

*Department of Defense
Defense Modeling and Simulation Office*



*High Level Architecture
Run-Time Infrastructure*

***RTI 1.3-Next Generation
Programmer's Guide
Version 3.2***

Sponsor:



Defense Modeling and Simulation Office
1901 N. Beauregard Street, Suite 504
Alexandria, VA 22311
(703) 998-0660
<http://www.dmsso.mil>

Developers:



Science Applications International Corporation
Distributed Computing Technology Division
5400 Shawnee Road, Suite 110
Alexandria, VA 22312
(703) 354-2063
<http://helpdesk.dctd.saic.com>



Virtual Technology Corporation
5400 Shawnee Road, Suite 203
Alexandria, VA 22312
(703) 658-7050
<http://www.virtc.com>



Object Sciences Corporation
P.O. Box 7175
Fairfax Station, VA 22039
(703) 250-4338
<http://www.objectsciences.com>



Dynamic Animations Systems
6035 Burke Center Parkway
Burke, VA 22015
(703) 503-0500
<http://www.d-a-s.com>

Errata – Programmer’s Guide (RTI 1.3NG-V3.2, 7 September 2000)

Revision 1 – 27 October 1999

Section 12.4.1, RTI Configuration Files – additional description and location information added to the FED and RID file sub-sections.

Section 12.4.1, RTI Executables – information updated to include RtiExec, FedExec, and Launcher table from Installation Guide.

Section 12.4.1, RTI Executables – information updated to include RtiExec and FedExec manual operation commands.

Section 12.4.1, Downloading the RTI Software – information updated to reflect current Software Distribution Center address.

Revision 2 – 10 December 1999

Section 12.3.5 Parameter Definition – RTI.rid parameter definitions updated to reflect current implementation.

Section 12.4 Removed – Repeated material from Installation Guide.

Appendix B.3.3 discoverObjectInstance – description corrected to reflect proper discovery criteria. (Third bullet deleted.)

Appendix C.1.1, AttributeHandleSet – exceptions corrected for AttributeHandleSet.

Revision 3 – 28 April 2000

Section 12.3.5 Parameter Definition – Change to actual RTI.rid file insertion to correctly reflect the RTI-NG 1.3v3 .rid file.

Preface

The *RTI Programmer's Guide* presents the Run-Time Infrastructure (RTI) – a fundamental component of the High Level Architecture (HLA). Readers of the guide are presumed to have modeling and simulation experience and prior exposure to the Department of Defense Modeling and Simulation Master Plan. Experienced developers should find the *RTI Programmer's Guide*, along with the companion “Hands-On Practicum” course, sufficient to begin using the RTI. Managers and System Architects porting old simulations to RTI 1.3-NG or planning new simulations should find this presentation helpful for identifying and assessing important issues. The guide examines the RTI application interface in considerable detail. The RTI software implements the *HLA Interface Specification*. Releases of the RTI software respond to releases of the *HLA Interface Specification*. Interesting releases are shown in Table P-1.

Interface Specification		RTI Software Release	Comments
Version 1.0	→	RTI F.0 (12/96)	No longer supported
Version 1.1 (02/97)	→	RTI 1.0 (05/97)	No longer supported
Version 1.2 (08/97)	X	None	
Version 1.3 (02/98)	→	RTI 1.3 (03/98)	Partial release
Version 1.3 (04/98)	→	RTI 1.3v2 (6/98) RTI 1.3v3 (7/98) RTI 1.3v4 (9/98) RTI 1.3v5 (12/98) RTI 1.3v6 (3/99)	
Version 1.3 (04/98)	→	RTI 1.3-NGv1 (9/99) RTI 1.3-NGv2 (12/99) RTI 1.3-NGv3 (4/00)	Verified Release

Table P-1. Mapping of Releases to Specifications

An RTI 1.1 release was originally planned to correspond to Version 1.2 of the interface specification, but the high frequency of specification and software releases and discrepancies in the release numbering schemes proved to be complicated. The Architecture Management Group (AMG) decided to postpone the 1.1 release of the RTI in order to implement the new 1.3 Interface Specification. A preliminary RTI 1.3 release (known as RTI 1.3 Beta) was distributed 01/98 to a small user community. The decision was made to synchronize RTI and interface version numbering as of the 1.3 release of the *HLA Interface Specification*. There is no RTI 1.2 release. The RTI 1.3-Next Generation (NG) release represents a “from scratch” implementation that builds upon the lessons learned from its predecessors. RTI 1.3-Next Generation version 1.1 is the first release on the RTI software to be fully verified against the 1.3 Interface Specification. This *RTI Programmer's Guide* covers only RTI 1.3-NG. The previous RTI 1.0 and RTI 1.3v6 Programmer's Guides are available on the DMSO website. Complete sets of methods reference pages for RTI 1.3-NG are provided in appendixes A through C. Service descriptions in the reference pages clearly delineate RTI 1.3-NG availability, syntax, and semantics. An index of these descriptions is provided at the end of this document. It references all service descriptions found in the three appendixes, in alphabetical order.

TABLE OF CONTENTS

PREFACE	I
1. INTRODUCTION TO HLA	1-1
1.1 FEDERATION RULES	1-3
1.2 INTERFACE SPECIFICATION.....	1-4
1.3 OBJECT MODEL TEMPLATE (OMT).....	1-4
1.4 CONCEPTUAL MODEL OF THE MISSION SPACE (CMMS).....	1-5
1.5 DATA STANDARDIZATION (DS).....	1-6
1.6 FURTHER READING.....	1-7
2. RTI SYNOPSIS	2-1
2.1 MAJOR COMPONENTS.....	2-1
2.2 RTIEXEC.....	2-2
2.3 FEDEXEC.....	2-3
2.4 LIBRTL.....	2-3
2.5 MANAGEMENT AREAS.....	2-4
2.5.1 <i>Federation Management</i>	2-6
2.5.2 <i>Declaration Management</i>	2-6
2.5.3 <i>Object Management</i>	2-7
2.5.4 <i>Ownership Management</i>	2-7
2.5.5 <i>Time Management</i>	2-8
2.5.6 <i>Data Distribution Management</i>	2-9
3. THE ROLE OF TIME	3-1
3.1 INTRODUCTION	3-1
3.2 TIME MANAGEMENT BASICS	3-1
3.3 "REGULATING" AND "CONSTRAINED"	3-2
3.3.1 <i>Regulating</i>	3-3
3.3.2 <i>Lookahead</i>	3-3
3.3.3 <i>TSO Event</i>	3-3
3.3.4 <i>Constrained</i>	3-3
3.3.5 <i>Lower bound time stamp (LBTS)</i>	3-3
3.4 ADVANCING TIME	3-4
3.4.1 <i>LBTS Constraint</i>	3-6
3.4.2 <i>Late Arriving Federate</i>	3-7
3.5 "RECEIVE-ORDERED" V. "TSO" EVENTS	3-8
3.5.1 <i>EXAMPLE 1</i>	3-9
3.5.2 <i>EXAMPLE 2</i>	3-10
3.5.3 <i>SUMMARY</i>	3-10
4. FOM/SOM DEVELOPMENT	4-1
5. FEDERATION MANAGEMENT	5-1
5.1 INTRODUCTION	5-1
5.2 PRIMARY FUNCTIONS	5-1
5.2.1 <i>RTIambassador::createFederationExecution()</i>	5-1
5.2.2 <i>RTIambassador::joinFederationExecution()</i>	5-2
5.2.3 <i>RTIambassador::tick()</i>	5-2
5.2.4 <i>RTIambassador::resignFederationExecution()</i>	5-2
5.2.5 <i>RTIambassador::destroyFederationExecution()</i>	5-3
5.3 FOODFIGHT EXAMPLE	5-3
5.4 FEDERATE SYNCHRONIZATION	5-6
5.5 SAVE/RESTORE.....	5-7

6.	TIME MANAGEMENT	6-1
6.1	INTRODUCTION	6-1
6.2	TOGGLING "REGULATING" AND "CONSTRAINED" STATUS	6-1
6.2.1.	<i>Regulation Policy</i>	6-2
6.2.2.	<i>Constrained Policy</i>	6-2
6.3	TIME ADVANCE REQUESTS	6-2
6.3.1.	<i>Time-Stepped Federates</i>	6-2
6.3.2.	<i>Event-Based Federates</i>	6-3
6.3.3.	<i>Optimistic Federates</i>	6-4
6.4	FOODFIGHT EXAMPLE	6-5
6.5	TIME-RELATED QUERIES	6-9
6.6	POLLING VS. ASYNCHRONOUSIO TICK() STRATEGIES	6-10
7.	DECLARATION MANAGEMENT	7-1
7.1	INTRODUCTION	7-1
7.2	OBJECT VOCABULARY REVIEW	7-1
7.3	OBJECT HIERARCHIES	7-2
7.4	PUBLISHING AND SUBSCRIBING OBJECTS	7-3
7.4.1.	<i>Object Publication</i>	7-4
7.4.2.	<i>Interaction Publication</i>	7-5
7.4.3.	<i>Object Subscription</i>	7-5
7.4.4.	<i>Interaction Subscription</i>	7-6
7.4.5.	<i>Control Signals</i>	7-6
7.5	OBJECT PUBLICATION AND SUBSCRIPTION	7-7
7.6	THROTTLING PUBLICATIONS	7-8
7.7	FOODFIGHT OBJECT DECLARATION	7-8
7.7.1.	<i>Excerpt from Student.h</i>	7-9
7.7.2.	<i>Dynamic Object Publication and Subscription</i>	7-11
7.8	PUBLISHING AND SUBSCRIBING INTERACTIONS	7-11
8.	OBJECT MANAGEMENT	8-1
8.1	REGISTERING, DISCOVERING, AND DELETING OBJECT INSTANCES	8-1
8.2	UPDATING AND REFLECTING OBJECT ATTRIBUTES	8-2
8.3	ENCODING AND OBJECT UPDATE	8-3
8.4	DECODING AND OBJECT REFLECTION	8-5
8.5	EXCHANGING INTERACTIONS	8-6
8.6	ADDITIONAL OBJECT CONTROL	8-7
8.6.1.	<i>Attribute Management</i>	8-8
8.6.2.	<i>Enable/Disable Attribute Management</i>	8-9
9.	OWNERSHIP MANAGEMENT	9-1
9.1	INTRODUCTION	9-1
9.1.1.	<i>Push v. Pull</i>	9-1
9.1.2.	<i>Privilege to Delete</i>	9-1
9.2	OWNERSHIP PULL	9-2
9.2.1.	<i>Attribute Ownership Acquisition</i>	9-4
9.2.2.	<i>Attribute Ownership Release</i>	9-4
9.3	OWNERSHIP PUSH	9-5
9.3.1.	<i>Unconditional Push</i>	9-5
9.3.2.	<i>Negotiated Push</i>	9-5
9.3.3.	<i>Complex Exchanges</i>	9-6
9.4	SUPPORTING FUNCTIONS	9-6
9.4.1.	<i>Cancellation</i>	9-6
9.4.2.	<i>Queries</i>	9-6

10.	DATA DISTRIBUTION MANAGEMENT	10-1
10.1	INTRODUCTION	10-1
10.2	EXAMPLE ROUTING SPACE	10-1
10.2.1.	<i>A Previous Example Revisited.....</i>	<i>10-1</i>
10.2.2.	<i>A Routing Space</i>	<i>10-2</i>
10.3	DEFINING ROUTING SPACES AND REGIONS.....	10-3
10.3.1.	<i>Routing Spaces.....</i>	<i>10-3</i>
10.3.2.	<i>Extents.....</i>	<i>10-4</i>
10.3.3.	<i>Calculation of Extents.....</i>	<i>10-4</i>
10.3.4.	<i>Creative Dimensions</i>	<i>10-5</i>
10.3.5.	<i>Regions and Attributes</i>	<i>10-6</i>
10.3.6.	<i>Oddly Shaped Regions</i>	<i>10-6</i>
10.3.7.	<i>Thresholds.....</i>	<i>10-7</i>
10.3.8.	<i>Default Routing Space.....</i>	<i>10-7</i>
10.4	CREATING REGIONS.....	10-7
10.5	BINDING OBJECT ATTRIBUTES TO REGIONS.....	10-8
10.5.1.	<i>Attribute Updates and Regions</i>	<i>10-8</i>
10.5.2.	<i>Attribute Subscriptions and Regions</i>	<i>10-9</i>
10.5.3.	<i>Requesting Updates.....</i>	<i>10-9</i>
10.5.4.	<i>Object Ownership and Regions.....</i>	<i>10-11</i>
10.5.5.	<i>Time and Regions.....</i>	<i>10-11</i>
10.6	BINDING INTERACTIONS TO REGIONS	10-11
11.	MANAGEMENT OBJECT MODEL	11-1
11.1	INTRODUCTION TO THE MANAGEMENT OBJECT MODEL.....	11-1
11.2	INTERACTIONS	11-2
	<i>Manager</i>	<i>11-2</i>
	<i>Manager.Federate.....</i>	<i>11-2</i>
	<i>Manager.Federate.Adjust.....</i>	<i>11-2</i>
	<i>Manager.Federate.Adjust.ModifyAttributeState.....</i>	<i>11-2</i>
	<i>Manager.Federate.Adjust.SetServiceReporting</i>	<i>11-2</i>
	<i>Manager.Federate.Adjust.SetExceptionLogging.....</i>	<i>11-3</i>
	<i>Manager.Federate.Adjust.SetTiming.....</i>	<i>11-3</i>
	<i>Manager.Federate.Report</i>	<i>11-3</i>
	<i>Manager.Federate.Report.Alert</i>	<i>11-3</i>
	<i>Manager.Federate.Report.ReportInteractionPublication</i>	<i>11-3</i>
	<i>Manager.Federate.Report.ReportInteractionsReceived.....</i>	<i>11-4</i>
	<i>Manager.Federate.Report.ReportInteractionsSent</i>	<i>11-4</i>
	<i>Manager.Federate.Report.ReportInteractionSubscription.....</i>	<i>11-4</i>
	<i>Manager.Federate.Report.ReportObjectInformation.....</i>	<i>11-4</i>
	<i>Manager.Federate.Report.ReportObjectPublication</i>	<i>11-5</i>
	<i>Manager.Federate.Report.ReportObjectsOwned</i>	<i>11-5</i>
	<i>Manager.Federate.Report.ReportObjectsReflected</i>	<i>11-5</i>
	<i>Manager.Federate.Report.ReportObjectSubscription.....</i>	<i>11-6</i>
	<i>Manager.Federate.Report.ReportObjectsUpdated.....</i>	<i>11-6</i>
	<i>Manager.Federate.Report.ReportReflectionsReceived</i>	<i>11-6</i>
	<i>Manager.Federate.Report.ReportServiceInvocation.....</i>	<i>11-7</i>
	<i>Manager.Federate.Report.ReportUpdatesSent</i>	<i>11-7</i>
	<i>Manager.Federate.Request</i>	<i>11-8</i>
	<i>Manager.Federate.Request.RequestInteractionsReceived</i>	<i>11-8</i>
	<i>Manager.Federate.Request.RequestInteractionsSent.....</i>	<i>11-8</i>
	<i>Manager.Federate.Request.RequestObjectInformation</i>	<i>11-8</i>
	<i>Manager.Federate.Request.RequestObjectsOwned</i>	<i>11-8</i>
	<i>Manager.Federate.Request.RequestObjectsReflected</i>	<i>11-8</i>
	<i>Manager.Federate.Request.RequestObjectsUpdated.....</i>	<i>11-8</i>

<i>Manager.Federate.Request.RequestPublications</i>	11-8
<i>Manager.Federate.Request.RequestReflectionsReceived</i>	11-8
<i>Manager.Federate.Request.RequestSubscriptions</i>	11-9
<i>Manager.Federate.Request.RequestUpdatesSent</i>	11-9
<i>Manager.Federate.Service</i>	11-9
<i>Manager.Federate.Service.ChangeAttributeOrderType</i>	11-9
<i>Manager.Federate.Service.ChangeAttributeTransportationType</i>	11-9
<i>Manager.Federate.Service.ChangeInteractionOrderType</i>	11-9
<i>Manager.Federate.Service.ChangeInteractionTransportionType</i>	11-9
<i>Manager.Federate.Service.DeleteObjectInstance</i>	11-10
<i>Manager.Federate.Service.DisableAsynchronousDelivery</i>	11-10
<i>Manager.Federate.Service.DisableTimeConstrained</i>	11-10
<i>Manager.Federate.Service.DisableTimeRegulation</i>	11-10
<i>Manager.Federate.Service.EnableAsynchronousDelivery</i>	11-10
<i>Manager.Federate.Service.EnableTimeConstrained</i>	11-10
<i>Manager.Federate.Service.EnableTimeRegulation</i>	11-10
<i>Manager.Federate.Service.FederateRestoreComplete</i>	11-10
<i>Manager.Federate.Service.FederateSaveBegun</i>	11-11
<i>Manager.Federate.Service.FederateSaveComplete</i>	11-11
<i>Manager.Federate.Service.FlushQueueRequest</i>	11-11
<i>Manager.Federate.Service.LocalDeleteObjectInstance</i>	11-11
<i>Manager.Federate.Service.ModifyLookahead</i>	11-11
<i>Manager.Federate.Service.NextEventRequest</i>	11-11
<i>Manager.Federate.Service.NextEventRequestAvailable</i>	11-11
<i>Manager.Federate.Service.PublishInteractionClass</i>	11-12
<i>Manager.Federate.Service.PublishObjectClass</i>	11-12
<i>Manager.Federate.Service.ResignFederationExecution</i>	11-12
<i>Manager.Federate.Service.SubscribeInteractionClass</i>	11-12
<i>Manager.Federate.Service.SubscribeObjectClassAttributes</i>	11-12
<i>Manager.Federate.Service.SynchronizationPointAchieved</i>	11-12
<i>Manager.Federate.Service.TimeAdvanceRequest</i>	11-13
<i>Manager.Federate.Service.TimeAdvanceRequestAvailable</i>	11-13
<i>Manager.Federate.Service.UnconditionalAttributeOwnershipDivestiture</i>	11-13
<i>Manager.Federate.Service.UnpublishInteractionClass</i>	11-13
<i>Manager.Federate.Service.UnpublishObjectClass</i>	11-13
<i>Manager.Federate.Service.UnsubscribeInteractionClass</i>	11-13
<i>Manager.Federate.Service.UnsubscribeObjectClass</i>	11-13
11.3 OBJECTS	11-13
<i>Manager</i>	11-13
<i>Manager.Federate</i>	11-14
<i>Manager.Federation</i>	11-15
12. MIGRATION DOCUMENT	12-1
12.1 INTRODUCTION TO MIGRATING RTI 1.3v6 FEDERATES TO RTI 1.3-NG	12-1
12.2 MANAGEMENT OBJECT MODEL	12-1
12.2.1. <i>General notes</i>	12-1
12.2.2. <i>Manager.Federate.Adjust.ModifyAttributeState</i>	12-1
12.2.3. <i>Manager.Federate.Adjust.SetExceptionLogging</i>	12-1
12.2.4. <i>Manager.Federate.Report.Alert</i>	12-1
12.2.5. <i>Manager.Federate.Report.ReportObjectSubscription</i>	12-1
12.2.6. <i>Manager.Federate.Report.ReportServiceInvocation</i>	12-1
12.2.7. <i>Manager.Federate object</i>	12-2
12.3 RTI INITIALIZATION DATA (EXTRACTED FROM THE RTI.RID FILE)	12-2
12.3.1. <i>Introduction to the RTI 1.3-NG RTI Initialization Date file</i>	12-2
12.3.2. <i>File Location</i>	12-3

12.3.3.	<i>File Format</i>	12-3
12.3.4.	<i>File Parameter Scoping</i>	12-3
12.3.5.	<i>Parameter Definition</i>	12-4
12.4	NOTES ON PORTING FOODFIGHT FROM RTI1.3V6 TO RTI1.3-NG	12-24
12.4.1.	<i>Migrating FoodFight for the Hands-On Practicum to RTI 1.3-NG</i>	12-24
12.4.2.	<i>Differences we noted while running the new FoodFight execution</i>	12-25
12.4.3.	<i>Getting help</i>	12-26

TABLE OF FIGURES

FIGURE 1-1. DoD M&S MASTER PLAN	1-1
FIGURE 1-2. COMMON TECHNICAL FRAMEWORK	1-2
FIGURE 1-3. HIGH LEVEL ARCHITECTURE MANDATE.....	1-2
FIGURE 1-4. HLA COMPONENT SUMMARY	1-3
FIGURE 1-5. OBJECT MODEL TEMPLATE.....	1-4
FIGURE 1-6. OBJECT MODEL SUMMARY	1-5
FIGURE 1-7. CONCEPTUAL MODEL OF THE MISSION SPACE.....	1-5
FIGURE 1-8. THE CMMS PROCESS.....	1-6
FIGURE 1-9. DATA STANDARDIZATION PRODUCTS.....	1-7
FIGURE 2-1. RTI OVERVIEW.....	2-1
FIGURE 2-2. RTI COMPONENTS AT-A-GLANCE	2-2
FIGURE 2-3. RTI COMPONENTS	2-2
FIGURE 2-4. RTI AND FEDERATE CODE RESPONSIBILITIES.....	2-3
FIGURE 2-5. FEDERATE – FEDERATION INTERPLAY	2-4
FIGURE 2-6. FEDEXEC LIFE CYCLE	2-5
FIGURE 2-7. MANAGEMENT AREAS PARTITIONED.....	2-5
FIGURE 2-8. FEDERATION MANAGEMENT.....	2-6
FIGURE 2-9. DECLARATION MANAGEMENT.....	2-6
FIGURE 2-10. OBJECT MANAGEMENT.....	2-7
FIGURE 2-11. OWNERSHIP MANAGEMENT.....	2-8
FIGURE 2-12. TIME MANAGEMENT.....	2-8
FIGURE 2-13. DATA DISTRIBUTION MANAGEMENT	2-9
FIGURE 3-2. SIX-AXIS DIAGRAM – LATE ARRIVAL.....	3-5
FIGURE 3-3. LBTS FOR CONSTRAINED FEDERATES.....	3-7
FIGURE 3-4. LATE-ARRIVING FEDERATE	3-8
FIGURE 3-5. PER FEDERATE QUEUES.....	3-9
FIGURE 4-1. THE FEDERATION DEVELOPMENT AND EXECUTION PROCESS (FEDEP) MODEL	4-1
FIGURE 5-1. FEDERATION MANAGEMENT LIFE CYCLE.....	5-1
FIGURE 5-3. FEDERATION MANAGEMENT SAVE	5-7
FIGURE 5-4. FEDERATION MANAGEMENT RESTORE	5-7
FIGURE 6-1. TOGGLING "REGULATING" AND "CONSTRAINED" STATUS	6-1
FIGURE 6-2. LOGICAL TIME ADVANCEMENT FOR A TIME-STEP FEDERATE	6-3
FIGURE 6-3. LOGICAL TIME ADVANCEMENT FOR AN EVENT-BASED FEDERATE	6-4
FIGURE 6-4. LOGICAL TIME ADVANCEMENT FOR AN OPTIMISTIC FEDERATE	6-5
FIGURE 7-1. CONTROL SIGNAL SCHEMA.....	7-1
FIGURE 7-2. CLASS HIERARCHY – VENN DIAGRAM	7-3
FIGURE 7-3. OBJECT PUBLISHING	7-4
FIGURE 7-4. OBJECT PUBLICATION AND SUBSCRIPTION	7-8
FIGURE 7-5. DECLARING INTERACTIONS	7-12
FIGURE 8-1. OBJECT MANAGEMENT METHODOLOGY.....	8-1
FIGURE 8-2. OBJECT MANAGEMENT UPDATES	8-3
FIGURE 8-3. EXCHANGING INTERACTIONS.....	8-7
FIGURE 8-4. ADDITIONAL OBJECT CONTROL.....	8-8
FIGURE 8-5. SCOPE INTERACTIONS	8-10
FIGURE 9-1. SHARED UPDATE RESPONSIBILITY.....	9-2
FIGURE 9-2. OWNERSHIP PULL INTERACTION DIAGRAM – ORPHANED ATTRIBUTE.....	9-3
FIGURE 9-3. OWNERSHIP PULL INTERACTION DIAGRAM – INTRUSIVE	9-4
FIGURE 9-4. OWNERSHIP PUSH INTERACTION DIAGRAM	9-5
FIGURE 10-1. PUBLICATION AND SUBSCRIPTION INTERSECTIONS.....	10-1
FIGURE 10-2. EXAMPLE ROUTING SPACE	10-2
FIGURE 10-3. NORMALIZATION OF A RANGE IN AN EXTENT	10-4
FIGURE 10-5. TWO-LAYER FILTERING.....	10-7
FIGURE 10-6. REGION METHODS	10-8

FIGURE 10-7. DDM ATTRIBUTES (PART 1 OF 3).....	10-10
FIGURE 10-8. DDM ATTRIBUTES (PART 2 OF 3).....	10-10
FIGURE 10-9. DDM ATTRIBUTES (PART 3 OF 3).....	10-11
FIGURE 10-10. INTERACTIONS AND DDM	10-12

1. Introduction to HLA

The DoD Modeling and Simulation Master Plan identifies six objectives for Modeling and Simulation (M&S), as shown in Figure 1-1. Objective 1 of the plan, development of a common technical framework for M&S, will be discussed in this chapter.

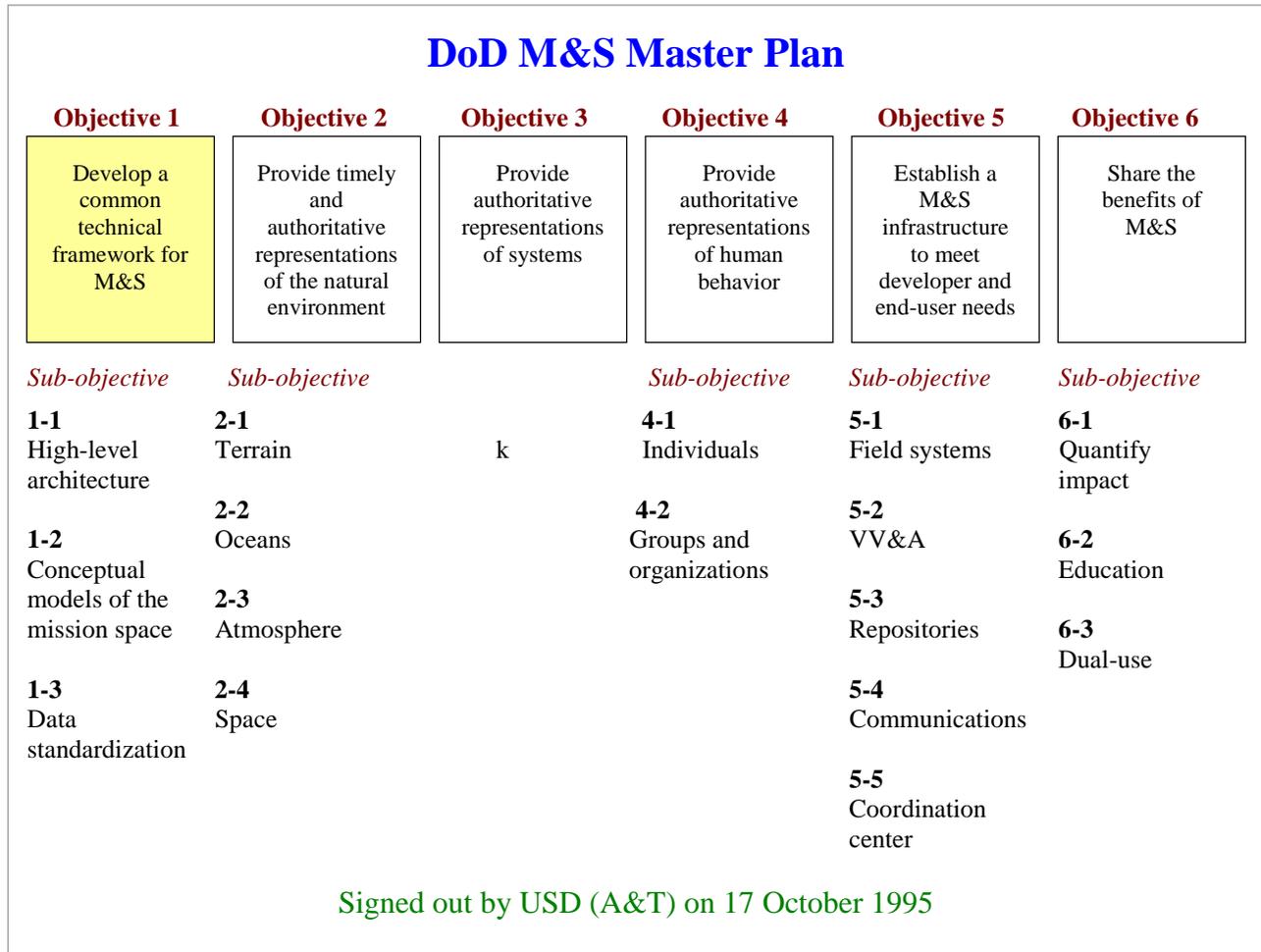


Figure 1-1. DoD M&S Master Plan

Objective 1 of the Modeling and Simulation Master Plan has three sub-objectives (Figure 1-1, DoD M&S Master Plan). These are (1) High-Level Architecture (HLA), (2) Conceptual Models of the Mission Space (CMMS), and (3) Data Standardization (DS). Figure 1-2 outlines the common technical framework components and candidate applications. Each component is described in the sections that follow.

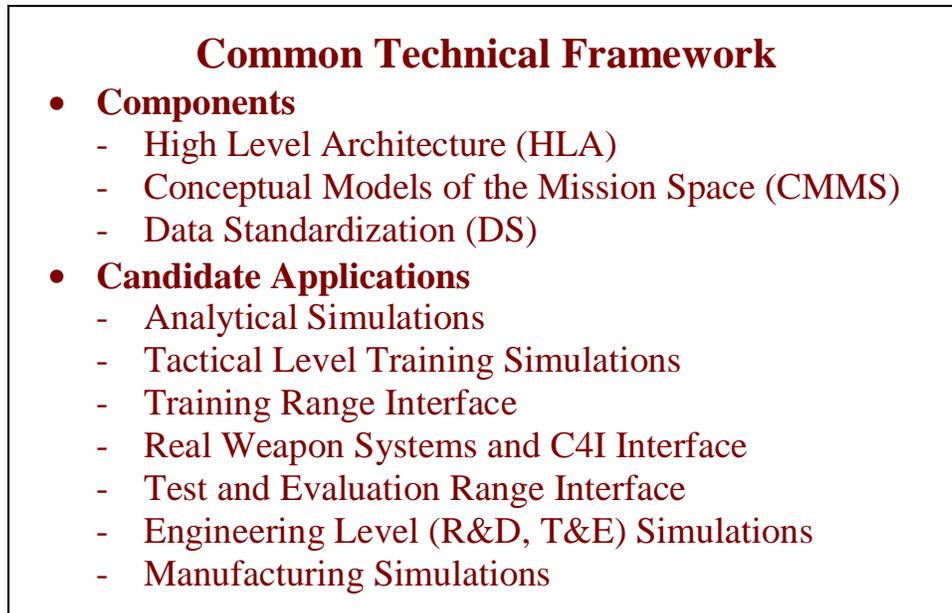


Figure 1-2. Common Technical Framework

The High-Level Architecture (HLA) mandate, shown in Figure 1-3, establishes a common high-level simulation architecture to facilitate the interoperability of all types of models and simulations among themselves and with C4I systems. The HLA is designed to promote standardization in the M&S community and to facilitate the reuse of M&S components.

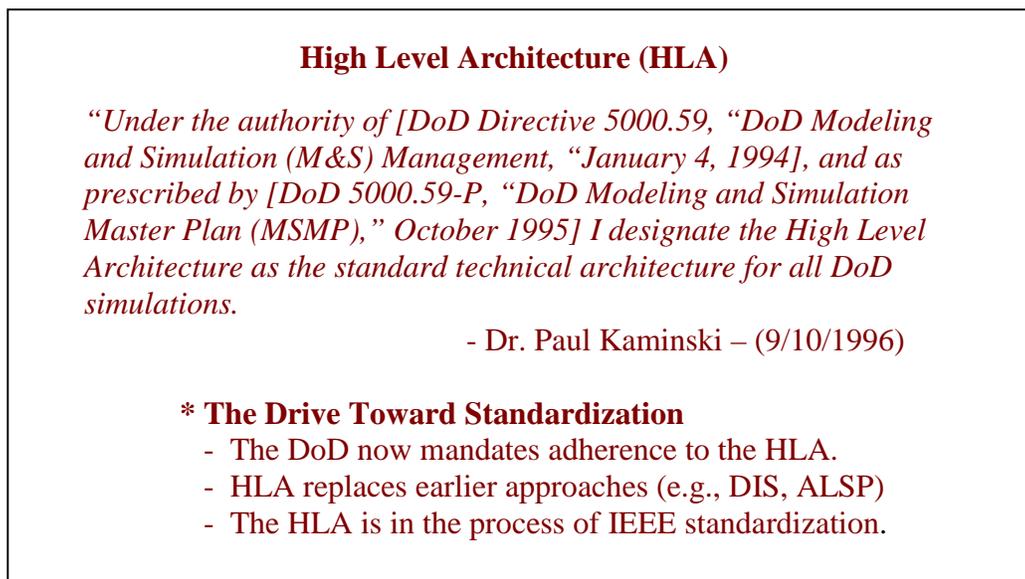


Figure 1-3. High Level Architecture Mandate

The HLA is defined by three components: (1) Federation Rules, (2) the HLA Interface Specification, and (3) the Object Model Template (OMT). Figure 1-4 summarizes the attributes of the HLA components.

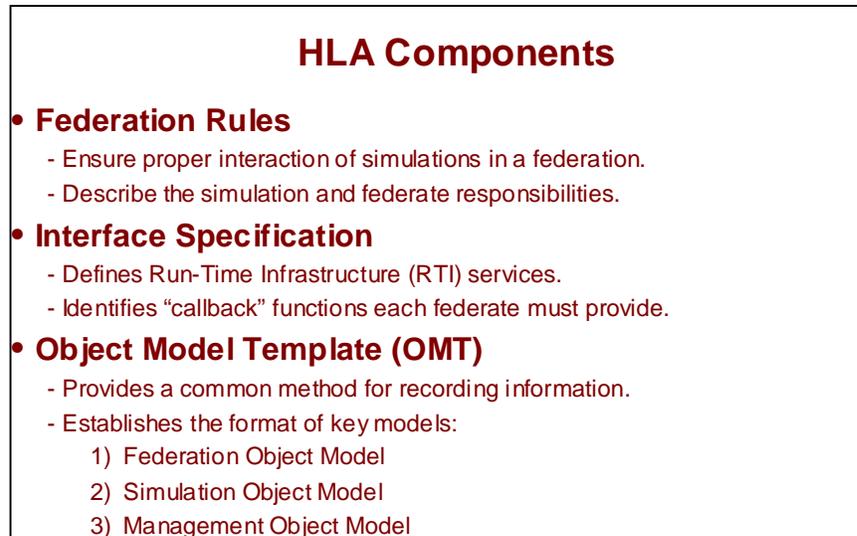


Figure 1-4. HLA Component Summary

The Run-Time Infrastructure (RTI) software implements the interface specification and represents one of the most tangible products of the HLA. It provides services in a manner that is comparable to the way a distributed operating system provides services to applications.

Within the HLA, *federations* are comprised of *federates* that exchange information in the form of *objects* and *interactions* – concepts that will be explained further in this guide.

1.1 Federation Rules

The Federation Rules describe the responsibilities of federates and their relationships with the RTI. There are ten rules. Five relate to the federation and five to the federate.

Federation Rules:

1. Federations shall have an HLA Federation Object Model (FOM), documented in accordance with the HLA Object Model Template (OMT).
2. In a federation, all representation of objects in the FOM shall be in the federates, not in the run-time infrastructure (RTI).
3. During a federation execution, all exchange of FOM data among federates shall occur via the RTI.
4. During a federation execution, federates shall interact with the run-time infrastructure (RTI) in accordance with the HLA interface specification.

HLA RTI 1.3-Next Generation

5. During a federation execution, an attribute of an instance of an object shall be owned by only one federate at any given time.

Federate Rules:

6. Federates shall have an HLA Simulation Object Model (SOM), documented in accordance with the HLA Object Model Template (OMT).
7. Federates shall be able to update and/or reflect any attributes of objects in their SOM and send and/or receive SOM object interactions externally, as specified in their SOM.
8. Federates shall be able to transfer and/or accept ownership of an attribute dynamically during a federation execution, as specified in their SOM.
9. Federates shall be able to vary the conditions (e.g., thresholds) under which they provide updates of attributes of objects, as specified in their SOM.
10. 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.

1.2 Interface Specification

The interface specification identifies how federates will interact with the federation and, ultimately, with one another. The specification is divided into six management areas, which are explored at length in subsequent chapters.

1.3 Object Model Template (OMT)

All objects and interactions managed by a federate, and visible outside the federate, are described according to the standard OMT. (See Figure 1-5.) The OMT provides a common method for representing HLA Object Model information.

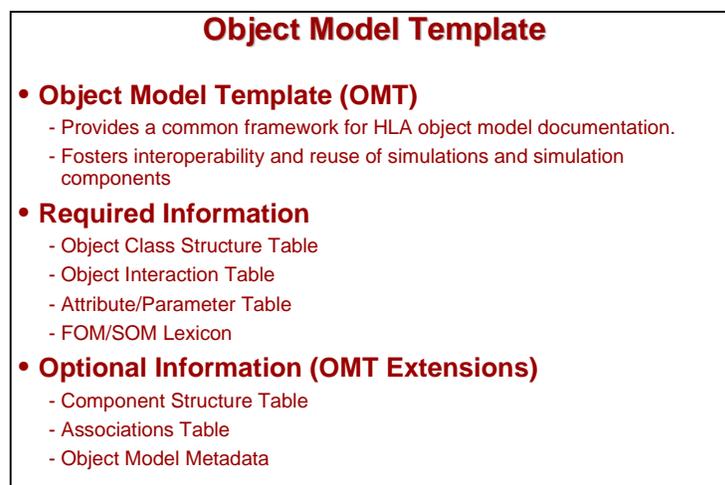


Figure 1-5. Object Model Template

The Federation Object Model (FOM), Simulation Object Model (SOM) and Management Object Model (MOM) are all defined using the OMT. Figure 1-6 summarizes these models.

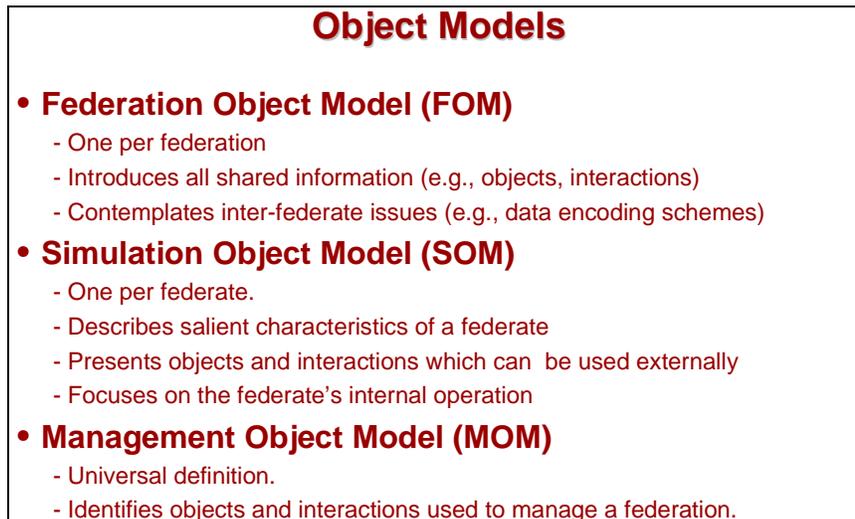


Figure 1-6. Object Model Summary

The HLA separates data and architecture. It prescribes that OMT objects and interactions defined according to the OMT can be constructed and exchanged with no adjustments to HLA-derived software.

1.4 Conceptual Model of the Mission Space (CMMS)

A Conceptual Model of the Mission Space (CMMS) is a first abstraction of the real world, which serves as a common framework for knowledge acquisition with validated, relevant actions and interactions organized by specific task and entity/organization. It is a simulation-independent hierarchical description of actions and interactions among the various entities associated with a particular mission area. See Figure 1-7.

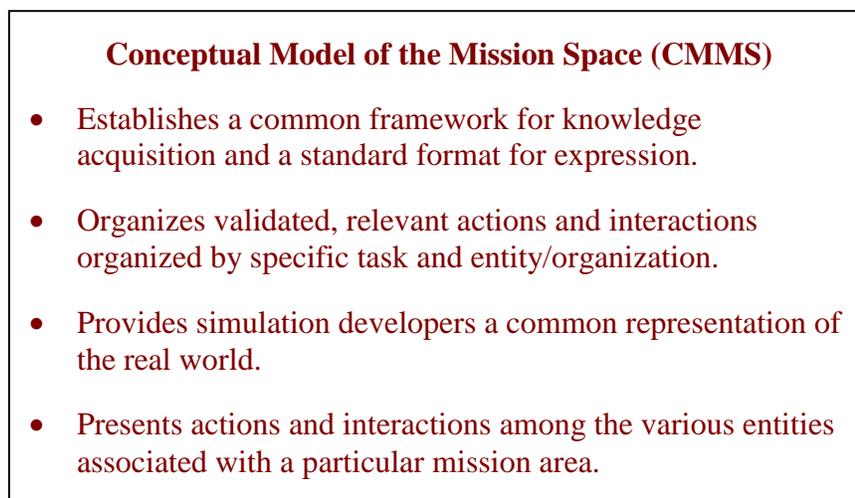


Figure 1-7. Conceptual Model of the Mission Space

Thus, conceptual models of the mission space provide simulation developers with a common baseline for constructing consistent and authoritative M&S representations. The primary purpose of CMMS is to facilitate interoperability and reuse of simulation components, particularly among DoD simulation developments, by sharing common, authoritative information between DoD simulations. The CMMS will provide a meta-model of fundamental knowledge about military operations. The CMMS System will capture and store this knowledge, and make it easily accessible to simulation developers and users. Figure 1-8 depicts the CMMS process.

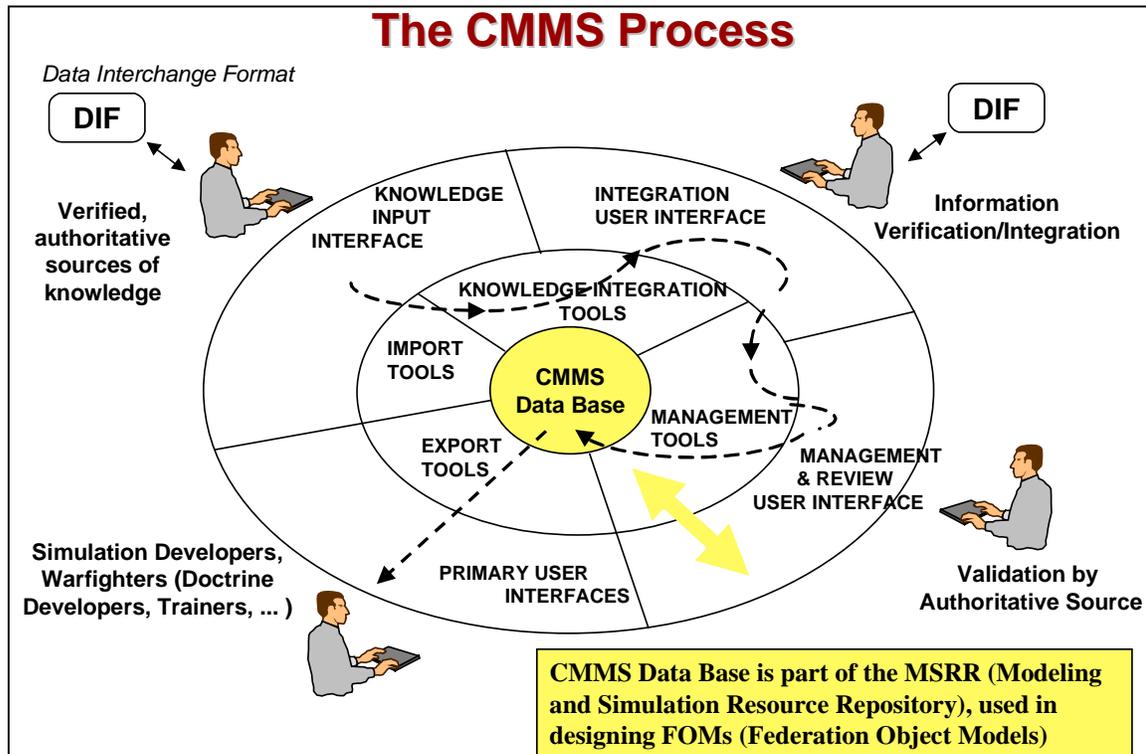


Figure 1-8. The CMMS Process

The mission space structure, tools, and resources will provide both an overarching framework and access to the necessary data and details to permit development of consistent, interoperable, and authoritative representations of the environment, systems, and human behavior in DoD simulation system.

