

Componentized Combat System Testbed Framework Design

8 January 2008

Adam Miller
Steve Kahn

1 Table of Contents

Componentized Combat System Testbed Framework Design Review Handout	1
1 Table of Contents	2
2 Overview	4
2.1 IR&D Objective	4
2.2 Design Review Scope	4
2.3 How to Read This Document	4
2.4 Framework Goals	6
2.5 Terms and Definitions	6
2.6 FY08 Federates	11
2.7 Similar Efforts at APL	13
3 Requirements	13
3.1 List of Requirements	13
3.2 Use Cases	20
4 Framework Design	20
4.1 Software Environment	20
4.2 Framework Software Components Overview	21
4.3 Design Notation	26
4.3.1 Sequence Diagram Legend	26
4.3.2 Class Stereotypes	27
4.3.3 Class Diagram Notes	28
4.4 Framework Processes	29
4.4.1 Control Application	29
4.4.2 Framework Process	29
4.4.3 Federate Process	29
4.4.4 Application Process	29
4.4.5 Interprocess Communication	29
4.4.6 Class Diagrams By Process	32
4.5 Major Functional Design Areas	36
4.5.1 Federate Management	36
4.5.2 Framework Initialization and Scenario Control	38
4.5.3 Time Management	82
4.5.4 I/O Management	110
4.6 HLA Middleware	134
4.6.1 RTI Implementation	134
4.6.2 HLA Functions	134
4.6.3 Class Descriptions	135
4.6.4 Class Diagram	136
4.7 Application Capture into Framework	138
4.7.1 Message Serialization	138
4.7.2 Application I/O Function Calls	138
4.7.3 Intercepting Time or Other Application Calls	140
4.7.4 Automated Code Searches	140
4.7.5 Automatic Application Code Modification	140
4.7.6 Advantages to Automatic Application Incorporation	140

5	Risk Areas	142
6	Potential Future Work.....	144
7	Related Resources	145
8	Appendix.....	146
8.1	Additional Use Case Details	146
8.2	Complete List of Use Cases	161

2 Overview

The Componentized Analysis Framework for Next Generation Combat System Components IR&D calls for development of a framework that allows disparate software applications to interact effectively to execute common scenarios, in order to evaluate networked and interacting sets of combat system components of increasing complexity. This document describes an initial software design for such a framework.

2.1 IR&D Objective

The primary objectives of the IR&D are to develop a framework that allows software applications, including tactical code as well as behavioral models, to interact with each other to execute a common scenario, faster than real-time if possible. As stated in the IR&D proposal, there are three primary objectives:

1. “Establishment of a framework that allows modules derived from both real-time tactical code and simulation models to operate together; and, evolution of this framework from simple model connections to more complex situations.”
2. “Development of a capability to allow models to run either real-time or as fast as possible to conduct statistically significant Monte-Carlo analysis of complex systems interacting.”
3. “Development and demonstration of a capability to evaluate networked and interacting sets of combat system components of increasing complexity.”

2.2 Design Review Scope

This design document describes a framework designed to accomplish objectives 1 and 2 above, and partially meets objective 3. The other part of objective 3 involves Aegis C&D modeling

There are several parallel efforts contributing to this IR&D including:

- Framework Development
- Aegis C&D Modeling
- Sensor Manager Development (possibly including WASP modifications)

This design document primarily covers the framework development aspect of the IR&D, however, relevant aspects of the applications targeted for the framework are discussed as necessary.

2.3 How to Read This Document

The authors are aware that this document provides a considerable amount of detail, and would be difficult to just sit down and read through. To help guide reviewers with specific interests, the table below maps topics of interest to the relevant sections of the document, but first, here are the sections we recommend for reviewers to take a look at, regardless of specific interest.

- Section 2, which provides an overview of the framework and what is expected for FY08. In particular, subsection 2.5 is useful because it describes terms used throughout the document.
- Section 3.1, which lists the top-level requirements.
- Section 4.1, which describes the operating system and programming languages used.
- Section 4.2, which gives a high-level overview of the framework components.
- Section 5, which outlines the major risk areas.
- Section 6, which lists areas of potential future work.

