should start updating interactions in this class that you published.

```
void CFederateAmbassador::turnInteractionsOff(
    RTI::InteractionClassHandle theInteraction)
    throw(RTI::InteractionClassNotPublished, RTI::FederateInternalError);
```

This function is called when no one on the network is interested in the interaction you have published anymore. You should stop updating interactions in this class that you published.

```
void CFederateAmbassador::discoverObjectInstance(
    RTI::ObjectHandle theObject, // supplied C1
    RTI::ObjectClassHandle theObjectClass, // supplied C1
    const char *theObjectName) // supplied C4
    throw(RTI::CouldNotDiscover,
    RTI::ObjectClassNotKnown, RTI::FederateInternalError);
```

This function is called when an object of a class that you subscribed to is registered on the network. You should create an object locally for it, and store “theObject”.

```
void CFederateAmbassador::reflectAttributeValues(
    RTI::ObjectHandle theObject, // supplied C1
    const RTI::AttributeHandleValuePairSet& theAttributes, // supplied C4
    const char *theTag) // supplied C4
    throw(RTI::ObjectNotKnown, RTI::AttributeNotKnown, RTI::FederateOwnsAttributes,
    RTI::InvalidFederationTime, RTI::FederateInternalError);
```

This function is called when an object that you discovered earlier is updated. The updated values are in “theAttributes” The object is identified by “theObject”, as specified in the function above.

```
void CFederateAmbassador::reflectAttributeValues(
    RTI::ObjectHandle theObject, // supplied C1
    const class RTI::AttributeHandleValuePairSet &theAttributes,
    const class RTI::FedTime &theTime,
    const char *theTag, struct RTI::EventRetractionHandle_s)
    throw(RTI::ObjectNotKnown, RTI::AttributeNotKnown,
    RTI::FederateOwnsAttributes, RTI::FederateInternalError);
```

This function is the same as the one above, except with time input.

```
void CFederateAmbassador::receiveInteraction(
    RTI::InteractionClassHandle theInteraction,
    const class RTI::ParameterHandleValuePairSet &theParameters, const char *theTag)
    throw(RTI::InteractionClassNotKnown, RTI::InteractionParameterNotKnown,
    RTI::InvalidFederationTime, RTI::FederateInternalError);
```

This function is called when an interaction of a class that you subscribed to is updated on the network. The updated values are in “theParameters”.

```
void CFederateAmbassador::receiveInteraction(
    RTI::InteractionClassHandle theInteraction,
    const class RTI::ParameterHandleValuePairSet &theParameters,
    const class RTI::FedTime &theTime,
    const char * theTag,
    struct RTI::EventRetractionHandle_s theHandle)
    throw(RTI::InteractionClassNotKnown, RTI::InteractionParameterNotKnown,
        RTI::FederateInternalError);
```

This function is the same as the one above, except with time input.

```
void CFederateAmbassador::removeObjectInstance(
    RTI::ObjectHandle theObject,
    const char *theTag)
    throw(RTI::ObjectNotKnown, RTI::InvalidFederationTime,
        RTI::FederateInternalError);
```

This function is called when an object that you discovered earlier is removed. The object is identified by “theObject”, as specified in “discoverObjectInstance”.

```
void CFederateAmbassador::removeObjectInstance(
    RTI::ObjectHandle theObject,
    const class RTI::FedTime &,
    const char *theTag, struct RTI::EventRetractionHandle_s)
    throw(RTI::ObjectNotKnown, RTI::FederateInternalError);
```

This function is the same as the one above, except with time input.

```
void CFederateAmbassador::provideAttributeValueUpdate(
    RTI::ObjectHandle theObject,
    const class RTI::AttributeHandleSet &theAttributes)
    throw(RTI::ObjectNotKnown, RTI::AttributeNotKnown,
        RTI::AttributeNotOwned, RTI::FederateInternalError);
```

This function is called when someone on the network requests that you update an object that you registered earlier. The object is identified by “theObject”, as specified in “discoverObjectInstance”.

```
void CFederateAmbassador::turnUpdatesOnForObjectInstance(
    RTI::ObjectHandle theObject,
    const class RTI::AttributeHandleSet &theAttributes)
    throw(RTI::ObjectNotKnown, RTI::AttributeNotOwned,
        RTI::FederateInternalError);
```

This function is called when someone on the network is now interested in an object that you registered earlier. The object is identified by “theObject”, as specified in “discoverObjectInstance”. You should start updating this object on the network.

```
void CFederateAmbassador::turnUpdatesOffForObjectInstance(
    RTI::ObjectHandle theObject,
    const class RTI::AttributeHandleSet &theAttributes)
    throw(RTI::ObjectNotKnown, RTI::AttributeNotOwned,
        RTI::FederateInternalError);
```

This function is called when no one on the network is interested in an object that you registered earlier anymore. The object is identified by “theObject”, as specified in “discoverObjectInstance”. You should stop updating this object on the network.