1.5 Data Standardization (DS)

The data standardization program seeks to facilitate reuse, interoperability, and data sharing among models, simulations, and C4I systems by establishing policies, procedures, and methodologies for data requirements, standards, sources, security, and verification, validation, and certification.

The primary products of the data standardization program are: (1) Common Semantics and Syntax (CSS), which define common lexicons, dictionaries, taxonomies, and tools for data

elements, and (2) Data Interchange Formats (DIF), the physical structures (BNF, SQL) used by programmers to actually interchange data.

Other supporting data standardization products are: (1) Authoritative Data Sources (ADS), the primary means for identifying data for reuse, (2) Data Quality (DQ) practices, a body of VV&A/C guidelines, and (3) Data Security (DS) practices, the policies pertaining to data protection and release. See Figure 1-9.

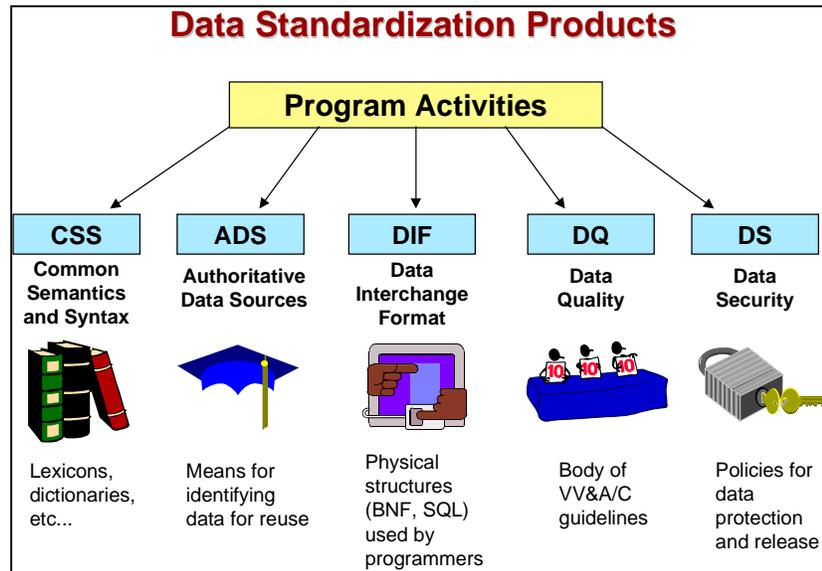


Figure 1-9. Data Standardization Products

1.6 Further Reading

Additional information may be obtained from the HLA Technical Library. Figure 1-10 provides the DMSO home page location and e-mail address for connections via the Internet.



Figure 1-10. HLA Technical Library

2. RTI Synopsis

This chapter introduces general characteristics of RTI 1.3-NG. It identifies major RTI components, examines the interplay between federates and the federation, and postulates some ground rules for using RTI software. Figure 2-1 summarizes the RTI definition described in the rest of this chapter.

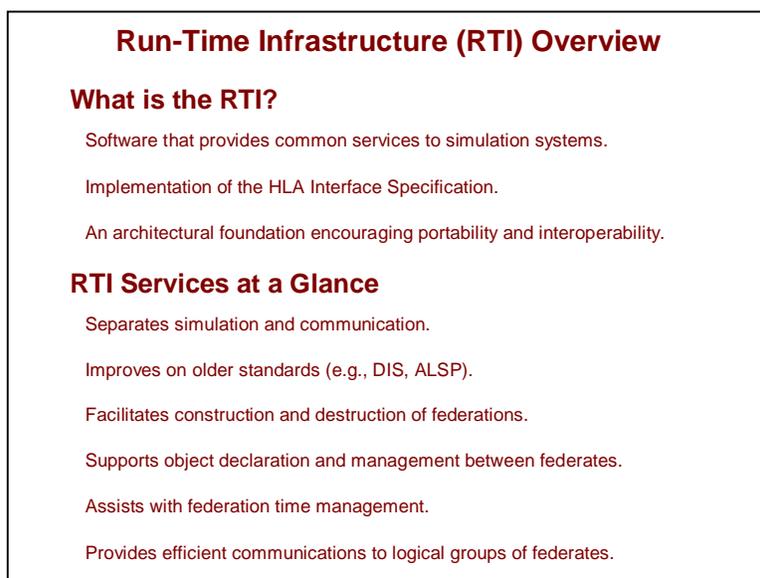


Figure 2-1. RTI Overview

RTI 1.3-NG implements Version 1.3 (Draft 10, 2 April 1998) of the *HLA Interface Specification*. The RTI 1.3-NG software having been rewritten “from the ground up”, may vary slightly from its predecessors. However, every effort has been made to ensure that RTI 1.3-NG maintains a “compile time” compatibility with the previous RTI 1.3v6 release.

RTI software is currently comprised of the RTI Executive process (RtiExec), the Federation Executive process (FedExec) and the libRTI library. As illustrated in Figure 2-2, each executable containing federates incorporates libRTI. Federates may exist as independent processes or be grouped into one or more processes. A federate may simultaneously participate in more than one federation.

2.1 Major Components

RTI software can be executed on a standalone workstation or executed over an arbitrarily complex network. The RtiExec process manages the creation and destruction of federation executions. Each executing federation is characterized by a single, global FedExec.

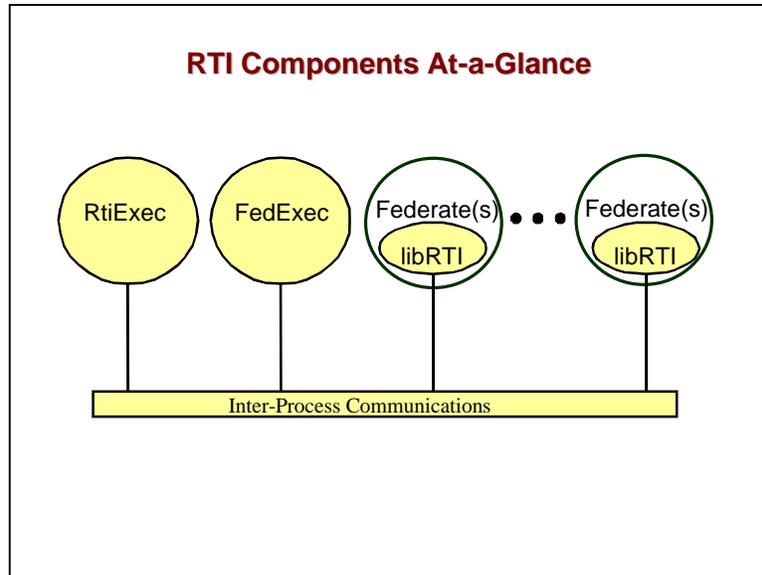


Figure 2-2. RTI Components At-a-Glance

The FedExec manages federates joining and resigning the federation. The libRTI library extends RTI services to federate developers. Services are accomplished through encapsulated communications between libRTI, RtiExec, and the appropriate FedExec. Figure 2-3 summarizes the activities supported by the components of the RTI.

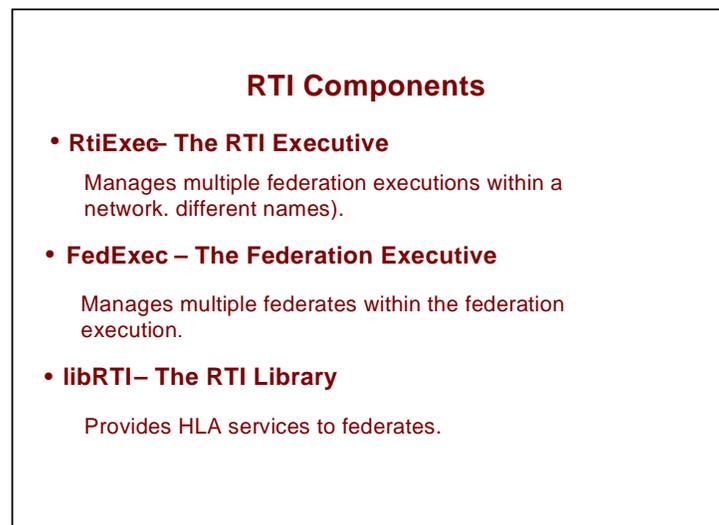


Figure 2-3. RTI Components

2.2 RtiExec

The RtiExec is a globally known process. Each application communicates with RtiExec to initialize RTI components. The RtiExec's primary purpose is to manage the creation and

destruction of FedExecs. An RtiExec directs joining federates to the appropriate federation execution. RtiExec ensures that each FedExec has a unique name.

2.3 FedExec

Each FedExec manages a federation. It allows federates to join and to resign, and facilitates data exchange between participating federates. A FedExec process is created by the first federate to successfully invoke the “createFederationExecution” service for a given federation execution name. Each federate joining the federation is assigned a federation wide unique handle.

2.4 libRTI

The C++ library, libRTI, provides the RTI services specified in the *HLA Interface Specification* to federate developers. The class diagrams in Figure 2-4 illustrates RTI and federates code responsibilities. Federates use libRTI (which communicates with the RtiExec, a FedExec, and other federates) to invoke HLA services.

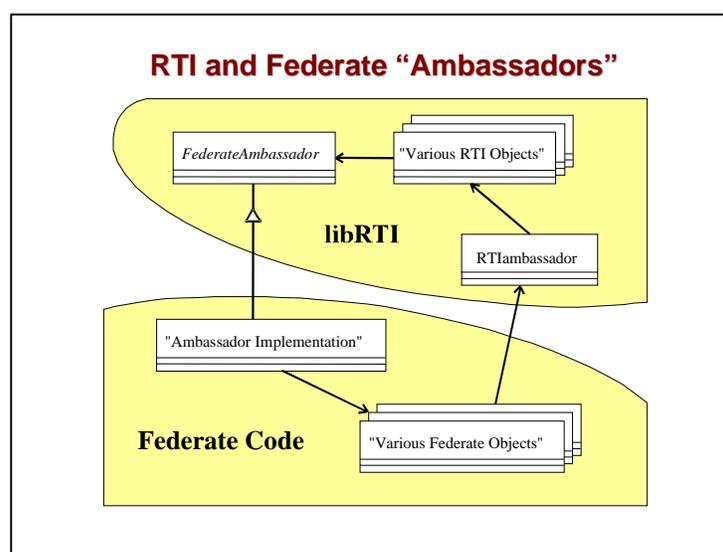


Figure 2-4. RTI and Federate Code Responsibilities

The *HLA Interface Specification* identifies the services provided by libRTI to each federates and the obligation each federate bears to the federation. Within libRTI, the class *RTIambassador* bundles the services provided by the RTI.¹ All requests made by a federate on the RTI take the form of an *RTIambassador* method call. The abstract class *FederateAmbassador* identifies the callback functions each federate is obliged to provide.

While both *RTIambassador* and *FederateAmbassador* ambassador classes are a part of libRTI, it is very important to understand that *FederateAmbassador* is abstract. The federate must

¹ Most RTI classes (e.g., *RTIambassador*, *FederateAmbassador*) are declared within the class *RTI* for namespace protection. The prefix “RTI:” will be required to access these classes (e.g., *RTI::RTIambassador*).

implement the functionality declared in *FederateAmbassador*. An instance of this federate-supplied class is required to join a Federation.

The federation (via libRTI) responds asynchronously to many federate requests. *FederateAmbassador* “callback” functions provide a mechanism for the federation to communicate back to the federate.

The header file “RTI.hh” that accompanies libRTI includes declarations for class *RTIambassador*, the abstract class *FederateAmbassador*, and a variety of supporting declarations and definitions. The *RTIambassador* is implemented in libRTI and must be incorporated into each federate executable. The RTI and Federate ambassadors are examined in detail in subsequent chapters.

2.5 Management Areas

Figure 2-5 presents a high level illustration of the interplay between a federate and a federation.

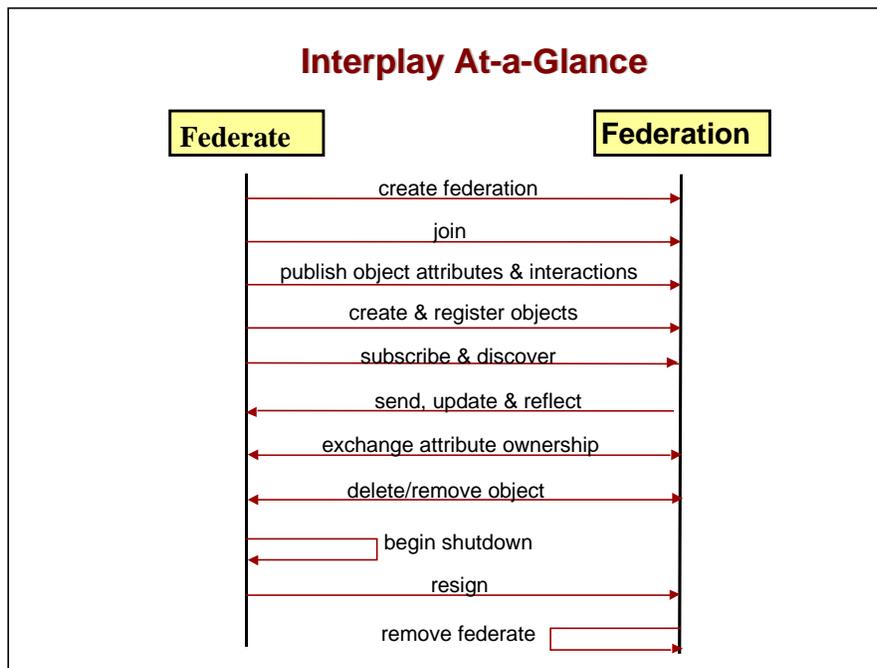


Figure 2-5. Federate – Federation Interplay

The *HLA Interface Specification* partitions the exchanges that take place between federate and federation into six management areas of the FedExec life cycle, as shown in Figure 2-6. The remaining figures offer a light overview of the management areas. Details will be explored in subsequent chapters. Figure 2-7 summarizes the objectives of each of the management areas.

Each of the management areas is described in one of the chapters that follow. Figures 2-8 through 2-13 present summary graphics for each management area to introduce the purpose and scope of each area and to provide a synopsis of the actions allocated to each management area. The applicable chapters that relate to each of the management areas are also provided in the following sections.

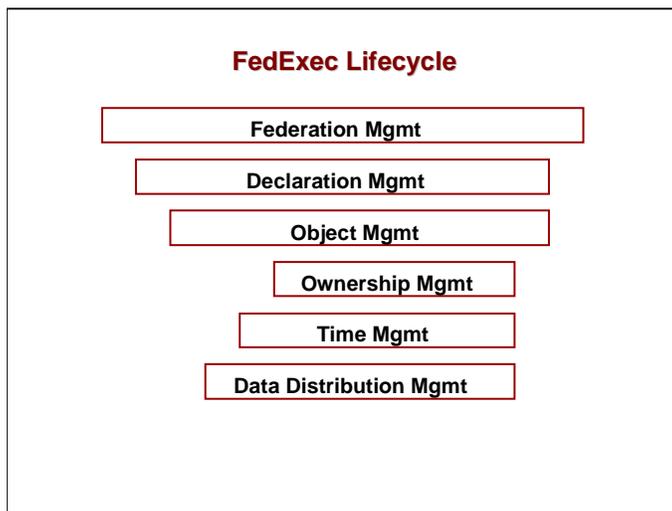


Figure 2-6. FedExec Life Cycle

Management Areas Partitioned	
<u>Management Area</u>	<u>Activities Described</u>
Federation Mgmt	Control an exercise
Declaration Mgmt	Define data publication and subscription
Object Mgmt	Exchange object and interaction data
Ownership Mgmt	Transfer attribute ownership
Time Mgmt	Control message ordering
Data Distribution Mgmt	Efficiently route data between producers and consumers
RTI Support Services	Assist service operations

Figure 2-7. Management Areas Partitioned

2.5.1. Federation Management

Federation management includes such tasks as creating federations, joining federates to federations, observing federation-wide synchronization points, effecting federation-wide saves and restores, resigning federates from federations, and destroying federations. Figure 2-8 summarizes the Federation Management profile. Chapter 5, *Federation Management*, describes these features.



Figure 2-8. Federation Management

2.5.2. Declaration Management

Declaration management includes publication, subscription, and supporting control functions. Federates that produce object class attributes or interactions must declare exactly what they are able to publish (i.e., generate). Figure 2-9 shows the main coordination tasks and synthesizes the actions accomplished by declaration management. Chapter 7, *Declaration Management*, discusses these tasks in detail.

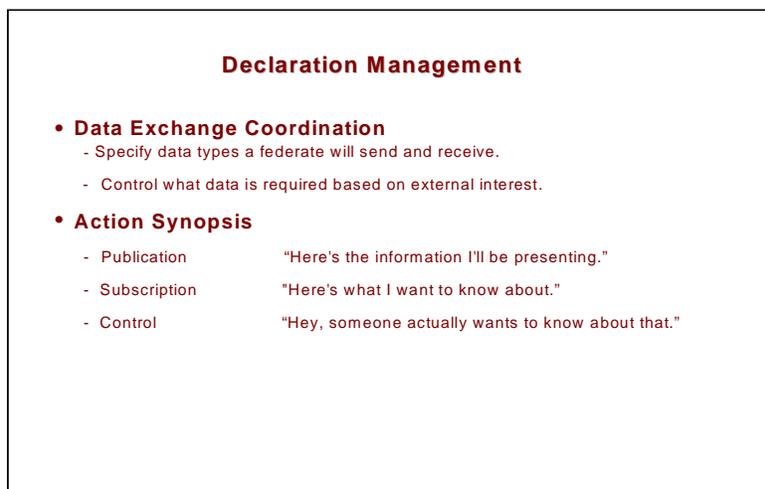


Figure 2-9. Declaration Management

2.5.3. Object Management

Object management includes instance *registration* and instance *updates* on the object producer side and instance *discovery* and *reflection* on the object consumer side. Object management also includes methods associated with sending and receiving interactions, controlling instance updates based on consumer demand, and other miscellaneous support functions. Figure 2-10 presents the object discovery principles and a synopsis of the actions effected by object management. These actions are discussed in detail in Chapter 8, *Object Management*.

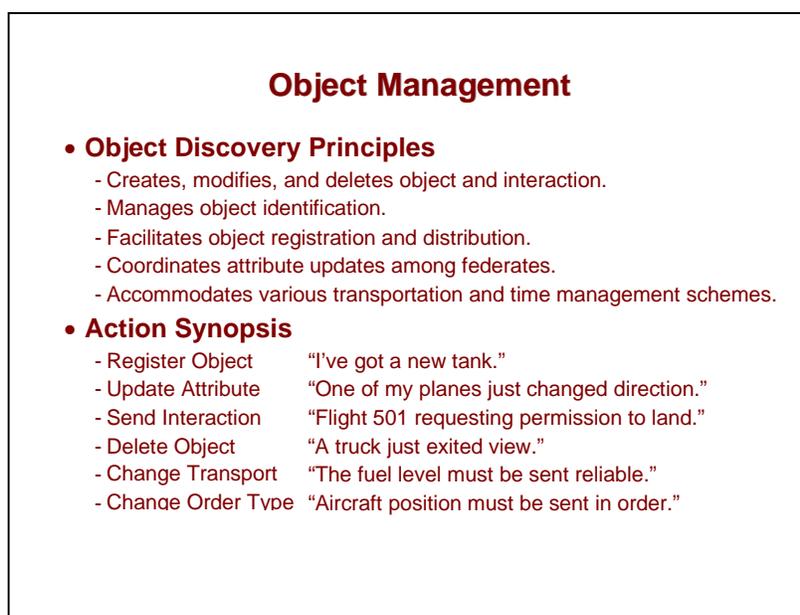


Figure 2-10. Object Management

2.5.4. Ownership Management

The RTI allows federates to distribute the responsibility for updating and deleting object instances with a few restrictions. It is possible for an object instance to be wholly owned by a single federate. In such cases, the owning federate has responsibility for updating all attributes associated with the object and for deleting the object instance. It is possible for two or more federates to share *update responsibility* for a single object instance. When update responsibility for an object instance is shared, each of the participating federates has responsibility for a mutually exclusive set of object attributes.² Only one federate can have update responsibility for an individual attribute of an object instance at any given time. In addition, only one federate has the *privilege to delete* an object instance at any given time. These object management tasks are summarized in Figure 2-11, and discussed in detail in Chapter 9, *Ownership Management*.

² For a given object instance, some attributes may be unowned – i.e., no federate has update responsibility.

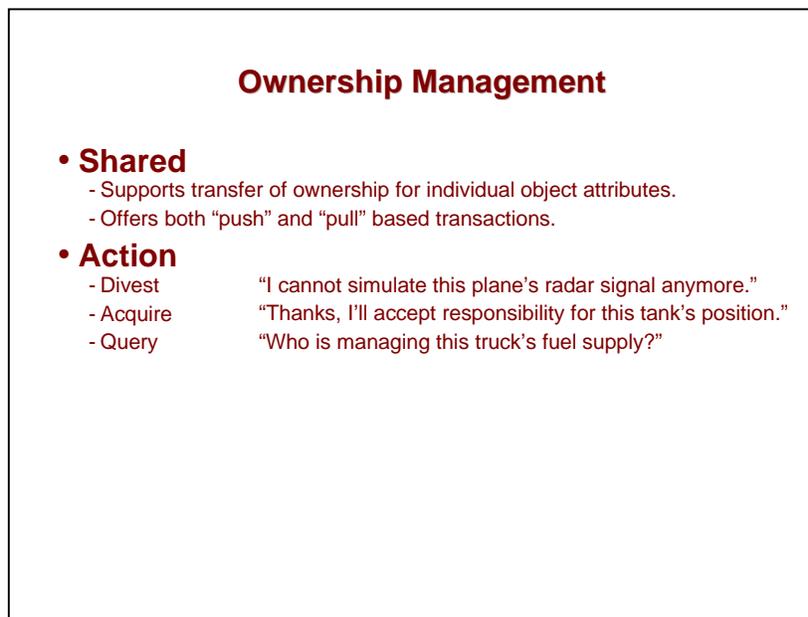


Figure 2-11. Ownership Management

2.5.5. Time Management

The focus of time management is on the mechanics required to implement time management policies and negotiate time advances. Chapter 6, *Time Management*, discusses these tasks in detail. Figure 2-12 displays a synopsis of the time management actions.

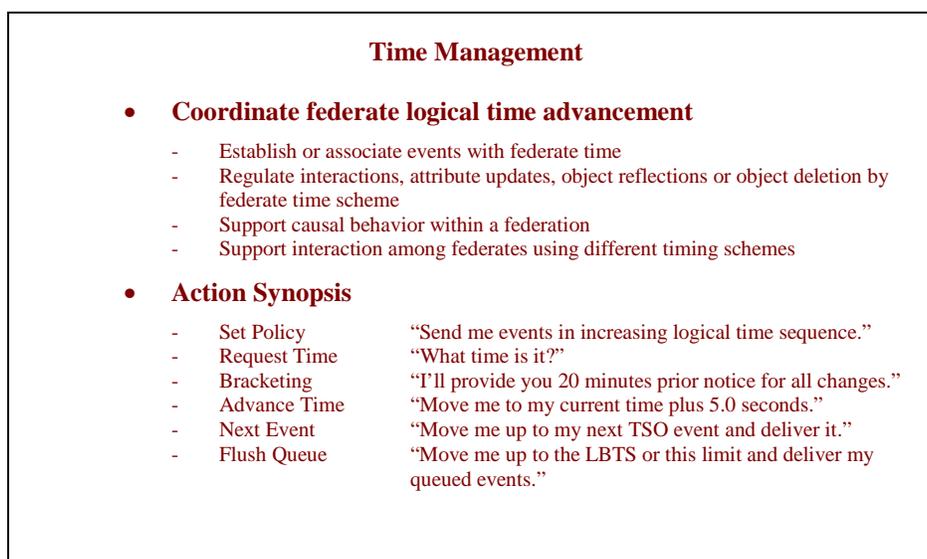


Figure 2-12. Time Management

2.5.6. Data Distribution Management

Data distribution management (DDM) provides a flexible and extensive mechanism for further isolating publication and subscription interests – effectively extending the sophistication of the RTI's routing capabilities. Figure 2-13 presents a synopsis of the DDM actions.

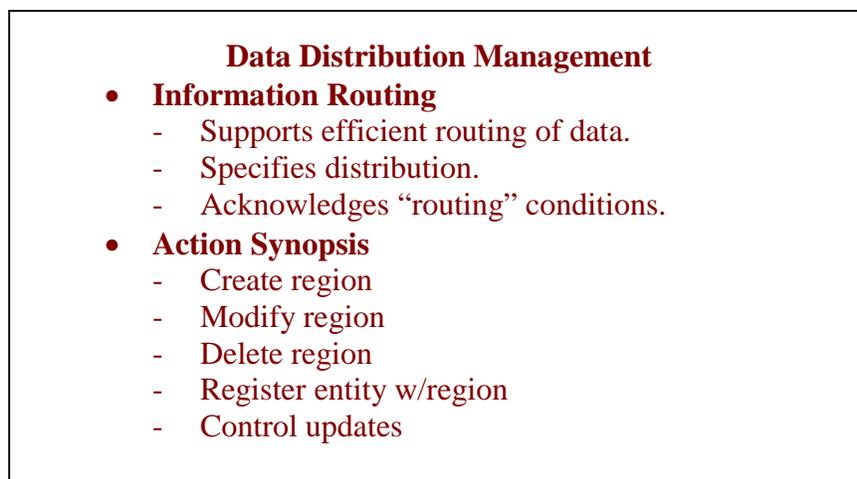


Figure 2-13. Data Distribution Management

3. The Role of Time

3.1 Introduction

This chapter introduces time management from a philosophical perspective and emphasizes RTI terminology.³ The RTI software supports a variety of time management policies. Time management services are optional. However, it is important to understand the time management models available in the RTI and the implication of exchanging events between federates with different time management policies. Chapter 6, *Time Management*, introduces specific RTI methods for setting time management policy and negotiating time advances.

3.2 Time Management Basics

The HLA accommodates a variety of time management policies. The RTI provides an optional time management service to coordinate the exchange of events between federates. Events can be associated with a point in time and the RTI can assist in ensuring causal behavior. It is also possible for one or more federates in a federation to fully ignore time. By default, the RTI does not attempt to coordinate time between federates. In addition, the HLA not only supports a variety of time management policies, but also facilitates interoperability between federates with different policies. Even if the optional time management services are ignored, it pays to understand available time management schemes.

In a federation, time always moves forward. However, the perception of the *current time* may differ among participating federates. Time management is concerned with the mechanisms for controlling the advancement of each federate along the federation time axis. In general, time advances must be coordinated with object management services so that information is delivered to each federate in a causally correct and ordered fashion.

In some situations, it is appropriate to constrain the progress of one federate based on the progress of another. In fact, any federate may be designated a *regulating* federate. Regulating federates regulate the progress in time of federates that are designated as *constrained*. In general, a federate may be "regulating," "constrained," "regulating and constrained," or "neither regulating nor constrained." By default, federates are neither regulating nor constrained. The RTI recognizes every federate as adopting one of these four approaches to time management. A federation may be comprised of federates with any combination of time management models. That is, a federation may consist of several federates that are regulating, several federates that are constrained, several federates that are regulating and constrained, or several federates that are not using the RTI time management services.

A federate that becomes "time regulating" may associate some of its activities (e.g., updating instance attribute values and sending interactions) with points on the federation time axis. Such events are said to have a "time-stamp." A federate that is interested in discovering events in a federation-wide, time-stamp order is said to be "time constrained." The time management

³ Portions of this chapter are lifted directly or paraphrased from the HLA 1.3 Interface Specification.

services coordinate event exchange among time regulating and time-constrained federates.⁴ Such coordination levies certain rules on participants.

Again, federates are neither time regulating nor time constrained by default. The activities of these federates are not coordinated (in time) with other federates by the RTI. Such federates need not make use of any of the time management services. However, these federates may participate in a federation where time-stamped events are exchanged. It is important to understand how time-stamped events are perceived by federates that are not constrained. Conversely, it is important to understand how events generated by a non-regulating federate are perceived by a constrained federate.

3.3 "Regulating" and "Constrained"

Figure 3-1, known as the "two-axis diagram," introduces the definitions of "regulating," "lookahead," "TSO event," "constrained," and "lower bound time stamp (LBTS)." Subsequent diagrams examine complex combinations of federates with various time management policies and explore these definitions in some depth.

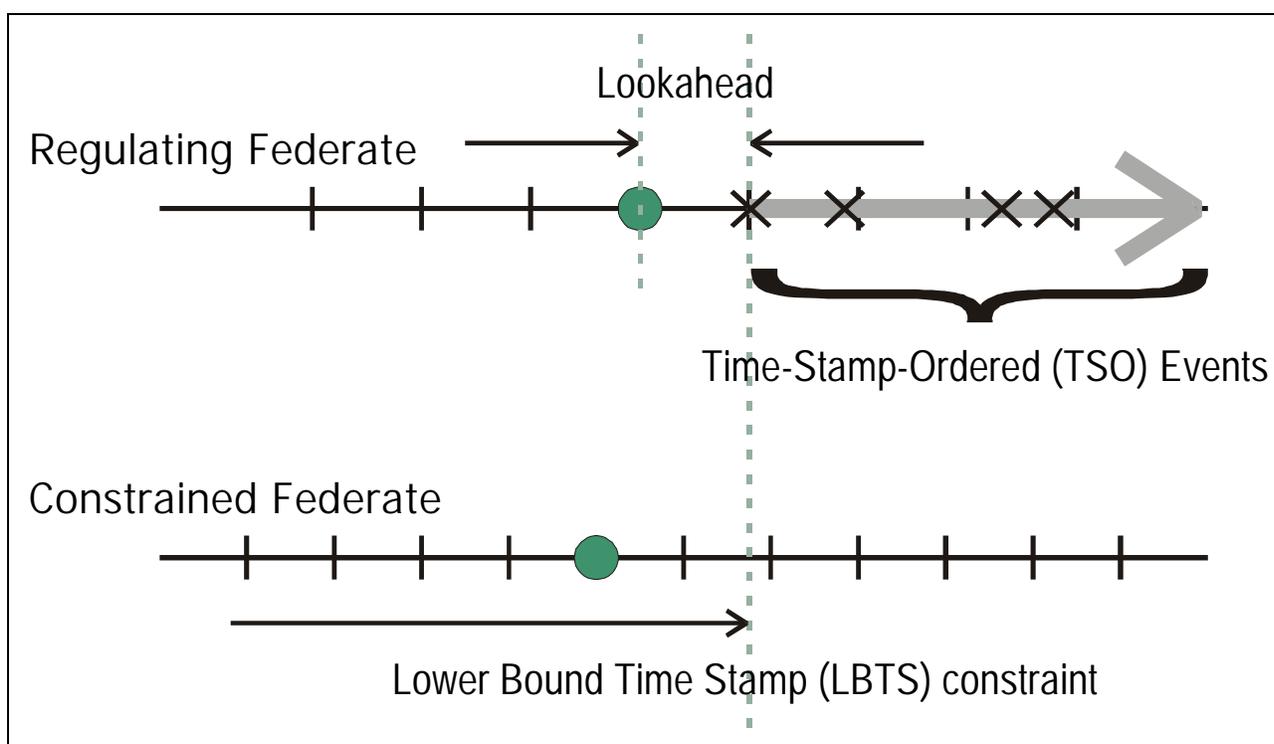


Figure 3-1. Two-Axis Diagram

⁴ Regulating federates generally produce time-stamped events, which their Local RTI Component (LRC) communicates to interested recipients. The LRC of each interested recipient orders all arriving time-stamped events by the time at which the events are said to occur.

3.3.1. Regulating

A federate that declares itself to be "regulating" is capable of generating time-stamp-ordered (TSO) events. TSO events are said to occur at a specific point in time. Federates that are not regulating can generate events, but there is no time associated with these events.⁵ A regulating federate coordinates time advances with the local RTI component (LRC). The regulating federate perceives the current time to be " t_{current} ." Federates can dynamically alter their status becoming regulating or non-regulating dynamically (i.e., "on-the-fly").

3.3.2. Lookahead

Each regulating federate establishes a "lookahead" value. The regulating federate promises that any TSO events it generates will occur equal to and no earlier than " $t_{\text{current}} + t_{\text{lookahead}}$." The lookahead value, $t_{\text{lookahead}}$, represents a contract between the regulating federate and the federation. It establishes the earliest possible TSO event the federate can generate relative to the current time, t_{current} .

Regulating federates must specify a lookahead value at the time they become regulating. Facilities exist to alter the lookahead value dynamically. It is possible to specify a lookahead value of zero. However, zero lookahead places extra constraints on a federate. When operating with a zero lookahead, the reference manual pages, for time management, should be read carefully to identify any special conditions and restrictions.

3.3.3. TSO Event

A TSO event is simply an event with an associated time-stamp. Only regulating federates can generate TSO events.⁶ A regulating federate can generate multiple TSO and/or non-TSO events, but all TSO events must occur at a time " $t_{\text{current}} + t_{\text{lookahead}}$ " or greater. Regulating federates need not generate TSO events in time-stamp order. That is, a regulating federate might generate an event at " $t_{\text{current}} + t_{\text{lookahead}} + 5$ " followed by another event at " $t_{\text{current}} + t_{\text{lookahead}} + 2$." It is the job of a constrained federate's LRC to order TSO events.

3.3.4. Constrained

A federate that declares itself to be "constrained" is capable of receiving TSO events. Federates that are not constrained still learn of TSO events, but absent the time-stamp information.⁷

3.3.5. Lower bound time stamp (LBTS)

Constrained federates have an associated LBTS.⁸ The LBTS specifies the time of the earliest possible time-stamp-ordered event the federate can receive. The LBTS is determined by looking

⁵ Such events are referred to as "Receive-Ordered" v. "Time-Stamp-Ordered" and will be discussed subsequently.

⁶ There are additional requirements on TSO events that are discussed subsequently.

⁷ Again, events with no time-stamp are termed "Receive-Ordered" and are discussed subsequently.

at the earliest possible message that might be generated by all other regulating federates. It changes as the regulating federates advance in time. A constrained federate cannot advance beyond its LBTS (i.e., this is the *constraint* from whence the name constrained), because the RTI can only guarantee there will be no more packets received prior to the LBTS.

3.4 Advancing Time

This section introduces a series of diagrams sometimes referred to as the six-axis diagrams. Each axis represents a federate in a federation. Each federate is using its own time management policy.

In Figure 3-2, five of six federates have joined an established federation. One of the federate's has not shown up yet – it is said to be *late arriving*. The small, solid circles represent the federation time as perceived by each federate. It is extremely important to understand that there is no universal "federation time" (at any given point each federate could have different "current times." Each federate is free to increment time independently. Some federates will apply the same time increment repeatedly. Other federates may jump through time based on the next available TSO event or some other criteria.

⁸ All federates, constrained or not, have an LBTS value. LBTS is really only meaningful to constrained federates or unconstrained federates planning to become constrained.

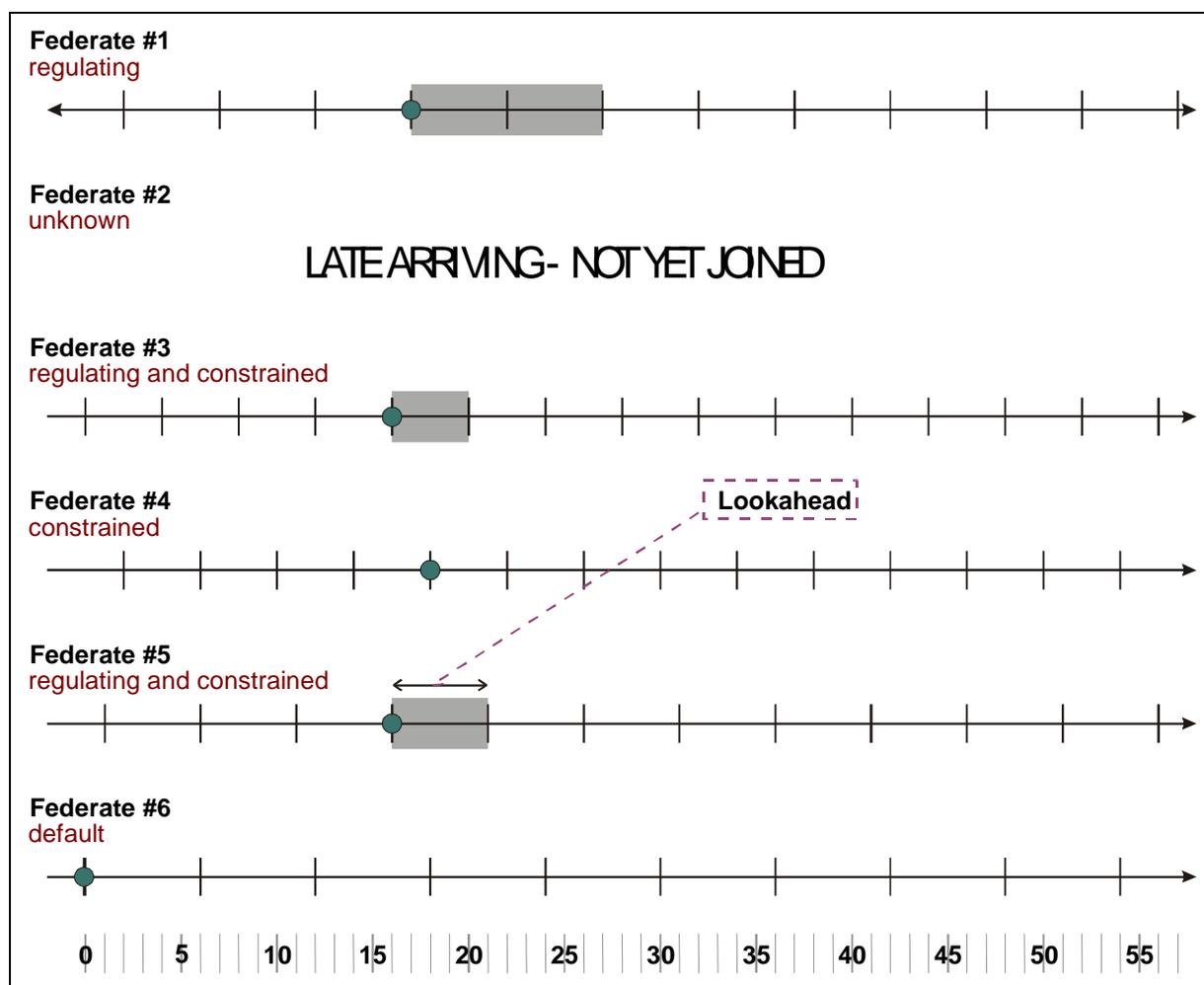


Figure 3-2. Six-axis Diagram – Late Arrival

The thick, shaded regions in the diagram represent the lookahead values specified by regulating federates. Federate #1's lookahead is twice its time step interval. Federate #3 and #5 have lookahead values that appear to be one time interval ahead. The lookahead values need not be related to a federate's time interval (as we will see when Federate #2 arrives).

Clearly, each federate in this federation has a unique perspective on the current time.

Federate 1	t = 17 seconds
Federate 2	not applicable
Federate 3	t = 16 seconds
Federate 4	t = 18 seconds
Federate 5	t = 16 seconds
Federate 6	t = 0 seconds

It's valuable to pose the question, "Is this combination of perceived times legitimate?" In general, unconstrained federates are free to progress through time. An unconstrained federate

has no requirement to request time advance grants through the RTI. For example, Federate #1 and Federate #6 can advance in time as fast as they want (or at least as fast as their simulation model can run). Should these unconstrained federates request permission to advance in time, their LRC realizes that they are unconstrained and grants permission to advance as a matter of course.

3.4.1. LBTS Constraint

Constrained federates cannot proceed beyond their current LBTS. The LBTS for a given federate is determined by calculating the earliest possible message a federate might receive from other regulating federates. Enforcing the LBTS constraint requires coordination between federate LRCs. As regulating federates advance, the LBTS of constrained federates increases. Figure 3-3 illustrates the LBTS for constrained federates.

The vertical dashed lines in Figure 3-3 represent the earliest possible TSO message that can be produced by each of the regulating federates – given their current time and their promised lookahead values. Below each constrained federate, a horizontal line is extended from “ $t = 0$ ” to the federate’s LBTS. In Figure 3-3, it is clear that the current time as perceived by each of the constrained federates is within their respective LBTS windows. Therefore, the “combination of perceived times” for each federate shown is legitimate!

Constrained federates are free to advance in time to their LBTS, but no further. In Figure 3-3, Federate #3 could increment to the next “tick mark” since the resulting time would be within its LBTS. However, Federate #4 and Federate #5 cannot proceed to their next “tick mark,” as each would have to move beyond its respective LBTS values.