Topic	Relevant Sections
Overall Framework Design	Section 3, Requirements, and section 4, which describes the framework design in detail.
FY08 Federates	Section 2.6
Use of HLA	Sections 4.2.1.4, 4.5.3, 4.5.4, and 4.6
Time-Related Functionality	Section 4.5.3
I/O-Related Functionality	Section 4.5.4
Initialization and Scenario Control Functionality	Section 4.5.2
Application Capture Functionality	Section 4.7
Executable Process Descriptions	Section 4.4
Framework Layers	Section 4.5.1
Major Risk Areas	Section 5
Future Work Areas	Section 6

Topic	Relevant Sections
Related Resources	Section 7

2.4 Framework Goals

The overall intent of the framework is to provide a mechanism for:

- Heterogeneous applications to interact with each other to execute a common scenario
- Applications to execute independently of real-time.
- Applications to be executed within the framework with little or ideally no modifications to the source code. This allows:
 - Simple incorporation of new versions of an existing application.
 - Reduced risk that applications behave differently in the framework than in their native environments.

Another goal of this effort is to provide guidance on effective techniques to easily add additional applications to the framework in the future. We want to end up with a generic framework, not one that only works for the applications targeted for FY08.

2.5 Terms and Definitions

This section introduces terms that will appear throughout this document..

Application

A software application that has been, or could be, incorporated into the framework. This could include tactical code, models of tactical systems, models of potential new systems or parts of systems. Examples of applications could include:

- CEP tactical code
- SSDS tactical code
- Aegis C&D model
- WASP
- CEP's tracker
- A model of a potential new tracking algorithm for CEP.

Applications can in theory be in any programming language, but the framework only supports C/C++ for FY08.

Application Process

An executable process that is part of an application. The number of application processes should not change as a result of incorporation into the framework, with the possible exception of an application that is delivered as a library (as the Aegis C&D model is) which carries no process of its own. In this case, a wrapper application process for the library would likely be added. Refer to section 4.4 for a more detailed description of the framework's executable processes.

Configuration

Information that describes the federates participating in the framework. Includes which federates are running on which computers, which interfaces each federate is publishing, which interfaces from other federates that each federate subscribes to, and any other information needed to execute the federate, possibly including the executable for the federate process.

Discrete-Time Mode

A timing mode in which each federate is executed as fast as possible by “jumping” ahead to the next time request made by a federate. For example, if a federate sleeps for one second, and no other federate is or will be processing within that second, the scenario time can be immediately advanced by one second.

Interposition Library

A unix/linux mechanism for having multiple shared libraries contain implementations of the same symbol (e.g. a function call), along with the ability for a function implementation in one library make calls to implementations of function calls into shared libraries that lie further down the shared library path (the path that is typically held by the LD_LIBRARY_PATH environment variable).

For example, the POSIX *nanosleep* call could be interposed by a user-defined implementation of nanosleep. which could decide when or how to call the “real” nanosleep. A shared library containing the user-defined nanosleep would be placed ahead of the real POSIX shared library in the LD_LIBRARY_PATH.

Refer to section 4.5.3.2 for a more detailed description of the use of interposition libraries in the framework.

Federate

An application that has been incorporated to execute within the framework, along with supporting framework code. The following conditions are true for federates:

- The framework is capable of starting and stopping the application.
- The framework is capable of sending “scenario control” information into the application, as needed.

- All time-related calls in the application are handled by the framework, instead of going directly to the operating system or some other middleware. In other words, the application is decoupled from wall-clock time.
- All inputs and outputs of the application are piped through the framework, except in cases in which no other federate cares about the input or output. For example, a CEP federate might output its DX data through its existing DX output portal, for use in the CEP DX/DR toolset. None of the other federates would need this data.

Most federates will likely include some sort of adaptation layer, either auto-generate or manually written, between the application and the framework.

Federate Instance

A specific instantiation of a federate within the framework. For example, two separate instances of a CEP federate could be executed simultaneously to represent two different units.