#### PublishAndSubscribettoObjects

// Some global variables first. They need to be accessible from CFederateAmbassador. Of course, they should be placed in a proper class.

```
RTI::ObjectClassHandle CountryClassHandle;
RTI::AttributeHandle NameHandle;
RTI::AttributeHandle PopHandle;
```

```
BOOL PublishAndSubscribettoObjects(RTI::RTIambassador *prtiAmb)
```

```
{
    CountryClassHandle = prtiAmb->
        getObjectClassHandle("Country");

    NameHandle = ms_rtiAmb->
        getAttributeHandle("Name", CountryClassHandle);

    PopHandle = prtiAmb->
        getAttributeHandle("Population", CountryClassHandle);

    //-----
    // To actually subscribe and publish we need to build
    // an AttributeHandleSet that contains a list of
    // attribute type ids (AttributeHandle).
    //-----
    RTI::AttributeHandleSet *pAttributes;
    pAttributes = RTI::AttributeHandleSetFactory::create(2);

    pAttributes ->add(NameHandle);
    pAttributes ->add(PopHandle);

    prtiAmb->subscribeObjectClassAttributes(CountryClassHandle,
        * pAttributes);
    prtiAmb->publishObjectClass(CountryClassHandle,
        * pAttributes);

    pAttributes->empty();

    delete pAttributes; // De-allocate the memory

    // Interactions can be done similarly
}
```

#### CheckExitSignal

This function’s prototype is “BOOL CheckExitSignal(void);”. It is a very simple function. If your application is a command line application, it may be simply

```
BOOL CheckExitSignal(void){ return (_kbhit() == 0); }
```

If it is a GUI application, it may simply be

```
BOOL CheckExitSignal(void){ return fExit; }
```

where BOOL fExit = FALSE initially, and set to TRUE when WM\_QUIT is received.

## Interface Class Reference

### HLA\_RTI:CProfile

This class is declared in HLA\_RTIProfile.h. All members in this class are static. All registry section name strings and entry name strings, along with default profile values are declared in it.

```
static UINT MS_FASTCALL GetInt(LPCTSTR lpszSection,  
                               LPCTSTR lpszEntry, int nDefault);
```

Example:

```
DWORD dwMaxPort = HLA_RTI:CProfile::GetInt(HLA_RTI:CProfile::m_szMCastSection,  
HLA_RTI:CProfile::m_szMCastMaxPortEntry, DEFAULT_MAX_MCASTPORT);
```

```
static CString MS_FASTCALL GetString(LPCTSTR lpszSection,  
                                     LPCTSTR lpszEntry, LPCTSTR lpszDefault);
```

Example:

```
CString szLobbyManagerAddress =  
    HLA_RTI:CProfile::GetInt(HLA_RTI:CProfile::m_szLobbySection,  
                             HLA_RTI:CProfile::m_szLobbyAddrEntry, "192.168.0.1");
```

```
static BOOL MS_FASTCALL WriteInt(LPCTSTR lpszSection,  
                                 LPCTSTR lpszEntry, int nValue);
```

Example:

```
HLA_RTI:CProfile::WriteInt(HLA_RTI:CProfile::m_szMCastSection,  
                           HLA_RTI:CProfile::m_szMCastMaxPortEntry, dwMaxPort);
```

```
static BOOL MS_FASTCALL WriteString(LPCTSTR lpszSection,  
                                    LPCTSTR lpszEntry, LPCTSTR lpszValue);
```

Example:

```
HLA_RTI:CProfile::WriteString(HLA_RTI:CProfile::m_szLobbySection,  
                              HLA_RTI:CProfile::m_szLobbyAddrEntry, szLobbyManagerAddress);
```

### HLA\_RTI:CLobbyManager

This class is declared in LobbyManager.h.

```
void DeInit(void);
```

Call this function when exiting RTI code.

```
BOOL Init(DWORD dwUDPPort, BOOL fSearch, DWORD dwTime, BOOL  
          fUseLocalAddress, BOOL fUsePublicServer, BOOL fUseMulticast);
```

DWORD dwUDPPort: specifies a port for UDP communication. If it is set to 0, the operating system will choose the next available port. Ports below 1024 should not be

used. They are used by the operating system. Valid values are between 1025 and 65535, unless the port is taken by another application. This parameter is useful when the UDP traffic must go through a firewall, or a router using NAT.