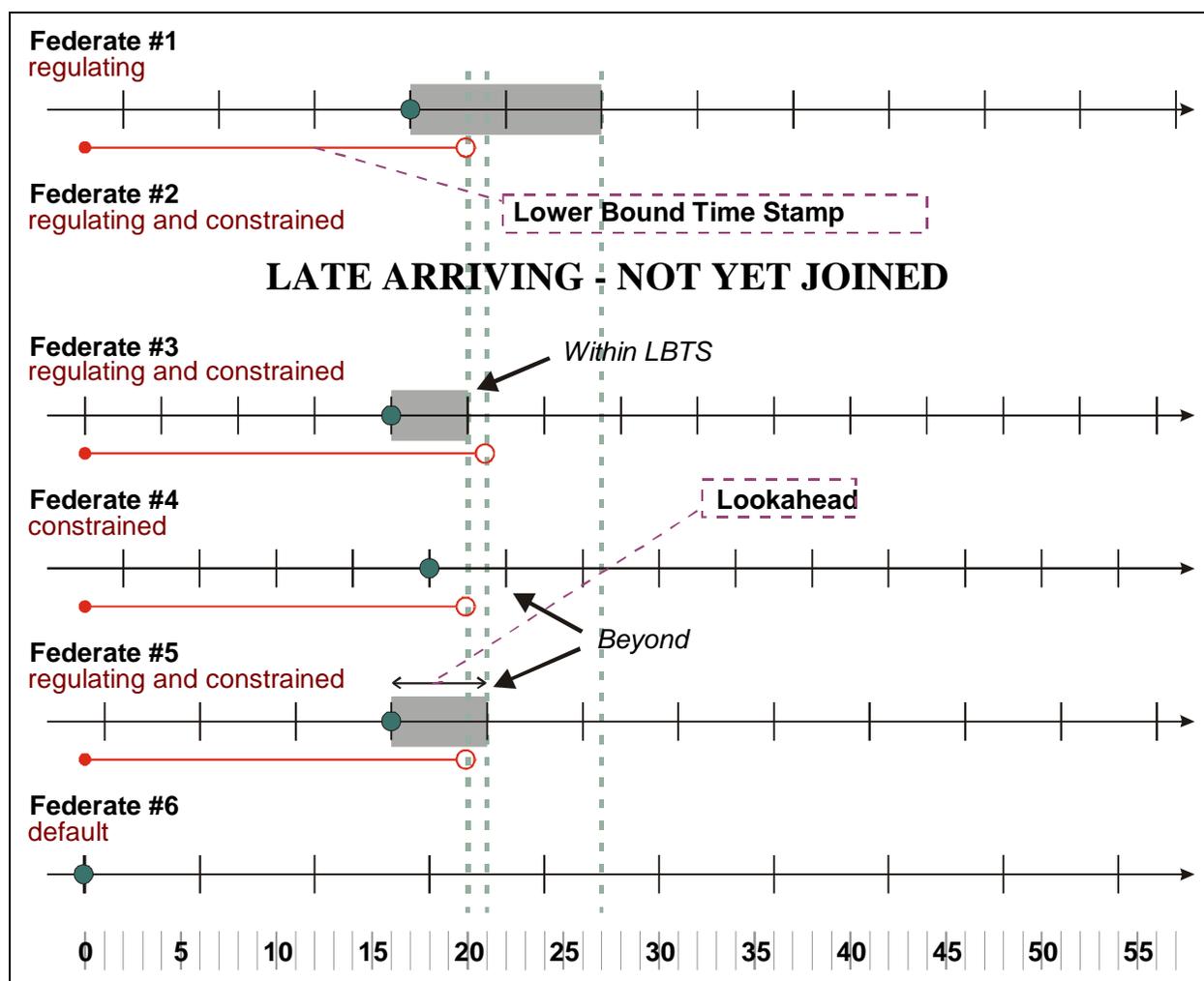


Figure 3-3. LBTS for Constrained Federates

3.4.2. Late Arriving Federate

Up to this point, Federate #2 has not arrived on the scene. If Federate #2 were to arrive at this point and insist upon being both regulating and constrained, it would be constrained as follows. At the time Federate #2 joins the federation, the LBTS of previously joined regulating federates will be calculated. Federate #2 must assume a time that ensures it will not generate a TSO message earlier than this LBTS. Figure 3-4 illustrates the arrival of Federate #2. When it joins the federation as a "regulating and constrained" federate, it is assigned an initial time of $t = 20$. Note that the Federate #2's lookahead value is ignored for purposes of assigning an initial time.

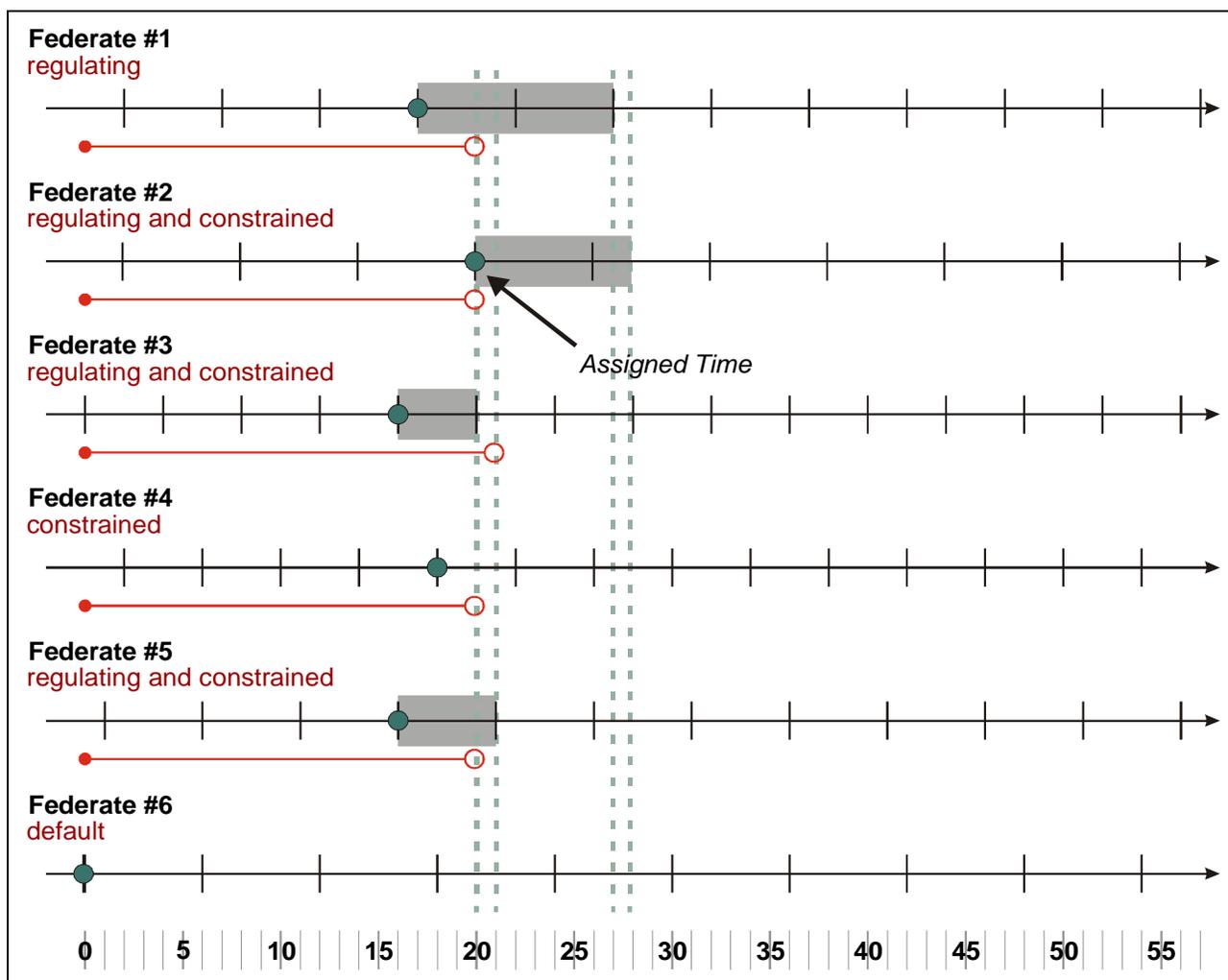


Figure 3-4. Late-Arriving Federate

3.5 "Receive-Ordered" v. "TSO" Events

In order for an event to be delivered TSO, four things must be true:

1. The sender must be "regulating."
2. The receiver must be "constrained."
3. The event itself must be identified as TSO.
4. The time on the TSO packed must be greater than the LBTS contribution of the sending regulating federate.

The third item refers to the time policy of the underlying event (e.g., an attribute update, an interaction). In the FED file, the time management default policy for object attribute and

interaction is specified as either "receive" or "timestamp."⁹ Attribute instances and interaction instances are delivered according to the time policy specified in the FED file, unless the default policy is overridden.¹⁰

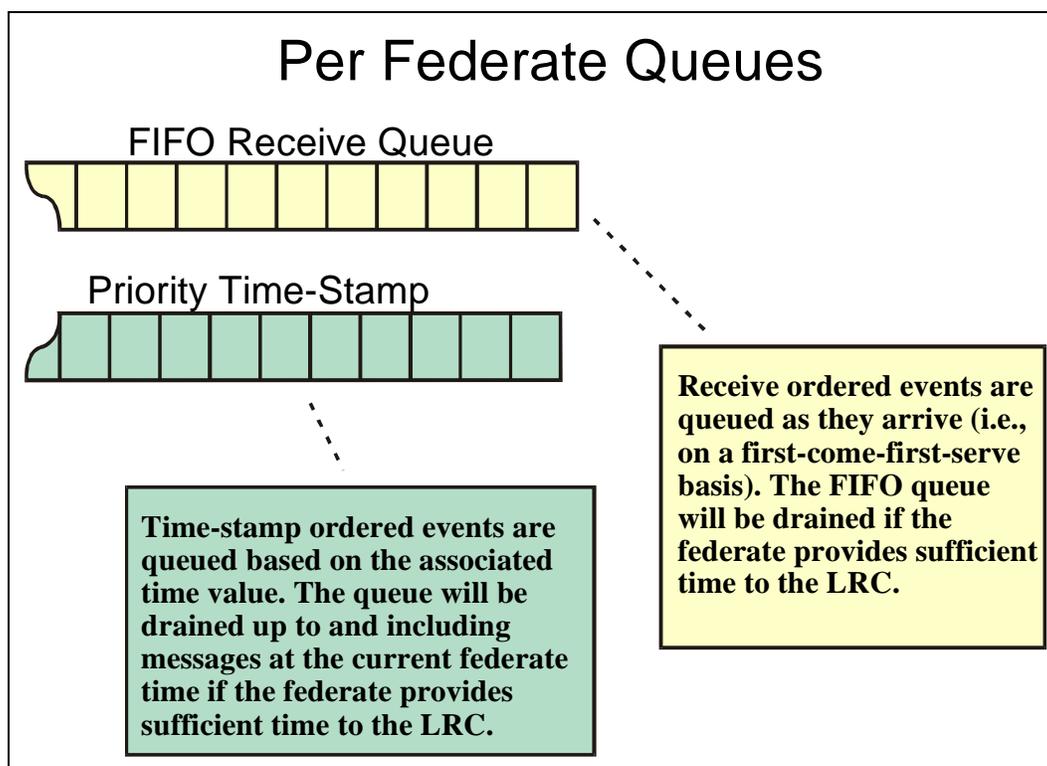


Figure 3-5. Per Federate Queues

As illustrated in Figure 3-5, each LRC maintains two queues. Events that meet the TSO criteria are placed in the time-stamp queue. The time-stamp queue orders incoming events based on the time stamp. Events that fail to meet the TSO criteria are placed in the receive queue in the order in which they arrive. Information in the receive-order queue is immediately available to the federate. The federate has access to all events in the TSO queue with time stamps less than or equal to the federate's perceived time.

3.5.1. EXAMPLE 1

If Federate #3 (in Figure 3-4) generates a TSO event, Federate #6 would see the event as a receive-ordered event. The event does not arrive as a TSO event because Federate #6 is unconstrained and therefore incapable of receiving events in time-stamped order. The same

⁹ It is not possible to set the time policy of individual parameters of an interaction. Policy is set at the interaction level (i.e., "all or nothing"). It is possible to specify time policy for individual attributes of an object. When an object with mixed time policies is updated, the update may result in both receive-ordered and time-stamp-ordered events.

¹⁰ It is possible to adjust time policy on a per attribute or per instance basis. See the RTIambassador methods `changeAttributeOrderType()` and `changeInteractionOrderType()` for more details.

event sent by Federate #3 and received by Federate #2 would be received as a TSO event because Federate #2 is constrained, and Federate #3 is regulating.

3.5.2. EXAMPLE 2

If Federate #4 attempts to generate an event that is TSO by default (i.e., according to the FED file), the event will be sent receive ordered since Federate #4 is not regulated. Only regulating federates may associate a time tag with an event.

3.5.3. SUMMARY

It is important to note that information can be exchanged between federates capable of communicating TSO events and federates that are not capable of communicating TSO events. However, the events are communicated as receive-ordered – a least common denominator approach.

4. FOM/SOM Development

The Federation Development and Execution Process (FEDEP) Model depicted in Figure 4-1, illustrates the major activities that should take place during the life cycle of a federation. This model starts with the definition of federation objectives through the federation development and concludes with the results of a running federation execution.

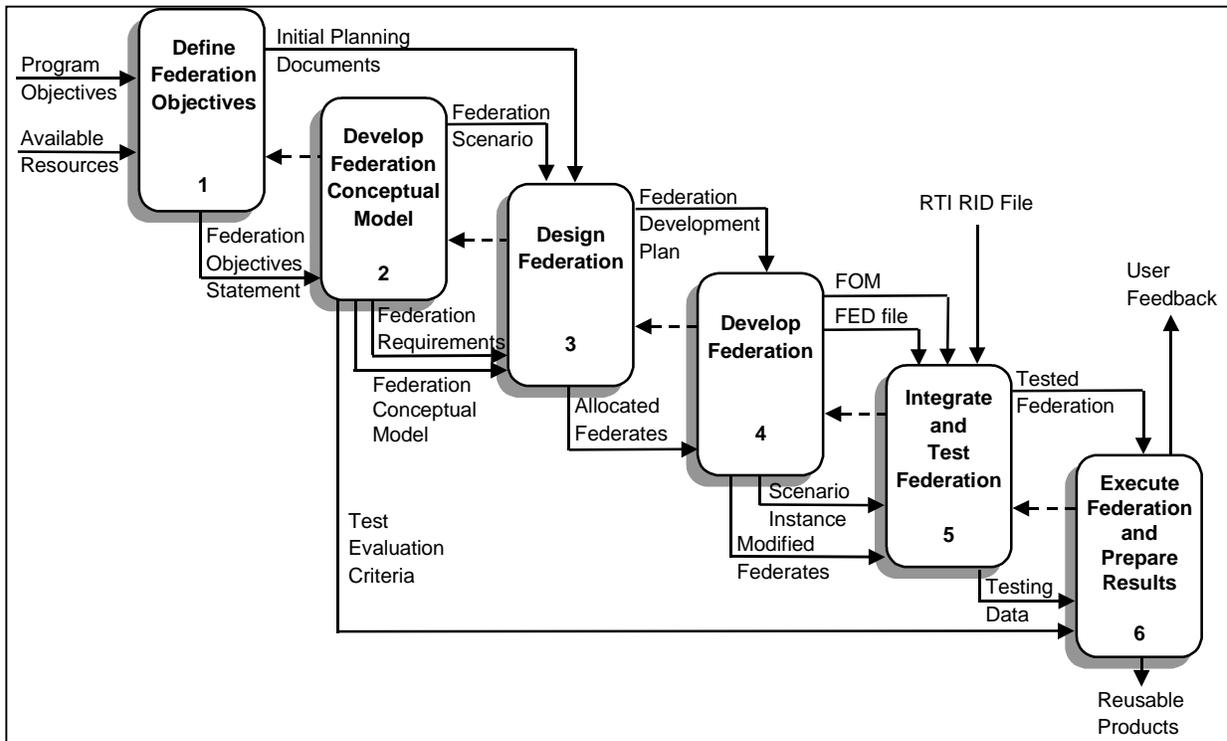


Figure 4-1. The Federation Development and Execution Process (FEDEP) Model

The HLA Federation Development and Execution Process (FEDEP) Model is intended to identify and describe the sequence of activities necessary to construct HLA federations. The HLA FEDEP Model description provided here has been heavily influenced by the experiences of the HLA prototype federations and other HLA user organizations. Federation developers may utilize the guidelines provided by the HLA FEDEP as a baseline process that may be tailored or modified as appropriate to meet specific objectives. In all cases, the development methodologies used to support the varying needs and interests of different application areas have been identified, and “best practices” merged into a single, broadly applicable, high-level framework

for HLA federation development and execution.¹¹ This document may be obtained via the DMSO website at <http://hla.dmsomil>.

DMSO has sponsored the development of several tools that automate various activities in the Federation Development and Execution Process. These tools are distributed freely via the HLA Software Distribution Center (SDC). Interested participants may visit the website at <http://hla.dmsomil> to become a registered user and obtain the freely available software.

The Object Model Development Tool (OMDT) is the first tool to be developed. It is currently available through the SDC. It supports the development of HLA compliant Simulation Object Models (SOMs) and Federation Object Models (FOMs).

FOM and SOM development is somewhat outside the scope of a Programmer's Guide for the Run-Time Infrastructure. Development of a SOM and FOM is a prerequisite to effectively using the RTI to facilitate interoperability between simulations. The FOM development process requires that the entire system be considered to determine things such as the object model that will describe the data communicated between the simulations, conditions for data update, and various other information that is pertinent to the specification of a simulation system for interoperability purposes. The Federation Execution Data (FED), which is required as an input to the RTI, is a subset of a FOM along with the specification of some default values for Ordering and Transport properties of data.

¹¹ Portions of this chapter are lifted directly or paraphrased from the HLA 1.3 Federation Development and Execution Process Model.

5. Federation Management

5.1 Introduction

This chapter introduces the RTIambassador services and FederateAmbassador callback functions that support federation management functionality. Federation management includes such tasks as creating federations, joining federates to federations, observing federation-wide synchronization points, effecting federation-wide saves and restores, resigning federates from federations, and destroying federations.

5.2 Primary Functions

Figure 5-1 illustrates the primary functions associated with the federation life cycle. The RTIambassador functions are presented alphabetically and in considerable detail in Appendix A, *RTI::RTIambassador*.

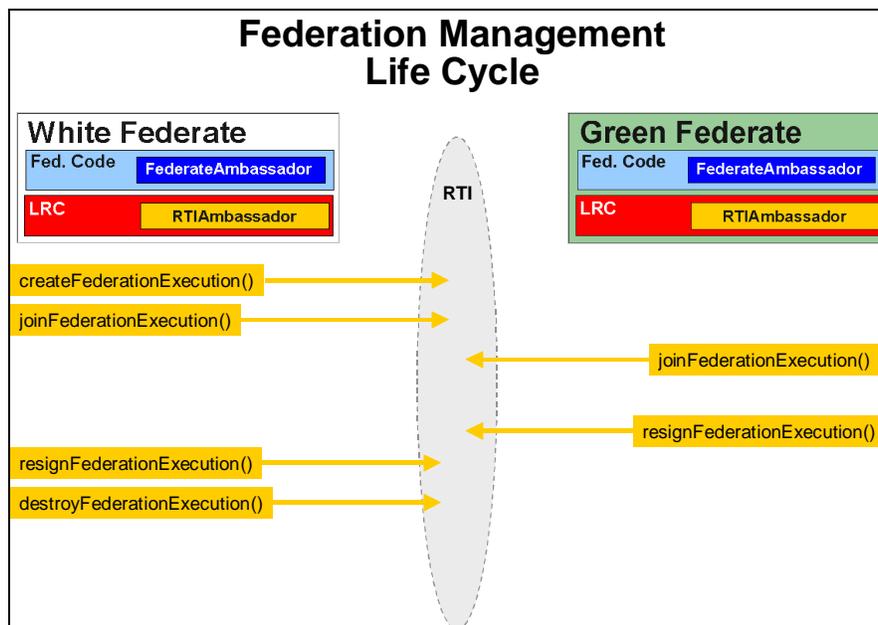


Figure 5-1. Federation Management Life Cycle

5.2.1. RTIambassador::createFederationExecution()

Because of calling the RTIambassador method `createFederationExecution()`, the Local RTI Component (LRC) communicates with the RtiExec process. If the specified federation does not exist, the RtiExec process creates a new FedExec process (as specified in Chapter 2, *RTI Synopsis*) and associates it with the supplied federation name. If the specified federation already exists, a `FederationExecutionAlreadyExists` exception is raised.

Frequently the same federate executable may be called upon to create a federation and at other times may be asked to participate in an established federation. This is certainly the case if the same simulation code is executed multiple times to function as multiple federates in a federation. If the `FederationExecutionAlreadyExists` exception is caught and ignored, then the call to

createFederationExecution() is robust – creating the federation if required and tolerating the existence of an existing federation execution.

5.2.2. RTIambassador::joinFederationExecution()

The joinFederationExecution() method is called to associate a federate with an existing federation execution. The method provides the non-unique name of the calling federate and the name of the federation execution that the federate is attempting to join. Additionally, a pointer to an instance of a class implementing the FederateAmbassador callback functions is required. The joinFederationExecution() method effectively says, "Here I am; and, here's how to get in touch with me."

5.2.3. RTIambassador::tick()

The tick() method is not a part of the Federation Management functionality identified in the *High Level Architecture Interface Specification Version 1.3*. It is, however, a very important part of the RTI 1.3-NG release. The LRC does a lot of work (e.g., exchanging information with counterparts) and needs time to do that work. The tick() method yields time to the RTI.

The tick() method exists in two forms – one taking zero arguments and another taking two arguments. The zero argument version yields time to each major activity within the LRC. A typical activity would be draining inbound event queues and providing callbacks to the federate via the FederateAmbassador. There is no guarantee as to the time required for a call to tick() to complete. The two argument version of tick() also yields time to the LRC, but suggests lower and upper bounds on the time being allotted to tick(). Like the no argument version, the two-argument version makes no guarantees as to its overall execution time. It, too, yields time to each major activity within the LRC, iterating as time permits.

Calling tick() is immensely important. Failure to tick() the LRC can lead to federation-wide problems. For example, while a late arriving federate is attempting to join an existing federation, information is being passed to the LRCs of the existing federates. If the existing federates are not ticking their LRC, the late arriving federate (and probably everyone else) is effectively blocked.

One final note, tick() is not an advancing time mechanism. See the Time Management section (Chapter 6) for time advancement methodology and services.

5.2.4. RTIambassador::resignFederationExecution()

The RTIambassador method resignFederationExecution() terminates a federate's participation in a federation. When a federate leaves a federation, something must be done with the objects for which the federate has update responsibility. Typically, this responsibility extends to (a) object instance (or object instance attributes) that the federate introduced (and has not negotiated away) and (b) additional responsibilities the federate has assumed¹².

¹² The specific details of object creation and ownership management are left to subsequent chapters.

The sole argument to `resignFederationExecution()` is a member of the `ResignAction` enumeration. A federate can "RELEASE_ATTRIBUTES," "DELETE_OBJECTS," "DELETE_OBJECTS_AND_RELEASE_ATTRIBUTES," or take "NO_ACTION." The `resignFederationExecution()` manual page provides additional details.

5.2.5. RTIambassador::destroyFederationExecution()

The `destroyFederationExecution()` method attempts to terminate an executing federation. If successful, the `FedExec` associated with the federation terminates. If the invoking federate is not the last federate to have resigned and there are still federates joined in the targeted federation, a `FederatesCurrentlyJoined` exception is raised.

5.3 FoodFight Example

The following code excerpt demonstrates typical code for creating and joining a federation.

```

01 void
02 CreateAndJoinFederation (Pstring federation_name,
03                          Pstring federate_name)
04 {
05     // Abstract
06     //     Attempt to create the FoodFight federation. Tolerate the fact
07     //     that the federation may already exist -- i.e., that this is a
08     //     "late arriving" federate.
09
10
11     // Attempt to create the FoodFight federation.
12     cout << "Creating the federation '" << federation_name << "'." << endl;
13
14
15     try
16     {
17         rti_ambassador.createFederationExecution(federation_name,
18                                                  "FoodFight.fed");
19     }
20     catch (RTI::FederationExecutionAlreadyExists&)
21     {
22         // Caught and ignored -- effectively allowing this condition.
23         cout << federation_name << " already exists." << endl;
24     }
25     catch (RTI::Exception& e)
26     {
27         cerr << "createFederationExecution() produced >" << &e << '<';
28         throw;
29     }
30     catch (...)
31     {
32         cerr << "createFederationExecution() produced unknown exception.";
33         throw;
34     }
35
36     cout << "RtiAmbWrapper: '" << federate_name << "' joining '"
37           << federation_name << "'." << endl;
38
39     // Attempt to join the federation.
40     for (int timer = RtiAmbWrapper::MAX_JOIN_TRIES; timer; /*NO-OP*/)
41     {

```

```

42     try
43     {
44         RTI::FederateHandle fed_handle =
45             rti_ambassador.joinFederationExecution(
46                 federate_name,
47                 federation_name,
48                 p_fedamb);
49
50         // If no exceptions encountered, abandon loop.
51         timer = 0;
52     }
53     catch (RTI::FederationExecutionDoesNotExist&)
54     {
55         if (--timer == 0)
56         {
57             cerr << "joinFederationExecution() failed.";
58             throw;
59         }
60         else
61         {
62             cout << "joinFederationExecution() failed, " << timer
63                 << " tries left." << endl;
64             ::SleepInSeconds(1);
65         }
66     }
67     catch (RTI::Exception& e)
68     {
69         cerr << "joinFederationExecution() produced >" << e << '<';
70         throw;
71     }
72     catch (...)
73     {
74         cerr << "joinFederationExecution() produced unknown exception.";
75         throw;
76     }
77 }
78 }

```

The important function calls in the preceding example occur in lines 17 and 45 where the federation is created and joined, respectively. The remaining code provides running commentary to `cout`¹³ and exception handling. The `FederationExecutionAlreadyExists` exception is caught and essentially ignored. Most remaining exceptions are caught, logged, and re-thrown.¹⁴

The call to `joinFederationExecution()` may produce the `FederationExecutionDoesNotExist` exception. Once `createFederationExecution()` is called, it takes time to create and initialize the resulting `FedExec` process. The preceding code, written for RTI 1.3v6, was designed to "spin" until the join is successful or until a predetermined number of join attempts is exhausted. However, this technique is no longer necessary. RTI 1.3-NG is ready to accept joining federates upon return from the `createFederationExecution()` invocation.

¹³ This code is lifted from a training example; therefore, a lot of information is printed to the standard output.

¹⁴ There are some benefits to logging exceptions at the point of occurrence and as the exception passes up the call stack. The resulting code is a little bulky, but the stack trace can simplify debugging.

The following code illustrates how a federate might resign from and destroy a federation.

```

79 void
80 ResignAndDestroyFederation (Pstring federation_name,
81                             Pstring federate_name)
82 {
83     cout << "Resigning from and attempting to destroy '" << federation_name
84           << "'." << endl;
85
86     try
87     {
88         rti_ambassador.resignFederationExecution(
89             RTI::DELETE_OBJECTS_AND_RELEASE_ATTRIBUTES);
90     }
91     catch (RTI::Exception& e)
92     {
93         cerr << "resignFederationExecution() produced >" << &e << '<';
94         throw;
95     }
96     catch (...)
97     {
98         cerr << "resignFederationExecution() produced unknown exception.";
99         throw;
100    }
101
102    try
103    {
104        rti_ambassador.destroyFederationExecution(federation_name);
105    }
106    catch (RTI::FederatesCurrentlyJoined&)
107    {
108        // We'll allow this condition -- catching and ignoring.
109    }
110    catch (RTI::Exception& e)
111    {
112        cerr << "destroyFederationExecution() produced >" << &e << '<';
113        throw;
114    }
115    catch (...)
116    {
117        cerr << "destroyFederationExecution() produced unknown exception.";
118        throw;
119    }
120 }

```

The code supporting the `destroyFederationExecution()` call tolerates (i.e., catches and ignores) the `FederatesCurrentlyJoined` exception. Other exceptions are caught, logged, and re-thrown.

Finally, the following code shows how `tick()` might be factored in.

```

121 void
122 PrimarySimulation (int regulating_flag,
123                  int constrained_flag)
124 {
125     // Abstract
126     //     This function produces the FoodFight simulation.
127
128     Pstring federation_name("FoodFight");
129     Pstring federate_name("ExampleFederate");
130 }

```

```

131 // Create and join the FoodFight federation.
132 ::CreateAndJoinFederation(federation_name, federate_name);
133
134 while (... stuff to do ...)
135 {
136     ... do some simulation work ...
137
138     // Yield some time to the RTI.
139     ::rti_ambassador.tick(1.0, 1.0);
140 }
141
142 // Resign from the federation execution and attempt to destroy.
143 ::ResignAndDestroyFederation(federation_name, federate_name);
144 }

```

This tick() example is a bit oversimplified, but introduces the notion of yielding time to the LRC.

5.4 Federate Synchronization

The RTI 1.3 specification provides functions for synchronizing activities between federates participating in a federation. The RTI provides mechanisms for exchanging information between federates. It is possible to associate times with exchanged information and thereby coordinate federate activities. The Federation Management synchronization functions allow federates to communicate explicit synchronization points. Figure 5-2 illustrates the RTIambassador service calls extended to a federate and the resulting FederateAmbassador callback functions that together support a synchronization capability. The RTIambassador method registerFederationSynchronizationPoint() accepts a label, a tag, and (optionally) a set of target federates. [By default, all federates are targeted.] The label and tag are communicated to targeted federates. The specific role of the label and tag are outlined in detail in the appendices.

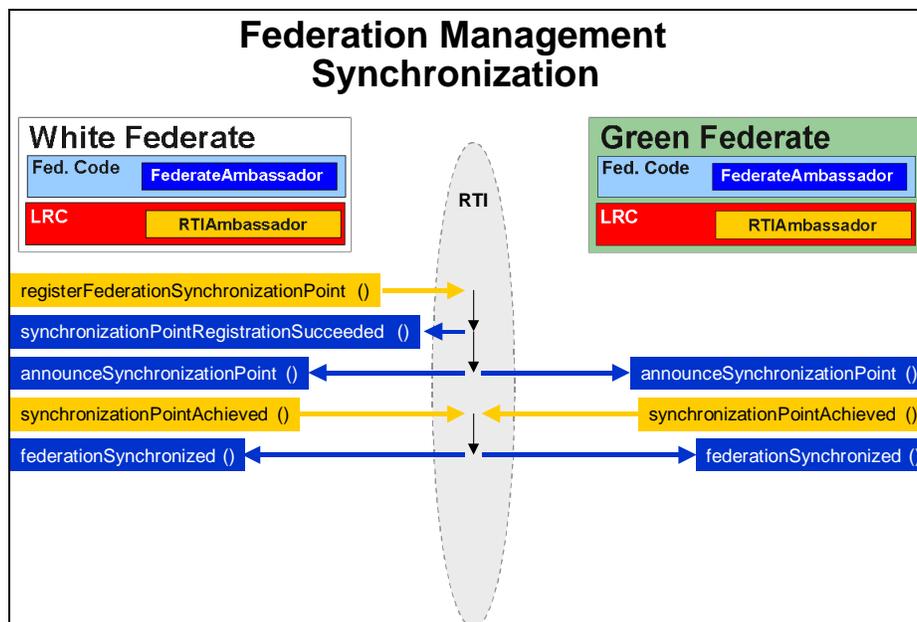


Figure 5-2. Federate Management Synchronization

5.5 Save/Restore

The RTI provides functions for coordinating federation-wide saves and restores. Figures 5-3 and 5-4 illustrate save and restore functions, respectively. The programmer reference pages included as Appendices A through C, should be consulted for syntactic and semantic details.

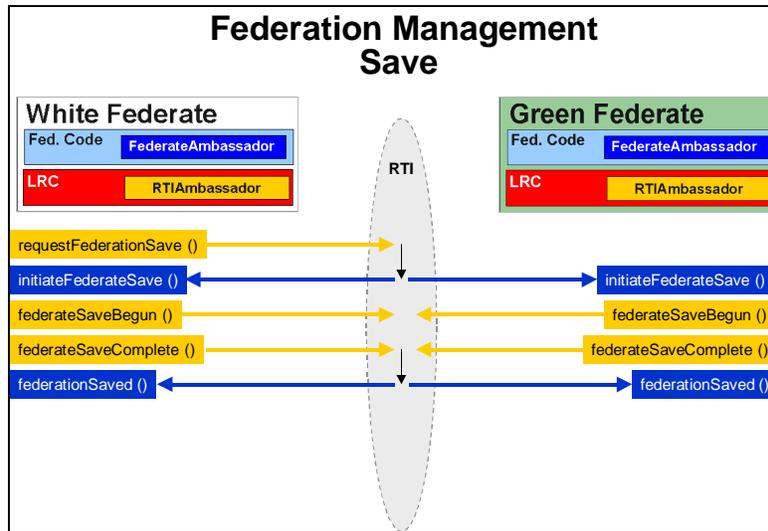


Figure 5-3. Federation Management Save

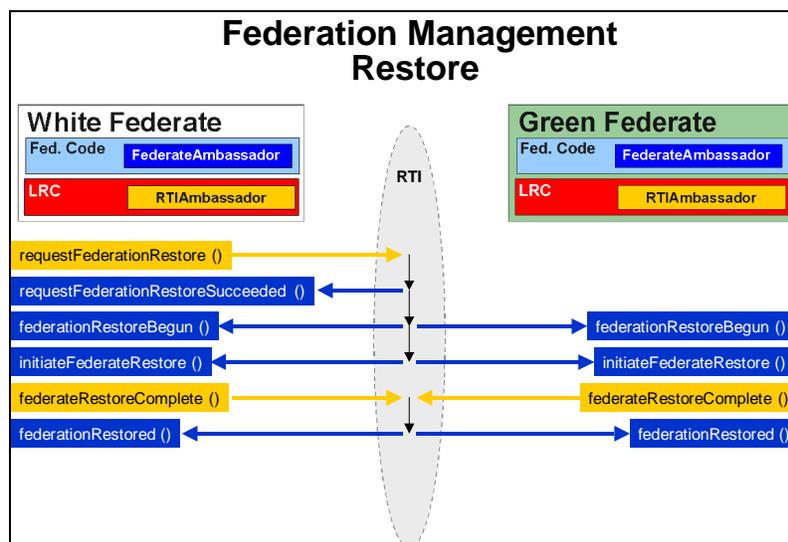


Figure 5-4. Federation Management Restore

6. Time Management

6.1 Introduction

This chapter introduces the RTIambassador service and FederateAmbassador callback methods that support time management functionality. The RTI provides a variety of optional time management services. Though optional, it is important to understand the time management models available in the RTI and the implication of exchanging events between federates with different time management policies. Chapter 3, *The Role of Time*, introduces the philosophy of time management. The focus here is on the mechanics required to implement time management policies and negotiate time advances.

6.2 Toggling "regulating" and "constrained" Status

Chapter 3, *The Role of Time*, presented the definitions for "regulating" and "constrained." Figure 6-1 identifies the RTIambassador and FederateAmbassador member functions associated with establishing whether a federate is regulating or not, and whether a federate is constrained or not. Key methods are presented briefly below and discussed in detail in the appendices.

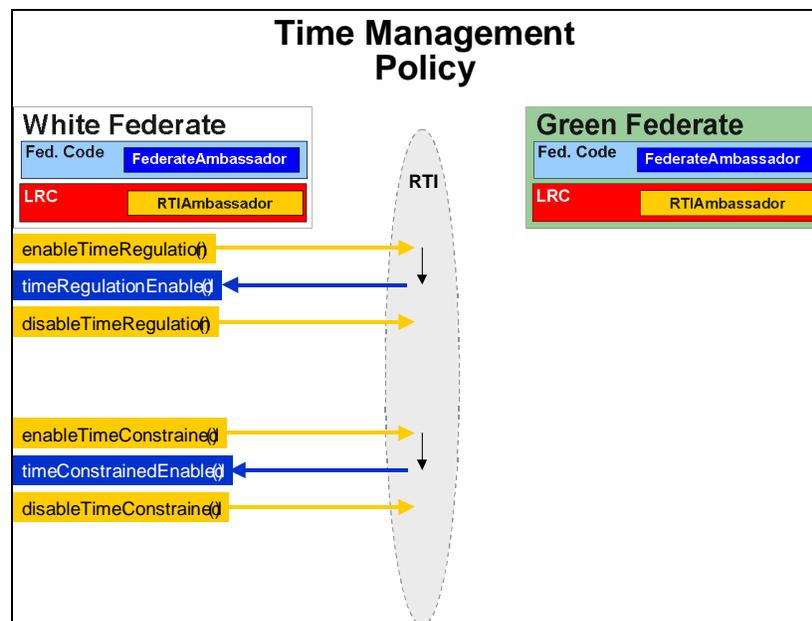


Figure 6-1. Toggling "regulating" and "constrained" Status

6.2.1. Regulation Policy

Federates have *regulation* disabled by default. A federate uses the RTIambassador member function `enableTimeRegulation()` to request that the federate be acknowledged as a regulating federate. The Local RTI Component (LRC) calls the FederateAmbassador callback `timeRegulationEnabled()` to inform a federate that the `enableTimeRegulation()` request has been granted and informs the federate of its (possibly new) logical time. In Section 3.4, *Advancing Time*, the effect of a late arriving federate wishing to be time regulating was discussed. In short, such a federate is obligated to advance to a time such that the current LBTS of existing federates is guaranteed to be honored.

It is possible to change the regulation policy dynamically. The RTIambassador method `disableTimeRegulation()` is the counterpart to `enableTimeRegulation()`. Unlike `enableTimeRegulation()`, `disableTimeRegulation()` takes effect immediately.

6.2.2. Constrained Policy

Federates have *constrained* disabled by default. A federate uses the RTIambassador member function `enableTimeConstrained()` to request that the federate be acknowledged as a constrained federate. The `timeConstrainedEnabled()` callback informs a federate that the `enableTimeConstrained()` request has been granted. It is possible to change the constrained policy dynamically. The RTIambassador method `disableTimeConstrained()` is the counterpart to `enableTimeConstrained()`. Unlike `enableTimeConstrained()`, `disableTimeConstrained()` takes effect immediately.

6.3 Time Advance Requests

Three variants of the time advancement service exist to provide the requisite functionality for time-step, event-based, and optimistic federates. Federates may employ any combination of time management scheme and time advancement services throughout the execution.

6.3.1. Time-Stepped Federates

Time-stepped federates will calculate values based on a point in time and then process all events that occur up to the next point in time (current time + time step). Figure 6-2 illustrates the functions used to advance a federate's logical time for a time-stepped simulation.

When a `timeAdvanceRequest()` or `timeAdvanceRequestAvailable()` service is used, the federate's LRC will be eligible to release all receive order messages from the FIFO Queue and all time-stamp ordered messages that have a time stamp less than or equal to the time requested from the TSO queue. After all TSO messages in a federation execution with time less than or equal to the requested time have been received, the federate will receive a `timeAdvanceGrant()` callback via the FederateAmbassador with time equal to that which was requested in the `timeAdvanceRequest()` or `timeAdvanceRequestAvailable()`. See the time management manual pages for a more detailed discussion of the released events and the granted time for the `timeAdvanceRequest()` and `timeAdvanceRequestAvailable()` services.

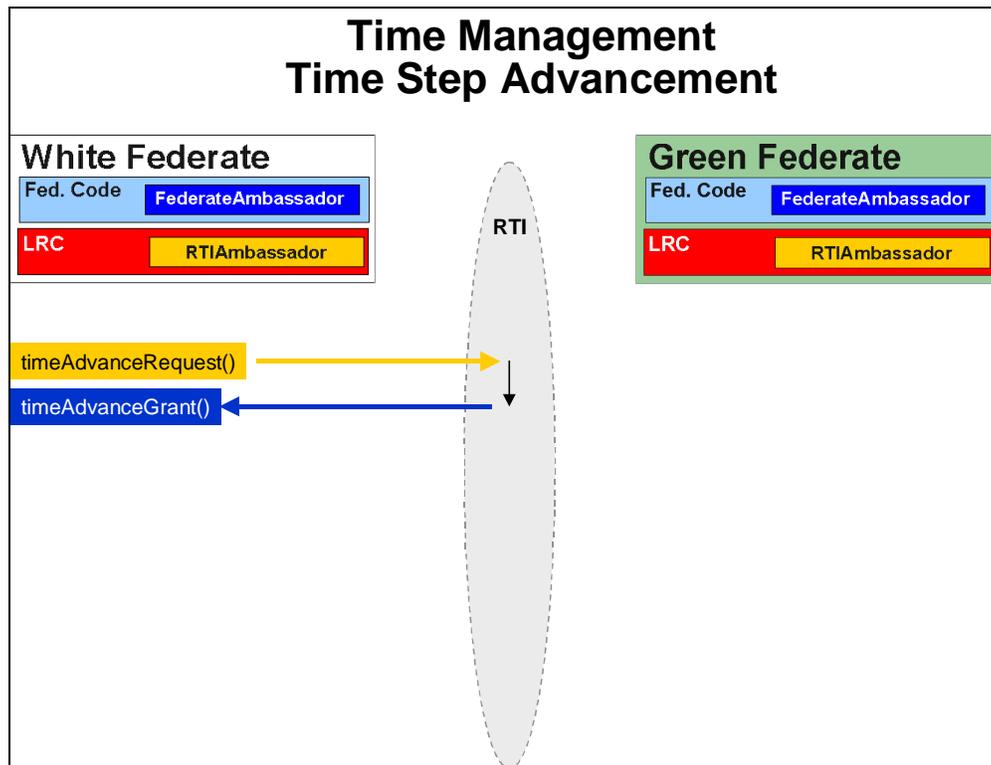


Figure 6-2. Logical Time Advancement for a Time-Step Federate

6.3.2. Event-Based Federates

Event-based federates will calculate values based on each event received from the federation execution. After an event is processed, the federate may need to send new events to the federation execution. This implies that the events may not happen on set time intervals but the times of events will be based on the time of the received events. Figure 6-3 illustrates the functions used to advance a federate's logical time for an event-based simulation.

When a `nextEventRequest()` or `nextEventRequestAvailable()` service is used, the federate's LRC will be eligible to release all receive order messages from the FIFO Queue and all time-stamp ordered messages that have a time stamp equal to the minimum next event time of any message that will be delivered as TSO.

After all possible TSO messages with time equal to the minimum next event time have been received, the federate will receive a `timeAdvanceGrant()` callback via the `FederateAmbassador` with time equal to the minimum next event time or the time requested in the `nextEventRequest()` or `nextEventRequestAvailable()`, whichever is less. See the programmer reference pages for a more detailed discussion of the released events and the granted time for the `nextEventRequest()` and `nextEventRequestAvailable()` services.

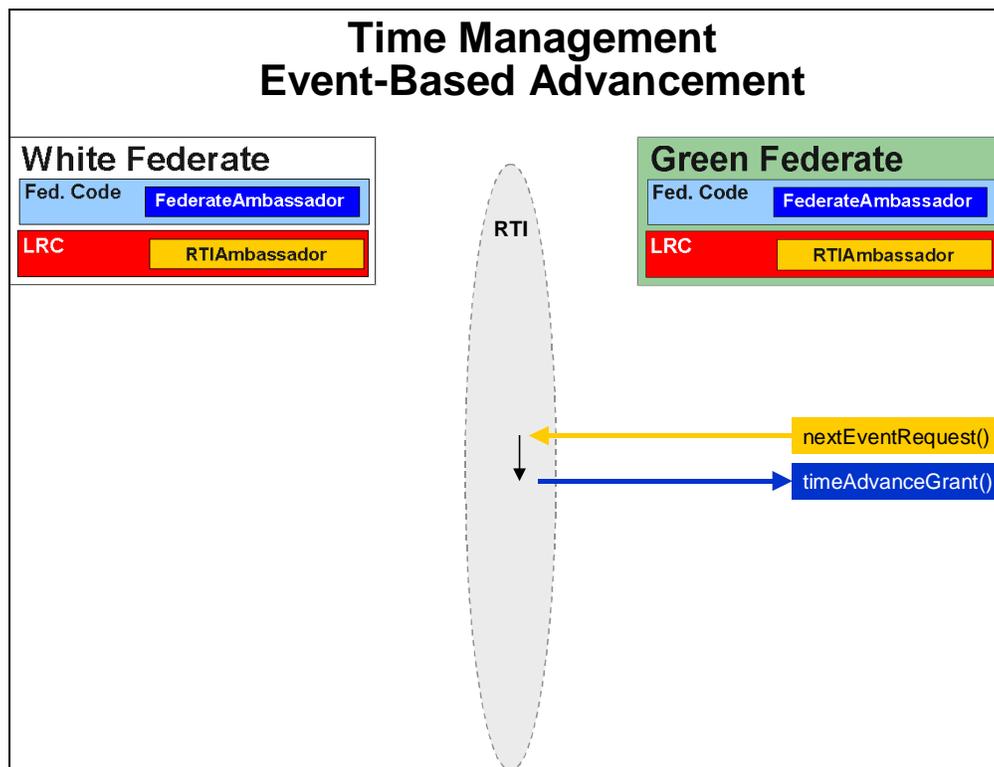


Figure 6-3. Logical Time Advancement for an Event-Based Federate

6.3.3. Optimistic Federates

Optimistic federates do not want to be constrained by the time advancement of regulating federates but instead will proceed ahead of LBTS to calculate and send events in the future. These federates will want to receive all of the events that have been sent in the federation execution regardless of the time-stamp ordering. A federate that uses the `flushQueueRequest()` service is likely to generate events that are in the future of messages that it has yet to receive. The messages that are received with a time-stamp less than messages already sent may invalidate the previous messages. In this case, the optimistic federate will need to retract the messages that have been invalidated and all federates that have received the invalid messages will receive a `requestRetraction()` callback on their `FederateAmbassador`. See the programmer reference pages for a detailed discussion of the `retract()` and `requestRetraction()` services.