Federate Message

Any collection of data exchanged between one or more federates. Typically a federate message is used to capture, send, and receive messages from existing application interfaces, e.g. a message between a CEP instance and an SSDS instance would be sent as a federate message. Refer to section 4.5.4 for a more detailed description of the I/O processing.

Federate Process

A process within the framework, dedicated to a specific federate instance, that is responsible for managing that federate instance and its application processes. For each active Federate, there is exactly one Federate process executing in the framework at any given time.

Federation

A collection of federates that interact with each other to execute a common scenario.

Framework

The source and executable code that:

- Allows federates to execute independently of wall-clock time, in scaled real-time or discrete time
- Supports inter-federate communications.
- Supports starting up and shutting down of federates.
- Delivers relevant scenario control data to the federates.

Framework Process

A process that supports execution of the framework and its federates. One framework process executes on each computer that is participating in a framework at any given time. Refer to section 4.4 for a more detailed description of the framework's executable processes.

High Level Architecture (HLA)

An IEEE standard that defines specifications for a simulation interoperability framework. HLA is commonly used in the department of defense (DoD) community.

Monte Carlo Analysis

An analysis technique that relies on repeated random samplings to obtain results, typically used when analysis with a deterministic algorithm is not feasible.

Open Architecture Simulation Interface Specification (OASIS)

A simulation interoperability framework specification developed jointly by JHU/APL and MIT Lincoln Lab.

Runtime Infrastructure (RTI)

Implementation of the HLA specification. An RTI will be used as middleware in the framework.

Portable Operating System Interface (POSIX)

A collection of related IEEE standards that defines an API for various operating system features, including but not limited to, threads, shared memory, and semaphores.

Probability of Raid Annihilation (PRA) Test-bed

A test-bed that incorporates a CEP, an SSDS, and other federates in a discrete-time framework. The discrete-time operation of this frameworks is based on the concepts and source code used in the PRA test-bed.

Scaled Real-Time Mode

A time mode in which time-related calls made by an application are modified before being sent to their intended target (most likely the operating system or a POSIX library). This is to make applications run faster or slower than real time. For example, if a time-scale factor of 2.0 is used, meaning twice as fast as real-time, a sleep call for one second would be scaled to one-half second before the calling the real sleep function. A time-scale factor of 1.0 would be a special case in which applications would run in actual real-time.

Scenario

The parameters that define an execution of a federation. This includes how long to run the federation, the time mode to run the scenario in, and any ground-truth data needed by specific federates, e.g. a previously scripted file name for the WASP to run. A scenario can be run multiple times consecutively (i.e. iterations) for Monte-Carlo analysis.

Sensor Manager

The federate within the framework that is responsible for driving the SSDS, CEP, and Aegis C&D federates. This federate will likely be primarily composed of WASP components.

Simitar

Implementation of OASIS, also developed jointly by JHU/APL and MIT Lincoln Lab.

Software Clock

The software clock is currently utilized in the PRA testbed to intercept time-related calls from application threads. These calls are treated as time requests – essentially when every thread in an application has made a time request, the software clock sends the smallest time request to the time server. One software clock exists per federate application process. The software clock concepts and source code are expected to be re-used in the framework.

Time Server

The time server is currently utilized in the PRA testbed to collect time advance requests from the Software Clock, and forward those requests to the RTI when necessary. One time server exists per federate. Each time server waits for time advance requests from its known software clocks. Once each software clock makes a time advance request, the time server forwards the smallest time request to the RTI. When the RTI grants time, the time server forwards the time grant on to the software clocks.

Unified Modeling Language (UML)

A general-purpose modeling language which consists of both graphical and textual components. UML is often used to express system and software designs. UML use cases, class diagrams, and sequence diagrams are used extensively in this design document.