BOOL fSearch: specifies whether to search for LobbyManager.exe via multicast ping. If you know LobbyManager.exe has already started, or you are going to start it yourself, you can set fSearch to FALSE.

DWORD dwTime: if fSearch is TRUE, this specifies how long to search for.

BOOL fUseLocalAddress: If this is TRUE, we assume that LobbyManager.exe is running locally. Otherwise we assume that it is running at IP address specified in the registry.

BOOL fUsePublicServer: If this is TRUE, we assume that LobbyManager.exe is running outside of your LAN.

BOOL fUseMulticast: If your network connection to the LobbyManager.exe does not support multicast, you must set this to FALSE.

```
static BOOL MS_FASTCALL IsLobbyManagerRunning(void);
```

Check if LobbyManager.exe is running.

```
static void MS_FASTCALL ShutDown(void);
```

This function will shutdown LobbyManager.exe, provided that it is running locally.

## Cybernet's RTI Extensions

### More than just the basic RTI

#### LobbyManager

In DMSO version of RTI, a list of active federation executions is maintained by an executable called RTIexec. Every federation execution is created and destroyed by RTIexec. In Cybernet's version of RTI, RTIexec is replaced by LobbyManager.

Besides replacing RTIexec, LobbyManager also has the following extended features.

#### Additional APIs

LobbyManager can be called directly from an RTI application via RPC to obtain information about the list of active federation executions. The following are member functions of HLA\_RTI::CLobbyManager class, which is declared in LobbyManager.h:

```
static BOOL FedexExist(const char *pExecutionName);
```

This function can be called to see if a federation named with pExecutionName already exists. It returns TRUE if it exists, and FALSE otherwise.

```
static int FindLobbyMember(const char *pLobbyMemberName, _SLobbyMember  
    *pLobbyMember);
```

This function can be called to retrieve information about a federation named with pExecutionName. It returns TRUE if successful, and FALSE otherwise. The requested information is returned in pLobbyMember. The memory space of pLobbyMember is provided by the caller.

```
static CString * GetHostID(const char *pszHostName);
```

This function retrieves the application-specified ID of a host of a federation named with pExecutionName. It returns NULL if failed.

```
static BOOL GetHostInfo(const char *pExecutionName, unsigned char szAddress[16],  
    unsigned char szPort[8]);
```

This function retrieves information needed for making a TCP connection to the host of a federation named with pExecutionName.

```
static CString *GetModelName(const char *pszHostName, const char *pszID);
```

This function retrieves the application-specified model name of an object with ID specified by pszID in a federation hosted by pszHostName. It returns NULL if failed.

```
Static BOOL GetFederateList (const char *pLobbyMemberName,  
    CTypedPtrList<CPtrList, CString *> *pStringList);
```

This function retrieves the list of federates in a federation hosted by pLobbyMemberName.

```
static int GetLobbyMemberCount(void);
```

This function retrieves the number o hosts available.

```
static int GetLobbyMemberNames(long IBufferSize, char *pBuffer);
```

This function retrieves the names of all available hosts. Each string is 32 bytes in length, but NULL-terminated.

```
static CString * GetScenarioTitle(const char *pszHostName);
```

This function retrieves the application specified scenario title of a federation hosted by pszHostName.

```
BOOL JoinGameLobby(const CScenarioHostInfo &ScenarioHostInfo, const CString  
    &szUserName);
```

This function is called implicitly if not called before creating a new federation. Calling it directly before creating a new federation gives the application a chance to store additional information about a federation in the federation list maintained by LobbyManager. The class CScenarioHostInfo is defined in LobbyManager.h. szUserName can be the name of the computer on the network, or something unique about the particular instance of the application that is calling this function.

```
void SetHostListChangeCallbackProc( __HostListChangeCallbackProc pProc);
```

This function allows application to setup a callback function. When there is a change in the list of federations, the application will be notified via the callback function.

### Additional Run-Time Features When Using Public Server

The following applies to installations that make use of servers out side of your local LAN. Such servers are provided by third party service providers such as Cybernet Systems Corporation.

LobbyManager can be placed on a "broker server" computer to manage a large network of federations over the Internet. The first feature for such purpose is user authentication. An application can use the "Login" member function of HLA\_RTI::CLobbyManager class to login to LobbyManager, and the "Logoff" member function to log off. HLA\_RTI::CLobbyManager is declared in LobbyManager.h.