When the `flushQueueRequest()` service is used, the federate's LRC will be eligible to release all receive order messages from the FIFO Queue and all time-stamp ordered messages from the TSO queue. After all TSO messages that were in the queue at the time of the `flushQueueRequest()` invocation have been released, the federate will receive a `timeAdvanceGrant()` callback via the `FederateAmbassador` with time equal to LBTS or the time

requested in the `flushQueueRequest()`, whichever is less. See the programmer reference pages for a more detailed discussion of the released events and the granted time for the `flushQueueRequest()` service. Figure 6-4 illustrates the functions used to advance a federate's logical time for an optimistic simulation.

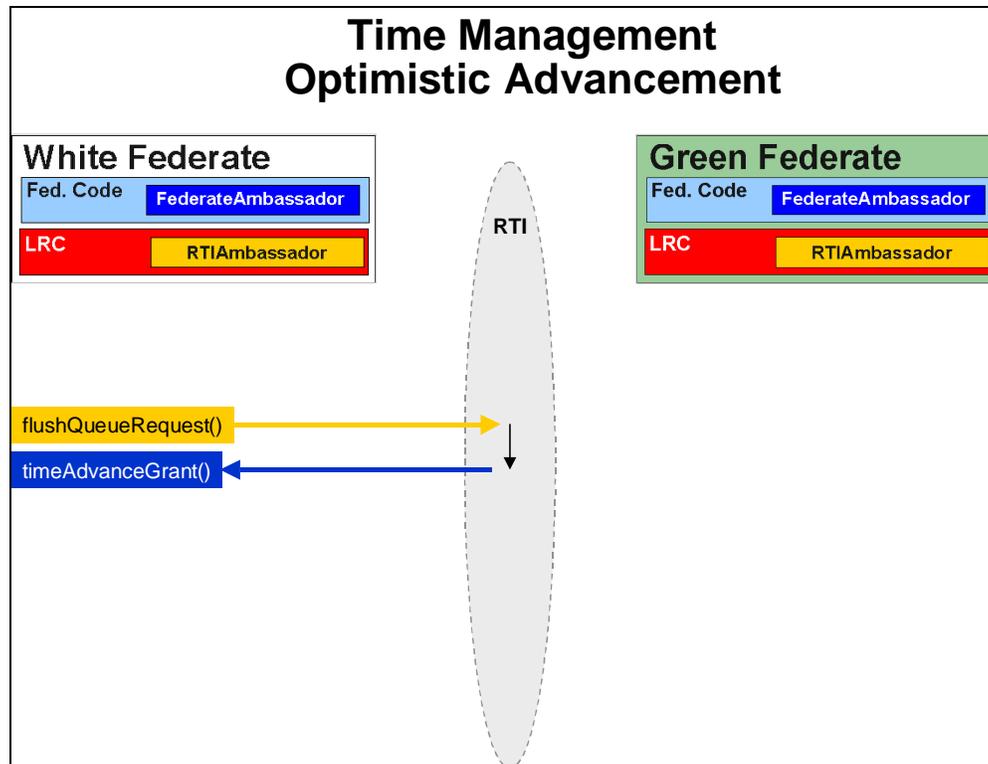


Figure 6-4. Logical Time Advancement for an Optimistic Federate

6.4 FoodFight Example

Time advance requests are made through the `RTIAmbassador` instance. Time advance grants are received through the `FederateAmbassador` instance. User code must unite the request and grant. This pattern is repeated throughout the RTI. One approach to uniting code is to communicate through global variables (globals).¹⁵ In the following examples, globals are used to tie the following service request, callback pairs:

```
RTIAmbassador::enableTimeRegulation()→FederateAmbassador::timeRegulationEnabled()
```

```
RTIAmbassador::enableTimeConstrained()→FederateAmbassador::timeConstrainedEnabled()
```

¹⁵ Solutions that rely on globals typically do not scale well. Globals introduce a variety of additional problems. None the less, globals are used here for compact examples where the emphasis is on the RTI application interface and not good programming practices. Good C++ programmers should immediately see alternatives to the use of globals and will likely adopt an alternative approach.

RTIambassador::timeAdvanceRequest()→FederateAmbassador::timeAdvanceGrant()

Whenever user-defined global variables or global functions are used in coding examples, they're preceded by the global scope resolution operator "::". While it is a bad practice to use globals, it is a good practice to identify any globals with this operator.

globals.h

```

145 #ifndef globals_h
146 #define globals_h
147
148 #include <RTI.hh>
149     :
150 // The time of the last time advance grant.
151 extern RTI::FedTime* p_current_time;
152
153 // The lookahead period promised by this federate. Set when federate
154 // attempts to become regulating.
155 extern RTI::FedTime* p_lookahead;
156
157 // Variable indicating whether we currently have a pending time advance
158 extern int time_advance_outstanding;
159
160 // Flag indicating if federate is constrained.
161 extern int regulation_enabled;
162
163 // Flag indicating if federate is constrained.
164 extern int constrain_enabled;
165     :
166 #endif
167 // globals_h

```

globals.cxx

```

168 #include "globals.h"
169     :
170 RTI::FedTime* p_current_time = new RTIfedTime(0.0);
171 RTI::FedTime* p_lookahead = 0; // Not set yet.
172 int time_advance_outstanding = 0;
173 int regulation_enabled = 0; // Disabled by default.
174 int constrain_enabled = 0; // Disabled by default.
175     :

```

::PrimarySimulation()

```

176 void
177 PrimarySimulation (int regulating_flag,
178                  int constrained_flag)
179 {
180     // Abstract
181     // This function produces the FoodFight simulation.
182
183     Pstring federation_name("FoodFight");
184     Pstring federate_name("ExampleFederate");
185
186     // Create and join the FoodFight federation.

```

```

187     ::CreateAndJoinFederation(federation_name, federate_name);
188
189     // Set up time-management services.
190     if (regulating_flag)
191     {
192         ::p_lookahead = new RTIfedTime(1.0); // One second.
193         ::rti_ambassador.enableTimeRegulation(*::p_current_time,*::p_lookahead);
194
195         // A request to become regulating is, effectively, a time advance
196         // request.
197         ::time_advance_outstanding = 1;
198     }
199
200     if (constrained_flag)
201     {
202         cout << "Federate is constrained!" << endl;
203         ::rti_ambassador.enableTimeConstrained();
204     }
205     :
206     :
207     // The time interval for this federate has been set (arbitrarily) to the
208     // lookahead value.
209     RTI::FedTime* p_interval = new RTIfedTime(0.0);
210     *p_interval = *::p_lookahead;
211
212     while (::local_students.entries())
213     {
214         if (!::time_advance_outstanding)
215         {
216             // Do one interval's worth of simulation.
217             :
218             :
219             // Attempt to advance federate's logical time. The logical time
220             // isn't officially advanced until a time advance grant is
221             // received. If a regulating federate is still attempting to
222             // generate events, it should pretend like the time advance has
223             // been granted for the purpose of observing its lookahead promise.
224             *::p_current_time += *p_interval;
225             ::rti_ambassador.timeAdvanceRequest(*::p_current_time);
226             ::time_advance_outstanding = 1;
227         }
228         :
229         :
230         // Work to be interleaved with tick only!
231         :
232         :
233         // Tick the RTI, initiating federate ambassador callbacks.
234         ::rti_ambassador.tick(1.0, 1.0);
235     }
236
237     // Resign from the federation execution and attempt to destroy.
238     ::ResignAndDestroyFederation(federation_name, federate_name);
239 }

```

The two-argument form of tick() is called in the preceding example. In the example, the goal is to slow the simulation so students can observe simulation progress. As an alternative to line 234, the no-argument version of tick() might be used. Between calls to tick() and prior to receiving a time advance grant, the federate may choose to interleave some preparatory work.

```
240         ::rti_ambassador.tick();    // Alternative to line 234 above.
```

FoodFightFedAmb.h

```
241 class FoodFightFedAmb : public DefaultFedAmb
242 {
243     // Abstract
244     //     The DefaultFedAmb defines all the federate ambassador methods
245     //     to "do nothing". Here, we override the ones we're interested
246     //     in.
247
248 public:
249     virtual void timeRegulationEnabled (const FedTime&)
250         throw (InvalidFederationTime, EnableTimeRegulationWasNotPending,
251             FederateInternalError);
252
253     virtual void timeConstrainedEnabled (const FedTime&)
254         throw (InvalidFederationTime, EnableTimeConstrainedWasNotPending,
255             FederateInternalError);
256
257     virtual void timeAdvanceGrant (const RTI::FedTime&)
258         throw (RTI::InvalidFederationTime, RTI::TimeAdvanceWasNotInProgress,
259             RTI::FederateInternalError);
260
261     :
```

FoodFightFedAmb.cxx

```
262     :
263 void
264 FoodFightFedAmb::
265 timeRegulationEnabled (const FedTime& time)
266 throw (InvalidFederationTime, EnableTimeRegulationWasNotPending,
267     FederateInternalError)
268 {
269     cout << "Federate acknowledged as regulating!" << endl;
270     ::regulation_enabled = 1;
271     ::time_advance_outstanding = 0;
272     *::p_current_time = time;
273 }
274
275 void
276 FoodFightFedAmb::
277 timeConstrainedEnabled (const FedTime&) // Argument ignored below.
278 throw (InvalidFederationTime, EnableTimeConstrainedWasNotPending,
279     FederateInternalError)
280 {
281     cout << "Federate acknowledged as constrained!" << endl;
282     ::constrain_enabled = 1;
283 }
284
285 void
286 FoodFightFedAmb::
287 timeAdvanceGrant (const RTI::FedTime& time)
288 throw (RTI::InvalidFederationTime,
289     RTI::TimeAdvanceWasNotInProgress, RTI::FederateInternalError)
290 {
291     if (!::time_advance_outstanding)
```

```

292     {
293         const char* err_msg = "Unexpected timeAdvanceGrant().";
294         cerr << err_msg;
295         throw RTI::TimeAdvanceWasNotInProgress(err_msg);
296     }
297
298     if (time < *::p_current_time)
299     {
300         const char* err_msg = "Old time passed in timeAdvanceGrant().";
301         cerr << err_msg;
302         throw RTI::InvalidFederationTime(err_msg);
303     }
304
305     try
306     {
307         *::p_current_time = time;
308         ::time_advance_outstanding = 0;
309
310         // Display current time.
311
312         char* p_string(0);
313         p_current_time.getPrintableString(p_string);
314
315         cout << "t = " << p_string << endl;
316
317         delete p_string;
318     }
319     catch (...)
320     {
321         const char* err_msg = "Exception caught in timeAdvanceGrant().";
322         cerr << err_msg;
323         throw RTI::FederateInternalError(err_msg);
324     }
325 }
326 :

```

6.5 Time-Related Queries

Several additional time management functions are available to query or fine tune time policy. Figure 6-4, Time Queries, shows additional functions. Consult the programmer reference pages for these functions for detailed overall description of services.

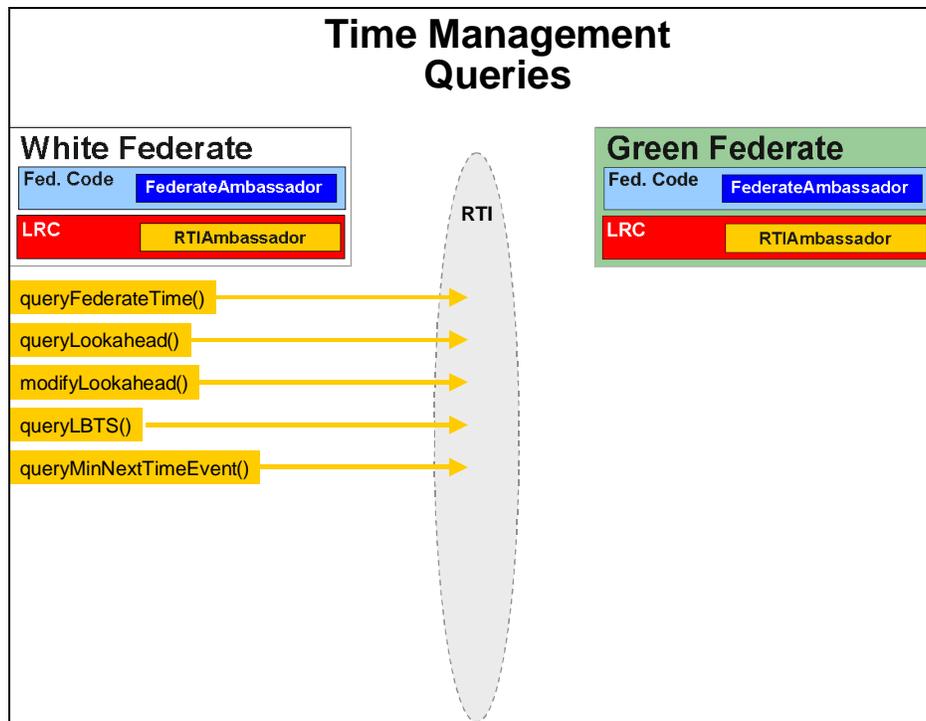


Figure 6-4. Time Queries

6.6 Polling vs. Asynchronous I/O Tick() Strategies

There are currently two process model strategies that are supported by the RTI; (1) polling process model and (2) asynchronous I/O process model. The polling process model uses a single thread of execution shared between the RTI and the federate. This strategy requires that the federate provide sufficient `tick()` invocations to transfer processor control to the LRC and allow the RTI to perform work. The federate must be aware that it can starve the RTI if `tick` is not called appropriately. The polling strategy model was only method provided by previous RTI releases. The asynchronous I/O process model uses an internal thread within the RTI to avoid starvation. This thread will periodically wake up and determine if it can perform any internal RTI work. In the asynchronous I/O strategy the federate only needs to invoke `tick` when it is prepared to handle callbacks. Use of the asynchronous I/O strategy requires the federate to consider two key points. First, using the asynchronous I/O process model does not prohibit the federate from calling `tick` anytime it is deemed appropriate. Second, data will be stored in the queue until `tick` is called allowing for delivery and storage clearing. If `tick` is not called, by either the RTI or the federate, there is a potential of memory exhaustion and data loss. Strategy selection is made via the `RTI.ProcessModel.StrategyToUse` parameter in the `RTI.rid` file. The default strategy is the asynchronous I/O process model.

7. Declaration Management

7.1 Introduction

This chapter introduces the RTIambassador service and FederateAmbassador callback methods that support declaration management. Declaration management includes publication, subscription and supporting control functions. Federates that produce objects instances (or object attributes) or that produce interactions must declare exactly what they are able to publish (i.e., generate). Federates that consume object instances (or object attributes) or that consume interactions must declare their subscription interests.

The RTI keeps track of what participating federates can produce and what they are interested in consuming and sends control signals to intelligently distribute notification of what is produced based on consumer interest. As depicted in Figure 7-1, the RTI uses control signals to inform producers exactly what they should transmit. The goal is to keep traffic off the communications network.

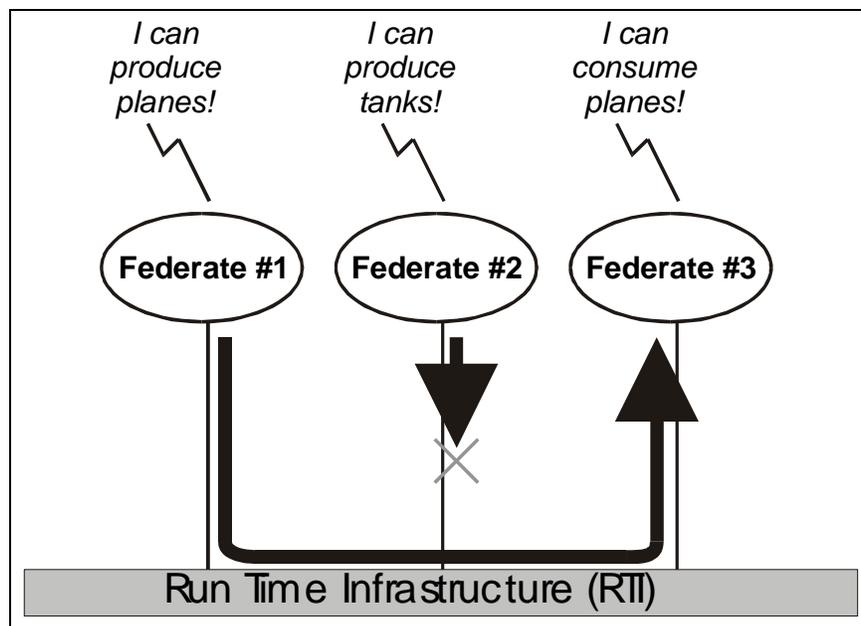


Figure 7-1. Control Signal Schema

7.2 Object Vocabulary Review

It is worth a moment to review some basic HLA terminology.

Object classes are comprised of attributes. Object classes describe *types* of things that can *persist*. For example, "tank" might be an object class. Objects of type "tank" have certain attributes (e.g., size, weight, and range). Actual, real tanks are *instances* of the object class tank. The term "object" standing alone is sometimes used to describe an instance of a particular object

class, but sometimes refers to the type information. Object classes may be related to cookie cutters and object instances to the cookies produced using the cookie cutters.

Interaction classes are comprised of parameters. Interaction classes describe *types* of events. Interaction instances are specific events. It is fair to say, "Objects are similar to interactions in so much as objects are comprised of attributes and interactions are comprised of parameters." The HLA recognizes this inherent symmetry and leverages it when appropriate. The primary difference between objects and interactions is *persistence*. Objects persist, interactions do not.

Would a missile be described by an object class or an interaction class? The answer depends on the simulation and the persistence of missile instances. A simulation that focuses on missile launchers and their targets may perceive missiles (or missile launches) as events. The launcher fires a missile, which effects some damage. The time that the missile is in the air may be trivial with respect to the simulation. Here, the missile could be modeled as an interaction – possibly between the launcher and the target. Another simulation may focus on the in-flight characteristics of missiles. The fact that the missile launches or impacts may be incidental. Here, the missile persists and should be modeled as an object.

7.3 Object Hierarchies

Figure 7-2 illustrates a class hierarchy and accompanying Venn diagram. Object classes and interaction classes can be constructed hierarchically. For example, assume that objects of type W are comprised of the attributes "a," "b," "c," and "d" – abbreviated "{a, b, c, d}." It is possible to define object classes that extend object class W. Object class W is extended to produce the object classes X and Z. Object class X is further extended to produce the object class Y.

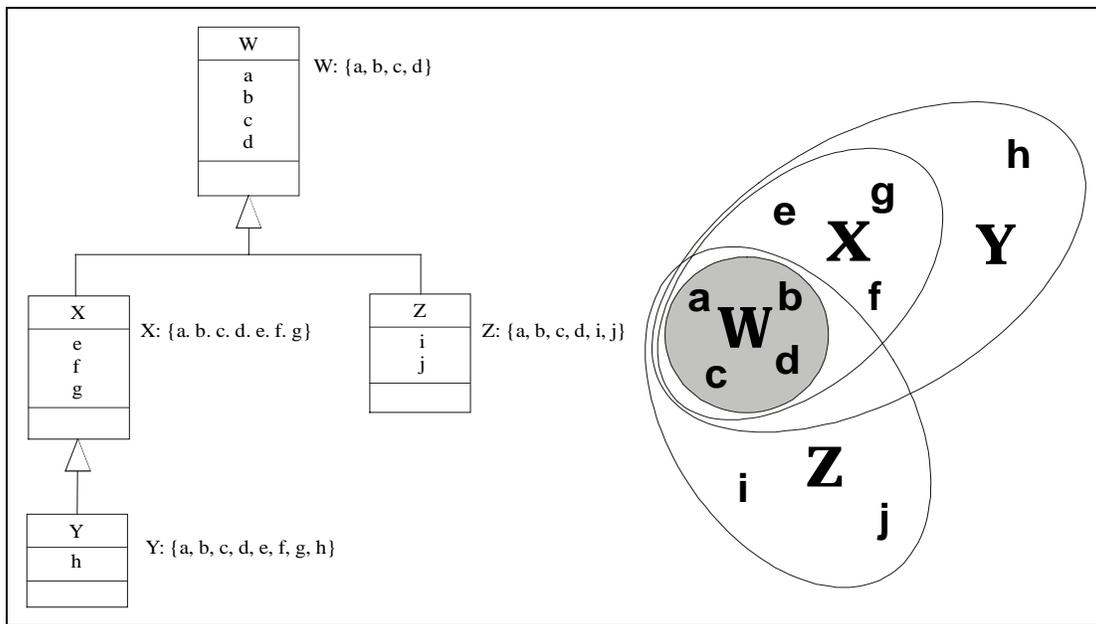


Figure 7-2. Class Hierarchy – Venn Diagram

Object-oriented programming enthusiasts will recognize such hierarchical representations.¹⁶ Various communities use different phrases to describe object hierarchies. Some examples include:

<i>X extends W.</i>	<i>Y inherits from X.</i>
<i>W is a base type.</i>	<i>W is an ancestor of Z.</i>
<i>X is derived from W.</i>	<i>X is a child of W.</i>
<i>Y is a descendant of W.</i>	<i>Y and Z are leaf objects.</i>
<i>W is the parent of Z.</i>	<i>W is the superclass of X.</i>
<i>X is a subclass of W.</i>	

The basic idea is that when an object class is extended to produce a new object class, the new object class contains all the attributes of the class being extended and possibly more. The object diagram and Venn diagram (Figure 7-2) illustrate the relationship between the object classes W, X, Y, and Z. Object class W has the four attributes {a, b, c, d}, class X adds the attributes {e, f, g} so instances of class X have attributes {a, b, c, d, e, f, g}.

7.4 Publishing and Subscribing Objects

Each federate must *publish* the object classes and interaction classes it plans to produce. It is possible for a federate to publish a subset of the available attributes for a given class.

¹⁶ Developers with a strong object-orientation should note that HLA "objects" are defined primarily by their constituent data elements rather than on behavior. In this way, HLA "objects" have more in common with relational models than object-oriented models.

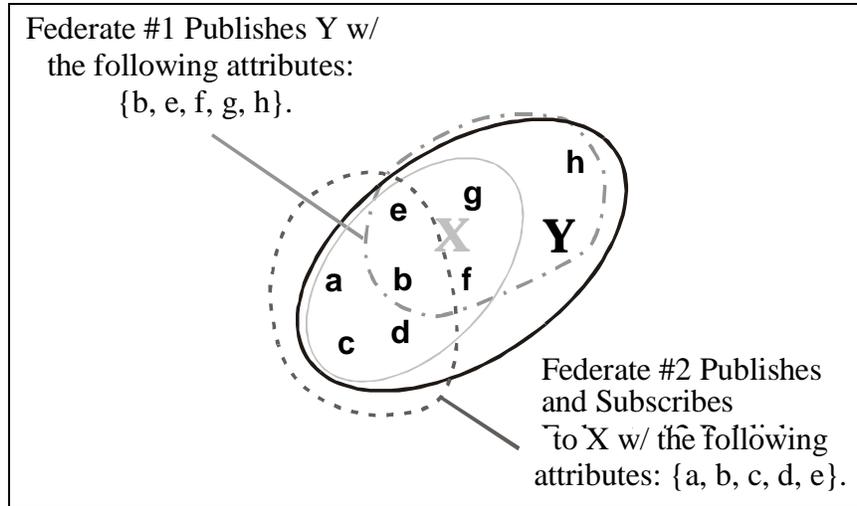


Figure 7-3. Object Publishing

7.4.1. Object Publication

In Figure 7-3, the object class Y contains the attributes {a, b, c, d, e, f, g, h}. A federate can create instances of object class Y, without specifying all of the attributes associated with Y. For example, Y might be a particular kind of aircraft. A given federate may know some information about aircraft instances (e.g., position information), but relies on other federates to "fill in" the missing pieces (e.g., intelligence about the aircraft). In such a case, the federate would indicate that it could publish particular attributes associated with Y. Here, Federate #1 indicates that it can publish attributes {b, e, f, g, and h} and Federate #2 indicates that it can publish attributes {a, b, c, d, and e} for object class Y.

Each federate must indicate explicitly which attributes it can produce (i.e., introduce or update) on a per class basis. Multiple federates may be able to publish Y instances. Federate #3 might publish all of the attributes associated with object class Y. Another federate, Federate # 7, may be able to publish attributes {a, c, f} for Y.

An implicit attribute, known by the name "privilegeToDelete," is included whenever a publication capability is registered for an object class. Only the federate that created a particular object instance is allowed to delete the instance unless the privilege to delete is conveyed to another Federate. Chapter 9, *Ownership Management*, explores the ability to exchange attribute update and object deletion responsibility among federates.

A federate must explicitly state every object class it intends to produce via the RTIambassador's publishObjectClass() method. A separate call to publishObjectClass() is required for every object class including objects that appear in class hierarchies. If Federate #9 wishes to produce instances of object classes W, X, Y, and Z, it must say so explicitly using four publication calls (one per object class).

7.4.2. Interaction Publication

As with object classes, each federate must state explicitly which interaction classes it intends to produce using the `publishInteractionClass()` method. Interactions are produced as "all or nothing." It is not possible to specify which parameters in an interaction will be published. If a federate indicates that it intends to publish an interaction, it must be capable of specifying all parameters associated with the interaction.

7.4.3. Object Subscription

Federates indicate their interest in certain object classes via the `RTIambassador` method `subscribeObjectClassAttributes()`. Object subscription differs from object publication. When a federate subscribes to an object class, it is expressing an interest in learning about all object instances of the class. For example, a federate subscribing to object class X (as shown in Figure 7-2) will discover all instances of class X produced by other federates in the federation. Additionally, a federate subscribing to X will discover all instances of class Y (produced by other federates) as though they were instances of class X. This is an example of *type promotion*.

Whenever a federate expresses a subscription interest in a particular object class, the RTI presumes that the federate is interested in instances of the descendant classes as well. A federate subscribing to class W would see external instances of classes X, Y, and Z as instances of class W. This can be a useful tool. Class W might represent all aircraft. Class X might represent military aircraft, while class Y represents commercial aircraft. A federate may wish to know about all aircraft, but not care about the details – including the military v. commercial designation.

A federate is informed about a new object instance if (a) the federate has subscribed to the object class of the instance or (b) the instance can be promoted (i.e., up the hierarchy) to a subscribed object class. When an object is promoted, attributes particular to the original class are dropped. An instance of object class Y has attributes {a, b, c, d, e, f, g, and h}.¹⁷ A federate subscribing to object class X can discover the Y instance as an X. Since attribute "h" is not present in instances of class X, that information is lost.

A federate can subscribe to multiple classes in a class hierarchy. If a federate subscribed to class W and X, the following would be true:

- Instances of object class W would be seen without promotion.
- Instances of object class X would be seen without promotion.
- Instances of object class Y would be seen as instances of object class X.
- Instances of object class Z would be seen as instances of object class W.

¹⁷ Some attributes may not have assigned values. It depends on what the originating federate has published for this object and the extent to which other federates have contributed to what's known about the instance.

When a federate discovers an object, it learns the object class of the instance. If the federate discovers the object instance to be of object class X, it will always believe the object's type to be X. If a federate subscribes to class X and not to class Y, it will discover Y instances as X instances. If the federate subsequently subscribes to class Y, object instances previously discovered as X instances (via promotion) will continue to be seen as X instances. Subsequently discovered instances of object class Y will be discovered as instances of object class Y.¹⁸

7.4.4. Interaction Subscription

As with object classes, each federate subscribes to the interaction classes it wishes to receive. It is not possible to subscribe to individual parameters of an interaction class. Again, interactions are "all or nothing." As with object classes, a federate is informed about a new interaction if (a) the federate has subscribed to the interaction class of the interaction or (b) the interaction can be promoted (i.e., up the hierarchy) to a subscribed interaction class. When an interaction instance is promoted, only the parameters of the subscribed class are presented to the receiving federate.

7.4.5. Control Signals

In Figure 7-3, above, Federate #1 indicated that it was capable of producing Y instances, but could only provide the attributes {b, e, f, g, and h}. In that same figure, Federate #2 subscribes to attributes {a, b, c, d, and e} for object class X. The Y instances produced by Federate #1 are discovered as X instances by Federate #2.

Federate #2 is only interested in a few of the Y attributes produced by Federate #1. As discussed previously, Federate #2 cannot access attribute "h" since the attribute is not a part of class X. Further, Federate #2 has no interest in attributes {f, g}. Of the information Federate #1 is able to produce Y:{b, e, f, g, h}, only the information Y:{b, e} is required – assuming Federate #2 is the only other federate in the federation.

The RTI issues control signals to indicate the information Federate #1 should produce. By default, a federate should refrain from producing object updates unless the Local RTI Component (LRC) has indicated that a consumer exists. If Federate #1 is first on the scene (i.e., there are no consumers), it will never be signaled to begin registering Y instance information.

Once Federate #2 arrives, the LRC will indicate to Federate #1 that it should register any instances of object class Y with the federation execution and it should start providing updates for Y:{b, e}. If Federate #2 goes away, Federate #1 will be told to stop registering instances of object class Y and to stop providing updates for Y:{b, e}.

Each LRC informs its federate (via callbacks) which object attributes and which interactions to start or stop producing based on consumer demand. Each federate's Simulation Object Model

¹⁸ Rediscovery of an object instance can be forced using the RTIambassador::localDeleteObjectInstance() service. After object class Y was subscribed to, a federate could "locally delete" all instances of object class X to rediscover the objects based on the federate's new subscriptions.

(SOM) will identify the extent to which the federate does or does not make use of the control signals provided by the LRC.

7.5 Object Publication and Subscription

Each federate is responsible for identifying its publication and subscription interests to the RTI LRC using the RTIambassador methods `subscribeObjectClassAttributes()` and `publishObjectClass()`. The interaction diagram shown in Figure 7-4, Object Publication and Subscription illustrates the procedure for building the information required to use these methods.

The `publish` and `subscribe` methods both require an `RTI::ObjectClassHandle` and an `RTI::AttributeHandleSet`. The LRC has an internal representation for object classes, object class attributes, interaction classes and interaction class parameter string representations that appear in the FED file. RTIambassador methods like `getObjectClassHandle()` and `getAttributeHandle()` translate character descriptions into LRC *handles*.

The (abstract) class `RTI::AttributeHandleSet` identifies a set of attributes – e.g., {a, b, c, d}. To express interest in publishing or subscribing to an object class, the following steps are required.

For each object class to be published:

- a) Obtain the handle for the current object class.
- b) Create a free-store allocated `AttributeHandleSet` using the static `create()` method in the class `AttributeHandleSetFactory`.
- c) For each attribute the federate can publish:
 - i) Obtain the handle for the current attribute.
 - ii) Add the handle to the `AttributeHandleSet`
- d) Publish/Subscribe the `AttributeHandleSet` for the object class.
- e) Empty and delete the set if no longer needed.

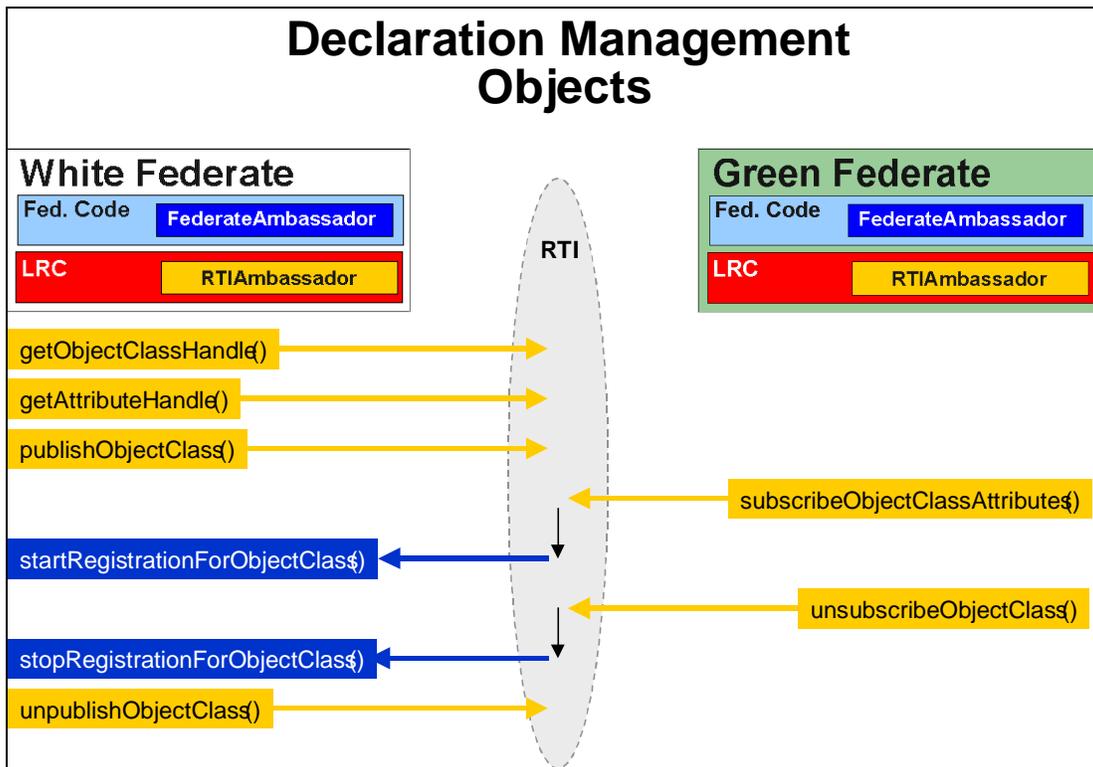


Figure 7-4. Object Publication and Subscription

7.6 Throttling Publications

The LRC signals a federate (via callbacks, as shown in Figure 7-4) to start or stop registering object instances for all published object classes and generating interactions for all published interaction classes.

7.7 FoodFight Object Declaration

The following code excerpts demonstrate publication and subscription to the object class "Student" which has attributes "LunchMoney," "Cleanliness," and "AmmoAmount." The class

name and attribute names would appear in the FED file.¹⁹ The student object class also has the hidden attribute "privilegeToDelete."²⁰

7.7.1. Excerpt from Student.h

The following excerpt is taken from the declaration of the C++ class "Student." In this example, a C++ class is used to realize the HLA object class "Student." This is but one possible way of realizing an HLA object class. Information about students could be maintained in a database or a C structure.

Student.h

```

327     :
328 class Student
329 {
330     friend ostream& operator<< (ostream&, const Student&);
331
332 public:
333     // In the following enumeration, ATTRIBUTE_COUNT denotes the end of the
334     // enumeration and is equal to the total number of attributes.
335     enum AttributeNames { PRIVILEGE_TO_DELETE = 0, LUNCH_MONEY,
336         CLEANLINESS, AMMO_AMOUNT, ATTRIBUTE_COUNT };
337
338     static const char* attribute_names[];
339
340     // Variables to store class handles.
341     static RTI::ObjectClassHandle class_handle;
342
343     // Array to store attribute/parameter handles.
344     static RTI::AttributeHandle attributes[];
345
346     static RTI::AttributeHandleSet* p_all_attribute_vector;
347
348     static void RegisterObject ();
349
350     :
351     :
352
353     // Student Characteristics
354     RTI::ObjectHandle getId () const { return id_self; }
355     double getLunchMoney () const { return lunch_money; }
356     double getCleanliness () const { return cleanliness; }
357     long getAmmoAmount () const { return ammo_amount; }
358
359 protected:
360     static const int BUY_AMMO_CHANCE;
361     static const double AMMO_COST_MEAN;

```

¹⁹ As of the RTI 1.3-NG release, names appearing in the FED file are case insensitive, so class "Student" could be specified "student," "STUDENT," or "StUdEnT." In general, the case of the names in the Federation Object Model (FOM), Federation Execution Data (FED), and federate source code should be considered as case-sensitive to ensure interoperability with all RTIs.

²⁰ The privilegeToDelete attribute does not have to appear explicitly in the attribute publication list. It is included in the Federation Execution Data (FED) file as the sole attribute of the objectRoot base class that all federation defined objects will extend.

```

362     static const double AMMO_COST_ADJ;
363     static const int ATTEMPT_LAUNCH_CHANCE;
364
365     RTI::ObjectHandle id_self; // RTI ID by which this student is known.
366     double lunch_money; // Funds for new ammo.
367     double cleanliness; // Food damage the student has sustained.
368     unsigned long ammo_amount; // Launchable food possessed.
369
370     :
371 };

```

The static function `Student::RegisterObject()` will eventually contain the code that registers this federate's publication and subscription requests with regard to the object class `Student`. To support this static registration process, several static variables are declared. The `Student::AttributeNames` enumeration provides an identifier for each attribute in the HLA object class `Student`. The static handle `Student::class_handle` will contain the LRC's internal representation of the `Student` object class (type). The static array `Student::attribute_names` will hold the string representation of each attribute in `Student`. The static array `Student::attributes` will house the LRC's internal handle for each `Student` attribute. Eventually, the pointer `Student::p_all_attribute_vector` will identify a free-store allocated attribute handle set containing all the attribute handles we wish to publish and subscribe.²¹

Student.cxx

Definitions for `Student` static variables are as follows.

```

372     :
373     const char* Student::attribute_names[Student::ATTRIBUTE_COUNT] = {
374         "privilegeToDelete", "LunchMoney", "Cleanliness", "AmmoAmount" };
375
376     RTI::ObjectClassHandle Student::class_handle
377     = RTI::ObjectClassHandle(); // Default constructor provides null handle.
378
379     RTI::AttributeHandle Student::attributes[Student::ATTRIBUTE_COUNT];
380
381     RTI::AttributeHandleSet* Student::p_all_attribute_vector = 0;
382     :

```

The publish and subscribe registration process is conducted by the static function `Student::RegisterObject()`.

```

383     void
384     Student::RegisterObject ()
385     {
386         // Abstract
387         // Register the class. Then, register each attribute recording the

```

²¹ In this particular example, the federate wishes to publish and subscribe all attributes of `Student`. In other applications a class might be published only, subscribed only, or the set of published attributes may be entirely different from the set of subscribed attributes.

```

388 //      corresponding handle in the vector of all attributes (e.g.,
389 //      p_all_attribute_vector).
390
391 const char* object_name = "Student";
392 cout << "Registering '" << object_name << "'." << endl;
393
394 class_handle = ::rtiAmb.getObjectClassHandle(object_name);
395
396 if (p_all_attribute_vector) delete p_all_attribute_vector;
397
398 p_all_attribute_vector
399     = RTI::AttributeHandleSetFactory::create(Student::ATTRIBUTE_COUNT);
400
401 for (int i = 0; i < ATTRIBUTE_COUNT; i++ )
402 {
403     cout << "\tGetting attribute handle for '" << attribute_names[i]
404         << "'." << endl;
405
406     RTI::AttributeHandle handle = ::rtiAmb.getAttributeHandle(
407         attribute_names[i],
408         class_handle);
409
410     // Here, we maintain an array of handles and an attribute vector
411     // that contains all the handles for this class.
412     attributes[i] = handle;
413     p_all_attribute_vector->add(handle);
414 }
415
416 // Publish and subscribe to all attributes.
417 ::rtiAmb.publishObjectClass(class_handle, *p_all_attribute_vector);
418 ::rtiAmb.subscribeObjectClassAttributes(
419     class_handle,
420     *p_all_attribute_vector);
421
422 // NB: For now, not bothering to delete p_all_attribute_vector. It will
423 //     live as long as the program does.
424 }

```

RegisterObject() closely follows the interactions identified in Figure 7.4, Object Publication and Subscription.

7.7.2. Dynamic Object Publication and Subscription

Each call to publishObjectClass() and subscribeObjectClassAttributes() for an object class replaces previous calls. The methods unpublishObjectClass() and unsubscribeObjectClass() should be called when a federate is no longer interested in any attributes of an object class.

7.8 Publishing and Subscribing Interactions

Registering publication and subscription interest in interaction classes is more straightforward than registering interest in object classes. Figure 7-5, Declaring Interactions, identifies RTIambassador declaration management methods. Unlike object registration, interactions do not have to be registered because they do not persist and you *cannot* specify interest in particular interaction parameters. Interactions are "all or nothing."

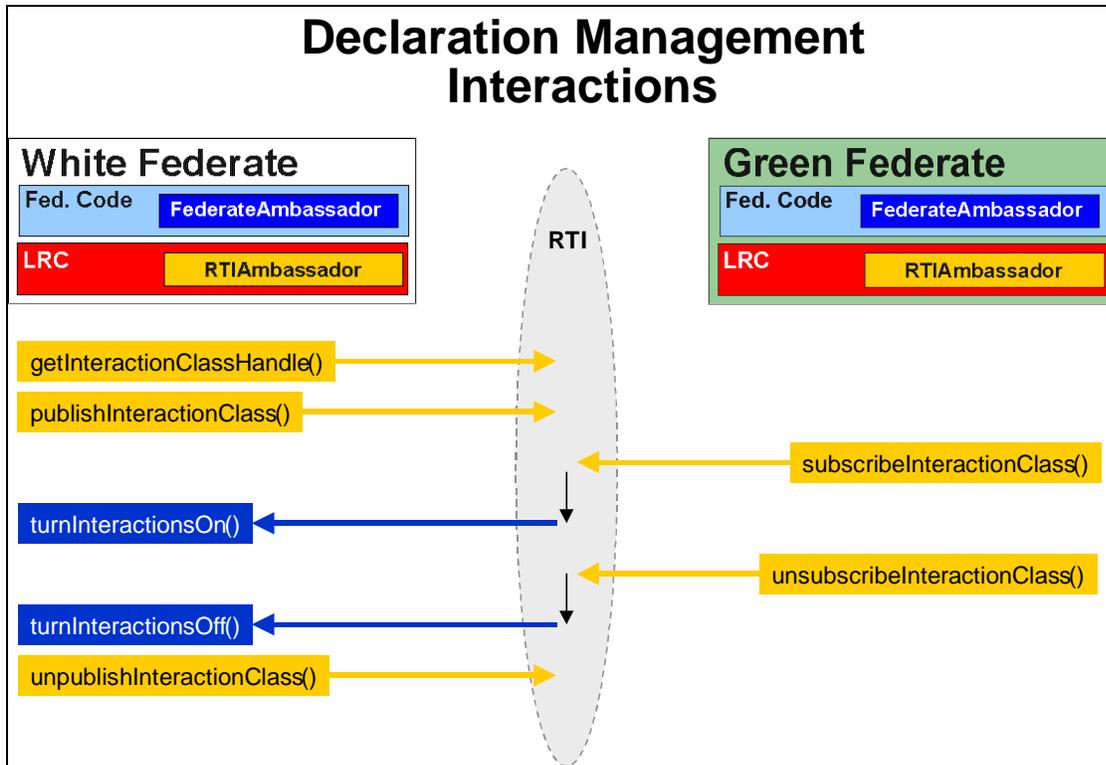


Figure 7-5. Declaring Interactions

Splat.h

```

425 class Splat
426 {
427     friend ostream& operator<< (ostream& os, const Splat& splat);
428
429 public:
430     enum ParameterNames { ENSUING_MESS, TARGET, PARAMETER_COUNT};
431
432     static const char* parameter_names[];
433
434     // Variables to store class handles.
435     static RTI::InteractionClassHandle interaction_handle;
436
437     // Array to store attribute/parameter handles.
438     static RTI::ParameterHandle parameters[];
439
440     static void RegisterInteraction ();
441     :

```

Splat.cxx

HLA RTI 1.3-Next Generation

```

442 const char* Splat::parameter_names[Splat::PARAMETER_COUNT] = {
443     "EnsnuingMess", "Target" };
444
445 RTI::InteractionClassHandle Splat::interaction_handle
446 = RTI::InteractionClassHandle(); // Null handle.
447
448 RTI::ParameterHandle Splat::parameters[Splat::PARAMETER_COUNT];
449     :
450     :
451 void
452 Splat::RegisterInteraction ()
453 {
454     const char* interaction_name = "Splat";
455     cout << "Registering '" << interaction_name << "'." << endl;
456
457     // Register the interaction.
458     interaction_handle = ::rtiAmb.getInteractionClassHandle(interaction_name);
459
460     for (int i = 0; i < PARAMETER_COUNT; i++ )
461     {
462         cout << "\tGetting parameter handle for '" << parameter_names[i]
463             << "'." << endl;
464
465         parameters[i] = ::rtiAmb.getParameterHandle(
466             parameter_names[i],
467             interaction_handle);
468     }
469
470     // Publish and subscribe interaction.
471     ::rtiAmb.publishInteractionClass(interaction_handle);
472     ::rtiAmb.subscribeInteractionClass(interaction_handle);
473 }

```

As with object class declaration, interaction interest can be declared dynamically. Each call to `publishInteractionClass()` and `subscribeInteractionClass()` for an interaction class replaces previous calls. The methods `unpublishInteractionClass()` and `unsubscribeInteractionClass()` should be called when a federate is no longer interested in an interaction class.