vxWorks Middleware

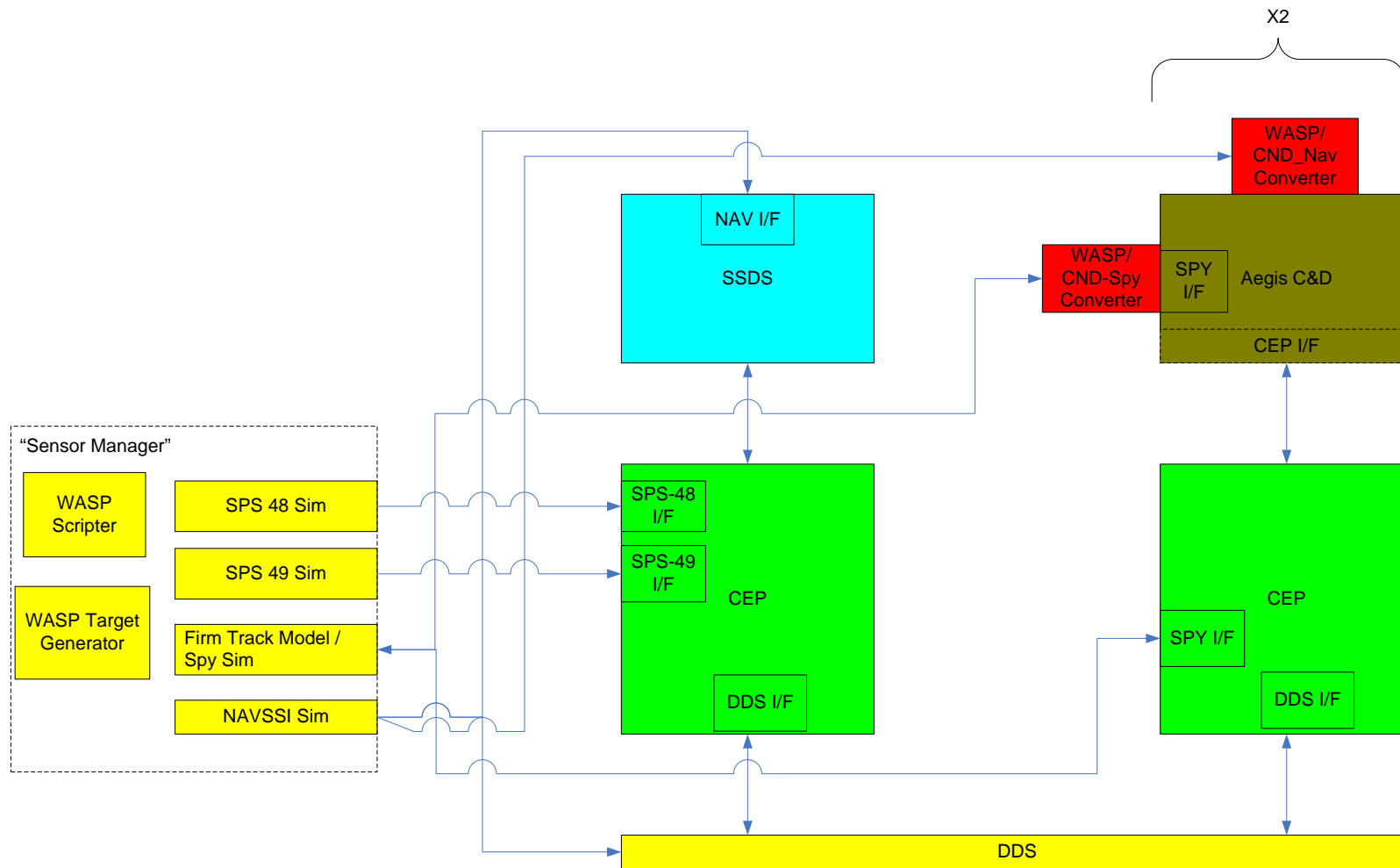
An implementation of the vxWorks real-time operating system API that converts vxWorks function calls into their POSIX equivalents. The vxWorks middleware is used to run vxWorks applications on Solaris and Linux workstations.

2.6 *FY08 Federates*

The federates that are planned for incorporation in FY08 are:

- CEP on a Workstation (CWS)
- SSDS on a Workstations (SSDSWS)
- Sensor Manager (primarily WASP on a Workstation (WWS))
- Aegis Command and Decision (C&D) model

The following diagram shows the interfaces between this federates. Note that the “x2” over the CEC/C&D unit indicates that two instances of each federate, representing two different units, are intended to be executed in the framework.