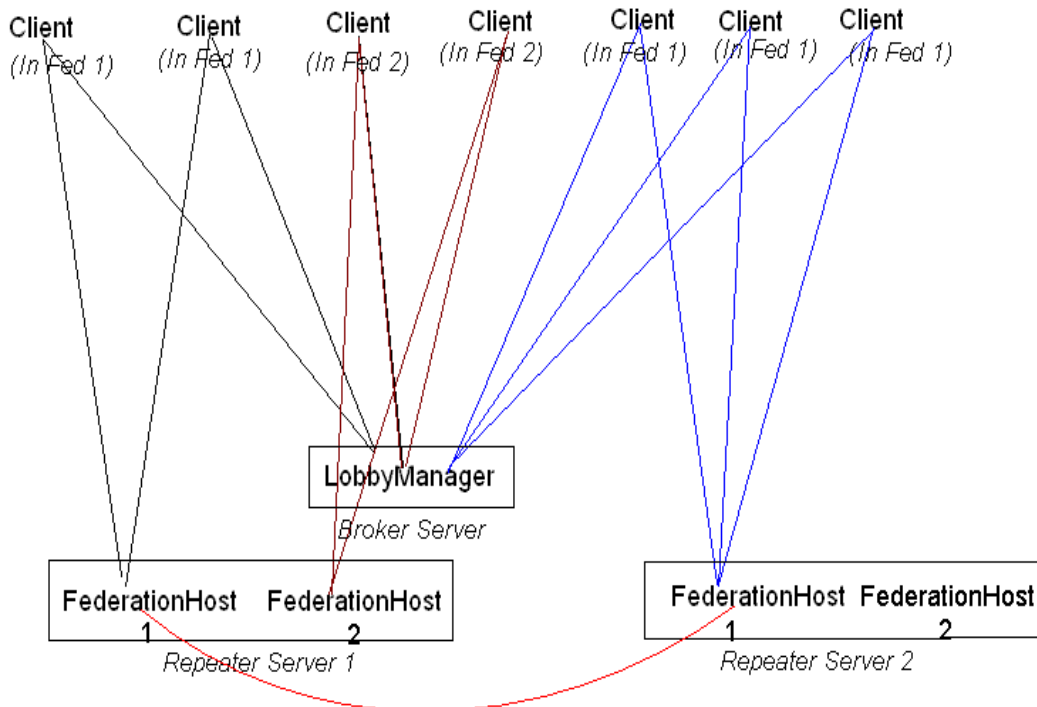
LobbyManager can keep track of a list of "repeater server" machines. Each "repeater server" machine will be running a copy of FedHost to be discussed later.

If a user requests to create a new federation, a new federation host process will be launched on each "repeater server" machine. If a user wants to join an existing federation, a least busy repeater server with the least ping time will be assigned to him. All FederationHost processes for the same federation on various repeater server machines will be able to communicate with each other, and each will have information about the entire federation.

For example, if a client requests LobbyManager to create a federation called "Fed1", LobbyManager makes sure that there is no federation called "Fed1" out there, and then create a hosting process for it on each repeater server, namely 1 and 2, called "FederationHost1". When another client requests to join "Fed1", it will be assigned to the least busy FederationHost1 on repeater server2. Based on the load percentage = number of clients / maximum number of allowed clients of each repeater server, the next client can be assigned to either repeater server 1 or repeater server 2. repeater server 1 and repeater server 2 share the same client list, the same object list, etc.

The reason that we do not start new federation host processes as needed is that when the number of federates is large (>= 100,000), starting new federation hosts can be very costly. You must update information for all federates on this new host. Thus adding new host processes can be used for system recovery after a partial crash, but not for dynamic client connections. However, new federation hosts can be added after a federation starts up when needed. They become available to users when all necessary updates are completed.

Suppose that a client requests to create another federation called "Fed2", LobbyManager will make sure that there is no federation called "Fed2" out there, and create new hosting processes called FederationHost2 on repeater server1 and repeater server 2, respectively. When another client requests to join "Fed2", it will be assigned to FederationHost2 on repeater server2, etc., just like for "Fed1".



LobbyManager run-time configuration is stored in a file named LobbyManager.ini. It must be stored in the same directory as LobbyManager itself. It is an ASCII file, and can be edited with a text editor. It contains 3 sections. The first section specifies NIC address and port number it uses. The second section specifies multicast address and port it uses. The