8. Object Management

This chapter introduces the RTIambassador service and FederateAmbassador callback methods that support object management. Object management includes instance *registration* and instance *updates* on the object production side and instance *discovery* and *reflection* on the object consumer side. Object management also includes methods associated with sending and receiving interactions, controlling instance updates based on consumer demand, and other miscellaneous support functions.

8.1 Registering, Discovering, and Deleting Object Instances

Figure 8-1 illustrates the required interactions for object instance registration and discovery. The RTIambassador method `registerObjectInstance()` informs the Local RTI Component (LRC) that a new object instance has come into existence. The method requires the object class of the new object instance and an optional object name. The method returns an `RTI::ObjectHandle` which the LRC uses to identify the particular object instance.

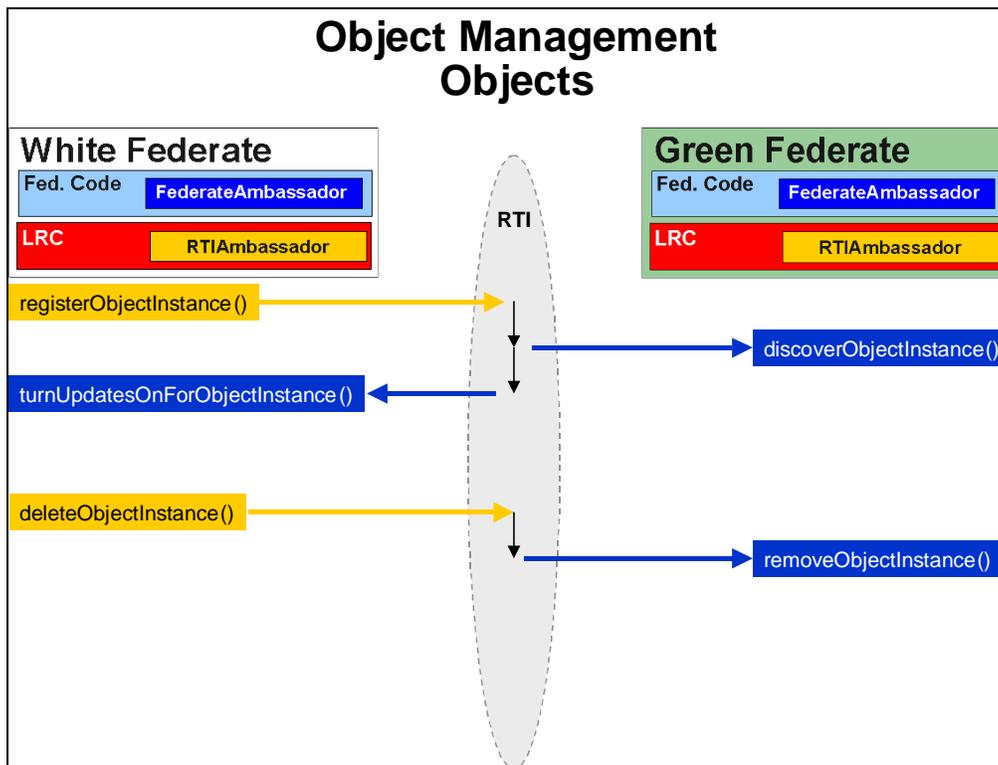


Figure 8-1. Object Management Methodology

Registration introduces an object instance to the federation. However, it does not provide attribute values for the instance. That requires a second step.

Each and every object can be deleted by exactly one federate. Initially, the federate that creates (registers) an object has the privilege to delete the object.²² In Figure 8-1, the RTIambassador method `deleteObjectInstance()` is called to remove a registered object. The FederateAmbassador `removeObjectInstance()` callback informs federates that a previously discovered object no longer exists. The RTIambassador method `localDeleteObjectInstance()` effectively "undiscovers" an object instance. This method does not ensure the object will be permanently undiscovered. This service is intended to be used when a federate discovers an object as an instance of an object class but would like to subscribe to object classes that extend the discovered class and then rediscover the instance based on the new subscriptions. The object instance will be rediscovered upon the next `updateAttributeValues()` invocation that meets the receiving federate's subscriptions.

8.2 Updating and Reflecting Object Attributes

To update one or more attributes associated with a registered object instance, a federate must prepare an `RTI::AttributeHandleValuePairSet`. This set is similar to the `RTI::AttributeHandleSet` discussed in Chapter 7, *Declaration Management*. An `AttributeHandleSet`, abbreviated AHS, identifies a set of attributes. An `AttributeHandleValuePairSet`, AHVPS, identifies a set of attributes and their values. The static function `RTI::AttributeSetFactory::create()` is used to construct a free-store allocated AHVPS instance.²³ In Chapter 7, *Declaration Management*, the notation {a, b, c, and d} was used to identify four attributes by name. The notation can be extended to accommodate attribute values – e.g., {a = 5, b = "Hello", c = 14.79821, d = -12}.

Attribute updates are provided for an object instance via the RTIambassador method `updateAttributeValues()`. The method requires an `ObjectHandle`, which the LRC uses to identify an object instance, an AHVPS and a descriptive character string (tag). An optional `FedTime` argument will have meaning if the federate is "regulating," and one or more contained attributes are TSO (see Chapter 3, *The Role of Time*, and Chapter 6, *Time Management*).

Figure 8-1 (previously introduced) and Figure 8-2, Object Management Updates, illustrates the interactions required to discover and to reflect updates for external object instances. Discovery is the counterpart to registration. Reflection is the counterpart to attribute updates. The FederateAmbassador callback method `discoverObjectInstance()` informs the federate that a new object instance has come into existence. The method provides an object handle, which will be used to identify the object for subsequent updates. The method also identifies the object class of the new object instance. It is important to note that the `ObjectHandle` is a global representation maintained by the LRC. The same object instance is known to all federates by its globally unique handle value.

²² Chapter 9, *Ownership Management*, explores functions for giving away the privilege to delete as well as the right to update various attributes.

²³ AHVPS is actually an abstract class; so, the factory function produces an AHVPS descendant (implementation).

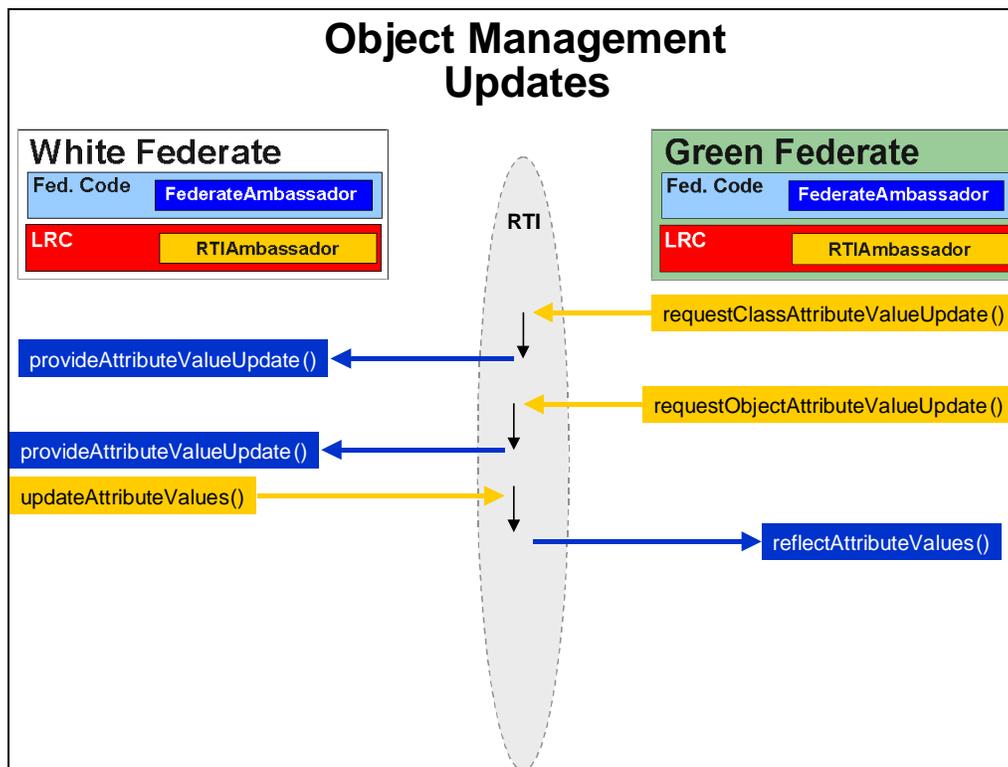


Figure 8-2. Object Management Updates

8.3 Encoding and Object Update

When producing an AHVPS, the federate is responsible for any data marshaling (encoding). The LRC knows nothing about data content. It knows the names of object classes, the names of attributes and the handle representations for object classes and attributes. The following code demonstrates how an AHVPS is produced for the Student class introduced in previous chapters. Data is encoded and the length of the encoding is communicated to the LRC.²⁴ Ultimately, the AHVPS is bound to an object instance handle in an `updateAttributeValues()` invocation.

In the example, each instance of the C++ class "Student" has an AHS named "require_update_vector". As the state of Student instances change, affected attributes are added to this update vector. The AHVPS is formed by iterating through the AHS update vector and building handle-value pairs.

²⁴ The AHVPS actually consist of triples, not pairs. The triple is (1) the attribute handle, (2) the corresponding value and (3) the length of the encoding.

In the following example, the macro REINTERPRET_CAST(TYPE, EXPR) should be defined as "reinterpret_cast<TYPE> (EXPR)" on platforms that support ANSI-style casts. With dated compilers, a traditional cast might be used instead – "(TYPE) (EXPR)".

```

474 RTI::AttributeHandleValuePairSet*
475 Student::getUpdatedValues ()
476 {
477     // Abstract
478     //     Get AHVPS containing entries for every handle in the update
479     //     vector (only!).
480
481     cout << id_self << " identifying updates." << endl;
482
483     RTI::AttributeHandleValuePairSet* p_set = 0;
484
485     if (require_update_vector.size()) // Is there work to do?
486     {
487         p_set = RTI::AttributeSetFactory::create(
488             require_update_vector.size());
489
490         for (unsigned long i = 0; i < require_update_vector.size(); ++i)
491         {
492             RTI::AttributeHandle handle
493                 = require_update_vector.getHandle(i);
494
495             if (handle == Student::attributes[LUNCH_MONEY])
496             {
497                 p_set->add(
498                     handle,
499                     REINTERPRET_CAST(const char*, &lunch_money),
500                     sizeof(double));
501             }
502             else if (handle == Student::attributes[CLEANLINESS])
503             {
504                 p_set->add(
505                     handle,
506                     REINTERPRET_CAST(const char*, &cleanliness),
507                     sizeof(double));
508             }
509             else if (handle == Student::attributes[AMMO_AMOUNT])
510             {
511                 p_set->add(
512                     handle,
513                     REINTERPRET_CAST(const char*, &ammo_amount),
514                     sizeof(unsigned long));
515             }
516             else if (handle == Student::attributes[PRIVILEGE_TO_DELETE])
517             {
518                 // Nothing to do. No reason to pass this (and it probably
519                 // shouldn't occur.
520             }
521             else
522             {
523                 const char* p_msg = "Student::getUpdatedValues() saw "
524                     "unrecognized handle.";
525                 cout << p_msg << endl;
526                 throw RTI::AttributeNotKnown(p_msg);
527             }
528         }

```

```

529
530     require_update_vector.empty();
531     }
532
533     return p_set;
534 }

```

The example demonstrates the cascading "if" statements required to identify an arbitrary attribute handle. It is tempting to try a switch() statement, but the LRC attribute values are not known prior – so, constant-based switching is ruled out. Clearly, the approach used in this code wouldn't scale for objects with large numbers of attributes. For such cases, it may be preferable to (a) use a map (i.e., a hashing dictionary) to store attributes or (b) use fewer, more complex attributes.

Attributes can be arbitrarily complex as long as they are documented properly in the FOM and SOM.²⁵ However, complicated attributes may be less reusable. It is a good idea to collect only those things that genuinely belong together both in terms of the information and the update frequency. For example, {longitude, latitude, altitude} might be combined into the single attribute {position}. But, combining {name, grossWeight, fuelSupply} into a single attribute would be a poor combination since the attribute "name" is likely to be updated at a very different rate than "grossWeight" and these attributes may not belong together.

The demonstration code above does not take any steps to ensure that data is encoded in a platform-independent way. This encoding strategy would not survive a federation with big endian and little endian federates.

8.4 Decoding and Object Reflection

The FederateAmbassador callback method reflectAttributeValues() provides an AHVPS. The following function decodes the AHVPS in a manner consistent with the encoding strategy.

```

535 void
536 Student::reflectExternalChanges (const RTI::AttributeHandleValuePairSet& set)
537 {
538     // Design Notes
539     //     Values are bit copied; so, examples will not work across
540     //     big/little endian boundaries.
541
542     cout << id_self << " incorporating reflected changes." << endl;
543
544     char buffer[MAX_BYTES_PER_VALUE];
545     unsigned long length;
546
547     // Iterate through the set, modifying corresponding attributes.
548     for (unsigned long i = 0; i < set.size(); ++i)
549     {
550         RTI::AttributeHandle handle = set.getHandle(i);
551
552         if (handle == Student::attributes[LUNCH_MONEY])

```

²⁵ An existing hardware component utilizing a complex data structure, might provide updates for the whole structure rather than structure components. In such cases, the whole structure might be combined to form a single attribute.

```

553     {
554         set.getValue(i, buffer, length);
555         lunch_money = *REINTERPRET_CAST(double*, buffer);
556     }
557     else if (handle == Student::attributes[CLEANLINESS])
558     {
559         set.getValue(i, buffer, length);
560         cleanliness = *REINTERPRET_CAST(double*, buffer);
561     }
562     else if (handle == Student::attributes[AMMO_AMOUNT])
563     {
564         set.getValue(i, buffer, length);
565         ammo_amount = *REINTERPRET_CAST(unsigned long*, buffer);
566     }
567     else if (handle == Student::attributes[PRIVILEGE_TO_DELETE])
568     {
569         // Nothing to do. Should not really be included in any update.
570     }
571     else
572     {
573         const char* p_msg = "Student::reflectExternalChanges() saw "
574             "unrecognized handle.";
575         cout << p_msg << endl;
576         throw RTI::AttributeNotKnown(p_msg);
577     }
578 }
579 }

```

8.5 Exchanging Interactions

Interactions are constructed in a similar fashion to the way attribute updates are constructed. Recall that objects persist, interactions do not. Each interaction is constructed, sent, and forgotten.²⁶ Interaction recipients receive, decode, and apply the interaction. Figure 8-3, Exchanging Interactions, lists the classes and methods involved in generating and consuming interactions. RTIambassador methods are discussed in Appendix A, *RTI::RTIambassador*, FederateAmbassador methods in Appendix B, *RTI::FederateAmbassador*, and the supporting types (e.g., ParameterHandleValuePairSet and ParameterSetFactory) in Appendix C, *Supporting Types and Classes*.

²⁶ Interactions can be retracted. See the manual pages for details.

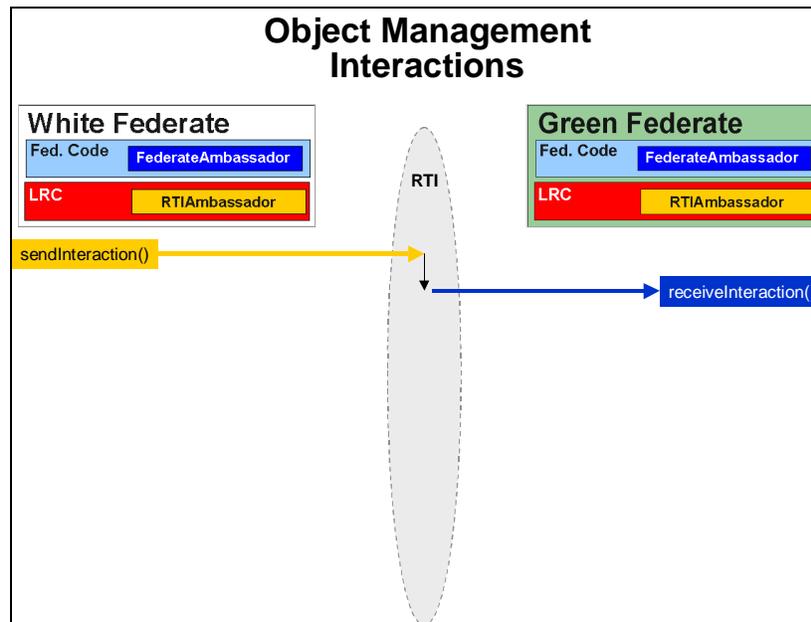


Figure 8-3. Exchanging Interactions

8.6 Additional Object Control

Object attribute updates and interactions are conveyed between federates using one of two data transportation schemes – "reliable" and "best effort". For objects, the transportation scheme is specified at the level of individual attributes. For interactions, the transportation scheme is specified at the interaction level (i.e., not the parameter level). By default, the transportation scheme is specified per object/attribute name and per interaction name in the Federation Execution Data (FED) file.

It is possible to change the transportation scheme dynamically for one or more attributes of a specific object instance using the RTIAmbassador method `changeAttributeTransportType()`. It is possible to change the transportation scheme dynamically for interactions by class name using the RTIAmbassador method `changeInteractionTransportType()`. Figure 8-4 illustrates these functions.

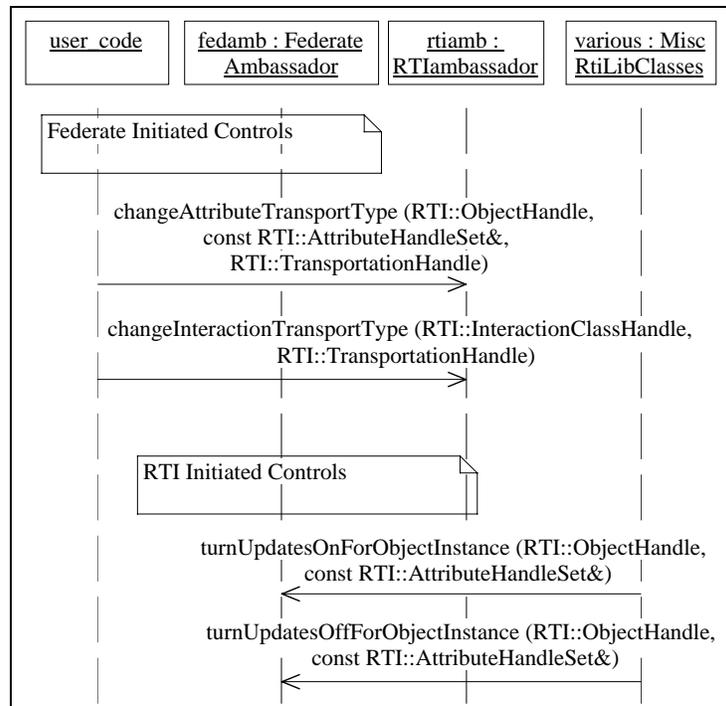


Figure 8-4. Additional Object Control

Figure 8-4 also shows two callback methods – `turnUpdatesOnForObjectInstance()` and `turnUpdatesOffForObjectInstance()`. These methods are used to inform a federate whether or not there is external interest in updates for specific attributes of specific object instances.²⁷

8.6.1. Attribute Management

A particular federate may have created and registered a particular F-15 fighter. If one or more federates are subscribed to overlapping attributes of this published object class, the LRC would issue the `turnUpdatesOnForObjectInstance()` callback to specify the particular attributes for which updates should be generated. If at some future point, there are no subscribed federates to the F-15 object class, the LRC would invoke `turnUpdatesOffForObjectInstance()` – informing the federate to cease updates for this particular object instance.

The federate should presume that there is no external interest (one or more subscribed federates) in an object unless or until `turnUpdatesOnForObjectInstance()` is issued. Calls to `turnUpdatesOnForObjectInstance()` and `turnUpdatesOffForObjectInstance()` are cumulative. Each call to `turnUpdatesOnForObjectInstance()` adds to the set of attributes that should be updated. Each call to `turnUpdatesOffForObjectInstance()` removes attributes from the set of attributes that should be updated.

²⁷ These functions are companions to the declaration management callback methods `startRegistrationForObjectClass()` and `stopRegistrationForObjectClass()` (see Chapter 7, *Declaration Management*).

8.6.2. Enable/Disable Attribute Management

It is possible to disable the `turnUpdatesOnForObjectInstance()` and `turnUpdatesOffForObjectInstance()` callbacks. . Two `RTIambassador` methods can be used to specify whether per object instance control signals are generated or suppressed. These methods are (1) `enableAttributeRelevanceAdvisorySwitch()` and (2) `disableAttributeRelevanceAdvisorySwitch()`.²⁸

Attribute Scopes

Prior to communicating attribute updates for a subscription with region to a particular object class, the LRC will (at the federate's discretion) provide the preliminary callback `attributesInScope()` announcing that subsequent attribute updates for the specified object instance with overlapping attributes may be forthcoming. A subsequent `attributesOutOfScope()` callback would inform the federate that subsequent attribute updates for the specified object and specified attribute set would no longer be provided. These signals will be generated or suppressed based on the "attribute scope advisory switch" that is set by the `RTIambassador` methods `enableAttributeScopeAdvisorySwitch()` and `disableAttributeScopeAdvisorySwitch()`. Figure 8-5 provides an interaction diagram for these methods.

²⁸ These methods are not shown in the accompanying interaction diagram.

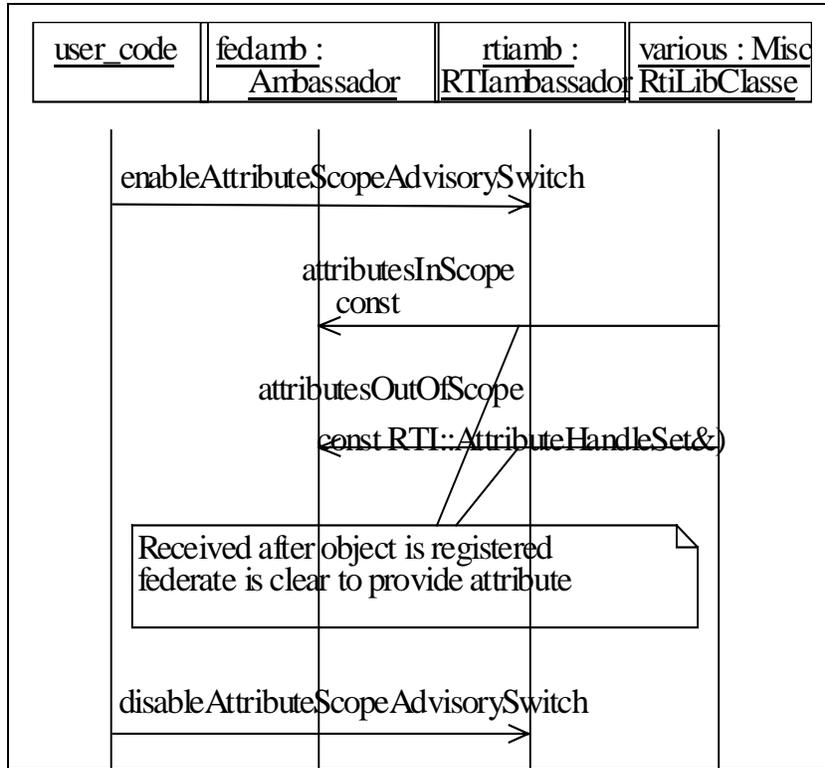


Figure 8-5. Scope Interactions

9. Ownership Management

9.1 Introduction

This chapter introduces the RTIambassador service and FederateAmbassador callback methods that support ownership management. Chapter 7, *Declaration Management*, presented declaration management methods supporting publication and subscription of objects and interactions. Chapter 8, *Object Management*, explored methods for registering and updating object instances.

The RTI allows federates to share the responsibility for updating and deleting object instances with a few restrictions. It is possible for an object instance to be wholly owned by a single federate. In such a case, the owning federate has responsibility for updating all attributes associated with the object and for deleting the object instance. It is possible for two or more federates to share *update responsibility* for a single object instance. When update responsibility for an object is shared, each of the participating federates has responsibility for a mutually exclusive set of object attributes.²⁹ Only one federate can have update responsibility for an individual attribute of an individual object at any given time. In addition, only one federate has the *privilege to delete* an object instance at any given time.

9.1.1. Push v. Pull

The ownership management methods provide a facility for exchanging attribute ownership among federates in a federation execution using a "push" and/or a "pull" model.³⁰ A federate can try to give away responsibility for one or more attributes of an object instance – or *push* ownership. Alternatively, a federate can try to acquire responsibility for one or more attributes of an object instance – or *pull* ownership. The push model cannot thrust ownership onto an unwitting federate. Similarly, the pull model cannot usurp ownership.

9.1.2. Privilege to Delete

The special attribute "privilegeToDelete" exists in all object instances (by default). The federate that "owns" this attribute for an object instance has the right to delete the object. Federates can exchange the "privilegeToDelete" attribute as they would any other attribute.

²⁹ For a given object instance, some attributes may be unowned – i.e., no federate has update responsibility.

³⁰ Push and pull models can be used in the same federation execution.

9.2 Ownership Pull

In Chapter 7, *Declaration Management*, Figure 7-2 introduced four object classes in a small hierarchy – including the object class Y with attributes {a, b, c, d, e, f, g, h, ~}. The “privilegeToDelete” attribute is shown graphically with a tilde. As mentioned above, multiple federates may share update responsibility for a given object instance. In Figure 9-1, Federate #1 declares that it can publish attributes {b, e, f, g, h, ~}. Federate #4 declares that it can publish attributes {a, c, d, ~}. Each federate implicitly publishes the “privilegeToDelete” attribute.³¹

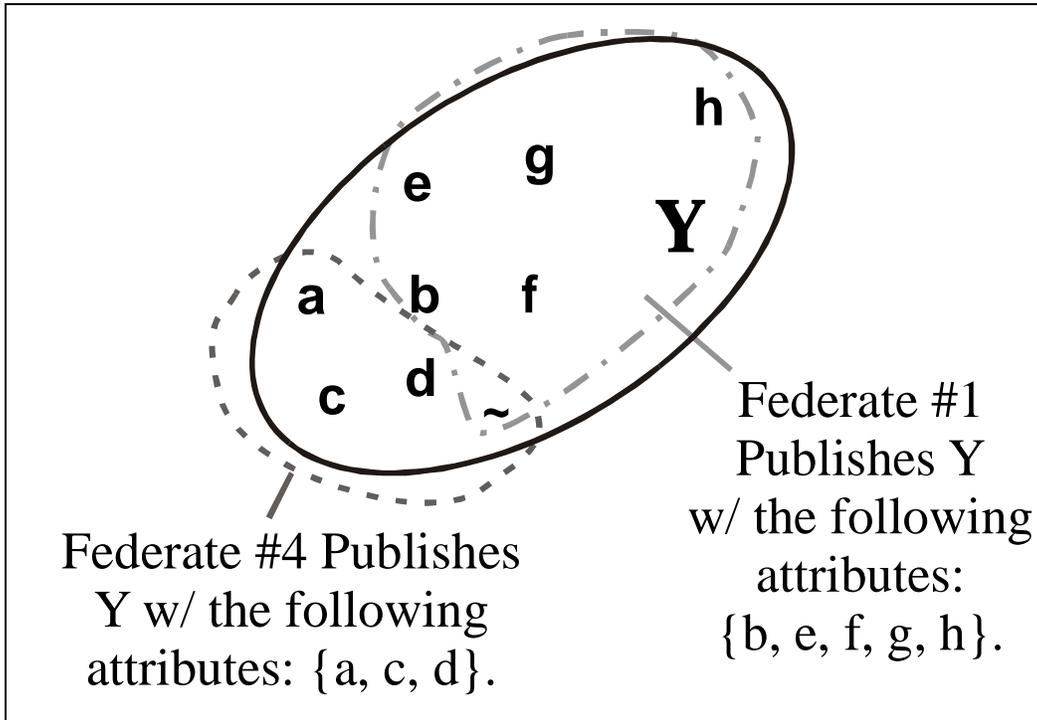


Figure 9-1. Shared Update Responsibility

In this particular example, there is no contention for attribute ownership since the two federates are interested in mutually exclusive attributes. However, only one federate can create a particular object instance. If Federate #1 creates an instance of Y named "Y_{alpha}," then it will "own" Y_{alpha}{b, e, f, g, h, ~} since it has published those attributes for object class Y. The attributes Y_{alpha}{a, c, d} are initially unowned.

If Federate #4 has subscribed to object class Y, it will discover Y_{alpha} as soon as it is registered by Federate #1.³² Federate #4 can attempt to acquire ownership (i.e., update responsibility) of any Y

³¹ When a federate indicates that it can publish an object class, the privilege to delete is assumed.

³² Federate #4 need not subscribe to the attributes produced by Federate #1 in order to discover Y_{alpha}.

attributes for Y_{α} . Figures 9-2 and 9-3 provide interaction diagrams that illustrate the pull ownership model for orphaned and obtrusive “pulls” respectively.

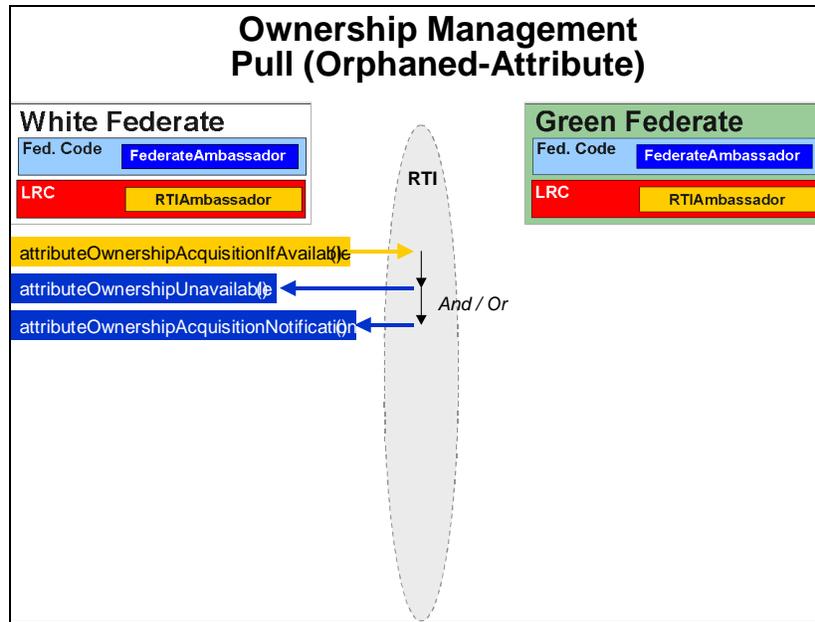


Figure 9-2. Ownership Pull Interaction Diagram – Orphaned Attribute

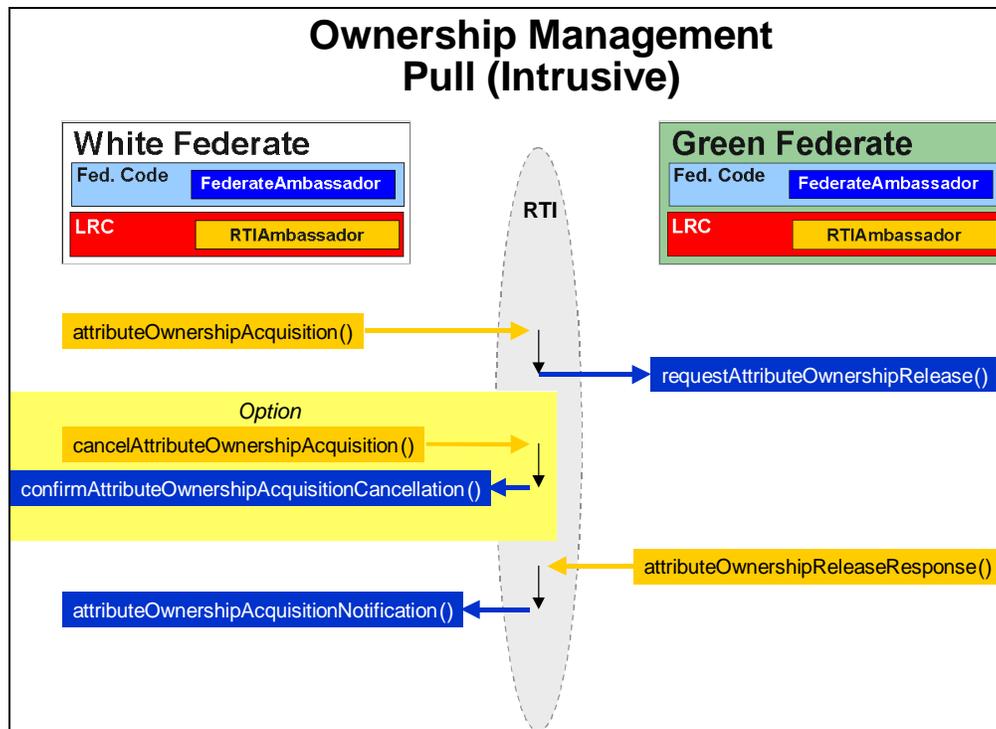


Figure 9-3. Ownership Pull Interaction Diagram – Intrusive

9.2.1. Attribute Ownership Acquisition

The RTIambassador method `attributeOwnershipAcquisition()` attempts to secure ownership of an attribute whether or not it is currently owned by another federate. As an alternative, the method `attributeOwnershipAcquisitionIfAvailable()` attempts to secure attributes that are not owned by another federate. A call to `attributeOwnershipAcquisition()` ultimately results in one or more `requestAttributeOwnershipRelease()` callback invocations if the requested attributes are owned by other federates. When `attributeOwnershipAcquisitionIfAvailable()` is called, any attributes that are already owned are reported via the `attributeOwnershipUnavailable()` callback. In order to request ownership of attributes for a particular object instance, the requesting federate must construct an attribute handle set. The procedure is outlined in Chapter 7, *Declaration Management*.

9.2.2. Attribute Ownership Release

As discussed in the previous paragraph, a call to `attributeOwnershipAcquisition()` will produce a `requestAttributeOwnershipRelease()` callback invocation on any federate that holds a requested attribute. A federate fielding this callback responds with the RTIambassador method `attributeOwnershipReleaseResponse()`. At a minimum, the federate should respond with a null attribute handle set – indicating that the attributes cannot or will not be released. The federate is free to give up none, some, or all of the requested attributes. The federate is released from

update and/or delete responsibility of all released attributes once it has called attributeOwnershipReleaseResponse().

9.3 Ownership Push

Ownership push suggests that a federate owns update responsibility and/or the privilege to delete for an object instance and wishes to transfer ownership to another federate. The ownership may be surrendered "unconditionally" or "negotiated." *Unconditional push* releases a federate from attribute update and/or deletion responsibility without any commitment from other federates to assume these responsibilities. *Negotiated push* is a formal exchange where a federate retains responsibility until a new owner is identified and a formal exchange process is completed. Typical ownership push interactions are presented in Figure 9-4.

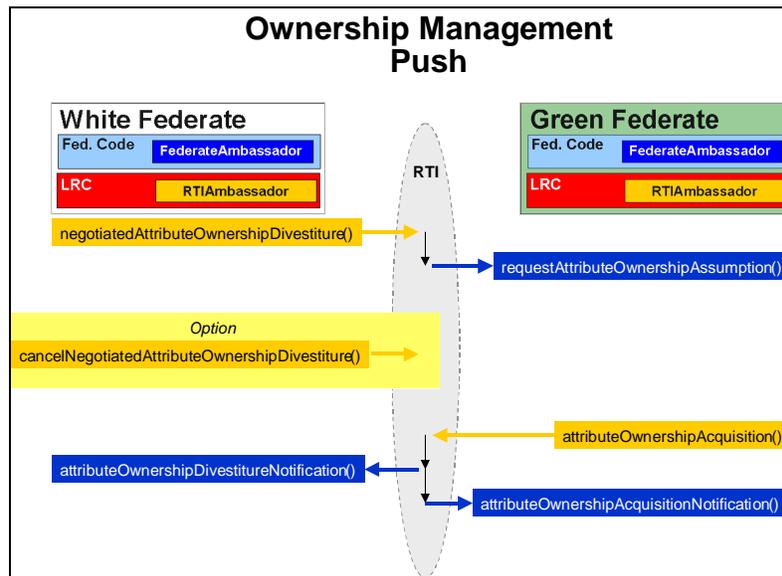


Figure 9-4. Ownership Push Interaction Diagram

9.3.1. Unconditional Push

A federate wishing to relieve itself immediately from attribute update responsibility for an object instance and/or the responsibility of deleting the object instance, can call the RTIAmbassador method `unconditionalAttributeOwnershipDivestiture()`. The federate is immediately free from the attribute responsibilities (including `privilegeToDelete` if listed) for the specified object instance.

9.3.2. Negotiated Push

A negotiated push is more involved than an unconditional push and is designed to ensure that attribute update and object deletion responsibilities are not dropped. The federate wishing to let go responsibilities calls the RTIAmbassador method `negotiatedAttributeOwnershipDivestiture()`. Federates that are capable of publishing any or all of the attributes being given away are notified

via the `FederateAmbassador` callback method `requestAttributeOwnershipAssumption()`. A federate wishing to acquire one or more of the offered attributes makes use of one of the pull methods – `attributeOwnershipAcquisition()` or `attributeOwnershipAcquisitionIfAvailable()`.

As federates are found to assume the responsibilities being given away, the federate that initiated the push receives the callback `attributeOwnershipDivestitureNotification()` – which informs the federate that it is no longer responsible for the listed attributes. The federate(s) gaining responsibility for the attributes is informed of its new responsibility with the callback method `attributeOwnershipAcquisitionNotification()`.

9.3.3. Complex Exchanges

Ownership exchange can be quite complex. In the push model, several federates may vie for ownership of offered attributes. The pushing federate may not succeed in giving all the requested attributes away. The contending federates may not get everything they ask for. A federate that does not get everything it wants may try to surrender the attributes it did receive. A federate that fails to get rid of everything it requested can let a negotiated divestiture stand or issue an unconditional divestiture. Divestiture calls for a specific object instance replace any previous calls for that instance.

9.4 Supporting Functions

9.4.1. Cancellation

Sometimes a federate reconsiders its decision during an ownership transfer. A federate attempting to push ownership may decide that there are not any takers or otherwise decides to retract the push offer. Push cancellation may be invoked by the `RTIambassador` method `cancelNegotiatedAttributeOwnershipDivestiture()`.

Similarly, a federate attempting to pull ownership of one or more attributes may wish to cancel the exchange. The method `cancelAttributeOwnershipAcquisition()` cancels a pull. It is acknowledged by the `confirmAttributeOwnershipAcquisitionCancellation()` callback method. The methods in Figures 9-3 and 9-4 are available to cancel an in-progress exchange.

9.4.2. Queries

Two mechanisms exist for determining attribute ownership. The `queryAttributeOwnership()` method seeks the federate currently responsible for a particular attribute of a particular object instance. It solicits the `informAttributeOwnership()` callback on the `FederateAmbassador` that delivers the handle of the owning federate.

The `RTIambassador` method `isAttributeOwnedByFederate()` returns a Boolean operator indicating whether the issuing federate owns or does not own the specified attribute for the specified object instance.

10. Data Distribution Management

10.1 Introduction

This chapter introduces Data Distribution Management (DDM). As discussed in Chapter 7, *Declaration Management*, the RTI uses publication and subscription information (declared by federates participating in a federation) to throttle the data placed on the network. Control signals issued by the RTI can be used to constrain type registration and instance updates. The RTI effectively serves as an intelligent switch – matching up producers and consumers of data, based on declared interests and without knowing details about the data format or content being transported.

DDM provides a flexible and extensive mechanism for further isolating publication and subscription interests – effectively extending the sophistication of the RTI's switching capabilities. In DDM, a federation "routing space" is defined. The *routing space* is a collection of "dimensions." The *dimensions* are used to define "regions." Each *region* is defined in terms of a set of "extents." An *extent* is a bounded range defined across the dimensions of a routing space. It represents a volume in the multi-dimensional routing space.

This chapter introduces DDM at a conceptual level and goes on to examine supporting RTI ambassador services and FederateAmbassador callbacks.

10.2 Example Routing Space

10.2.1. A Previous Example Revisited

If all this seems a bit confusing, perhaps an example will help. Chapter 7, *Declaration Management*, presented a declaration management example. Figure 7-3 (repeated here as Figure 10-1) illustrates the publication interest of Federate #1 and the subscription interest of Federate #2. Object class Y is a descendant of object class X (Figure 7-2, not reprinted).

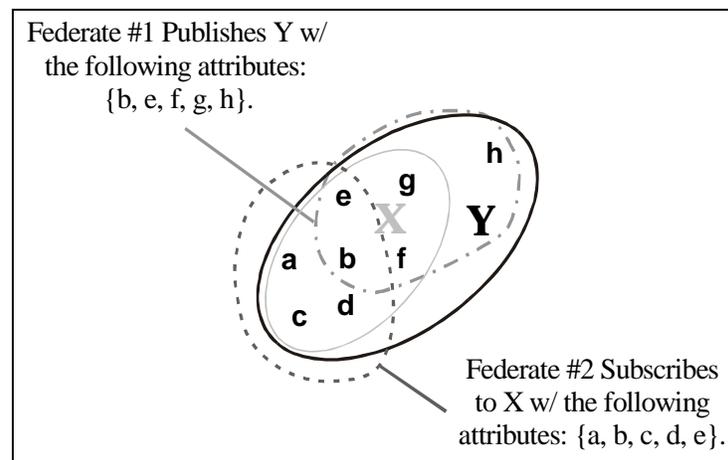


Figure 10-1. Publication and Subscription Intersections

Federate #1 indicates that it is able to publish $Y:\{b, e, f, g, h\}$. Federate #2 indicates that it wishes to subscribe to $X:\{a, b, c, d, e\}$. The RTI will promote instances of object class Y such that Federate #2 sees these instances as X instances. Since there is a consumer for the information produced by class Y , the RTI informs Federate #1 that it should register Y instances. As suggested in Chapter 8, *Object Management*, the RTI can provide additional information to Federate #1 indicating the specific attributes of specific object instances for which a subscription interest (i.e., a consumer) exists.

10.2.2. A Routing Space

Consider a routing space defined by the three dimensions "longitude," "latitude," and "altitude." Figure 10-2 illustrates such a routing space. For the examples in this chapter, this routing space is indicated with the shorthand notation $R\{\text{longitude, latitude, and altitude}\}$.