2.7 Similar Efforts at APL

OASIS/Simitar

Simulation framework developed by APL, as described previously. OASIS/Simitar was considered as a candidate middleware for the framework, and could still be utilized in the future.

Total Ship Computing Environment Infrastructure (TSCEI)

TSCEI is Raytheon's infrastructure on which all shipboard applications for the DDX will run. There is an effort underway at APL to investigate the possibility of running TSCEI independently of real-time. Though this effort is programmatically unrelated to this IR&D, there are many technical similarities.

SSDS Software in the Loop (SIL)

The SIL project involves running recent versions of the SSDS on a Workstation in real time.

3 Requirements

The following table lists the top-level requirements for this framework. These requirements were identified during an initial research phase at the beginning of FY08. The requirements that are not mandatory for FY08 are denoted as "Nice to have" in the priority column.

3.1 List of Requirements

Category	Requirement	Number	Examples/Comments	Priority	Risk/Difficulty
Configuration Setup	Specify which components are going to be executed, whom interacts with whom, etc	10		Have to have	Low
Configuration Setup	Run multiple scenarios independently and concurrently, if hardware permits.	11	If 100 monte-carlo runs desired, could do two sets of 50 simultaneously.	Nice to have	Medium
Scenario Generation	All components must support a common scenario (received via framework)	20		Have to have	Medium
Scenario Generation	Unit groundtruth data scripted into scenario	21		Have to have	Low
Scenario Generation	Target groundtruth data scripted into scenario	22		Have to have	Low
Scenario Generation	Radar contact data scripted into scenario.	23		Have to have	Low
Scenario Generation	Navigation data scripted into scenario.	24		Have to have	Low
Scenario Generation	ID data scripted into scenario.	25		Nice to have	Low
Scenario Generation	IFF data scripted into scenario.	26		Nice to have	Low
Scenario Generation	Add noise/randomness for monte-carlo runs.	27		Not Sure Yet	Medium
Scenario Control	Capability to load a scenario.	30		Have to have	Medium
Scenario Control	Capability to start a scenario.	31		Have to have	Medium
Scenario Control	Capability to abort a scenario gracefully.	32		Have to have	Medium
Scenario Control	Scenario ends at its scripted time.	33		Have to have	Medium

Category	Requirement	Number	Examples/Comments	Priority	Risk/Difficulty
Scenario Control	Run multiple scenarios from a batch file	34		Nice to have	Medium
Scenario Control	Capability of pausing a running scenario.	35	Since we have to control time anyway, this shouldn't be too difficult. Will look impressive for demos.	Nice to have	Low
Framework Execution	Execute all federates.	40		Have to have	Medium
Framework Execution	Support scenario control: Initialize/shutdown federates.	41	(may have to handle legacy run-time states within legacy components)	Have to have	Medium
Framework Execution	Message passing between federates.	42		Have to have	Medium
Framework Execution	Provide mechanism to exchange any data among framework and application components.	43	Periodic events	Have to have	Medium
Framework Execution	Flexible to support various modes of operation: Real-time	44		Have to have	Medium
Framework Execution	Flexible to support various modes of operation: Scaled real-time.	45	2x real-time, 4x...	Not Sure Yet	High
Framework Execution	Flexible to support various modes of operation: Discrete time.	47		Have to have	Medium
Framework Execution	Fast as possible: components execute simultaneously.	46	Restart scenario when a component falls behind	Have to have	High
Framework Execution	Support components on distributed computers.	48		Have to have	High
Framework Execution	Provide current time to components.	49	e.g. component uses POSIX calls to retrieve time, framework must implement that call.	Have to have	High
Framework Execution	Need to intercept all calls to system clock	50	What other calls do we need to intercept? Any call with a time value in it?	Have to have	High