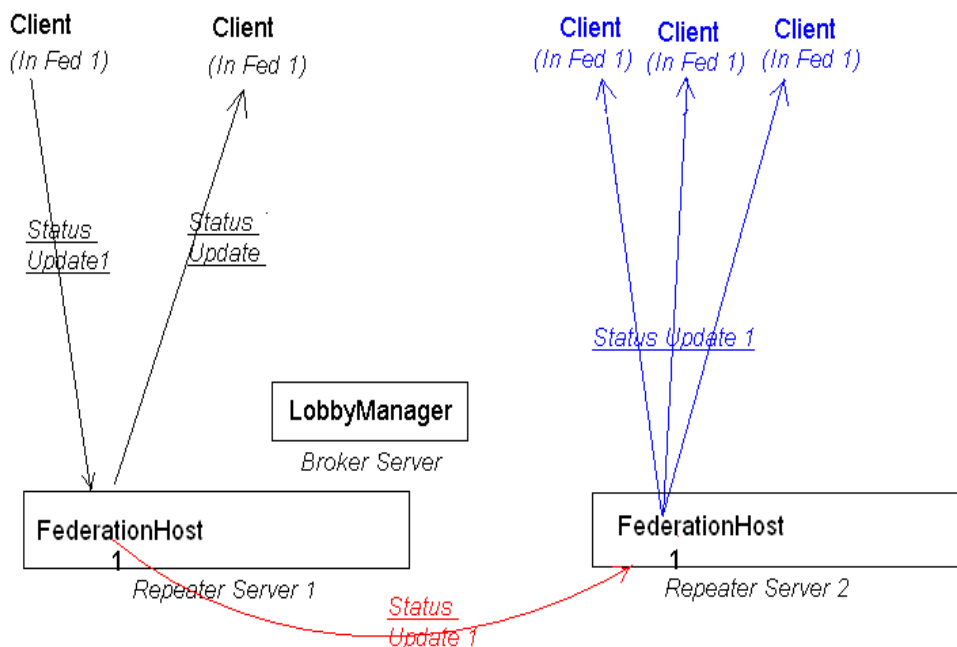
third section lists all repeater servers that it is supposed to manager. An actual copy of a LobbyManager.ini file can be found in Appendix A.

## FederationHost

As mentioned in previous section, when managing a large network of federations over the internet, LobbyManager will spawn FederationHost processes to host a federation. There is an executable called FedHost running on each repeater server at all times. When LobbyManager needs to spawn new FederationHost processes, it sends a command to each FedHost, and each FedHost will in turn spawn an actual FederationHost process on the repeater server that it resides on. *The actual code for FederationHost is contained in the FedHost executable.*

FederationHost will perform all the host functions in the existing RTI code. They communicate with each other via TCP/IP connections, IP multicast, and UDP datagrams. They will each maintain a complete list of federates, but each will communicate directly with a limited number of these clients.

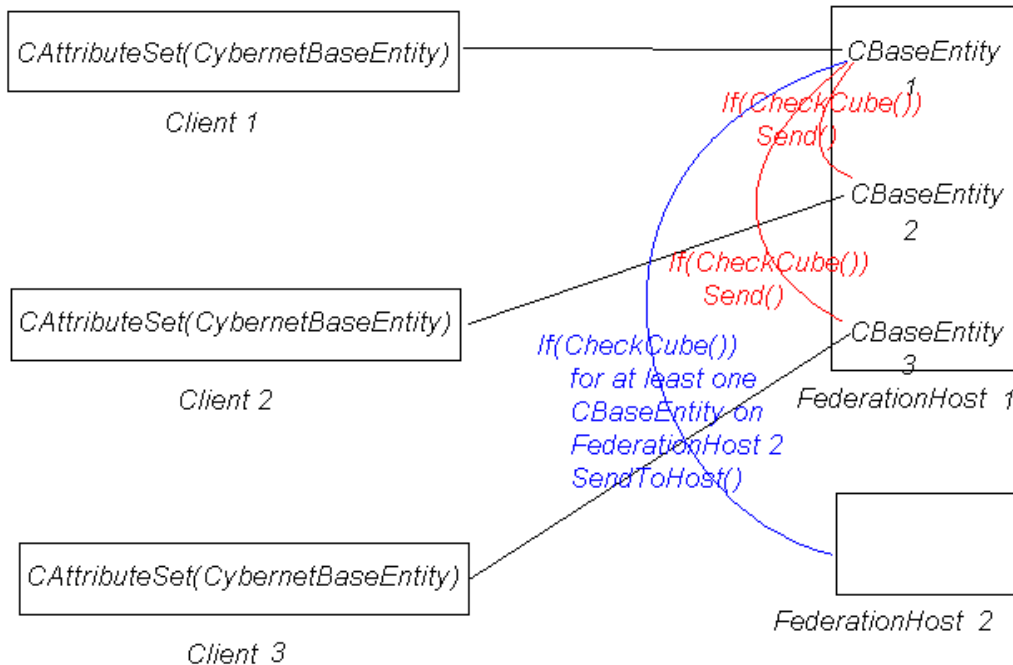
LobbyManager determines exactly which clients will communicate with a given FederationHost. The client code in HLA\_RTI.DLL receives the IP address of a repeater server, and a port address of a FederationHost Process from the LobbyManager after it logs in. Then the client code will establish a TCP connection with the FederationHost, as well as UDP communication.