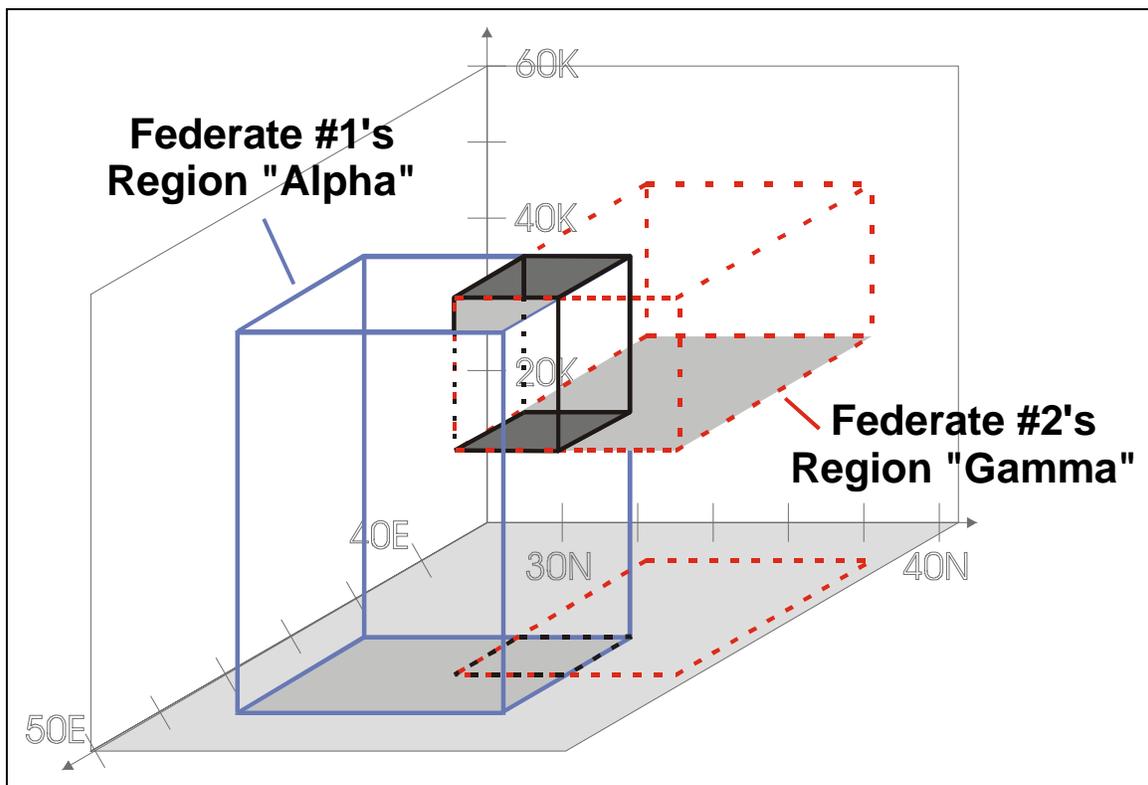


Figure 10-2. Example Routing Space

Federates can fine-tune their subscription declarations and data updates in terms of regions within the routing space. For example, Federate #1 might associate the attributes of an object class $Y:\{b, e, f, g, h\}$ with the following region:

$$R_{\text{Alpha}}\{\text{longitude: } 44^{\circ}\text{E} - 48^{\circ}\text{E, latitude: } 30^{\circ}\text{N} - 37^{\circ}\text{N, altitude: } 0 - 50,000 \text{ ft}\}$$

Similarly, Federate #2 might subscribe in $X:\{a, b, c, d, e\}$ with the region:

$R_{\text{Gamma}}\{\text{longitude: } 40^{\circ}\text{E} - 46^{\circ}\text{E}, \text{latitude: } 34^{\circ}\text{N} - 40^{\circ}\text{N}, \text{altitude: } 30,000 \text{ ft} - 50,000 \text{ ft}\}$

The overlap between the regions R_{Alpha} and R_{Gamma} is relatively small. In fact, the intersection of these two regions is:

$R_{\text{Alpha}} \cap \text{Gamma}\{\text{longitude: } 44^{\circ}\text{E} - 46^{\circ}\text{E}, \text{latitude: } 34^{\circ}\text{N} - 37^{\circ}\text{N}, \text{altitude: } 30,000 \text{ ft} - 50,000 \text{ ft}\}$

However, since the regions do intersect, the RTI will ensure that the data is communicated from Federate #1 to Federate #2.

10.3 Defining Routing Spaces and Regions

As suggested in the preceding example, federates associate data with regions. The High Level Architecture (HLA), on which the RTI is based, maintains a separation between the data and the code that manipulates the data (i.e., the RTI). DDM introduces a generic means of defining routing spaces and regions that do not require the RTI to have knowledge about a federation's data.

10.3.1. Routing Spaces

A routing space is essentially the problem space. Routing spaces identify all of the dimensions on which a region might be defined. The previous example used the routing space $R\{\text{longitude, latitude, and altitude}\}$.³³ The example routing space has three dimensions. All federates in a federation that elect to use routing spaces must agree upon the dimensions that form the routing space as well as the worst case upper and lower bounds along each dimension. The FED file specifies the routing spaces and the dimensions available to each federate within the federation.³⁴

In the sample problem, the federates might have agreed upon a routing space bounded by a longitude of 40°E to 50°E, a latitude of 30°N to 40°N, and an altitude of 0 to 50,000 feet. The FED file includes parameters that identify the routing space (by name) and the dimensions (by name). Beyond that federates must know the upper and lower bounds along each dimension in the routing space.³⁵

```

580      :
581      :
582 ;;   (spaces
583 ;;     (space <name>
584 ;;       (dimension <name>)
585 ;;         . . .
586 ;;       (dimension <name>)
587 ;;     )
588 ;;   . . .
589 ;;   (space <name>

```

³³ This is a rather obvious routing space. Some less obvious choices will be discussed subsequently.

³⁴ Note that the FED file may specify multiple routing spaces – all of which are available to federates.

³⁵ This is very similar to the requirement that the federates must "know" how to encode and decode attribute values. The range of possible values for each dimension is specified in the FOM routing space tables.

```

590 ;;      (dimension <name>)
591 ;;      . . .
592 ;;      (dimension <name>)
593 ;;      )
594 ;;      )

```

10.3.2. Extents

An extent is a volume defined by a range (minimum and maximum) along each dimension of a routing space. For DDM to support arbitrary dimensions, a generic representation scheme is needed to express extents. The scheme adopted by the RTI is as follows: The full range along a given dimension is mapped onto the interval [MIN_EXTENT, MAX_EXTENT].³⁶ Figure 10-3 illustrates a formula for translating a value "v" on the dimension "D" to a number in the range [MIN_EXTENT, MAX_EXTENT]. In order to specify a range, two such values must be mapped – one specifying the minimum value of the range and another specifying the maximum value of the range.

Federate #1 (from in the example above (paragraph 10.2.2) specified the region:

$R_{\text{Alpha}}\{\text{longitude: } 44^{\circ}\text{E} - 48^{\circ}\text{E}, \text{latitude: } 30^{\circ}\text{N} - 37^{\circ}\text{N}, \text{altitude: } 0 - 50,000 \text{ ft}\}$

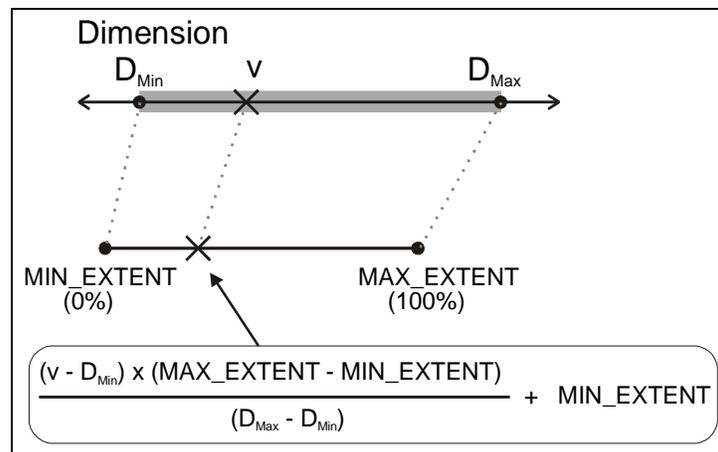


Figure 10-3. Normalization of a Range in an Extent

10.3.3. Calculation of Extents

The region was specified in terms of one extent containing a range for each dimension. Here, the extent is expressed in terms of range values on the dimension axis.

$R_{\text{Alpha}}\{\text{longitude: } 44^{\circ}\text{E} - 48^{\circ}\text{E}, \text{latitude: } 30^{\circ}\text{N} - 37^{\circ}\text{N}, \text{altitude: } 0 - 50,000 \text{ ft}\}$

³⁶ The values of MIN_EXTENT and MAX_EXTENT are defined by macros in the RTI header files and should be treated as implementation-specific.

Each range in the extent must be mapped onto the generic range (i.e., bounded by MIN_EXTENT and MAX_EXTENT) for submittal to the RTI. In order to compute the mapping, the minimum and maximum values along each dimension must be calculated.

$$\alpha_{\text{long}_{\text{min}}} = \frac{(4-4) \times (\text{MAX_EXTENT} - \text{MIN_EXTENT})}{(5-4)} + \text{MIN_EXTENT}$$

$$\alpha_{\text{long}_{\text{max}}} = \frac{(48-40) \times (\text{MAX_EXTENT} - \text{MIN_EXTENT})}{(50-40)} + \text{MIN_EXTENT}$$

$$\alpha_{\text{lat}_{\text{min}}} = \frac{(30-30) \times (\text{MAX_EXTENT} - \text{MIN_EXTENT})}{(40-30)} + \text{MIN_EXTENT}$$

$$\therefore \alpha_{\text{lat}_{\text{min}}} = \text{MIN_EXTENT}$$

$$\alpha_{\text{lat}_{\text{max}}} = \frac{(37-30) \times (\text{MAX_EXTENT} - \text{MIN_EXTENT})}{(40-30)} + \text{MIN_EXTENT}$$

$$\alpha_{\text{altitude}_{\text{min}}} = \frac{(0-0) \times (\text{MAX_EXTENT} - \text{MIN_EXTENT})}{(50,000-0)} + \text{MIN_EXTENT}$$

$$\therefore \alpha_{\text{altitude}_{\text{min}}} = \text{MIN_EXTENT}$$

$$\alpha_{\text{altitude}_{\text{max}}} = \frac{(50,000-0) \times (\text{MAX_EXTENT} - \text{MIN_EXTENT})}{(50,000-0)} + \text{MIN_EXTENT}$$

$$\therefore \alpha_{\text{altitude}_{\text{max}}} = \text{MAX_EXTENT} - \text{MIN_EXTENT} + \text{MIN_EXTENT} = \text{MAX_EXTENT}$$

10.3.4. Creative Dimensions

Federations are free to introduce dimensions that suit the needs of constituent federates. For example, two candidate dimensions may be Frequency {with enumerated ranges defined as 1, 2, 3, 4, 5} or Military Ranks {with enumerated ranges defined as Private, Corporal, Sergeant, Sergeant Major, so forth.}. Extents might be defined on each and incorporated into region definitions. A radio frequency spectrum might serve as a dimension. Federates would define regions that include a frequency range. Even a discrete, ordinal series might serve as a dimension.

A dimension could be introduced identifying the "UpdateFrequency" of certain updates. It might contain such values as "once per second," "once per 10 seconds," and "once per 60 seconds." A producing federate capable of issuing update information every second could publish the updates every second to a region including an UpdateFrequency with an extent that covered "once per second." Every ten seconds, the federate would publish to a region that included an

UpdateFrequency extent that covered "once per second" and "once per 10 seconds." A federate wishing to receive information every 10 seconds would construct its regions accordingly.³⁷

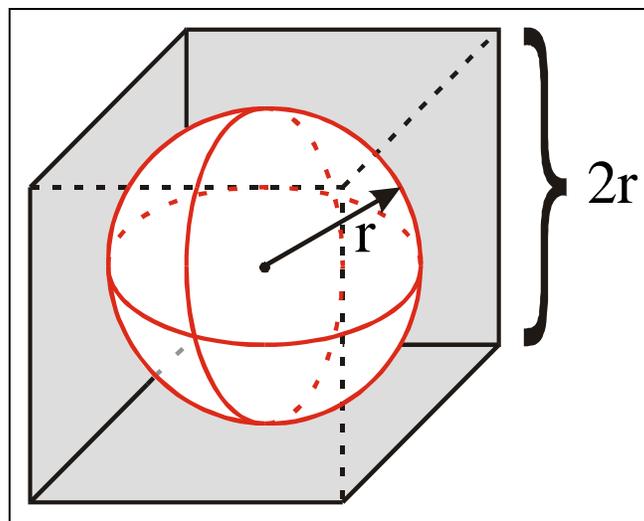
10.3.5. Regions and Attributes

The RTI makes no intuitive connections between regions and attributes. For example, an airplane object class might contain the attributes "longitude," "latitude," and "altitude." The routing space might contain the dimensions "longitude," "latitude," and "altitude." The RTI does not make any automatic associations between the class attribute "longitude" and the routing space dimension "longitude." It is entirely up to the producing federate to associate events (e.g., object updates, interactions) with regions!

10.3.6. Oddly Shaped Regions

The RTI supports the specification of a rectangle-shaped region. Some simulations are interested in oddly shaped regions. Complex areas can be defined by collecting multiple extents within a region. However, use of numerous extents or artificially complex regions may have a negative impact on performance. A federate may also use the RTI to specify initial thresholds and go on to perform additional filtering within its simulation.

Figure 10-5 illustrates a cube-shaped region. A sphere appears within the cube. A federate might subscribe to certain events within this cube-shaped region. All activities outside the cube are suppressed by the RTI. The federate, however, is only interested in events within the inner sphere. In this case, the federate must use additional information (e.g., object attribute values, interaction parameter values) to discern whether received events are applicable or not.



³⁷ Clearly, some consideration would have to be given as to whether the ten second updates were differential or exhaustive. Other schemes are also possible.

Figure 10-5. Two-Layer Filtering

10.3.7. Thresholds

The RTI.rid file contains threshold values that can cause region specification to be treated as *fuzzy*. The description of the RTI implementation class "Region" in Appendix C discusses how thresholds can effectively extend regions.

10.3.8. Default Routing Space

The RTI provides a "default routing space." Events and requests that are not associated with a particular routing space are automatically associated with the default routing space. The RTI associates an RTI::SpaceHandle (i.e., a numeric representation) with every routing space. The default routing space will have the value returned by RTI::SpaceHandle().

10.4 Creating Regions

Federates call the RTIambassador method createRegion() to construct a new region on a specified routing space (see Appendix A, *Class RTI::RTIambassador*). The routing space must be declared in the FED file. The createRegion() method returns a pointer to a free-store allocated instance of the RTI::Region class (see Appendix C, *Supporting Types and Classes*). Regions must be deleted with the RTIambassador method deleteRegion().

The following RTI::Region methods allow the federate to get and/or set the minimum and maximum values of each extent range – one extent at a time:

getRangeLowerBound()	getRangeUpperBound()
setRangeLowerBound()	setRangeUpperBound()

Functions also exist to identify the routing space with which the region is associated. Once a region has been modified locally, the changes must be communicated to the RTI. The RTIambassador method notifyAboutRegionModification() exists for that purpose. Figure 10-6 illustrates the interactions required to create, alter, and delete a region.

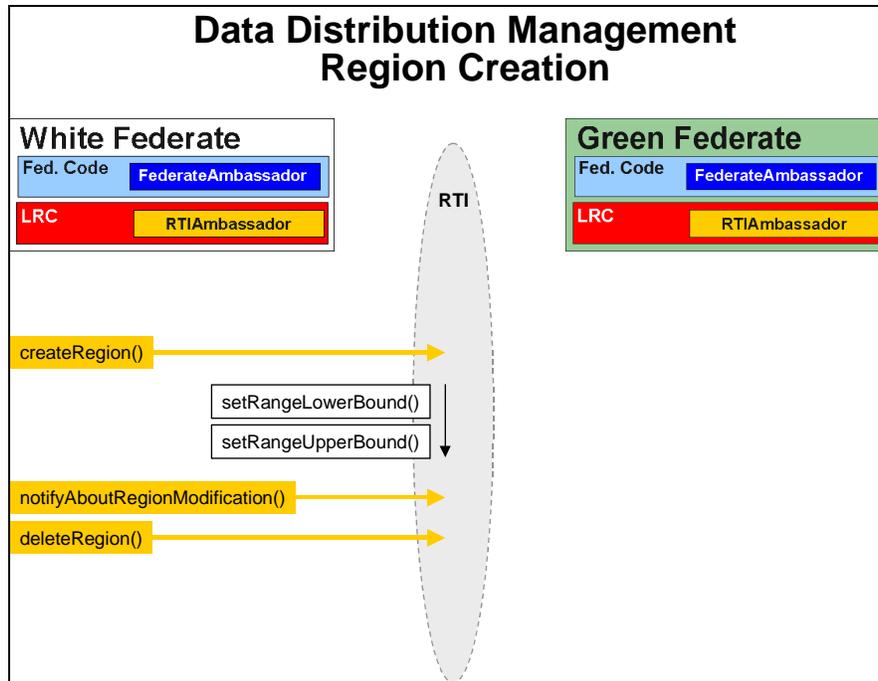


Figure 10-6. Region Methods

10.5 Binding Object Attributes to Regions

Object instance updates and interactions can be tied to regions on the sending federate and subscriptions can be tied to regions on the receiving federate. Each federate maintains its own regions. Federates do not know anything about the regions of their peers.³⁸

10.5.1. Attribute Updates and Regions

An "attribute instance" is a particular attribute of a particular object instance. The FED file specifies the routing space for each attribute of an object class. A given attribute instance is only associated with one region at any given time and the region must be specified on the appropriate routing space.

The RTIAmbassador method `associateRegionForUpdates()` ties a set of attributes for a particular object instance to a specified region. The counterpart method `unassociateRegionForUpdates()` removes the association between a region and an object instance. In the event that an attribute

³⁸ Receiving federates can use the `RTI::AttributeHandleValuePairSet::getRegion(RTI::Ulong index)` method to get the update region of each attribute as well as the `RTI::ParameterHandleValuePairSet::getRegion(void)` method to get the update region for an interaction.

instance is not explicitly bound to a region, the RTI implicitly binds such instances to the default region on the appropriate routing space.³⁹

For attribute instances that are associated with different regions, multiple calls to `associateRegionForUpdates()` are required. As an alternative, the RTIambassador method `registerObjectInstanceWithRegion()` allows a federate to specify attribute-to-region mapping for some or all attributes – i.e., without multiple calls to `associateRegionForUpdates()`.

10.5.2. Attribute Subscriptions and Regions

Associating a region with a subscription is similar to associating a region with updates. The RTIambassador method `subscribeObjectClassAttributesWithRegion()` allows a federate to associate a set of attributes with a region for a given object class. The call is similar to the declaration management call `subscribeObjectClassAttributes()`, but with a few important changes. Whereas repeated calls to `subscribeObjectClassAttributes()` replaced prior calls, multiple calls to `subscribeObjectClassAttributesWithRegion()` accrue – with the caveat that individual attributes can only be associated with one region and that region will be the region most recently specified. The method `unsubscribeObjectClassWithRegion()` removes interest in certain attributes.⁴⁰

10.5.3. Requesting Updates

The RTIambassador method `requestClassAttributeValueUpdateWithRegion()` solicits an attribute update the same way as an `requestClassAttributeValueUpdate()`, but associates the request with a region. It effectively solicits updates for the named attributes of all objects of a given class that are associated with a region that intersects the identified region. Figures 10-7 through 10-9 illustrate the methods for managing attributes.

³⁹ Recall, the default region for any given routing space, includes the entire routing space. Since the FED file specifies the appropriate routing space per object attribute, the RTI knows which routing space and what default region to use.

⁴⁰ It's tempting to expect this function to remove the association with a specific region in favor of a default region. This is not the case. Interest in the specified attributes is abandoned all together.

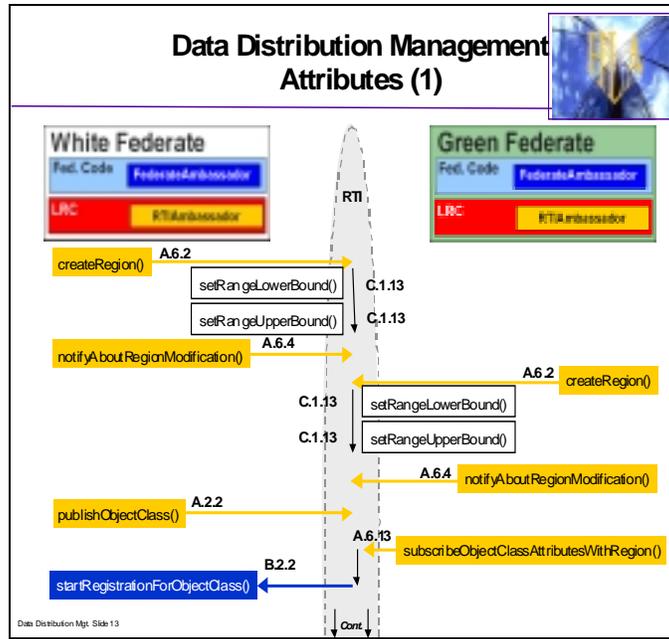


Figure 10-7. DDM Attributes (Part 1 of 3)

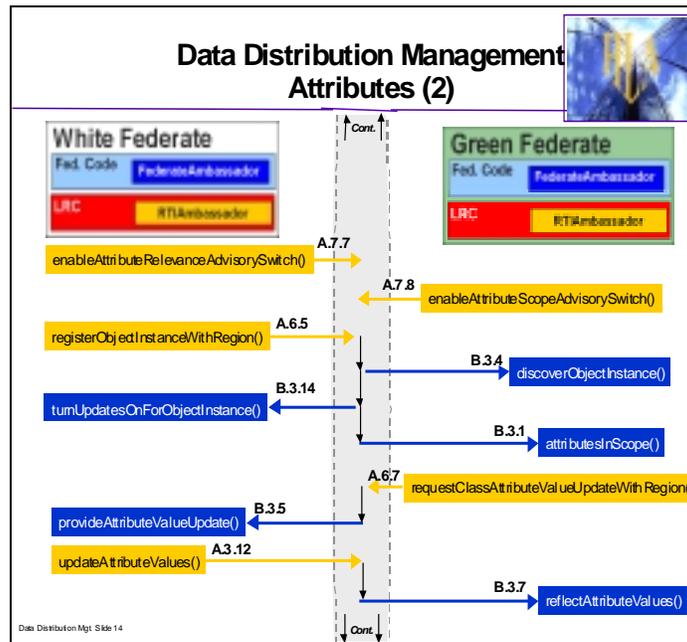


Figure 10-8. DDM Attributes (Part 2 of 3)

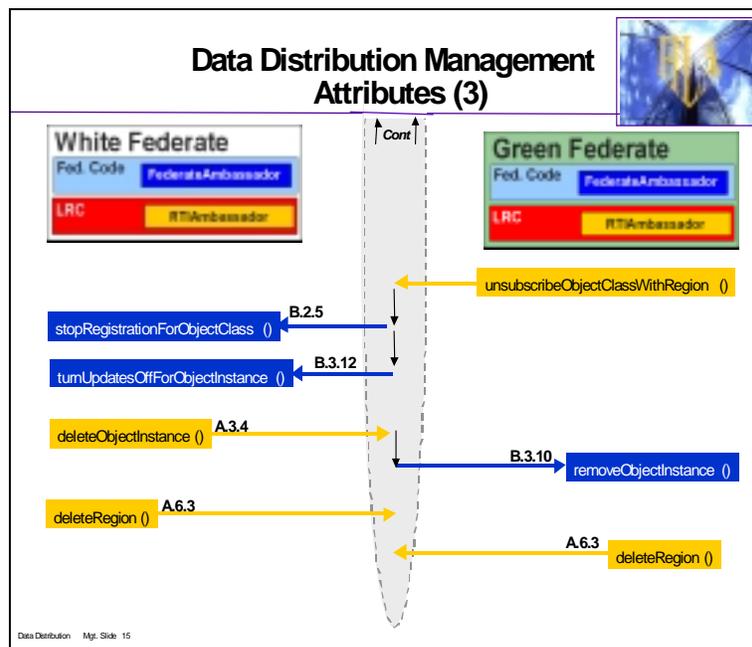


Figure 10-9. DDM Attributes (Part 3 of 3)

10.5.4. Object Ownership and Regions

When federates exchange ownership of attribute instances, the region associations for the attribute instances will not be maintained for the federate acquiring ownership.

10.5.5. Time and Regions

As with the declaration management services, methods that associate regions with attribute instances or with subscriptions take place immediately and are not subject to time management (i.e., such specifications cannot be tied to a future time).

10.6 Binding Interactions to Regions

Interactions may be bound to regions; however, such bindings are "all or nothing." It is not possible to associate specific interaction parameters with different regions. DDM methods for interactions are presented in Figure 10-10. The RTIAmbassador method `sendInteractionWithRegion()` allows a producing federate to tie an interaction to a region. The methods `subscribeInteractionClassWithRegion()` and `unsubscribeInteractionClassWithRegion()` can be used to tie a region to an interaction subscription (i.e., on the interaction recipient side).

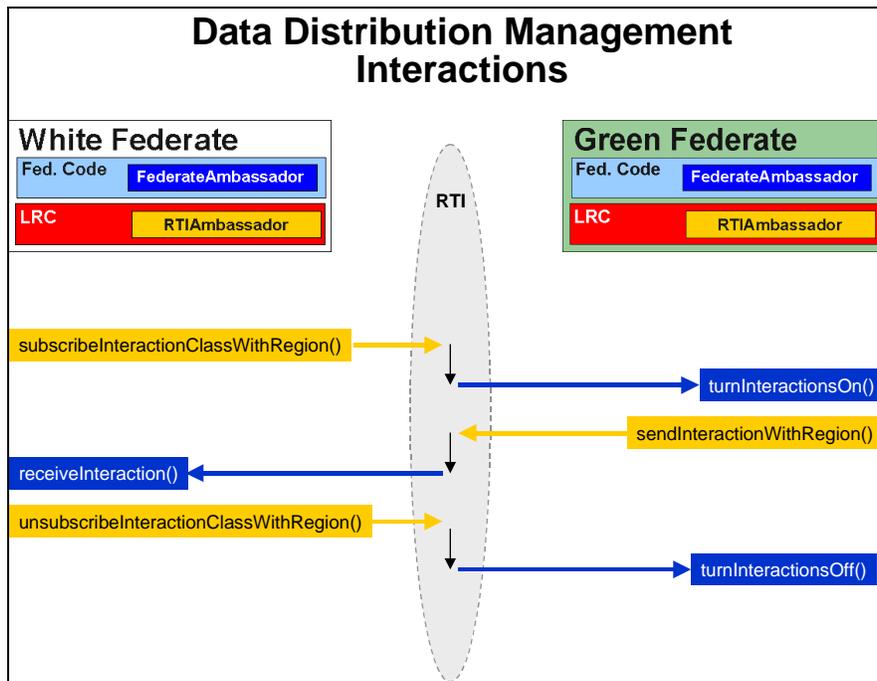


Figure 10-10. Interactions and DDM

11. Management Object Model

11.1 Introduction to the Management Object Model

The Management Object Model (MOM) consists of a number of object and interaction classes that must be present in the encoded FOM hierarchy of any FED file intended for use with the RTI. These classes constitute a mechanism by which federates may obtain information about the internal and external characteristics of the LRCs comprising the federation. Typically, this information will be combined by a “manager” federate and used to monitor and tune the operation of an active federation.

The RTI 1.3-NG software implements the MOM hierarchy described in *HLA Interface Specification* version 1.3. The HLA 1.3 MOM consists of the following primary components.

- A *Manager.Federate* object class, which is instantiated exactly once per federate by the federate’s LRC.
- A *Manager.Federation* object class, which is instantiated exactly once per federation execution by the RTI.
- A *Manager.Federate.Adjust* hierarchy of interactions, which may be sent by federates to effect changes in the internal and external characteristics of the LRCs comprising the federation.
- A *Manager.Federate.Request* hierarchy of interactions, which may be sent by federates to solicit reports on the internal and external characteristics of the LRCs comprising the federation.
- A *Manager.Federate.Report* hierarchy of interactions, which are sent by LRCs in response to information requests initiated by federates.
- A *Manager.Federate.Service* hierarchy of interactions, which may be sent by federates to invoke services and callbacks on remote LRCs and federates, respectively.

A federate’s LRC will automatically publish and subscribe various classes on behalf of the federate. The publications and subscriptions are independent of any federate-level publications and subscriptions. A federate must publish the appropriate interaction classes before sending out interaction instances, and may subscribe to MOM object and interaction classes to receive reflections of MOM events.

All parameters and attributes of MOM classes are represented as null-terminated strings. Numeric values are encoded as strings suitable for conversion using `atoi()` or `atof()`. Lists of values and composite types (i.e., C++ *structs*) are typically encoded as comma-delimited sequences containing the elements comprising the list or composite entity. Federation Time parameters in the MOM interactions sent by the user should be encoded using the `RTI::FedTime.encode()` method; however, for received interactions, the Federation Time parameters are encoded with `RTI::FedTime.getPrintableString()`. See the descriptions of specific MOM attributes and parameters for more details.

11.2 Interactions

Manager

SYNOPSIS

```
(class Manager reliable receive
  (class ...)
)
```

DESCRIPTION

This class is the root of the MOM interaction class hierarchy. It has no parameters and is not intended to be directly subscribed or instantiated.

Manager.Federate

SYNOPSIS

```
(class Federate reliable receive
  (parameter Federate)
  (class ...)
)
```

DESCRIPTION

This class is the root of the hierarchy of MOM interactions that are generated by or intended for an entity associated with a specific federate handle (either the federate itself or the LRC associated with the federate.) In RTI 1.3, all interactions fall into this category. This class is not intended to be directly subscribed or instantiated.

In the *Adjust*, *Request*, and *Service* sub-hierarchies, the federate-handle parameter denotes the intended recipient of the interaction. In the *Report* sub-hierarchy, the federate-handle parameter denotes the sender of the interaction.

PARAMETERS

Federate

the federate handle (as returned by `joinFederationExecution()` of the federate or LRC sender or recipient of the interaction)

Manager.Federate.Adjust

SYNOPSIS

```
(class Adjust reliable receive
  (class ...)
)
```

DESCRIPTION

This class is the root of the hierarchy of MOM interaction classes used to modify internal characteristics of an LRC. This class should not be directly subscribed or instantiated. Subclasses of this class are intended to be generated by federates and reacted to by LRCs. The *Federate* parameter inherited from *Manager.Federate* specifies the recipient LRC of an interaction instance.

Manager.Federate.Adjust.ModifyAttributeState

SYNOPSIS

```
(class ModifyAttributeState reliable receive
  (parameter ObjectInstance)
  (parameter Attribute)
  (parameter AttributeState)
)
```

DESCRIPTION

Interactions of this class may be generated by a federate to cause an instance-attribute to become owned or unowned by a specified federate, independent of RTI ownership management services. This may result in the instance-attribute being lost to the federation. However, if the attribute is owned by another federate, any attempt to assume ownership will fail. The attribute must first be set to unowned.

The object instance affected by this interaction must be known by the federate for which ownership is being toggled. The class-attribute corresponding to the affected instance-attribute need not be published by the federate. The affected federate will receive no indication that the ownership status of the instance-attribute has been modified.

This interaction is intended to be used to recover instance-attributes lost to a federation during a crash.

All three parameters must be present for an instance of this interaction to be valid. If one or more parameters are missing, the interaction has no effect.

PARAMETERS

ObjectInstance

the instance name of the object instance for which to modify the state of the instance-attribute

Attribute

an attribute handle (taken in context of the actual class of the object instance) representing the instance-attribute whose state is to be modified

AttributeState

a string equal to “owned” or “unowned” (case-insensitive), indicating the new state for the instance-attribute at the receiving LRC

Manager.Federate.Adjust.SetServiceReporting

SYNOPSIS

```
(class SetServiceReporting reliable receive
  (parameter ReportingState)
)
```

DESCRIPTION

Interactions of this class may be generated by a federate to toggle reporting of service calls by a specified LRC. When service reporting is enabled at an LRC, it will send a *Manager.Federate.Report.ReportServiceInvocation* interaction for each federate- or RTI-initiated service call.

By default, service reporting is turned off for all LRCs.

It is illegal for a federate to have service reporting enabled and to be subscribed to the *Manager.Federate.Reporting.ReportServiceInvocation* interaction, an Alert will be sent.

PARAMETERS

ReportingState

a string equal to “true” or “false” (case-insensitive), indicating the new toggle state of service reporting at the receiving LRC

Manager.Federate.Adjust.SetExceptionLogging

SYNOPSIS

```
(class SetExceptionLogging reliable receive
  (parameter LoggingState)
)
```

DESCRIPTION

Interactions of this class may be generated by a federate to toggle logging of exceptions by a specified LRC. Turning logging off stops all exceptions from being written to the federate log file. By default, the log file is written to the federate's current directory, in a file named

<File Prefix>-<Fed Name>

where

<File Prefix> is the file prefix specified by the ExceptionLoggingFilePrefix RID parameter (default value: "RtiMomExceptionLoggingFile")

<Fed Name> is the federate's name as specified in the call to joinFederationExecution().

Each exception entry lists the date and time that the exception is logged, followed by the exception name and its description.

By default, logging is turned off for all LRCs.

PARAMETERS

LoggingState

a string equal to "true" or "false" (case-sensitive), indicating the new toggle state of logging at the receiving LRC

Manager.Federate.Adjust.SetTiming

SYNOPSIS

```
(class SetTiming reliable receive
  (parameter ReportPeriod)
)
```

DESCRIPTION

Interactions of this class may be generated by a federate to set the frequency at which a specified LRC will generate updates for the *Manager.Federate* object representing its local federate. A value of zero may be specified to disable updates by the specified LRC.

By default, an LRC does not generate periodic updates for its local *Manager.Federate* object.

PARAMETERS

ReportPeriod

a positive integer value representing a time (in seconds) used to set the update period, or zero to disable updates

Manager.Federate.Report

SYNOPSIS

```
(class Report reliable receive
  ...
)
```

DESCRIPTION

This class is the root of the hierarchy of MOM interaction

HLA RTI 1.3-Next Generation

classes generated by LRCs to report various characteristics of LRC and federate state. This class should not be directly subscribed or instantiated. Subclasses of this class are indented to be subscribed by federates and generated by LRCs. The *Federate* parameter inherited from *Manager.Federate* specifies the LRC sender of an interaction instance.

Manager.Federate.Report.Alert

SYNOPSIS

```
(class Alert reliable receive
  (parameter AlertSeverity)
  (parameter AlertDescription)
  (parameter AlertID)
)
```

DESCRIPTION

Interactions of this class are generated by an LRC whenever it throws an exception.

PARAMETERS

AlertSeverity

A text string representing the severity of the exception thrown by the LRC; it will be one of the following:

- "RTI exception"
- "RTI internal error"
- "Federate internal error"
- "Warning" (not supported)
- "Diagnostic" (not supported)

AlertDescription

the text associated with the exception; this includes the type-name of the exception class and a string description of the reason for the exception

AlertID

an integer indicating the alert count; this count is incremented after each Alert is sent

Manager.Federate.Report.ReportInteractionPublication

SYNOPSIS

```
(class ReportInteractionPublication reliable
  receive
  (parameter InteractionClassList)
)
```

DESCRIPTION

Interactions of this class are generated by an LRC in response to *Manager.Federate.Request.RequestPublications* interactions. This interaction reports only the interaction classes published by the federate itself (i.e., it does not include interaction classes published by the LRC on behalf of the federate.)

PARAMETERS

InteractionClassList

a comma-delimited list of interaction class handles

being published by the reporting federate (null if no interaction classes were published).

InteractionCounts

a comma-delimited list of pairs of the form “<class>/<count>” where *class* is an interaction class handle and *count* is the number of interactions that have been sent by the federate of that class; only classes that have a non-zero count are listed (null if no interactions were sent)

Manager.Federate.Report.ReportInteractionsReceived

SYNOPSIS

```
(class ReportInteractionsReceived reliable receive
  (parameter TransportationType)
  (parameter InteractionCounts)
)
```

DESCRIPTION

Interactions of this class are generated by an LRC in response to *Manager.Federate.Request.RequestInteractionsReceived* interactions. Two reports will be generated in response to such a request: one for best-effort transport and one for reliable transport. Each report details the interactions that have been delivered to the federate, tabulated according to the actual classes of the interactions (which is not necessarily the same as the classes by which the interactions were actually presented to the federate.) These counts do not include MOM interactions that were not delivered to the federate, nor do they include interactions used for internal RTI communications.

PARAMETERS

TransportationType

a string equal to “Reliable” or “Best Effort” depending on the transportation service category being reported

InteractionCounts

a comma-delimited list of pairs of the form “<class>/<count>” where *class* is an interaction class handle and *count* is the number of interactions that have been delivered to the federate of that class; only classes that have a non-zero count are listed (null if no interactions were received)

Manager.Federate.Report.ReportInteractionsSent

SYNOPSIS

```
(class ReportInteractionsSent reliable receive
  (parameter TransportationType)
  (parameter InteractionCounts)
)
```

DESCRIPTION

Interactions of this class are generated by an LRC in response to *Manager.Federate.Request.RequestInteractionsSent* interactions. Two reports will be generated in response to such a request: one for best-effort transport and one for reliable transport. Each report details the interactions that have been sent by the federate, tabulated according to the classes of the interactions. These counts do not include MOM interactions that sent by the LRC on behalf of the federate, nor do they include interactions used for internal RTI communications.

PARAMETERS

TransportationType

a string equal to “Reliable” or “Best Effort” depending on the transportation service category being reported

Manager.Federate.Report.ReportInteractionSubscription

SYNOPSIS

```
(class ReportInteractionSubscription reliable receive
  (parameter InteractionClassList)
)
```

DESCRIPTION

Interactions of this class are generated by an LRC in response to *Manager.Federate.Request.RequestSubscriptions* interactions. This interaction reports only the interaction classes subscribed by the federate itself (i.e., it does not include interaction classes subscribed by the LRC on behalf of the federate.)

PARAMETERS

InteractionClassList

a comma-delimited list of interaction class handles being subscribed by the reporting federate (null if no interaction classes were subscribed)

Manager.Federate.Report.ReportObjectInformation

SYNOPSIS

```
(class ReportObjectInformation reliable receive
  (parameter ObjectInstance)
  (parameter OwnedAttributeList)
  (parameter RegisteredClass)
  (parameter KnownClass)
)
```

DESCRIPTION

Interactions of this class are generated by an LRC in response to *Manager.Federate.Request.RequestObjectInformation* interactions.

PARAMETERS

ObjectInstance

the object name corresponding to the instance subject of the report

OwnedAttributeList

a comma-delimited list of attribute handles (in the context of the actual object class of the instance) representing any instance-attributes of the object owned by the reporting federate (null if the object instance is invalid)

RegisteredClass

the class handle of the actual (registered) object class of the object instance (null if the object instance is invalid)

KnownClass

the class handle of the object class by which the reporting federate has discovered the object, or the actual class if the federate owns the object (null if the object instance is invalid)

Manager.Federate.Report.ReportObjectPublication

SYNOPSIS

```
(class ReportObjectPublication reliable receive
  (parameter NumberOfClasses)
  (parameter ObjectClass)
  (parameter AttributeList)
)
```

DESCRIPTION

Interactions of this class are generated by an LRC in response to *Manager.Federate.Request.RequestPublications* interactions. This interaction reports only the object classes published by the federate itself (i.e., it does not include object classes published by the LRC on behalf of the federate.)

Each publication request will result in a separate *ReportObjectPublication* interaction for each object class published by the federate. The *NumberOfClasses* parameter – which is the same for each interaction comprising the response – indicates the total number of reports sent in response.

PARAMETERS

NumberOfClasses

an integer indicating the total number of object-publication reports in the sequence this report is part of

ObjectClass

the object class handle of the object class published by the reporting federate

AttributeList

the attribute handles of the class-attributes of the specified object class published by the reporting federate (null if no object classes were published)

Manager.Federate.Report.ReportObjectsOwned

SYNOPSIS

```
(class ReportObjectsOwned reliable receive
  (parameter ObjectCounts)
)
```

DESCRIPTION

Interactions of this class are generated by an LRC in response to *Manager.Federate.Request.RequestObjectsOwned* interactions. An object instance is considered owned by the reporting federate if and only if the federate owns the *privilegeToDelete* instance-attribute for the object. Objects owned by the LRC on behalf of the federate (e.g., the *Manager.Federate* object instance corresponding to the federate) are not included in the count.

The report is tabulated according to the actual (registered) object class of the object instances, which may not be the same as the object classes by which they are known to the federate.

HLA RTI 1.3-Next Generation

PARAMETERS

ObjectCounts

a comma-delimited list of pairs of the form “<class>/<count>” where *class* is an object class handle and *count* is the number of instances of the class for which the reporting federate holds the privilege to delete (null if no objects are owned)

Manager.Federate.Report.ReportObjectsReflected

SYNOPSIS

```
(class ReportObjectsReflected reliable receive
  (parameter ObjectCounts)
)
```

DESCRIPTION

Interactions of this class are generated by an LRC in response to *Manager.Federate.Request.RequestObjectsReflected* interactions. This report indicates the number of reflections that have been delivered to the reporting federate by object class (i.e., it does not include reflections processed internally by the LRC.) If multiple *reflectAttributeValues()* callbacks are made in response to a single update (e.g., if different attributes are sent reliably vs. best effort), they will be tallied individually.

The report is tabulated according to the actual (registered) object class of the object instances that were subjects of reflections, which may not be the same as the object classes by which they are known to the federate.

PARAMETERS

ObjectCounts

a comma-delimited list of pairs of the form “<class>/<count>” where *class* is an object class handle and *count* is the number of reflections delivered to the federate for instances of the class; only non-zero counts are listed (null if no objects were reflected)

Manager.Federate.Report.ReportObjectSubscription**SYNOPSIS**

```
(class ReportObjectSubscription reliable receive
  (parameter NumberOfClasses)
  (parameter ObjectClass)
  (parameter AttributeList)
  (parameter Active)
)
```

DESCRIPTION

Interactions of this class are generated by an LRC in response to *Manager.Federate.Request.RequestSubscriptions* interactions. This interaction reports only the interaction classes subscribed by the federate itself (i.e., it does not include interaction classes subscribed by the LRC on behalf of the federate.)

Each subscription request will result in a separate *ReportObjectSubscription* interaction for each object class subscribed by the federate. The *NumberOfClasses* parameter – which is the same for each interaction comprising the response – indicates the total number of reports sent in response.

PARAMETERS*NumberOfClasses*

an integer indicating the total number of object-publication reports in the sequence this report is part of

ObjectClass

the object class handle of the object class published by the reporting federate

AttributeList

the attribute handles of the class-attributes of the specified object class published by the reporting federate (null if no object classes were subscribed)

Active

a string equal to “True” or “False”, depending on the type of the subscription

Manager.Federate.Report.ReportObjectsUpdated**SYNOPSIS**

```
(class ReportObjectsUpdated reliable receive
  (parameter ObjectCounts)
)
```

DESCRIPTION

Interactions of this class are generated by an LRC in response to *Manager.Federate.Request.RequestObjectsUpdated* interactions. This report indicates the number of object instances for which the federate owns at least one instance-attribute.

The report is tabulated according to the actual (registered) object class of the object instances that the federate has updated. This may differ from the object classes by which the instances are actually known to the federate.