Category	Requirement	Number	Examples/Comments	Priority	Risk/Difficulty
Framework Execution	In addition to system clock calls, need to determine what other system calls to intercept and how to handle them	53	(e.g. blocking semTake).	Have to have	High
Framework Execution	Framework must keep components in sync	51		Have to have	High
Framework Execution	Support endianness incompatibilities	52	At a minimum, this could simply require each component to send all its outputs in network byte order (big endian)	Have to have	Medium
Framework Execution: Instrumentation	Determine when components "fall behind"	60	Preferably automated.	Nice to have	High
Framework Execution: Instrumentation	Measure performance of components.	61	measure wall-time taken to processes a given event.	Nice to have	High
Framework Execution: Instrumentation	Ideally try to measure an optimum speed for a given scenario.	62		Nice to have	High
Framework Execution: Instrumentation	Ideally determine at run time if a component is processing when it shouldn't be	63	i.e. figure out if we failed to remove a real-time dependency.	Nice to have	High
Framework Execution: Data Extraction	Extract all data that passes through framework.	64	For test and analysis, possible playback capability, or message insertion.	Have to have	Low
Framework Execution: Data Extraction	File bookkeeping for multiple runs	66		Have to have	Low
Framework Execution: Data Extraction	Use existing DX capabilities.	67	e.g. CEC DX	Nice to have	Low

Category	Requirement	Number	Examples/Comments	Priority	Risk/Difficulty
Component Requirements: Converting Real-Time to Discrete	No direct calls to system clock - all system clock calls need to go through framework	70	Might be duplicate of requirement 50	Have to have	High
Component Requirements: Converting Real-Time to Discrete	No infinite loops	71	e.g. a repeating while loop that checks a message queue, which may not have a sleep call.	Have to have	High
Component Requirements: Converting Real-Time to Discrete	Identify programming constructs that, if left alone, may cause application to keep processing when it shouldn't, i.e. continuous processing when framework is running in a discrete mode.	72	See requirement 53. This might be a duplicate requirement.	Have to have	High
Component Requirements: Converting Real-Time to Discrete	All inputs received through framework	73	Possible exceptions for components that already have dedicated interfaces?	Not Sure Yet	Medium
Component Requirements: Converting Real-Time to Discrete	All outputs go to framework	74		Have to have	Medium
Component Requirements: Converting Real-Time to Discrete	Purge all data in response to framework	75	Required for monte-carlo runs	Have to have	High
Component Requirements: Converting Real-Time to Discrete	Common way for each component to define its message specs	76	ideally automated	Not Sure Yet	Medium
Advanced Features	Discovery	80	e.g. "I need a composite tracker, who can do that?"	Nice to have	High

Category	Requirement	Number	Examples/Comments	Priority	Risk/Difficulty
Advanced Features	Dynamic data semantics incompatibility resolution	81	e.g. at runtime figure out things like "this component uses LLA, this one uses ECEF, automatically plug in the converter". Coordinate frames and units relatively easy, more difficult challenges could include uncorrected data vs corrected data.	Nice to have	High
Advanced Features	Automated process to modify existing applications to conform to framework.	83		Nice to have	High
Advanced Features	Automagically support endianness incompatibilities.	84		Nice to have	High
Advanced Features	Create a library of components	85		Nice to have	Medium
Advanced Features	Create a library of reusable software functions	86	e.g. an adaptive layer between legacy tactical software and the framework.	Nice to have	Medium
Advanced Features	Ensure repeatable results across monte-carlo runs.	87		Nice to have	High
Display	Define interface between framework and display	90		Not Sure Yet	Low
Display	Create framework display	91		Not Sure Yet	Low
Display	Incorporate available legacy displays	92	If we can do this easily...for example CEC_WS comes with this, SSDS_WS does not.	Nice to have	
Forward Compatibility	Integrate with OASIS.	100		Nice to have	
Forward Compatibility	Integrate with HLA.	101		Have to have	
Forward Compatibility	Integrate with NSSAT.	102		Nice to have	

Category	Requirement	Number	Examples/Comments	Priority	Risk/Difficulty
Forward Compatibility	Integrate with ARTEMIS	103		Nice to have	
Forward Compatibility	Integrate with ANTARES	104		Nice to have	
Documentation	Maintain paper trail of modifications to existing code to conform to framework.	110		Have to have	
Documentation	Justification of why OASIS/Simitar isn't sufficient on its own.	111	The "distributed" issue should cover this, if nothing else.	Have to have	
Documentation	Justification of why HLA is not sufficient on its own.	112		Have to have	
Documentation	Create and document approach for incorporating future applications	113		Have to have	

3.2 Use Cases

Based on the requirements listed in the previous section, a number of UML use cases were developed in order to document the expected usage of the framework. Starting with a notional class design, the required behavior for each use case was traced through the class design, leading to newly defined classes, associations between classes, and class method definitions. Sequence diagrams were created for many of the use cases. The most significant use cases and sequence diagrams are presented in section 4.