Because clients do not communicate directly with each other, network traffic is greatly reduced. For example, if FederationHost 1 on repeater server 1 is hosting 10 users, and FederationHost 1 on repeater server 2 is hosting 10 users, for the update of the status of a single client connected to repeater server 1, there is going to be one and only one transmission of data from repeater server 1 to repeater server 2. The status update does not need to be done 10 times for each client connected to repeater server 2 between the repeater servers.

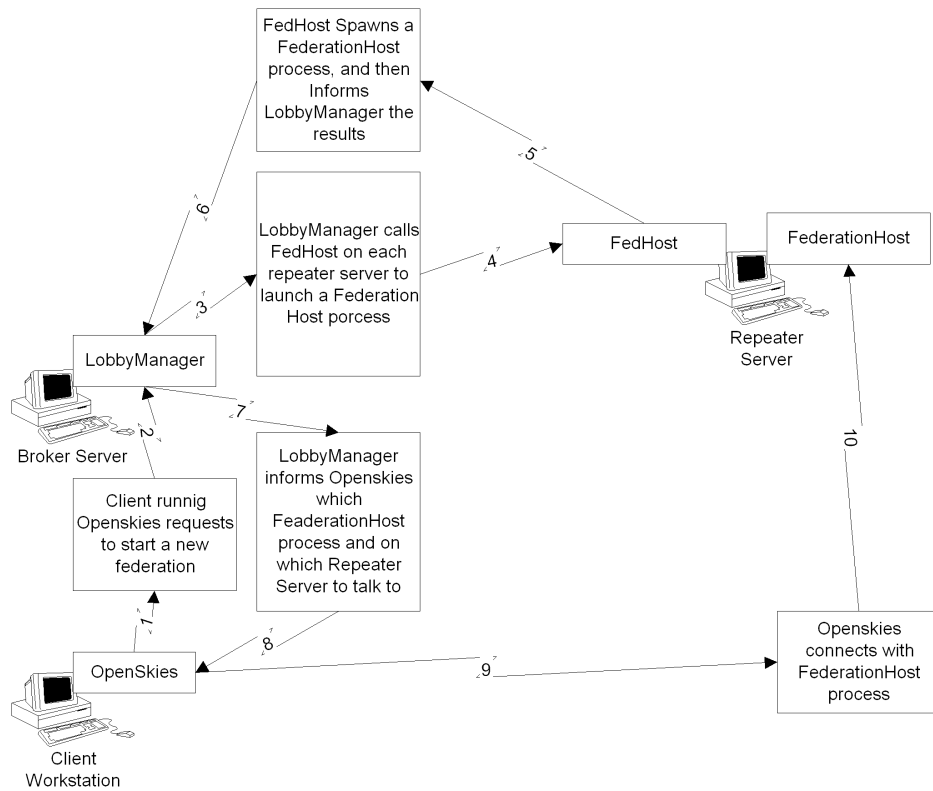
FederationHost will also perform the functions of culling to further reduce network traffic. Culling functions are defined as "member functions" of various attribute sets in FED files, which reside on repeater servers. These attribute set definitions will be provided by Cybernet in a FOM(federation object model) library. Application developers can also develop their own culling routines in DLL's, or "so" files under LINUX. Each function can be turned on and off at run-time. Because one can derive new attribute sets from existing ones, just like C++ class derivation with single inheritance, we can create other attribute sets, and not limit ourselves to what are in the FOM library. We will have a complete list of attribute set definitions in the next section.

For example, if we start with "CybernetBaseEntity", which consists of double Altitude, double Latitude, and double Longitude, we can override the "Cull" member function, and define it in such a way that it returns TRUE if "Altitude-Altitude0>=a1 and Altitude-Altitude0<=a2 and Latitude-Latitude0>=b1 and Latitude-Latitude0<=b2 and Longitude-Longitude0>=c1 and Longitude-Longitude0<=c2", and FALSE if otherwise. (Altitude0, Latitude0, Longitude0) is a "CybernetBaseEntity" that belongs to the receiving federate.