PARAMETERS*ObjectCounts*

a comma-delimited list of pairs of the form

HLA RTI 1.3-Next Generation

“<class>/<count>” where *class* is an object class handle and *count* is the number of updates initiated by the federate for instances of the class; only non-zero counts are listed (null if no objects were updated)

Manager.Federate.Report.ReportReflectionsReceived**SYNOPSIS**

```
(class ReportReflectionsReceived reliable receive
  (parameter TransportationType)
  (parameter ReflectCounts)
)
```

DESCRIPTION

Interactions of this class are generated by an LRC in response to *Manager.Federate.Request.RequestReflectionsReceived* interactions. Two reports will be generated in response to such a request: one for best-effort transport and one for reliable transport. Each report indicates the number of reflections that have been delivered to the reporting federate by object class (i.e., it does not include reflections processed internally by the LRC.) If multiple *reflectAttributeValues()* callbacks are made in response to a single update (e.g., if different attributes are sent reliably vs. best effort), they will be tallied individually.

The report is tabulated according to the actual (registered) object class of the object instances that were subjects of reflections, which may not be the same as the object classes by which they are known to the federate.

PARAMETERS*TransportationType*

a string equal to “Reliable” or “Best Effort” depending on the transportation service category being reported

ReflectCounts

a comma-delimited list of pairs of the form “<class>/<count>” where *class* is an object class handle and *count* is the number of reflections that have been delivered to the federate for instances of the class; only classes that have non-zero counts are listed (null if no reflections were received)

Manager.Federate.Report.ReportServiceInvocation

SYNOPSIS

```
(class ReportServiceInvocation reliable receive
  (parameter Service)
  (parameter Initiator)
  (parameter SuccessIndicator)
  (parameter SuppliedArgument1)
  (parameter SuppliedArgument2)
  (parameter SuppliedArgument3)
  (parameter SuppliedArgument4)
  (parameter SuppliedArgument5)
  (parameter ReturnedArgument)
  (parameter ExceptionDescription)
  (parameter ExceptionID)
)
```

DESCRIPTION

If service logging is enabled for an LRC, the LRC will generate an interaction of this class for every RTI- and federate-ambassador service invocation made by/to the local federate. The string representation of the various types of arguments is as follows:

Type	Representation
integers and longs	string suitable for conversion using <code>atoi()</code>
strings	the string value, passed as-is
RTI::FedTime	The string returned by <code>RTI::FedTime.getPrintableString()</code>
RTI::Boolean	“True” or “False”
RTI::EventRetractionHandle	integers suitable for conversion using <code>atoi()</code> , separated by commas, representing the serial number and sending federate, respectively
RTI::AttributeHandleSet, RTI::AttributeHandleValuePairSet, RTI::ParameterHandleValuePairSet	comma-delimited list of integers suitable for conversion using <code>atoi()</code> (attribute/parameter values are not represented)
RTI::Region	The memory address of the region

It is illegal for a federate to have service reporting enabled and to be subscribed to the `Manager.Federate.Report.ReportServiceInvocation` interaction. If a federate has service reporting enabled and attempts to subscribe to the `Manager.Federate.Report.ReportServiceInvocation` interaction, a `FederateLoggingServiceCalls` exception is thrown

PARAMETERS

Service

the name of the C++ method implementing the service

Initiator

a string, “FED” or “RTI”, indicating an RTI- or federate-ambassador service, respectively

SuccessIndicator

a string, “True” or “False” indicating whether the service completed successfully

SuppliedArgument1

a string representation of the first argument to the service method (null if the service has no first argument)

SuppliedArgument2

a string representation of the second argument to the service method (null if the service has no second argument)

SuppliedArgument3

a string representation of the third argument to the service method (null if the service has no third argument)

SuppliedArgument4

a string representation of the fourth argument to the service method (null if the service has no fourth argument)

SuppliedArgument5

a string representation of the fifth argument to the service method (null if the service has no fifth argument)

ReturnedArgument

a string representation of the return value of the service method (null if the service has a void return argument or if `SuccessIndicator` is false)

ExceptionDescription

the text associated with the exception thrown (null if `SuccessIndicator` is true)

ExceptionID

A string containing a zero (null if `SuccessIndicator` is true)

Manager.Federate.Report.ReportUpdatesSent

SYNOPSIS

```
(class ReportUpdatesSent reliable receive
  (parameter TransportationType)
  (parameter UpdateCounts)
)
```

DESCRIPTION

Interactions of this class are generated by an LRC in response to `Manager.Federate.Request.RequestUpdatesSent` interactions. Two reports will be generated in response to such a request: one for best-effort transport and one for reliable transport. Each report indicates the number of updates that have been initiated by the reporting federate by object class (i.e., it does not include updates sent by the LRC for internal RTI needs.) If multiple physical updates result from a single `updateAttributeValues()` service invocation (e.g., if different attributes are sent reliably vs. best effort), they will be tallied individually.

The report is tabulated according to the actual (registered) object class of the object instances that were subjects of updates, which may not be the same as the object classes by which they are known to the federate.

PARAMETERS

TransportationType

a string equal to “Reliable” or “Best Effort” depending on the transportation service category being reported

UpdateCounts

a comma-delimited list of pairs of the form “<class>/<count>” where *class* is an object class

handle and *count* is the number of updates that have been initiated by the federate for instances of the class; only classes that have non-zero counts are listed (null if no updates were sent)

Manager.Federate.Request

SYNOPSIS

```
(class Request reliable receive
  ...
)
```

DESCRIPTION

This class is the root of the hierarchy of MOM interaction classes generated by federates in order to solicit reports of various characteristics of LRC and federate state. This class should not be directly subscribed or instantiated. Subclasses of this class are indented to be generated by federates and reacted to by LRCs. The *Federate* parameter inherited from *Manager.Federate* specifies the LRC recipient of an interaction instance

Manager.Federate.Request.RequestInteractionsReceived

SYNOPSIS

```
(class RequestInteractionsReceived reliable receive)
```

DESCRIPTION

Interactions of this class may be generated by a federate to solicit *Manager.Federate.Report.ReportInteractionsReceived* interactions from an LRC. Two instances of this report will be sent in response: one for best-effort transport and one for reliable transport.

Manager.Federate.Request.RequestInteractionsSent

SYNOPSIS

```
(class RequestInteractionsSent reliable receive)
```

DESCRIPTION

Interactions of this class may be generated by a federate to solicit *Manager.Federate.Report.ReportInteractionsSent* interactions from an LRC. Two instances of this report will be sent in response: one for best-effort transport and one for reliable transport.

Manager.Federate.Request.RequestObjectInformation

SYNOPSIS

```
(class RequestObjectInformation reliable receive
  (parameter ObjectInstance)
)
```

DESCRIPTION

Interactions of this class may be generated by a federate to solicit a *Manager.Federate.Report.ReportObjectInformation* report from an LRC. If the object instance is not known by the target federate, a *ReportObjectInformation* is sent with empty attributes.

PARAMETERS

ObjectInstance
the name of the object instance for which a report is solicited

Manager.Federate.Request.RequestObjectsOwned

SYNOPSIS

```
(class RequestObjectsOwned reliable receive)
```

DESCRIPTION

Interactions of this class may be generated by a federate to solicit a *Manager.Federate.Report.ReportObjectsOwned* report from an LRC.

Manager.Federate.Request.RequestObjectsReflected

SYNOPSIS

```
(class RequestObjectsReflected reliable receive)
```

DESCRIPTION

Interactions of this class may be generated by a federate to solicit a *Manager.Federate.Report.ReportObjectsReflected* report from an LRC.

Manager.Federate.Request.RequestObjectsUpdated

SYNOPSIS

```
(class RequestObjectsUpdated reliable receive)
```

DESCRIPTION

Interactions of this class may be generated by a federate to solicit a *Manager.Federate.Report.ReportObjectsUpdated* report from an LRC.

Manager.Federate.Request.RequestPublications

SYNOPSIS

```
(class RequestPublications reliable receive)
```

DESCRIPTION

Interactions of this class may be generated by a federate to solicit *Manager.Federate.Report.ReportInteractionPublication* and *Manager.Federate.Report.ReportObjectPublication* reports from an LRC. Such a request will result in a single report of the former type, and a separate report of the later type for each object class published by the respondent.

Manager.Federate.Request.RequestReflectionsReceived

SYNOPSIS

```
(class RequestReflectionsReceived reliable receive)
```

DESCRIPTION

Interactions of this class may be generated by a federate to solicit *Manager.Federate.Report.ReportReflectionsReceived* interactions from an LRC. Two instances of this report will be sent in response: one for best-effort transport and one for reliable transport.

Manager.Federate.Request.RequestSubscriptions**SYNOPSIS**

```
(class RequestSubscriptions reliable receive)
```

DESCRIPTION

Interactions of this class may be generated by a federate to solicit *Manager.Federate.Report.ReportInteractionSubscription* and *Manager.Federate.Report.ReportObjectSubscription* reports from an LRC. Such a request will result in a single report of the former type, and a separate report of the later type for each object class published by the respondent.

Manager.Federate.Request.RequestUpdatesSent**SYNOPSIS**

```
(class RequestUpdatesSent reliable receive)
```

DESCRIPTION

Interactions of this class may be generated by a federate to solicit *Manager.Federate.Report.ReportUpdatesSent* interactions from an LRC. Two instances of this report will be sent in response: one for best-effort transport and one for reliable transport.

Manager.Federate.Service**SYNOPSIS**

```
(class Service reliable receive
  ...
)
```

DESCRIPTION

This class is the root of the hierarchy of MOM interaction classes generated by federates in order to invoke RTI ambassador services on remote LRCs. This class should not be directly subscribed or instantiated. Subclasses of this class are intended to be generated by federates and reacted to by LRCs. The *Federate* parameter inherited from *Manager.Federate* specifies the LRC recipient of an interaction instance.

A service invocation made via a subclass of the *Service* interaction class is the same as one made using the local API interface, except that service-call reporting is not done for remote invocations. If an exception occurs as a result of a remote invocation, an Alert report is sent.

Instances of subclasses of the *Service* interaction must include values for all parameters defined for the interaction class. Incomplete interactions will be discarded upon receipt; an Alert report will be sent if at least the *Federate* parameter was provided.

Manager.Federate.Service.ChangeAttributeOrderType**SYNOPSIS**

```
(class ChangeAttributeOrderType reliable receive
  (parameter ObjectInstance)
  (parameter AttributeList)
  (parameter OrderingType)
)
```

DESCRIPTION

Interactions of this class may be generated by a federate to invoke the *changeAttributeOrderType()* service on a remote LRC.

PARAMETERS*ObjectInstance*

the name of the object instance to affected by the service invocation

AttributeList

a comma-delimited list of string-encoded integers suitable for conversion using *atoi()*

OrderingType

the name of the ordering service, either “receive” or “timestamp” (case insensitive)

Manager.Federate.Service.ChangeAttributeTransportationType**SYNOPSIS**

```
(class ChangeAttributeTransportationType reliable
  receive
  (parameter ObjectInstance)
  (parameter AttributeList)
  (parameter TransportationType)
)
```

DESCRIPTION

Interactions of this class may be generated by a federate to invoke the *changeAttributeTransportationType()* service on a remote LRC.

PARAMETERS*ObjectInstance*

the name of the object instance to be affected by the service invocation

AttributeList

a comma-delimited list of string-encoded integers suitable for conversion using *atoi()*

TransportationType

the name of the transportation service, either “best_effort” or “reliable” (case insensitive)

Manager.Federate.Service.ChangeInteractionOrderType**SYNOPSIS**

```
(class ChangeInteractionOrderType reliable receive
  (parameter InteractionClass)
  (parameter OrderingType)
)
```

DESCRIPTION

Interactions of this class may be generated by a federate to invoke the *changeInteractionOrderType()* service on a remote LRC.

PARAMETERS*InteractionClass*

a string-encoded integer, suitable for conversion using *atoi()*, representing an interaction class handle.

OrderingType

the name of the ordering service, either “receive” or “timestamp” (case insensitive)

Manager.Federate.Service.ChangeInteractionTransportationType**SYNOPSIS**

```
(class ChangeInteractionTransportationType reliable
  receive
  (parameter InteractionClass)
  (parameter TransportationType)
```

)

DESCRIPTION

Interactions of this class may be generated by a federate to invoke the `changeInteractionTransportationType()` service on a remote LRC.

PARAMETERS*InteractionClass*

a string-encoded integer, suitable for conversion using `atol()`, representing an interaction class handle

TransportationType

the name of the transportation service, either "best_effort" or "reliable" (case insensitive)

Manager.Federate.Service.DeleteObjectInstance**SYNOPSIS**

```
(class DeleteObjectInstance reliable receive
  (parameter ObjectInstance)
  (parameter FederationTime)
  (parameter Tag)
)
```

DESCRIPTION

Interactions of this class may be generated by a federate to invoke the `deleteObjectInstance()` service on a remote LRC.

PARAMETERS*ObjectInstance*

the name of the object instance to be affected by the service invocation

FederationTime

federation time parameters are encoded using the `RTI::FedTime.encode()` method

Tag

a string corresponding to the user-specified field for the service invocation

Manager.Federate.Service.DisableAsynchronousDelivery**SYNOPSIS**

```
(class DisableAsynchronousDelivery reliable receive)
```

DESCRIPTION

Interactions of this class may be generated by a federate to invoke the `disableAsynchronousDelivery()` service on a remote LRC.

Manager.Federate.Service.DisableTimeConstrained**SYNOPSIS**

```
(class DisableTimeConstrained reliable receive)
```

DESCRIPTION

Interactions of this class may be generated by a federate to

HLA RTI 1.3-Next Generation

invoke the `disableTimeConstrained()` service on a remote LRC.

Manager.Federate.Service.DisableTimeRegulation**SYNOPSIS**

```
(class DisableTimeRegulation reliable receive)
```

DESCRIPTION

Interactions of this class may be generated by a federate to invoke the `disableTimeRegulation()` service on a remote LRC.

Manager.Federate.Service.EnableAsynchronousDelivery**SYNOPSIS**

```
(class EnableAsynchronousDelivery reliable receive)
```

DESCRIPTION

Interactions of this class may be generated by a federate to invoke the `enableAsynchronousDelivery()` service on a remote LRC.

Manager.Federate.Service.EnableTimeConstrained**SYNOPSIS**

```
(class EnableTimeConstrained reliable receive)
```

DESCRIPTION

Interactions of this class may be generated by a federate to invoke the `enableTimeConstrained()` service on a remote LRC.

Manager.Federate.Service.EnableTimeRegulation**SYNOPSIS**

```
(class EnableTimeRegulation reliable receive
  (parameter FederationTime)
  (parameter Lookahead)
)
```

DESCRIPTION

Interactions of this class may be generated by a federate to invoke the `enableTimeRegulation()` service on a remote LRC.

PARAMETERS*FederationTime*

federation time parameters are encoded using the `RTI::FedTime.encode()` method

Lookahead

federation time parameters are encoded using the `RTI::FedTime.encode()` method

Manager.Federate.Service.FederateRestoreComplete**SYNOPSIS**

```
(class FederateRestoreComplete reliable receive
  (parameter SuccessIndicator)
)
```

DESCRIPTION

Interactions of this class may be generated by a federate to invoke the `federateRestoreComplete()` service on a remote LRC.

PARAMETERS*SuccessIndicator*

the string “true” or “false” (case insensitive) indicating whether the restoration of federate-managed state succeeded (corresponding to the `federateRestoreComplete()` and `federateRestoreNotComplete()` services, respectively)

Manager.Federate.Service.FederateSaveBegun**SYNOPSIS**

```
(class FederateSaveBegun reliable receive)
```

DESCRIPTION

Interactions of this class may be generated by a federate to invoke the `federateSaveBegun()` service on a remote LRC.

Manager.Federate.Service.FederateSaveComplete**SYNOPSIS**

```
(class FederateSaveComplete reliable receive
 (parameter SuccessIndicator)
 )
```

DESCRIPTION

Interactions of this class may be generated by a federate to invoke the `federateSaveComplete()` service on a remote LRC.

PARAMETERS*SuccessIndicator*

the string “true” or “false” (case insensitive) indicating whether the save of federate-managed state succeeded (corresponding to the `federateSaveComplete()` and `federateSaveNotComplete()` services, respectively)

Manager.Federate.Service.FlushQueueRequest**SYNOPSIS**

```
(class FlushQueueRequest reliable receive
 (parameter FederationTime)
 )
```

DESCRIPTION

Interactions of this class may be generated by a federate to invoke the `flushQueueRequest()` service on a remote LRC.

PARAMETERS*FederationTime*

federation time parameters are encoded using the `RTI::FedTime.encode()` method

Manager.Federate.Service.LocalDeleteObjectInstance**SYNOPSIS**

```
(class LocalDeleteObjectInstance reliable receive
 (parameter ObjectInstance)
 )
```

*HLA RTI 1.3-Next Generation***DESCRIPTION**

Interactions of this class may be generated by a federate to invoke the `localDeleteObjectInstance()` service on a remote LRC.

PARAMETERS*ObjectInstance*

the name of the object instance to be affected by the service invocation

Manager.Federate.Service.ModifyLookahead**SYNOPSIS**

```
(class ModifyLookahead reliable receive
 (parameter Lookahead)
 )
```

DESCRIPTION

Interactions of this class may be generated by a federate to invoke the `modifyLookahead()` service on a remote LRC.

PARAMETERS*Lookahead*

Federation Time parameters are encoded using the `RTI::FedTime.encode()` method.

Manager.Federate.Service.NextEventRequest**SYNOPSIS**

```
(class NextEventRequest reliable receive
 (parameter FederationTime)
 )
```

DESCRIPTION

Interactions of this class may be generated by a federate to invoke the `nextEventRequest()` service on a remote LRC.

PARAMETERS*FederationTime*

federation time parameters are encoded using the `RTI::FedTime.encode()` method

Manager.Federate.Service.NextEventRequestAvailable**SYNOPSIS**

```
(class NextEventRequestAvailable reliable receive
 (parameter FederationTime)
 )
```

DESCRIPTION

Interactions of this class may be generated by a federate to invoke the `nextEventRequestAvailable()` service on a remote LRC.

PARAMETERS*FederationTime*

federation time parameters are encoded using the `RTI::FedTime.encode()` method

Manager.Federate.Service.PublishInteractionClass**SYNOPSIS**

```
(class PublishInteractionClass reliable receive
  (parameter InteractionClass)
)
```

DESCRIPTION

Interactions of this class may be generated by a federate to invoke the `publishInteractionClass()` service on a remote LRC.

PARAMETERS

InteractionClass

a string-encoded integer, suitable for conversion using `atol()`, representing an interaction class handle

Manager.Federate.Service.PublishObjectClass**SYNOPSIS**

```
(class PublishObjectClass reliable receive
  (parameter ObjectClass)
  (parameter AttributeList)
)
```

DESCRIPTION

Interactions of this class may be generated by a federate to invoke the `publishObjectClass()` service on a remote LRC.

PARAMETERS

ObjectClass

a string-encoded integer, suitable for conversion using `atol()`, representing an object class handle

AttributeList

a comma-delimited list of string-encoded integers suitable for conversion using `atol()`

Manager.Federate.Service.ResignFederationExecution**SYNOPSIS**

```
(class ResignFederationExecution reliable receive
  (parameter ResignAction)
)
```

DESCRIPTION

Interactions of this class may be generated by a federate to invoke the `resignFederationExecution()` service on a remote LRC.

PARAMETERS

ResignAction

a text string (case insensitive), corresponding to a valid resign action; the value can be one of the following:

“release attributes”

“delete objects”

“delete objects and release attributes”

“no action”

Manager.Federate.Service.SubscribeInteractionClass**SYNOPSIS**

```
(class SubscribeInteractionClass reliable receive
  (parameter InteractionClass)
  (parameter Active)
)
```

HLA RTI 1.3-Next Generation

)

DESCRIPTION

Interactions of this class may be generated by a federate to invoke the `subscribeInteractionClass()` service on a remote LRC.

PARAMETERS

InteractionClass

a string-encoded integer, suitable for conversion using `atol()`, representing an interaction class handle

Active

a string equal to “true” or “false” (case insensitive) indicating corresponding to an active or passive subscription, respectively

Manager.Federate.Service.SubscribeObjectClassAttributes**SYNOPSIS**

```
(class SubscribeObjectClassAttributes reliable receive
  (parameter ObjectClass)
  (parameter AttributeList)
  (parameter Active)
)
```

DESCRIPTION

Interactions of this class may be generated by a federate to invoke the `subscribeObjectClassAttributes()` service on a remote LRC.

PARAMETERS

ObjectClass

a string-encoded integer, suitable for conversion using `atol()`, representing an object class handle

AttributeList

a comma-delimited list of string-encoded integers suitable for conversion using `atol()`

Active

a string equal to “true” or “false” (case insensitive) indicating corresponding to an active or passive subscription, respectively

Manager.Federate.Service.SynchronizationPointAchieved**SYNOPSIS**

```
(class SynchronizationPointAchieved reliable receive
  (parameter Label)
)
```

DESCRIPTION

Interactions of this class may be generated by a federate to invoke the `synchronizationPointAchieved()` service on a remote LRC.

PARAMETERS

Label

a string uniquely identifying the synchronization point

Manager.Federate.Service.TimeAdvanceRequest**SYNOPSIS**

```
(class TimeAdvanceRequest reliable receive
  (parameter FederationTime)
)
```

DESCRIPTION

Interactions of this class may be generated by a federate to invoke the `timeAdvanceRequest()` service on a remote LRC.

PARAMETERS

FederationTime

federation time parameters are encoded using the `RTI::FedTime.encode()` method

Manager.Federate.Service.TimeAdvanceRequestAvailable**SYNOPSIS**

```
(class TimeAdvanceRequestAvailable reliable receive
  (parameter FederationTime)
)
```

DESCRIPTION

Interactions of this class may be generated by a federate to invoke the `timeAdvanceRequestAvailable()` service on a remote LRC.

PARAMETERS

FederationTime

federation time parameters are encoded using the `RTI::FedTime.encode()` method

Manager.Federate.Service.UnconditionalAttributeOwnershipDivestiture**SYNOPSIS**

```
(class UnconditionalAttributeOwnershipDivestiture
  reliable receive
  (parameter ObjectInstance)
  (parameter AttributeList)
)
```

DESCRIPTION

Interactions of this class may be generated by a federate to invoke the `unconditionalAttributeOwnershipDivestiture()` service on a remote LRC.

PARAMETERS

ObjectInstance

the name of the object instance to be affected by the service invocation

AttributeList

a comma-delimited list of string-encoded integers suitable for conversion using `atol()`

Manager.Federate.Service.UnpublishInteractionClass**SYNOPSIS**

```
(class UnpublishInteractionClass reliable receive
  (parameter InteractionClass)
)
```

DESCRIPTION

Interactions of this class may be generated by a federate to

HLA RTI 1.3-Next Generation

invoke the `unpublishInteractionClass()` service on a remote LRC.

PARAMETERS

InteractionClass

a string-encoded integer, suitable for conversion using `atol()`, representing an interaction class handle

Manager.Federate.Service.UnpublishObjectClass**SYNOPSIS**

```
(class UnpublishObjectClass reliable receive
  (parameter ObjectClass)
)
```

DESCRIPTION

Interactions of this class may be generated by a federate to invoke the `unpublishObjectClass()` service on a remote LRC.

PARAMETERS

ObjectClass

a string-encoded integer, suitable for conversion using `atol()`, representing an object class handle

Manager.Federate.Service.UnsubscribeInteractionClass**SYNOPSIS**

```
(class UnsubscribeInteractionClass reliable receive
  (parameter InteractionClass)
)
```

DESCRIPTION

Interactions of this class may be generated by a federate to invoke the `unsubscribeInteractionClass()` service on a remote LRC.

PARAMETERS

InteractionClass

a string-encoded integer, suitable for conversion using `atol()`, representing an interaction class handle

Manager.Federate.Service.UnsubscribeObjectClass**SYNOPSIS**

```
(class UnsubscribeObjectClass reliable receive
  (parameter ObjectClass)
)
```

DESCRIPTION

Interactions of this class may be generated by a federate to invoke the `unsubscribeObjectClass()` service on a remote LRC.

PARAMETERS

ObjectClass

a string-encoded integer, suitable for conversion using `atol()`, representing an object class handle

11.3 Objects**Manager****SYNOPSIS**

```
(class Manager
  ...
)
```

)

DESCRIPTION

This class is the root of the MOM object class hierarchy. It has no attributes and is not intended to be directly subscribed or instantiated.

Manager.Federate**SYNOPSIS**

```
(class Federate
  (attribute FederateHandle reliable receive)
  (attribute FederateType reliable receive)
  (attribute FederateHost reliable receive)
  (attribute RTIversion reliable receive)
  (attribute FEDid reliable receive)
  (attribute TimeConstrained reliable receive)
  (attribute TimeRegulating reliable receive)
  (attribute AsynchronousDelivery reliable receive)
  (attribute FederateState reliable receive)
  (attribute TimeManagerState reliable receive)
  (attribute FederateTime reliable receive)
  (attribute Lookahead reliable receive)
  (attribute LBTS reliable receive)
  (attribute MinNextEventTime reliable receive)
  (attribute ROLength reliable receive)
  (attribute TSOLength reliable receive)
  (attribute ReflectionsReceived reliable receive)
  (attribute UpdatesSent reliable receive)
  (attribute InteractionsReceived reliable receive)
  (attribute InteractionsSent reliable receive)
  (attribute ObjectsOwned reliable receive)
  (attribute ObjectsUpdated reliable receive)
  (attribute ObjectsReflected reliable receive)
)
```

DESCRIPTION

A single instance of this object class is registered and updated by an LRC on behalf of its federate. Periodic updates are sent out with the frequency specified by the most recent *Manager.Federate.Adjust.SetTiming* interaction received by the LRC. By default, no periodic updates are made.

ATTRIBUTES*FederateHandle*

a string-encoded integer, suitable for conversion using `atoi()`, representing the numeric handle of the federate

FederateType

a string identifying the category of the federate, as provided as an argument to `joinFederationExecution()`

FederateHost

the hostname of the node on which the federate is executing, as determined by the `gethostname()` call

RTIVersion

the string #defined as `RTI_VERSION` in the *RTItypes.hh* file of the RTI library employed by the local federate

FEDid

the FED data-designator, as specified to `createFederationExecution()`

TimeConstrained

“True” if time constraint is enabled for the federate, otherwise “False”

TimeRegulating

“True” if time regulation is enabled for the federate,

otherwise “False”

AsynchronousDelivery

“True” if asynchronous delivery of receive-ordered events is enabled for the federate, otherwise “False”

FederateState

a text string representing the current run state of the federate; it will be one of the following:

“Running”

“Saving”

“Save Pending”

“Restoring”

“Restore Pending”

TimeManagerState

a text string representing the current time-advancement state of the federate; it will be one of the following:

“Idle”

“Advance Pending”

FederateTime

a string, encoded using the `getPrintableString()` method of the `RTI::FedTime` implementation in use by the federate, corresponding to the current logical time of the federate

Lookahead

a string, encoded using the `getPrintableString()` method of the `RTI::FedTime` implementation in use by the federate, corresponding to the length of the current lookahead interval in effect for the federate

LBTS

a string, encoded using the `getPrintableString()` method of the `RTI::FedTime` implementation in use by the federate, corresponding to the current federation lower-bound time-stamp from the perspective of the local federate

MinNextEventTime

a string, encoded using the `getPrintableString()` method of the `RTI::FedTime` implementation in use by the federate, corresponding to the current minimum next-event time from the perspective of the local federate

ROLength

a string-encoded integer representing the number of events queued for receive-ordered delivery

TSOLength

a string-encoded integer representing the number of events currently queued for time-stamp-ordered delivery

ReflectionsReceived

a string-encoded integer representing the number of reflections delivered to the local federate

UpdatesSent

a string-encoded integer representing the number of updates initiated by the local federate

InteractionsReceived

a string-encoded integer representing the number of interactions delivered to the local federate (best-effort and reliable combined)

InteractionsSent

a string-encoded integer representing the number of interactions initiated by the local federate (best-effort and reliable combined)

ObjectsOwned

a string-encoded integer representing the number of object instances for which the local federate holds the privilege to delete (all object classes combined)

ObjectsUpdated

a string-encoded integer representing the number of object instances for which there exist one or more instance-attributes that the local federate owns and has been advised to update

ObjectsReflected

a string-encoded integer representing the number of object instances for which the local federate reflects updates of at least one attribute (best-effort and reliable combined)

ATTRIBUTES*FederationName*

the unique string name of the federation, as specified to `createFederationExecution()`

FederatesInFederation

a comma-delimited list of string-encoded integers suitable for conversion using `atol()`

RTIversion

the string #defined as `RTI_VERSION` in the *RTItypes.hh* file of the RTI library employed by the federation

FEDid

the FED data-designator, as specified to `createFederationExecution()`

LastSaveName

the label associated with the most-recently completed federation save (or the empty string if no saves have been completed)

LastSaveTime

the logical time, encoded using `FedTimeFactory::getPrintableString()`, associated with the most recently completed federation save (or time zero if no saves have been completed or the most recently completed save was not associated with a logical time)

NextSaveName

the label associated with the currently pending federation save (or the empty string if no save is currently pending)

NextSaveTime

the logical time, encoded using `FedTimeFactory::getPrintableString()`, associated with the currently pending federation save (or time zero if no save is pending or the pending save is not associated with a logical time)

Manager.Federation**SYNOPSIS**

```
(class Federation
  (attribute FederationName reliable receive)
  (attribute FederatesInFederation reliable receive)
  (attribute RTIversion reliable receive)
  (attribute FEDid reliable receive)
  (attribute LastSaveName reliable receive)
  (attribute LastSaveTime reliable receive)
  (attribute NextSaveName reliable receive)
  (attribute NextSaveTime reliable receive)
)
```

DESCRIPTION

A single instance of this object class is registered and updated by the federation (in reality, each LRC locally maintains the state of this object for the benefit of its local federate.) The *Federation* object instance is updated upon request.

12. Migration Document

12.1 Introduction to Migrating RTI 1.3v6 Federates to RTI 1.3-NG

This section is intended as a “quick start guide” to making the transition from RTI 1.3v6 to 1.3-NG as smooth as possible by pointing out the most commonly encountered differences and pitfalls. It is not intended to introduce features and functionality exclusive to RTI version 1.3-NG, nor is it intended to be an exhaustive list of differences between RTI 1.3v6 and RTI 1.3-NG.

12.2 Management Object Model

12.2.1. General notes

The Manager.Federate.Service interaction `ChangeAttributeTransportType` was corrected to be `ChangeAttributeTransportationType`.

The Manager.Federate.Service interaction `ChangeInteractionTransportType` was corrected to be `ChangeInteractionTransportationType`.

The Manager.Federate.Service interactions `RegisterFederationSynchronizationPoint`, `RequestFederationRestore` and `RequestFederationSave` were removed because they did not appear in the interface specification.

The Manager.Federate.Service interactions `FederateRestoreComplete`, `FederateSaveBegun` and `FederateSaveComplete` currently cannot be used without throwing an exception.

12.2.2. Manager.Federate.Adjust.ModifyAttributeState

An attribute must be unowned in order for a federate to assume ownership via this service.

12.2.3. Manager.Federate.Adjust.SetExceptionLogging

Now, only exceptions are logged to the file. Its default state is off.

12.2.4. Manager.Federate.Report.Alert

This is sent only when exceptions occur.

`AlertSeverity` has only three possible values instead of five. These values are now strings which will be one of the following:

- “RTI exception”
- “RTI internal error”
- “Federate internal error”

`AlertID` now represents a count which is incremented after each alert is sent.

12.2.5. Manager.Federate.Report.ReportObjectSubscription

The `Active` field will now contain "True" or "False" instead of "Active" or "Passive", respectively.

12.2.6. Manager.Federate.Report.ReportServiceInvocation

All parameters will be included in each `ReportServiceInvocation`, not just those that are relevant.

All RTI::FedTime types are now represented as the string returned by RTI::FedTime.getPrintableString()

All RTI::Region types are now represented as the memory address of the region instance.

The ExceptionID parameter will always contain a zero when SuccessIndicator is false, or null when SuccessIndicator is true.

Note: New to v1.1: A federate will now report an invocation to the joinFederationExecution service if the reporting federate's RID file has the RID parameter FederationSection.MOM.EnableServiceReporting set to Yes."

12.2.7. Manager.Federate object

No initial update of this object is sent out automatically; the user must first request an attribute update.

FederateState currently can never have the values of save pending, restoring or restore pending because no object updates can be sent during those states.

12.3 RTI Initialization Data (Extracted from the RTI.rid file)

12.3.1. Introduction to the RTI 1.3-NG RTI Initialization Data file

This file contains configuration parameters that control the operation of the RTI software. All parameters have a default setting that is used in the event that a parameter value is not specified in the RID file or a RID file is not specified. If a RID file is not present, RTI 1.3-NG will use the directory from which the RtiExec was launched as the current directory for launch of the FedExec and for saving information during a save or restore invocation. In addition, the user should expect to see multiple warning messages. Unless a minimum RTI.rid file is used, directing the RTI to turn off warnings, the user can expect to see multiple warning messages printed to the screen. These messages will be in the following format:

"<File>", line <x>: <RID param> not found in RID, using default value <RID value>.

Example Warning Message:

"G:\Release_Views\ng_v1.1_dev_nt_vc\rti\priv\pkg\interactionMgt\priv\src\RtiInteractionManagerIncoming.cpp", line 174: Valid value for Advisories.

InteractionRelevanceAdvisorySwitchDefault not found in RID. Valid values are [Enabled | Disabled]. Using value of `ENABLED`

The following snippet would represent the minimum RTI.rid file that would use default settings without displaying default parameter warnings to the screen:

```
(RTIdebug
  (WarningMessages
    (ViewPackageSet 0xffffffff7fff)
  )
) ;; End of RTI_Debug
```

12.3.2. File Location

The RTI-NG software looks for the environment variable, `RTI_RID_FILE`, which defines the name and location of the RID file to be used by the application. The file location may be absolute or relative using the appropriate convention for the particular operating system. The file name is not required to have a special name or prefix, it only needs to be readable by the application and provide the correct syntax.

If the `RTI_RID_FILE` environment variable is not set, the `rtiexec` process will attempt to open a file named "RTI.rid" in the directory from which the application was launched.

12.3.3. File Format

The format used for the RID file has several rules related to valid parsing of the file. The first rule is that any text to the right of the comment token, (two semi-colons “;;”), is ignored by the parser. The second rule is that the left and right parentheses are used for scoping, and must always be used in matching pairs.

Within a pair of parentheses, there can be either the scope name or a parameter name and value pair. The scope name is used to organize parameters that are conceptually related and to ensure uniqueness (in case a parameter name is used multiple times within different scopes). If a parameter name is not unique, only the last value will be used for the scoping. The parameter name is case insensitive. The value is parsed as a character string and subsequently interpreted according to the particular parameter type (e.g., integer, floating point, string).

12.3.4. File Parameter Scoping

Each RID parameter is identified by a scope name in which the scoping is broken into three major categories according to the granularity of the internal RTI components. The RTI-NG instantiates components when an RTI process is initially started (the first create or join), when a federation comes into existence within the process (first create or join of a new federation), and when a particular federate joins a federation. These scope names are defined below.

ProcessSection - process level component parameters
 FederationSection - federation level component parameters
 FederateSection - federate level component parameters

It is possible that a RID file used by a particular application will need to support multiple federations and federates within a single process using different RID parameter values for each federation or federate. This RID structure can support this situation by creating a scope within the federation or federate section with the scope name the same as the name of the federation or name of the federate, respectively.

As an example, assume that an application needs to support two different federations named FederationA and FederationB. The RID parameter for the multicast base address for FederationB needs to be different from the address of all other federations. An example RID is shown below where the BaseAddress used for FederationB is "224.100.0.1" and for all other federations the value is "224.2.0.1".

```
(FederationSection
...
(BaseAddress 224.2.0.1)
```

```

...
(FederationB
...
(BaseAddress 224.100.0.1)
...
)
)

```

12.3.5. Parameter Definition

Each parameter contained in the RID file provides a description of the effect that the parameter value has on the operation of the RTI. The RANGE defines the valid parameter values and the DEFAULT VALUE defines the default value. As previously mentioned, if the parameter and value is not specified within the RID file the default value will be used by the RTI.

(RTI

```

;; The RTI scope serves as a namespace for the RID user parameters. No
;; parameter entries should be made at this level.

```

(ProcessSection

```

;; Entries in this section apply to the process level components.

```

(RtiExecutive

```

;; The RTI Executive is a logically centralized process that is used as a
;; network wide resource manager to handle such items as the uniqueness of
;; federation names. It is logically centralized since redundant processes
;; can be used for fault tolerance (although this feature is currently not
;; supported). The parameters associated with the RTI Executive control
;; how the process is found on the network.

```

```

;; PARAMETER: ProcessSection.RtiExecutive.RtiExecutiveEndpoint
;; DESCRIPTION: The RTI Executive endpoint defines the network address and
;; port number used by the RTI Executive process (and hence the RTI Naming
;; Service). The network address can be a hostname or an IP address. The
;; endpoint is only necessary when the multicast discovery mechanism is not
;; used and the endpoint must match the value provided when the RTI Executive
;; process is started.
;; RANGE: A valid hostname or IP address followed by a colon and then the
;; port number.
;; DEFAULT VALUE: None, will use multicast discovery mechanism.
;;
;;; (RtiExecutiveEndpoint hostname:port)

```

```

;; PARAMETER: ProcessSection.RtiExecutive.
;; RtiExecutiveMulticastDiscoveryEndpoint

```

```
:: DESCRIPTION: The RTI Executive discovery parameter defines the multicast
:: address and port number used for the multicast discovery protocol to find
:: the RTI Naming Service which is located in the RTI Executive process
:: The naming service will then enable the application to locate distributed
:: RTI components (e.g., RTI Executive).
:: RANGE: A valid multicast IP address (or hostname) followed by a colon and
:: then the port number.
:: DEFAULT VALUE: 224.9.9.2:22605
::
;;; (RtiExecutiveMulticastDiscoveryEndpoint 224.9.9.2:22605)

:: PARAMETER: ProcessSection.RtiExecutive.NumberOfAttemptsToFindRtiExecutive
:: DESCRIPTION: The NumberOfAttemptsToFindRtiExecutive parameter is used to
:: control how many attempts the application should use to locate the RTI
:: Naming Service using the multicast discovery mechanism.
:: RANGE: An integer value greater than zero.
:: DEFAULT VALUE: 10
::
;;; (NumberOfAttemptsToFindRtiExecutive 10)

:: PARAMETER: ProcessSection.RtiExecutive.TimeToWaitAfterEachAttemptInSeconds
:: DESCRIPTION: The TimeToWaitAfterEachAttemptInSeconds parameter is used to
:: control how long the application should wait between attempts to find the
:: RTI Executive using the multicast discovery mechanism.
:: RANGE: A floating point value greater than zero.
:: DEFAULT VALUE: 2.0
::
;;; (TimeToWaitAfterEachAttemptInSeconds 2.0)
) ;; End of ProcessSection.RtiExecutive
```

(Networking

```
:: The Networking section is used to define the communication configuration
:: information associated with all of the RTI components within the
:: application using this RID file.

:: PARAMETER: ProcessSection.Networking.FederateEndpoint
:: DESCRIPTION: The Networking endpoint defines the network address and port
:: number used by the federate application process using this RID file. The
:: network address can be a hostname or an IP address. The federate endpoint
:: is used by other distributed RTI components to communicate with internal
:: modules within this application. Typically the federate endpoint does not
```

:: need to be defined unless the computer has multiple network interfaces.
:: If an environmental variable named RTI_FEDERATE_ENDPOINT is found, its
:: value will be used in favor of what is specified here.
:: RANGE: A valid hostname or IP address followed by a colon and then the
:: port number.
:: DEFAULT VALUE: The default network card and the port.
::
::: (FederateEndpoint hostname:port)

(MulticastOptions

:: The networking multicast options define the parameters that control the
:: behavior of UDP communication within the RTI that is used for Best Effort
:: transport.

:: PARAMETER: ProcessSection.Networking.MulticastOptions.Interface
:: DESCRIPTION: The Interface is used to specify which ethernet
:: interface shall be used to send and receive multicast traffic. On
:: most systems the possible interfaces can be listed with the netstat
:: command). If no interface is specified, the default is used.
:: NOTE: This parameter does not effect multicast name service discovery.

:: DEFAULT VALUE: None.
::
::: (Interface "eth0")

(Fragmentation

:: The UDP communication protocol (used for Best Effort transport) does
:: not fragment and reassemble data. For messages larger than the UDP
:: fragmentation size the RTI must fragment the message into smaller
:: packets on the send side and then reassemble the packets on the
:: receiver side.

:: PARAMETER: ProcessSection.Networking.MulticastOptions.Fragmentation.FragmentSize
:: DESCRIPTION: The FragmentSize is used to define the maximum number of
:: bytes that can be used as the payload in a UDP packet. Different
:: networks may be capable of supporting different UDP Maximum Transfer
:: Unit (MTU) values.
:: RANGE: An integer greater than zero representing the number of bytes.
:: DEFAULT VALUE: 62000
::
::: (FragmentSize 62000)

```

;; PARAMETER: ProcessSection.Networking.MulticastOptions.Fragmentation.ReassemblyTimerIntervalInSeconds
;; DESCRIPTION: TheReassmeblyTimerIntervalInSeconds parameter is used to control how long
;; control how long the receiver will wait to receive all of the fragments
;; that make up a single message. Since UDP is not a reliable communication
;; protocol the fragments can be lost and the receiver needs to know how long
;; to wait before discarding incomplete fragments. For performance reasons
;; the RTI does not create a timer for each fragment set, instead a common
;; timer is used and each incomplete fragment set is incremented and removed
;; after MaxTimeouts.
;; RANGE: A floating point value greater than zero.
;; DEFAULT VALUE: 1.0
;;
;;; (ReassemblyTimerIntervalInSeconds 1.0)

;; PARAMETER: ProcessSection.Networking.MulticastOptions.Fragmentation.MaxTimeouts
;; DESCRIPTION: The MaxTimeouts parameter is used to control how long the
;; receiver will wait to receive all of the fragments that make up a
;; single message. Since UDP is not a reliable communication protocol the
;; fragments can be lost and the receiver needs to know how long to wait
;; before discarding incomplete fragments. For performance reasons the RTI
;; does not create a timer for each fragment set, instead a common timer is
;; used and each incomplete fragment set is incremented and removed after
;; MaxTimeouts.
;; RANGE: An integer value greater than zero.
;; DEFAULT VALUE: 3
;;
;;; (MaxTimeouts 3)
) ;; End of ProcessSection.Networking.MulticastOptions.Fragmentation
) ;; End of ProcessSection.Networking.MulticastOptions
) ;; End of ProcessSection.Networking

```

(ProcessModel

```

;; The process model controls the mechanism used by the RTI to obtain
;; processing cycles and support callbacks to the federate during the tick
;; call.

```

```

;; PARAMETER: ProcessSection.ProcessModel.StrategyToUse
;; DESCRIPTION: There are currently two process model strategies that are
;; supported by the RTI; (1) polling process model and (2) asynchronous I/O
;; process model. The polling process model uses a single thread of

```

```

;; execution shared between the RTI and the federate. Only when the federate
;; calls tick is the RTI able to perform work. This strategy can starve the
;; RTI if tick is not called appropriately. The asynchronous I/O process
;; model uses an internal thread within the RTI to avoid starvation. This
;; thread will periodically wake up and determine if it can perform any
;; internal RTI work. In the asynchronous I/O strategy the federate only
;; needs to invoke tick when it is prepared to handle callbacks.
;; RANGE: An enumeration value {Polling, AsynchronousIO}.
;; DEFAULT VALUE: AsynchronousIO

```

```

;;; (StrategyToUse AsynchronousIO)
) ;; End of ProcessSection.ProcessModel

```

(Scheduler

```

;; The Scheduler section contains parameters associated with the behavior of
;; the tick service.