Some of the defined use cases were deemed redundant with others; the details of some of those are presented in section 8.1 in the appendix. The complete list of use cases is also presented in the appendix, in section 8.2.

4 Framework Design

4.1 Software Environment

The framework will be developed in the C++ programming language for FY08. C++ is the language of choice for several reasons:

- all the FY08 federate applications have been developed in either C, C++, or a combination thereof
- C++ is a high-performance programming language that is commonly used for real-time applications
- HLA RTIs provide a C++ API
- source code from the PRA testbed and vxWorks middleware is in C++

The operating systems (OSs) supported in FY08 have been selected based on the operating systems on which the federate applications are supported, as well as the operating systems supported by the RTIs. The following table lists the FY08 federates and their operating system requirements.

Federate	OS	Comments
CWS	Red Hat Enterprise Linux 4	Also runs on Solaris. Using Linux to show multiple OSs.
SSDSWS	Solaris 8 or 10 on SPARC hardware	Hosting of SSDSWS on Solaris 10 is underway. Solaris 10 is preferable to 8; Solaris 8 is not supported on new SPARC hardware.
Sensor Manager (WWS)	Solaris 8 or 10 on SPARC hardware.	WWS builds and runs on Solaris 10, though release versions are still built on 8.

Federate	OS	Comments
Aegis C&D	Windows XP or Linux	C&D model is developed on Windows. Intent is for framework to run on Linux / Unix environment, however, developing framework to execute on Windows may turn out to be easier than porting C&D to Unix/Linux.

The MÄK RTI, which is likely the RTI of choice for FY08, runs on all of the above operating systems. However, the latest version of MÄK does not run on Solaris 8, and support for MÄK on Solaris 8 is only expected to last for approximately one more year.

4.2 Framework Software Components Overview

Software designs/implementations typically describe/produce large amounts of source code, which are then divided into components (or packages, or modules, or any other term developers like to use). Essentially the software is divided up to make it easier to manage. The framework software can be divided up in a number of different ways, depending on the criteria. Different criteria for dividing up the framework software, and the resulting components, are described below. Note that some framework class names are provided below as examples; those classes will be described in detail later in this document.

4.2.1.1 Packaging by Functionality

If the framework software is divided up by functionality, the packages are:

- Federate Management
- Framework Initialization and Scenario Control
- Time
- I/O
- Framework Controlling Application
- Application Capture*
- Utilities

This is the way use cases and the associated sequence diagrams are divided up in this document. A class diagram for each of these areas will also be presented. Federate Management is something of a special case in that it is related to the other functionalities. Therefore there are no use cases directly associated with Federate Management, but class diagrams will still be shown.

*Application capture is not a run-time feature.

4.2.1.2 Packaging by Executable Process

Another way to divide up the framework software is by executable process. If this is the criteria used, the packages would be:

- Framework Process
- Federate Process
- Application Process

This document shows class diagrams for each of these hypothetical process packages.

4.2.1.3 Packaging by Layer

Yet another way to divide up the framework software is in a layered approach. For example:

- Federate-specific software. This would be code to handle a specific federate type's I/O, such as the Federate I/O Handler generated for a specific federate type.
- Generic framework software. This is software not coupled with any application, nor any underlying middleware. The abstract Post Office and Time Machine classes are examples of this.
- Middleware-specific software. This is software necessary to utilize specific underlying middleware. The HLA Post Office and HLA Time Machine classes are examples of this.

This layering is important because the framework is intended to be implemented in such a way that different applications can be incorporated into the framework by adding to the federate-specific layer, without changing any other layers, and middleware implementations for different services (e.g. time or I/O) can be replaced out as long as equivalent functionality can be supplied. Altering the middleware would not affect the other layers, thus the actual federates are not at all affected by changes in middleware.