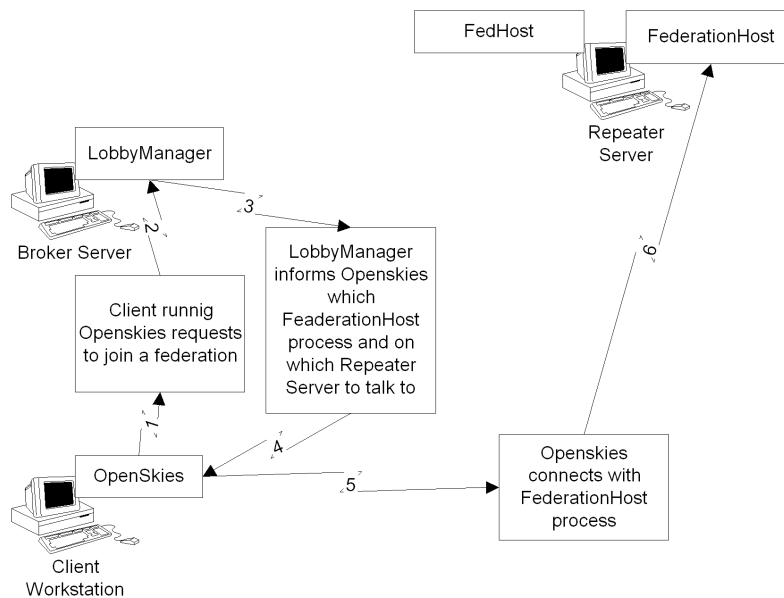


## The FedHost Executable

FedHost is an executable running on a repeater server. Each repeater server will have one and only one FedHost executable launched. When LobbyManager needs to launch a new FederationHost process on a repeater server, it connects to FedHost on that repeater server using TCP/IP, and sends the request. FedHost will launch the requested new FederationHost process and return the status of the new process to LobbyManager, so that a client application such as Openskies can connect with the FederationHost process (please see the diagrams below). *The code for each FederationHost process is inside the FedHost executable.*



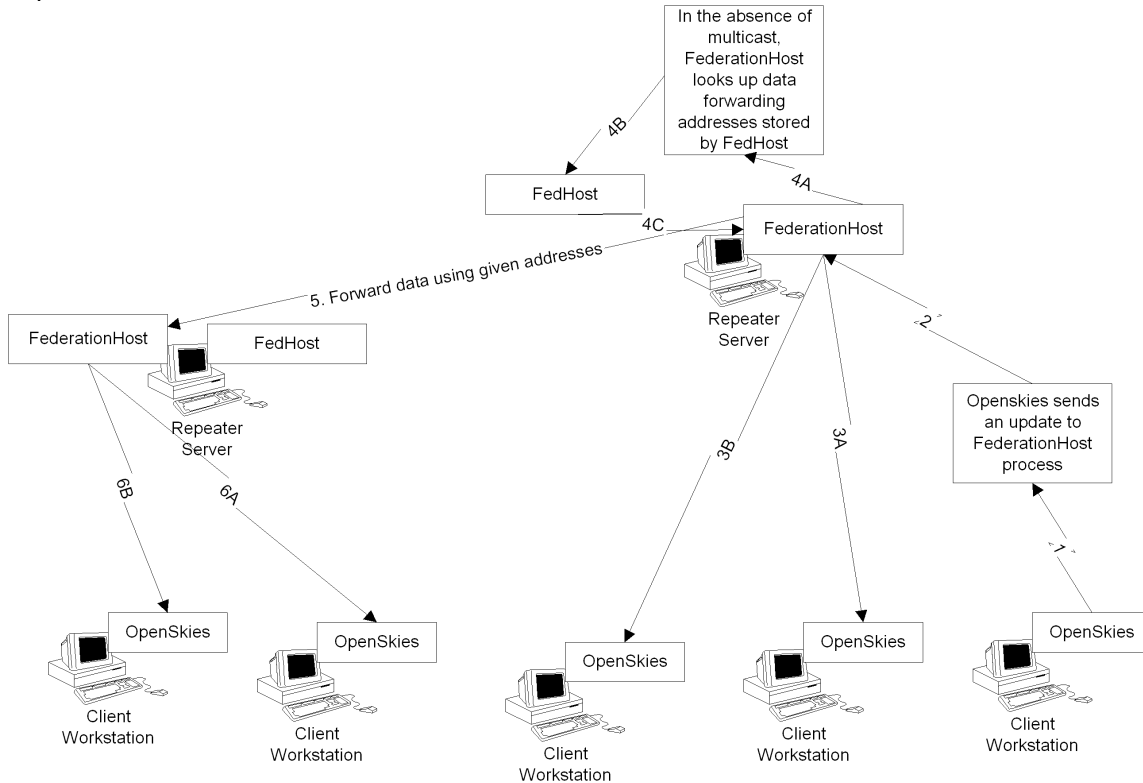
• Figure 1 Launching a New FederationHost Process



• Figure 2 Join a Federation

Run-time configuration of FedHost is stored in an ASCII file named FedHost.ini. It must be stored in the same directory as FedHost itself. It contains 3 sections. The first section contains the NIC address and port number it uses. It must match what is listed in the LobbyManager.ini file. The second section specifies socket options. The last section is for standard culling rules in the Cybernet FOM library. An actual copy of a FedHost.ini file can be found in Appendix B.