```

```

;; PARAMETER: ProcessSection.Scheduler.SingleCallbackPerTick
;; DESCRIPTION: When using the tick service without the minimum and maximum
;; time arguments the RTI can be directed to return a single callback or
;; provide all available callbacks. Using this service with a setting of No
;; indicates to the RTI that all available callbacks should be delivered to
;; the federate in a single tick() call.
;; RANGE: An enumeration value {Yes, No}.
;; DEFAULT VALUE: No.
;;
;;; (SingleCallbackPerTick No)

```

```

) ;; End of ProcessSection.Scheduler
) ;; End of ProcessSection

```

(FederationSection

```

;; Entries in this section apply to the federation level components.

```

(FederationExecutive

```

;; The FederationExecutive section contains parameters related to the
;; federation executive process that is launched when a federation is
;; created.

```

```

;; PARAMETER: FederationSection.FederationExecutive.FederationExecutiveEndpoint
;; DESCRIPTION: The Federation Executive endpoint defines the network address

```

```

;; and port number used by the Federation Executive process. The network
;; address can be a hostname or an IP address. Typically, the endpoint only
;; needs to be defined when the Federation Executive needs to use the
;; non-default network interface, or when the Federation Executive (i.e.,
;; fedex) is to run on a different host than the RTI Executive.
;; RANGE: A valid hostname or IP address followed by a colon and then the
;; port number.
;; DEFAULT VALUE: Will use same default hostname on the host where the RTI
;; Executive is running, with a system generated port number.
;;
;;; (FederationExecutiveEndpoint hostname:port)

```

```

;; PARAMETER: FederationSection.FederationExecutive.FileNameOfFederationExecutiveExecutable
;; DESCRIPTION: This parameter defines the relative or absolute path to the
;; Federation Executive executable (i.e., fedex). The relative path is
;; defined relative to the location of the RTI Executive.
;; RANGE: A valid path to the fedex executable.
;; DEFAULT VALUE: fedex
;;
;;; (FileNameOfFederationExecutiveExecutable fedex)

```

```

;; PARAMETER:
FederationSection.FederationExecutive.TimeToWaitBeforeCommunicatingWithFederationExecutiveInSeconds
;; DESCRIPTION: This parameter is used to allow the federate application to
;; wait a period of time before trying to connect to the Federation Executive
;; process. A small delay may be necessary when the Federation Executive is
;; being created.
;; RANGE: An integer greater than or equal to zero.
;; DEFAULT VALUE: 3
;;
;;; (TimeToWaitBeforeCommunicatingWithFederationExecutiveInSeconds 3)

```

```

;; PARAMETER: FederationSection.FederationExecutive.NumberOfAttemptsToFindFederationExecutive
;; DESCRIPTION: The federate application may attempt to connect to the
;; Federation Executive process multiple times.
;; RANGE: An integer number greater than zero.
;; DEFAULT VALUE: 10
;;
;;; (NumberOfAttemptsToFindFederationExecutive 10)

```

```

;; PARAMETER: FederationSection.FederationExecutive.TimeToWaitAfterEachAttemptInSeconds
;; DESCRIPTION: When the federate application fails to initially connect to

```

```
;; the Federation Executive process it can wait a fixed period of time.
;; RANGE: A floating point number greater than or equal to zero.
;; DEFAULT VALUE: 2.0
;;
;;; (TimeToWaitAfterEachAttemptInSeconds 2.0)

;; PARAMETER: FederationSection.FederationExecutive.FilenameToRedirectStdout
;; DESCRIPTION: This parameter can be used to direct standard output to a
;; file, rather than the default output device.
;; RANGE: A valid filename.
;; DEFAULT VALUE: None, will use the standard output device.
;;
;;; (FilenameToRedirectStdout fedex.stdout)

;; PARAMETER: FederationSection.FederationExecutive.FilenameToRedirectStderr
;; DESCRIPTION: This parameter can be used to direct standard error to a
;; file, rather than the default error device.
;; RANGE: A valid filename.
;; DEFAULT VALUE: None, will use the standard error device.
;;
;;; (FilenameToRedirectStderr fedex.stderr)

;; PARAMETER: FederationSection.FederationExecutive.DirectoryForSaveAndRestoreFiles
;; DESCRIPTION: This parameter provides the pathname to the directory used
;; when producing new saved files or processing existing saved files.
;; RANGE: A valid directory pathname.
;; DEFAULT_VALUE: .
;;
;;; (DirectoryForSaveAndRestoreFiles .)

) ;; End of FederationSection.FederationExecutive

;; PARAMETER: FederationSection.TimeIntervalToCheckForUnresponsiveFederationInSeconds
;; DESCRIPTION: The RTI Executive employs a simple heartbeat model as a means
;; to clear the name of an unresponsive federation from the RTI Naming
;; Service so that the name can be reused. This parameter sets the time
;; interval in seconds at which the RTI Executive checks to see whether it has
;; heard from the Federation Executive.
;; RANGE: A floating point value greater than zero.
;; DEFAULT_VALUE: 60.0
;;
```

```

;;; (TimeIntervalToCheckForUnresponsiveFederationInSeconds 60.0)

;; PARAMETER: FederationSection.TimeToWaitBeforeDeclaringFederationDeadInSeconds
;; DESCRIPTION: The RTI Executive employs a simple heartbeat model as a means
;; to clear the name of an unresponsive federation from the RTI Naming Service
;; so that the name can be reused. This parameter sets the time to wait in
;; seconds before the RTI Executive removes the name of a federation from
;; which it has not received a ping reply.
;; RANGE: A floating point value greater than zero.
;; DEFAULT_VALUE: 90.0
;;
;;; (TimeToWaitBeforeDeclaringFederationDeadInSeconds 90.0)

```

(FederationInterconnect

```

;; The FederationInterconnect section contains parameters associated with the
;; configuration of reliable transport configuration channels.

```

```

;; PARAMETER: FederationSection.FederationInterconnect.StrategyToUse
;; DESCRIPTION: The interconnection of RTI nodes for reliable traffic can use
;; either a CollocatedEventChannel or a CentralizedEventChannel strategy. The
;; CollocatedEventChannel strategy allows each RTI node (i.e., each federate
;; application) to contain a "TCP exploder" that is connected to every other
;; node. The CentralizedEventChannel strategy uses a single node to receive
;; all of the federation reliable traffic and transmit this data to the
;; interested receivers. The CollocatedEventChannel adds some processing
;; requirements to each node to perform the TCP writes, but it avoids latency
;; when going through a centralized node which has the processing burden of
;; the entire federation.
;; RANGE: An enumeration value { CollocatedEventChannels,
;; CentralizedEventChannel }
;; DEFAULT VALUE: CollocatedEventChannels
;;
;;; (StrategyToUse CollocatedEventChannels)

```

```

;; PARAMETER: FederationSection.FederationInterconnect.PingIntervalInSeconds
;; DESCRIPTION: Controls the rate at which the interconnect sends ping
;; requests to each node. This is used to determine the "wellness" of the
;; nodes.
;; RANGE: A real number greater than 1.0 or Off
;; DEFAULT VALUE: 5.0
;; Values

```

```
::  
;;; (PingIntervalInSeconds 5.0)  
) ;; End of FederationSection.FederationInterconnect
```

(Networking

```
;; This section contains parameters related to networking information related  
;; to the federation components.
```

(BundlingOptions

```
;; Federation data can be bundled by the sender in order to improve  
;; throughput, at the expense of latency.
```

(UDP

```
;; RTI best effort transport uses User Datagram Protocol (UDP).
```

```
;; PARAMETER: FederationSection.Networking.BundlingOptions.UDP.MaxTimeBeforeSendInSeconds
```

```
;; DESCRIPTION: This parameter is the maximum amount of time that the RTI  
;; will wait before the RTI flushes the data. If set to 0, bundling will  
;; be disabled.
```

```
;; RANGE: A floating point value greater than or equal to zero.
```

```
;; DEFAULT VALUE: 0.005
```

```
::
```

```
;;; (MaxTimeBeforeSendInSeconds 0.005)
```

```
;; PARAMETER: FederationSection.Networking.BundlingOptions.UDP.MaxBytesBeforeSend
```

```
;; DESCRIPTION: This parameter is the maximum number of bytes that will be  
;; bundled before the RTI flushes the data.
```

```
;; RANGE: An integer value greater than or equal to zero.
```

```
;; DEFAULT VALUE: 63000
```

```
::
```

```
;;; (MaxBytesBeforeSend 63000)
```

```
) ;; End of FederationSection.Networking.BundlingOptions.UDP
```

(TCP

```
;; RTI Reliable transport uses Transfer Control Protocol (TCP).
```

```
;; PARAMETER: FederationSection.Networking.BundlingOptions.TCP.MaxTimeBeforeSendInSeconds
```

```
;; DESCRIPTION: This parameter is the maximum amount of time that the RTI  
;; will wait before the RTI flushes the data. If set to 0, bundling will  
;; be disabled.
```

```
;; RANGE: A floating point value greater than or equal to zero.
;; DEFAULT VALUE: 0.005
;;
;;; (MaxTimeBeforeSendInSeconds 0.005)

;; PARAMETER: FederationSection.Networking.BundlingOptions.TCP.MaxBytesBeforeSend
;; DESCRIPTION: This parameter is the maximum number of bytes that will be
;; bundled before the RTI flushes the data.
;; RANGE: An integer value greater than or equal to zero.
;; DEFAULT VALUE: 63000
;;
;;; (MaxBytesBeforeSend 63000)

) ;; End of FederationSection.Networking.BundlingOptions.TCP

) ;; End of FederationSection.Networking.BundlingOptions
```

(MulticastOptions

```
;; The networking multicast options define the parameters that control the
;; behavior of UDP communication within the RTI that is used for Best Effort
;; transport.
```

```
;; PARAMETER: FederationSection.Networking.MulticastOptions.PortNumber
;; DESCRIPTION: The port number of the socket used for sending multicast
;; traffic in support of Best Effort transport.
;; RANGE: An integer value representing a valid port number, or
;; the string RTI-selected to have the RTI select an available port
;; DEFAULT VALUE: RTI-selected
;;
;;; (PortNumber 2000)
```

```
;; PARAMETER: FederationSection.Networking.MulticastOptions.BaseAddress
;; DESCRIPTION: The base network address (IP four decimal address or
;; hostname) for sending multicast traffic. The Best Effort traffic will be
;; segmented into different multicast addresses when using the Data
;; Distribution Management (DDM) services. The maximum number of multicast
;; addresses used by the RTI will be defined by the available addresses
;; between the MaxAddress and the BaseAddress, although the actual number
;; used may be far less (see DDM parameters).
;; RANGE: Any valid IP multicast address, e.g., from 224.0.0.3 to
;; 239.255.255.255
```

```

;; DEFAULT VALUE: 224.1.0.0
;;
;;; (BaseAddress 224.1.0.0)

;; PARAMETER: FederationSection.Networking.MulticastOptions.MaxAddress
;; DESCRIPTION: The maximum network address (IP four decimal address or
;; hostname) for sending multicast traffic. The Best Effort traffic will be
;; segmented into different multicast addresses when using the Distribution
;; Management (DDM) services. The maximum number of multicast addresses
;; used by the RTI will be defined by the available addresses between the
;; MaxAddress and the BaseAddress, although the actual number used may be
;; far less (see DDM parameters).
;; RANGE: Any valid IP multicast address greater than or equal to the base
;; address.
;; DEFAULT VALUE: 239.255.255.255
;;
;;; (MaxAddress 239.255.255.255)

;; PARAMETER: FederationSection.Networking.MulticastOptions.TimeToLive
;; DESCRIPTION: To prevent infinite routing loops, UDP multicast packets are
;; marked with a counter that is decremented each time a router sees the
;; packet. This counter is called TTL (Time To Live). Routers will not
;; pass any packets with a TTL less than 2. Consequently, to pass UDP
;; multicast between LAN's, not only must the routers be properly
;; configured, but the TTL must be at least 2 as well.
;; RANGE: An integer from 0 to 255
;; DEFAULT VALUE: 1
;;
;;; (TimeToLive 1)
) ;; End of FederationSection.Networking.MulticastOptions
) ;; End of FederationSection.Networking

```

(Advisories

```

;; The advisories section contains parameters related to the RTI advisory
;; mechanisms. The scope advisories that inform the federate when certain
;; attributes owned by other federates are in or out of scope with respect to
;; the federate's subscriptions. The relevance advisories inform the
;; publishing federate whether other federates in the federation are
;; interested in particular object or interaction classes and particular
;; object attribute instances.

```

```
:: PARAMETER: FederationSection.Advisories.ClassRelevanceAdvisorySwitchDefault
:: DESCRIPTION: This parameter controls the switch to define if the object
:: class relevance advisory mechanism is enabled when the federate begins.
:: The class relevance advisories will inform the federate if there exists
:: any other federate within the federation that is subscribed to a
:: particular class. The relevance advisories could be beneficial in large
:: scale federations, although there may be significant performance cost in
:: calculating these advisories.
:: RANGE: An enumeration value {Enabled, Disabled}
:: DEFAULT VALUE: Enabled
::
;;; (ClassRelevanceAdvisorySwitchDefault Enabled)
```

```
:: PARAMETER: FederationSection.Advisories.AttributeRelevanceAdvisorySwitchDefault
:: DESCRIPTION: This parameter controls the switch to define if the object
:: attribute relevance advisory mechanism is enabled when the federate
:: begins. The attribute relevance advisories will inform the federate if
:: there exists any other federate within the federation that is subscribed
:: such that they would receive updates from a particular attribute instance.
:: The relevance advisories could be beneficial in large scale federations,
:: although there may be significant performance cost in calculating these
:: advisories.
:: RANGE: An enumeration value {Enabled, Disabled}
:: DEFAULT VALUE: Disabled
::
;;; (AttributeRelevanceAdvisorySwitchDefault Disabled)
```

```
:: PARAMETER: FederationSection.Advisories.AttributeScopeAdvisorySwitchDefault
:: DESCRIPTION: This parameter controls the switch to define if the object
:: attribute scope advisory mechanism is enabled when the federate begins.
:: The attribute scope advisories will inform the federate if a particular
:: attribute instance being updated from another federate matches the
:: federates subscriptions. If the attribute is in scope the federate will
:: receive any updates, and if the attribute is out of scope any updates will
:: not be reflected.
:: RANGE: An enumeration value {Enabled, Disabled}
:: DEFAULT VALUE: Disabled
::
;;; (AttributeScopeAdvisorySwitchDefault Disabled)
```

```
:: PARAMETER: FederationSection.Advisories.InteractionRelevanceAdvisorySwitchDefault
```

```

;; DESCRIPTION: This parameter controls the switch to define if the
;; interaction relevance advisory mechanism is enabled when the federate
;; begins. The interaction relevance advisories will inform the federate if
;; there exists any other federate within the federation that is subscribed
;; to a particular interaction class. The relevance advisories could be
;; beneficial in large scale federations, although there may be significant
;; performance cost in calculating these advisories.
;; RANGE: An enumeration value {Enabled, Disabled}
;; DEFAULT VALUE: Enabled
;;
;;; (InteractionRelevanceAdvisorySwitchDefault Enabled)

;; PARAMETER: FederationSection.Advisories.RelevanceAdvisoryInteractionClassHeartbeatInSeconds
;; DESCRIPTION: The relevance advisory interaction class heartbeat parameter
;; controls how often each federate will broadcast interaction subscription
;; information to the other RTI nodes in order to calculate if there is a
;; federate that has an interest in a particular interaction class. The
;; heartbeat mechanism is used as a more scalable approach than requiring
;; each federate to buffer global subscription knowledge for the entire
;; federation. If interaction relevance advisories are not used within the
;; federation then the value of Off can be used. If the interaction
;; relevance advisories are being used the heartbeat rate should be the
;; maximum delay in receiving the interaction advisory that is tolerable.
;; RANGE: A floating point value greater than zero, or an enumeration value
;; of Off.
;; DEFAULT VALUE: 10.0
;;
;;; (RelevanceAdvisoryInteractionClassHeartbeatInSeconds 10.0)

;; PARAMETER: FederationSection.Advisories.RelevanceAdvisoryInteractionClassTimeoutInSeconds
;; DESCRIPTION: The relevance advisory interaction timeout parameter defines
;; how long a federate will wait to receive a relevance interaction
;; subscription heartbeat that matches a particular interaction class.
;; A match is necessary to inform the federate that there is
;; interest. The heartbeat mechanism is used as a more scalable approach
;; than requiring each federate to buffer global subscription knowledge for
;; the entire federation. If interaction relevance advisories are not used
;; within the federation then the value of Off can be used.
;; If the relevance advisories are being used the timeout rate should be
;; at least twice that of the heartbeat rate.
;; RANGE: A floating point value greater than zero, or an enumeration value

```

```
;; of Off.  
;; DEFAULT VALUE: 30.0  
;;  
;;; (RelevanceAdvisoryInteractionClassTimeoutInSeconds 30.0)  
  
;; PARAMETER: FederationSection.Advisories.RelevanceAdvisoryObjectClassHeartbeatInSeconds  
;; DESCRIPTION: The relevance advisory object class heartbeat parameter  
;; controls how often each federate will broadcast object class subscription  
;; information to the other RTI nodes in order to calculate if there is a  
;; federate that has an interest in a particular object class. The  
;; heartbeat mechanism is used as a more scalable approach than requiring  
;; each federate to buffer global subscription knowledge for the entire  
;; federation. If object class relevance advisories are not used within the  
;; federation then the value of Off can be used. If the object class  
;; relevance advisories are being used the heartbeat rate should be the  
;; maximum delay in receiving the advisory that is tolerable.  
;; RANGE: A floating point value greater than zero, or an enumeration value  
;; of Off.  
;; DEFAULT VALUE: 10.0  
;;  
;;; (RelevanceAdvisoryObjectClassHeartbeatInSeconds 10.0)  
  
;; PARAMETER: FederationSection.Advisories.RelevanceAdvisoryObjectClassTimeoutInSeconds  
;; DESCRIPTION: The relevance advisory object class timeout parameter  
;; defines how long a federate will wait to receive a relevance object  
;; class subscription heartbeat that matches a particular object class.  
;; A match is necessary to inform the federate that there is  
;; interest. The heartbeat mechanism is used as a more scalable approach  
;; than requiring each federate to buffer global subscription knowledge for  
;; the entire federation. If object class relevance advisories are not used  
;; within the federation then the value of Off can be used.  
;; If the relevance advisories are being used the timeout rate should be at  
;; least twice that of the heartbeat rate.  
;; RANGE: A floating point value greater than zero, or an enumeration value  
;; of Off.  
;; DEFAULT VALUE: 30.0  
;;  
;;; (RelevanceAdvisoryObjectClassTimeoutInSeconds 30.0)  
  
;; PARAMETER: FederationSection.Advisories.RelevanceAdvisoryAttributeInstanceHeartbeatInSeconds  
;; DESCRIPTION: The relevance advisory attribute instance heartbeat parameter
```

```
:: controls how often each federate will broadcast attribute instance
:: subscription information to the other RTI nodes in order to calculate if
:: there is a federate that has an interest in a particular attribute
:: instance. The heartbeat mechanism is used as a more scalable approach
:: than requiring each federate to buffer global subscription knowledge for
:: the entire federation. If attribute instance relevance advisories
:: are not used within the federation then the value of Off can be used.
:: If the attribute instance relevance advisories are being used the
:: heartbeat rate should be the maximum delay in receiving the advisory that
:: is tolerable.
:: RANGE: A floating point value greater than zero, or an enumeration value
:: of Off.
:: DEFAULT VALUE: 10.0
::
;;; (RelevanceAdvisoryAttributeInstanceHeartbeatInSeconds 10.0)
```

```
:: PARAMETER: FederationSection.Advisories.RelevanceAdvisoryAttributeInstanceTimeoutInSeconds
:: DESCRIPTION: The relevance advisory attribute instance timeout parameter
:: defines how long a federate will wait to receive a relevance attribute
:: instance subscription heartbeat that matches a particular attribute
:: instance. A match is necessary to inform the federate that there is
:: interest. The heartbeat mechanism is used as a more scalable approach
:: than requiring each federate to buffer global subscription knowledge for
:: the entire federation. If attribute instance relevance advisories
:: are not used within the federation then the value of Off can be used.
:: If the relevance advisories are being used the timeout rate should be at
:: least twice that of the heartbeat rate.
:: RANGE: A floating point value greater than zero, or an enumeration value
:: of Off.
:: DEFAULT VALUE: 30.0
::
;;; (RelevanceAdvisoryAttributeInstanceTimeoutInSeconds 30.0)
```

```
:: PARAMETER: FederationSection.Advisories.ProvideAttributeValueUpdateDelayTimeInSeconds
:: DESCRIPTION: The provide attribute value update delay time parameter
:: defines the amount of time that a federate will "hold onto" a
:: object-instance-level provideAttributeValueUpdate command before
:: delivering it to the federate ambassador. While the command is being
:: held, if identical provideAttributeValueUpdate commands are generated,
:: then only one is eventually delivered. This behavior potentially
:: reduces the number of identical provideAttributeValueUpdate commands a
```

```

;; federate must process in a given time period. Furthermore, the reduced
;; network traffic due fewer resulting updateAttributeValue commands
;; can also greatly improve performance.
;; RANGE: A non-negative floating point value.
;; DEFAULT VALUE: 0.0
;;
;;; (ProvideAttributeValueUpdateDelayTimeInSeconds 0.0)

) ;; End of FederationSection.Advisories

```

(TimeManagement

```

;; This section contains parameters related to the RTI Time Management
;; services.

```

```

;; PARAMETER: FederationSection.TimeManagement.TimeToWaitForLbtsCalculationsBeforeErrorInSeconds
;; DESCRIPTION: This parameter is used to control how long the LBTS (Lower
;; Bound on Time Stamp) calculation will stall before reporting an error.
;; The event synchronization algorithm requires coordination with all of the
;; federates when any federate becomes time regulating. If in the process of
;; calculating the LBTS a federate becomes unresponsive the calculation will
;; stall and report an error after waiting the amount of time defined by this
;; parameter.
;; RANGE: A floating point value greater than zero.
;; DEFAULT VALUE: 10.0
;;
;;; (TimeToWaitForLbtsCalculationsBeforeErrorInSeconds 10.0)

```

```

;; PARAMETER: FederationSection.TimeManagement.
;;           TimeToWaitInBetweenLbtsCalculationsInSeconds
;; DESCRIPTION: This parameter controls how long the synchronization process
;; will wait after completing an LBTS computation before initiating another
;; calculation. A small value will allow the federation to advance logical
;; time more quickly, but will also increase the rate of computations and
;; communications. The value should be based on the maximum rate (advances
;; per seconds of wallclock time) at which the federation would ever be
;; required to advance time.
;; RANGE: A floating point value greater than or equal to zero.
;; DEFAULT VALUE: 0.010
;;
;;; (TimeToWaitInBetweenLbtsCalculationsInSeconds 0.010)

```

) ;; End of FederationSection.TimeManagement

(DataDistribution

;; This section contains parameters related to the RTI Data Distribution
 ;; Management services.

;; PARAMETER: FederationSection.DataDistribution.StrategyToUse
 ;; DESCRIPTION: The routing of federation data using the RTI Data
 ;; Distribution Management (DDM) services can be implemented using different
 ;; techniques. Each technique will have different characteristics that may
 ;; effect the performance of DDM depending on the particular operating
 ;; conditions of the federation. The initial data routing strategies offered
 ;; are:
 ;; Simple - Uses two data channels, one for Best Effort and one for
 ;; Reliable transport. This scheme provides no segmentation of data based on
 ;; DDM usage and is useful for testing purposes.
 ;; StaticSpacePartitioned - Uses two channels per routing space, one for Best
 ;; Effort and one for Reliable transport. This scheme segments data
 ;; according to the routing space that a particular class or attribute is
 ;; assigned to in the RID file. This mechanism provides a simplification to
 ;; the grid partitioned scheme in which segmentation is only required based on
 ;; the routing space, not the particular update and subscription regions
 ;; within a space.
 ;; StaticGridPartitioned - Uses two channels (Best Effort and Reliable) for
 ;; each hypercube formed by partitioning each dimension of a routing space in a
 ;; grid like fashion. The number of partitions in each dimension is user
 ;; defined. This scheme offers the federation control of how to statically
 ;; segment federation traffic for their particular exercise.
 ;; RANGE: An enumeration value {Simple, StaticSpacePartitioned,
 ;; StaticGridPartioned}.
 ;; DEFAULT VALUE: StaticGridPartitioned
 ;;
 ;;;; (StrategyToUse StaticGridPartitioned)

(Options

;; This section contains parameters associated with the DDM implementations.

;; PARAMETER: FederationSection.DataDistribution.BestEffortChannelType
 ;; DESCRIPTION: This parameter allows for the specification of the
 ;; particular channel type for all Best Effort traffic. Currently a UDP
 ;; multicast and TCP channel type are supported. Additional channel types

```

;; may be available in future releases.
;; RANGE: An enumeration value {UDPMulticast, TCP}.
;; DEFAULT_VALUE: UDPMulticast
;;
;;; (BestEffortChannelType UDPMulticast)

;; PARAMETER: FederationSection.DataDistribution.ReliableChannelType
;; DESCRIPTION: This parameter allows for the specification of the
;; particular channel type for all Reliable traffic. Currently a UDP
;; multicast and TCP channel type are supported. Additional channel types
;; may be available in future releases.
;; RANGE: An enumeration value {UDPMulticast, TCP}.
;; DEFAULT_VALUE: TCP
;;
;;; (ReliableChannelType TCP)

```

(StaticGridPartitionedStrategyOptions

```

;; This section contains parameters related to the static grid strategy.

```

```

;; PARAMETER:
FederationSection.DataDistribution.StaticGridPartitionedStrategyOptions.MaxNumberOfDataChannelsToUse
;; DESCRIPTION: This parameter is used to define the maximum number of data
;; channels to be used by the segmentation of all routing spaces. A
;; larger number of channels may provide more effective segmentation of the
;; federation data, but for Best Effort traffic using UDP the segmentation
;; can be limited by the number of available multicast addresses (see
;; MulticastOptions in the FederationSection). If the maximum number of
;; channels exceed the available multicast addresses the algorithm will
;; reuse the addresses and therefore reducing segmentation efficiency.
;; The TCP mechanism for Reliable traffic has a 2^32 limit on the number of
;; channels due to the fact that a 32 bit quantity is used to address the
;; data.
;; RANGE: An integer value greater than zero.
;; DEFAULT_VALUE: 64
;;
;;; (MaxNumberOfDataChannelsToUse 64)

```

```

;; PARAMETER:
FederationSection.DataDistribution.StaticGridPartitionedStrategyOptions.NumPartitionsPerDimension
;; DESCRIPTION: This parameter is used to define the default number of
;; partitions that are used for each dimension to segment the space. A
;; larger number of partitions will increase segmentation (provided there

```

```

;; is not a limit with the number of channels to use or a limit in the
;; number of multicast addresses for UDP). Further refinement to the space
;; partitioning can be accomplished by having different decimation for
;; different dimensions in different spaces using the parameters in the
;; SpaceOptions section.
;; RANGE: An integer value greater than zero.
;; DEFAULT_VALUE: 1
;;
;;; (NumPartitionsPerDimension 1)

```

(SpaceOptions

```

;; The SpaceOptions Section enables the number of partitions on any given
;; dimension within any space to override the default. Note that to
;; control these parameters they need to be defined based on the
;; particular routing space names used in the FED (Federation Execution
;; Data) file. The examples below are shown for two spaces "SpaceA" and
;; "SpaceB", but these names would have to be replaced for use within a
;; particular federation. The NumPartitions parameter is associated with a
;; particular numbered dimension (e.g., dimension1, dimension2) and
;; defines the decimation for that dimension in the appropriate routing
;; space.

```

(SpaceA

(DimensionOptions

```

;;; (dimension1 (NumPartions 2))

```

```

;;; (dimension2 (NumPartions 3))

```

```
)
```

```
)
```

(SpaceB

(DimensionOptions

```

;;; (dimension1 (NumPartions 5))

```

```
)
```

```
)
```

```
) ;; End of ...StaticGridPartitionedStrategyOptions.SpaceOptions
```

```
) ;; End of ...DataDistribution.StaticGridPartitionedStrategyOptions
```

```
) ;; End of ...DataDistribution.Options
```

```
) ;; End of FederationSection.DataDistribution
```

(MOM

```
;; This section contains parameters related to MOM.
```

```

;; PARAMETER: FederationSection.MOM.MomServiceAvailable
;; DESCRIPTION: Although MOM services are very useful for federation
;; monitoring and control there is an overhead associated with MOM. This
;; parameter allows a federation to turn off MOM services if they are not
;; being used within the federation.
;; RANGE: An enumeration value { Yes, No}.
;; DEFAULT_VALUE: Yes
;;
;;; (MomServiceAvailable Yes)

;; PARAMETER: FederationSection.MOM.MomTickingIntervalInSeconds
;; DESCRIPTION: This parameter is to control how often MOM will obtain
;; processing cycles to perform work. It should be set at the maximum amount
;; of time the federation can tolerate for waiting for MOM responses.
;; RANGE: A floating point value greater than zero.
;; DEFAULT_VALUE: 3.0
;;
;;; (MomTickingIntervalInSeconds 3.0)

;; PARAMETER: FederationSection.MOM.ExceptionLoggingFilePrefix
;; DESCRIPTION: This parameter specifies the file prefix for the exception
;; log. The file is written into the federate's current directory and has
;; the form: <Prefix>_<fedName> where <fedName> is the federate identifier as
;; specified in the call to joinFederationExecution
;; RANGE: A valid file name.
;; DEFAULT_VALUE: RtiMomExceptionLoggingFile
;;
;;; (ExceptionLoggingFilePrefix RtiMomExceptionLoggingFile)

;; PARAMETER: FederationSection.MOM.EnableServiceReporting
;; DESCRIPTION: This controls whether the federate has service reporting
;; enabled upon startup.
;; RANGE: An enumeration value { Yes, No}.
;; DEFAULT_VALUE: No
;;
;;; (EnableServiceReporting No)
) ;; End of FederationSection.MOM

```

(FederateSection

```

;; Entries in this section apply to federate level components. They can
;; also be used to override upper section parameters.

```

```

(EventRetractionHandleCacheOptions
;; Each federate keeps a cache of Event Retraction Handles that it has
;; generated. To prevent this cache from growing without bound, Event
;; Retraction Handles with a timestamp in the past are periodically purged.

;; PARAMETER: FederateSection.EventRetractionHandleCacheOptions.MinimumCacheSizeBeforePerformingPurge
;; DESCRIPTION: If the event retraction cache has fewer than
;; MinimumCacheSizeBeforePerformingPurge entries, purging will be skipped.
;; RANGE: An unsigned long
;; DEFAULT_VALUE: 65535
;;
;;; (MinimumCacheSizeBeforePerformingPurge 65535)

;; PARAMETER:
FederateSection.EventRetractionHandleCacheOptions.NumberOfEventRetractionHandlesToCreateBeforeStartingNewPurgeCycle
;; DESCRIPTION: Purging of the Event Retraction Handle cache may occur if at
;; least NumberOfEventRetractionHandlesToCreateBeforeStartingNewPurgeCycle
;; Event Retraction Handles have been added to the cache since the last time
;; the cache was purged.
;; RANGE: An unsigned long
;; DEFAULT_VALUE: 1024
;;
;;;(NumberOfEventRetractionHandlesToCreateBeforeStartingNewPurgeCycle 1024)
) ;; End of ...FederateSection.EventRetractionHandleCacheOptions
) ;; End of FederationSection.FederateSection
) ;; End of FederationSection
) ;; End of RTI

```

12.4 Notes on porting FoodFight from RTI1.3v6 to RTI1.3-NG

12.4.1 Migrating FoodFight for the Hands-On Practicum to RTI 1.3-NG

1. Prior to porting your code, it would be wise to review the RTI.hh file. Because of our desire to allow the use the Standard C++ fstream header file or to use of the legacy fstream.h header file. An immediate issue arose with whether ostream is in the global namespace or in namespace std. Our solution was to add a RTI_USES_STD_FSTREAM flag which triggers the following code snippet.

```

#ifdef RTI_USES_STD_FSTREAM
# include <fstream>
# define RTI_STD std
#else

```

```
# include <fstream.h>
# define RTI_STD /* nothing */
#endif
```

2. During the compilation of FoodFight.cpp we experienced no other difficulties.
3. Review the .rid file parameters above:
 - RTI 1.3-NG requires that you identify the location of the fedex.exe file in the RTI_FederationExecutive.PathOfTheFullExecutable section of the .rid file.
 - RTI 1.3-NG allows you to specify the host name for the launch of the RtiExec and FedExec in the RTI_FederationExecutive.FederationStrategy section of the .rid file. If not specified, the RTI will use the local host.
4. In addition, RTI 1.3-NG does not require that the .fed and .rid files be maintained in the %RTI_CONFIG% directory. Instead, these files should be located in the federate code directory or the path provided during the create federation invocation (see createFederationExecution in the programmer's reference pages).

12.4.2. Differences we noted while running the new FoodFight execution

1. When the federate was launched without an operational RtiExec it did not spin until a RtiExec was started. Instead, the federate hung until the resolution timeout completed its cycle and dispatched an exception message to the screen. The user was prompted with sufficient information to determine that the RtiExec was not operating at initiation.
2. The RtiExec and FedExec provide minimum manual operations. The [shift][?] feature for help is not available.
3. Exception messages are longer and more descriptive.
4. There was no delaying action required after the createFederationExecution invocation. The FedExec was ready to accept joining federates immediately after return.
5. Rti 1.3-NG does not depend on the %RTI_SAVE_PATH% variable. Instead, the directory expected to contain the saved state is given by the path to the directory specified by the user in the RID file parameter FullPathOfSaveDirectory under the RTI_FederationExecutive. As shipped the RTI.rid file defines FullPathOfSaveDirectory as ".". Unless this parameter was updated by the user, the RTI saved the state files in the same directory that launched the "rtiexec".
6. For each federate and the fedex, a successful save produced a file with the name generated using the name of the federation, the save label, the federate type and the federate handle. In the case of the fedex application, "fedex" was used instead of the federate type and handle. For example, the following would have been created for the federation named, "Verification", using a save label of "Save2Feds", with two federates and the fedex.

```
Verification_Save2Feds_agent1_1.save
Verification_Save2Feds_agent2_3.save
Verification_Save2Feds_fedex.save
```

12.4.3. Getting help

A web based help desk system on the RTI1.3NG Support Page (<http://helpdesk.dctd.saic.com/>) has been established to allow users to submit problem reports, provide enhancement requests, or get help. To submit a problem report, enhancement request, or pose a question follow the **Submit Problem** link. You will then be prompted to enter a user name and password to access the HelpDesk software. Enter the user name and password that you were assigned when you downloaded the software. Upon successful login, you will be presented with a problem report form. Please choose the appropriate **Case Type** (Problem Report or Enhancement Request) and problem category from the **Case Category - Type – Item** selection boxes and enter a short description and the details of the problem. Shortly after you submit your problem report, you should receive email confirming its submission and containing the case ID for the report and a link to the problem report. You may use link in the email to go directly to the problem report view or you can view your problem report status using the **View Problems** link. When the problem has been solved or an answer posed for your question you will receive an email with a link to the solution of your problem.

INDEX

Term	Page #
~RTIambassador()	A.7-1
A	
announceSynchronizationPoint()	B.1-1
associateRegionForUpdates()	A.6-1
AttributeHandleSet	C.1-1
AttributeHandleValuePairSet	C.1-3
attributesNotOwned()	B.4-1
attributesOwnedByFederate()	A.4-1
attributeOwnedByRTI()	B.4-2
attributeOwnershipAcquisition()	A.4-2
attributeOwnershipAcquisitionIfAvailable()	A.4-4
attributeOwnershipAcquisitionNotification()	B.4-3
attributeOwnershipDivestitureNotification()	B.4-5
attributeOwnershipReleaseResponse()	A.4-5
attributeOwnershipUnavailable()	B.4-7
attributesInScope()	B.3-1
attributesOutOfScope()	B.3-2
C	
cancelAttributeOwnershipAcquisition()	A.4-6
cancelNegotiatedAttributeOwnershipDivestiture()	A.4-8
changeAttributeOrderType()	A.5-1

changeAttributeTransportType()	A.3-1
changeInteractionOrderType()	A.5-3
changeInteractionTransportType()	A.3-3
confirmAttributeOwnershipAcquisitionCancellation()	B.4-8
createFederationExecution()	A.1-1
createRegion()	A.6-2

D

deleteObject()	A.3-4
deleteObjectInstance()	A.3-5
deleteRegion()	A.6-3
dequeueFIFOasynchronously()	A.7-2
destroyFederationExecution()	A.1-3
disableAsynchronousDelivery()	A.5-5
disableAttributeRelevanceAdvisorySwitch()	A.7-3
disableAttributeScopeAdvisorySwitch()	A.7-4
disableClassRelevanceAdvisorySwitch()	A.7-5
disableInteractionRelevanceAdvisorySwitch()	A.7-6
disableTimeConstrained()	A.5-6
disableTimeRegulation()	A.5-7
discoverObject()	B.3-3
discoverObjectInstance()	B.3-4

E

enableAsynchronousDelivery()	A.5-8
------------------------------	-------

enableAttributeRelevanceAdvisorySwitch()	A.7-7
enableAttributeScopeAdvisorySwitch()	A.7-8
enableClassRelevanceAdvisorySwitch()	A.7-9
enableInteractionRelevanceAdvisorySwitch()	A.7-10
enableTimeConstrained()	A.5-9
enableTimeRegulation()	A.5-11
Enumerated Types	C.2-1
EventRetractionHandle	C.2-8
Exception	C.1-6
Exceptions	C.2-2

F

Factory Classes	C.2-6
FederateHandleSet	C.1-7
federateRestoreComplete()	A.1-4
federateRestoreNotComplete()	A.1-5
federateSaveAchieved()	A.1-8
federateSaveBegun()	A.1-6
federateSaveComplete()	A.1-9
federateSaveNotAchieved()	A.1-10
federateSaveNotComplete()	A.1-11
federationNotRestored()	B.1-2
federationNotSaved()	B.1-3
federationRestoreBegun()	B.1-4
federationRestored()	B.1-5

federationSaved()	B.1-6
federationSynchronized()	B.1-7
FedTime	C.1-8
flushQueueRequest()	A.5-13

G

getAttributeHandle()	A.7-11
getAttributeName()	A.7-12
getAttributeRoutingSpaceHandle()	A.7-13
getDimensionHandle()	A.7-14
getDimensionName()	A.7-15
getInteractionClassHandle()	A.7-16
getInteractionClassName()	A.7-17
getInteractionRoutingSpaceHandle()	A.7-18
getObjectClass()	A.7-19
getObjectClassHandle()	A.7-20
getObjectClassName()	A.7-21
getObjectInstanceHandle()	A.7-22
getObjectInstanceName()	A.7-23
getOrderingHandle()	A.7-24
getOrderingName()	A.7-25
getParameterHandle()	A.7-26
getParameterName()	A.7-27
getRegion()	A.7-28
getRegionToken()	A.7-29

getRoutingSpaceHandle()	A.7-30
getRoutingSpaceName()	A.7-31
getTransportationHandle()	A.7-32
getTransportationName()	A.7-33

I

informAttributeOwnership()	B.4-9
initiateFederateRestore()	B.1-8
initiateFederateSave()	B.1-9
initiatePause()	B.1-11
initiateRestore()	B.1-12
initiateResume()	B.1-13
isAttributeOwnedByFederate()	A.4-9

J

joinFederationExecution()	A.1-12
---------------------------	--------

L

localDeleteObjectInstance()	A.3-7
-----------------------------	-------

M

modifyLookahead()	A.5-15
-------------------	--------

N

negotiatedAttributeOwnershipDivestiture()	A.4-10
---	--------

nextEventRequest()	A.5-16
nextEventRequestAvailable()	A.5-18
notifyAboutRegionModification()	A.6-4

P

ParameterHandleValuePairSet	C.1-10
pauseAchieved()	A.1-14
Pound-Defined Constants	C.2-7
provideAttributeValueUpdate()	B.3-5
publishInteractionClass()	A.2-1
publishObjectClass()	A.2-2

Q

queryAttributeOwnership()	A.4-12
queryFederateTime()	A.5-20
queryLBTS()	A.5-21
queryLookahead()	A.5-22
queryMinNextEventTime()	A.5-23

R

receiveInteraction()	B.3-6
reflectAttributeValues()	B.3-8
reflectRetraction()	B.3-10
Region	C.1-13
registerFederationSynchronizationPoint()	A.1-15

registerObject()	A.3-8
registerObjectInstance()	A.3-9
registerObjectInstanceWithRegion()	A.6-5
removeObject()	B.3-11
removeObjectInstance()	B.3-12
requestAttributeOwnershipAcquisition()	A.4-14
requestAttributeOwnershipAssumption()	B.4-10
requestAttributeOwnershipDivestiture()	A.4-16
requestAttributeOwnershipRelease()	B.4-12
requestClassAttributeValueUpdate()	A.3-11
requestClassAttributeValueUpdateWithRegion()	A.6-7
requestFederateTime()	A.5-24
requestFederationRestore()	A.1-17
requestFederationRestoreFailed()	B.1-14
requestFederationRestoreSucceeded()	B.1-15
requestFederationSave()	A.1-19
requestFederationTime()	A.5-25
requestID()	A.3-12
requestLBTS()	A.5-26
requestLookahead()	A.5-27
requestMinNextEventTime()	A.5-28
requestObjectAttributeValueUpdate()	A.3-13
requestPause()	A.1-21
requestRestore()	A.1-22
requestResume()	A.1-23

requestRetraction()	B.5-1
resignFederationExecution()	A.1-24
restoreAchieved()	A.1-25
restoreNotAchieved()	A.1-26
resumeAchieved()	A.1-27
retract()	A.5-29
RTIambassador()	A.7-34

S

sendInteraction()	A.3-14
sendInteractionWithRegion()	A.6-9
setLookahead()	A.5-30
setTimeConstrained()	A.5-31
startInteractionGeneration()	B.2-1
startRegistrationForObjectClass()	B.2-2
startUpdates()	B.2-3
stopInteractionGeneration()	B.2-4
stopRegistrationForObjectClass()	B.2-5
stopUpdates()	B.2-6
subscribeInteractionClass()	A.2-4
subscribeInteractionClassWithRegion()	A.6-11
subscribeObjectClassAttribute()	A.2-6
subscribeObjectClassAttributes()	A.2-8
subscribeObjectClassAttributesWithRegion()	A.6-13
synchronizationPointAchieved()	A.1-28

synchronizationPointRegistrationFailed()	B.1-16
synchronizationPointRegistrationSucceeded()	B.1-17

T

tick()	A.7-35
timeAdvanceGrant()	B.5-2
timeAdvanceRequest()	A.5-32
timeAdvanceRequestAvailable()	A.5-34
timeConstrainedEnabled()	B.5-3
timeRegulationEnabled()	B.5-4
turnInteractionsOff()	B.2-7
turnInteractionsOn()	B.2-8
turnRegulationOff()	A.5-36
turnRegulationOn()	A.5-37
turnRegulationOnNow()	A.5-38
turnUpdatesOffForObjectInstance()	B.3-13
turnUpdatesOnForObjectInstance()	B.3-14
Typedefs	C.2-9

U

unassociateRegionForUpdates()	A.6-15
unconditionalAttributeOwnershipDivestiture()	A.4-18
unpublishInteractionClass()	A.2-10
unpublishObjectClass()	A.2-11
unsubscribeInteractionClass()	A.2-13

unsubscribeInteractionClassWithRegion()	A.6-16
unsubscribeObjectClass()	A.2-14
unsubscribeObjectClassAttribute()	A.2-15
unsubscribeObjectClassWithRegion()	A.6-17
updateAttributeValues()	A.3-16