Besides acting as a process launcher, FedHost will also establish IP multicast traffic forwarding route from one repeater server to another when an IP multicast connection is not available amongst some Repeater Servers. It will select routes of least travel for all forwarded data.



• Figure 3. Forward Data Packets to Other FederationHosts

By relaying data amongst repeater servers, we can reduce network traffic through the Internet backbone considerably. If the topology amongst repeater servers is such that a datagram can reach every node with a single pass on every connection segment, the amount of data sent across the Internet is simply proportional to the number of clients. Consider the example of a flight simulator. An aircraft needs to transmit its altitude, latitude, longitude in doubles, heading, pitch, bank in floats, and ID in 32 bit integer for a total of 76 bytes at 10 Hz, i.e., 760 bytes/sec, in UDP datagrams. This number can be further reduced by not sending the high 32 word of each double every frame, for example. So the number becomes 64 bytes at 10 Hz, i.e., 640 bytes/sec. If there are 100 players, the amount of data sent across the Internet backbone is  $100 * 640 \text{ bytes/sec} = 64 \text{ kilobytes/sec}$ . For 500 users, it is about 320 KB/sec. And so on. Local traffic at each repeater server would still be  $n(=100)\text{-squared}$  times 640 bytes/sec with the absence of culling.

Due to the limited bandwidth that is available to each end user, we must rely on culling to significantly reduce the amount of data sent to each user. Assume that the user is using a 56kb/sec modem, the number of aircraft it can handle is about 10-15. Since we must leave room for infrequently transmitted data, plus things such as missiles and other projectiles, we should limit the number of planes to 10.

## Cybernet FOM Library

Cybernet FOM library is provided for network traffic culling. For detailed specifications of general-purpose attribute set definitions in a FED files, please see [Federation Execution Details \(FED\) Files Specification](#). The only deviation from this official FED specification in OpenSkies is the addition of culling rules. Culling rules allow the network designer to make associations between culling modules, which are loaded by the FedHost, and attribute set data (i.e. client to simulation messages).

At the beginning of a CybernetRTI FED file, there can be a “culling-rules” section that lists all available culling modules. The following is an example from an actual FED file:

```
(culling-rules
  (module main)
  (module CullingRulesII.so)
)
```

It lists two available culling modules: main and CullingRulesII.so.

Each culling module is a dynamic linking library that can be loaded by FedHost at run time. Within each culling module there are C++ classes that do the actual culling of attribute set data. These culling modules can be provided by the application developers, and placed on repeater servers, to be loaded by FedHost at run time. A special culling module called “main” is built into FedHost itself. For example, a class named “CFastVehicle” is included in “CullingRulesII.so”. It is designed to cull attribute sets of type “FastVehicle” in a certain FED file. To use classes defined in “CullingRulesII.so” for culling “FastVehicle” attribute sets, we must add an entry in the definition of FastVehicle in the FED file:

```
(class FastVehicle
  (attribute x besteffort receive)
  (attribute y besteffort receive)
  (attribute z besteffort receive)
  (culling-module CullingRulesII.so)
)
```

This will make all C++ classes in CullingRulesII.so that are for culling FastVehicle attribute sets available to FedHost. Actual run time configurations of these C++ classes are in an ASCII text file named CullingRulesII.ini. Alternatively, these run time configurations can also be altered via network communication at run time, e.g., if we add an attribute to FastVehicle called “EableCulling”:

```
(class FastVehicle
  (attribute EnableCulling besteffort receive)
  (attribute x besteffort receive)
  (attribute y besteffort receive)
  (attribute z besteffort receive)
  (culling-module CullingRulesII.so)
)
```

and let C++ classes in CullingRulesII.so check the status of this “EnableCulling” variable before culling, we can then turn culling on or off at run time from any client.

The following is a complete list of the content of Cybernet FOM library in alphabetical order. Their culling module “main” is built into FedHost itself. They can be configured in FedHost.ini.

1. BaseEntity2D

```
(class BaseEntity2D
  (attribute x besteffort receive)
  (attribute y besteffort receive)
  (culling-module main)
)
```

2. BaseEntity. Derived from BaseEntity2D.

```
(class BaseEntity
  (attribute z besteffort receive)
  (culling-module main)
)
```