The following figures show an examples of both layering and functionality divisions.

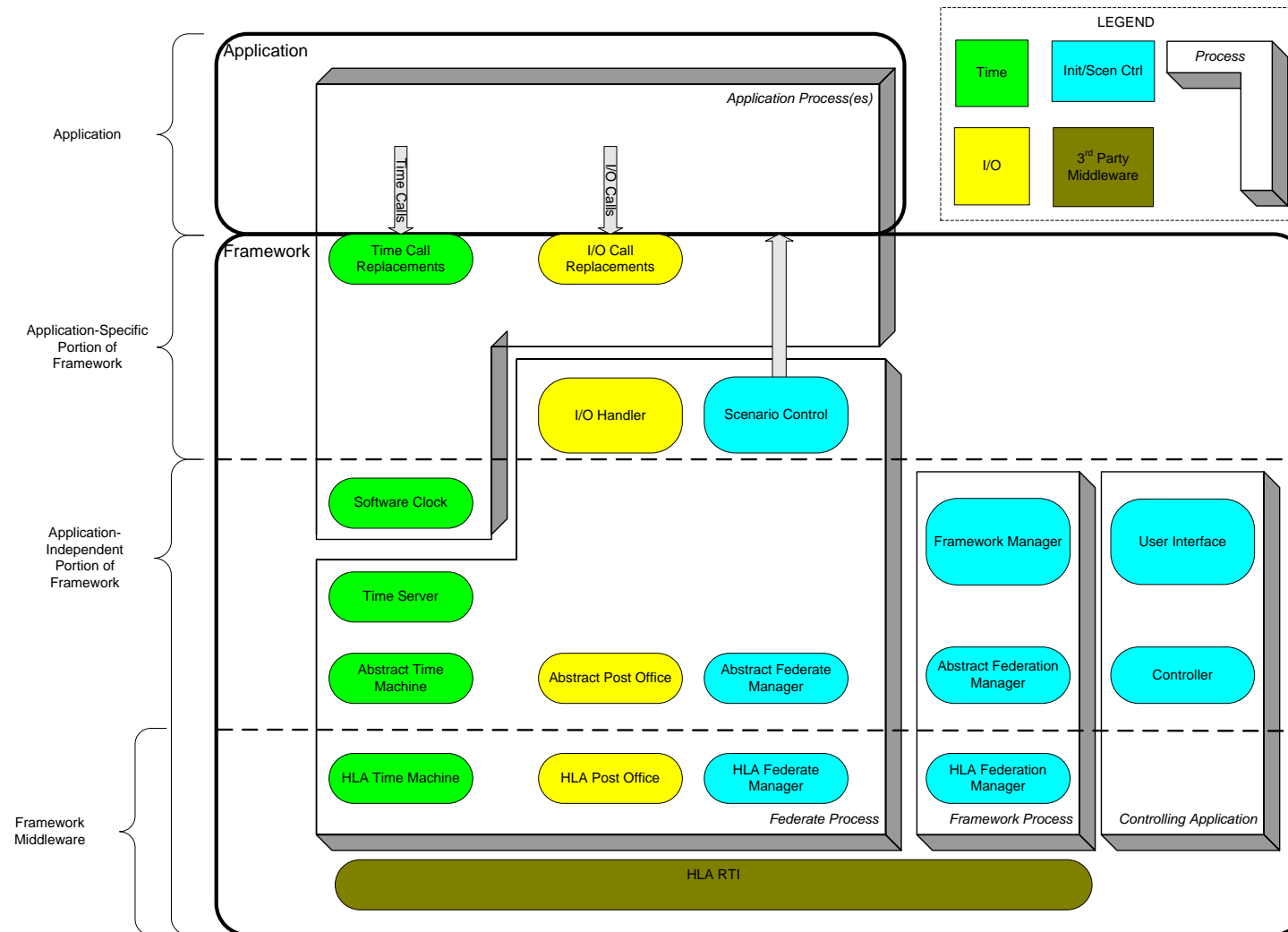


Figure 1: Overall Architecture

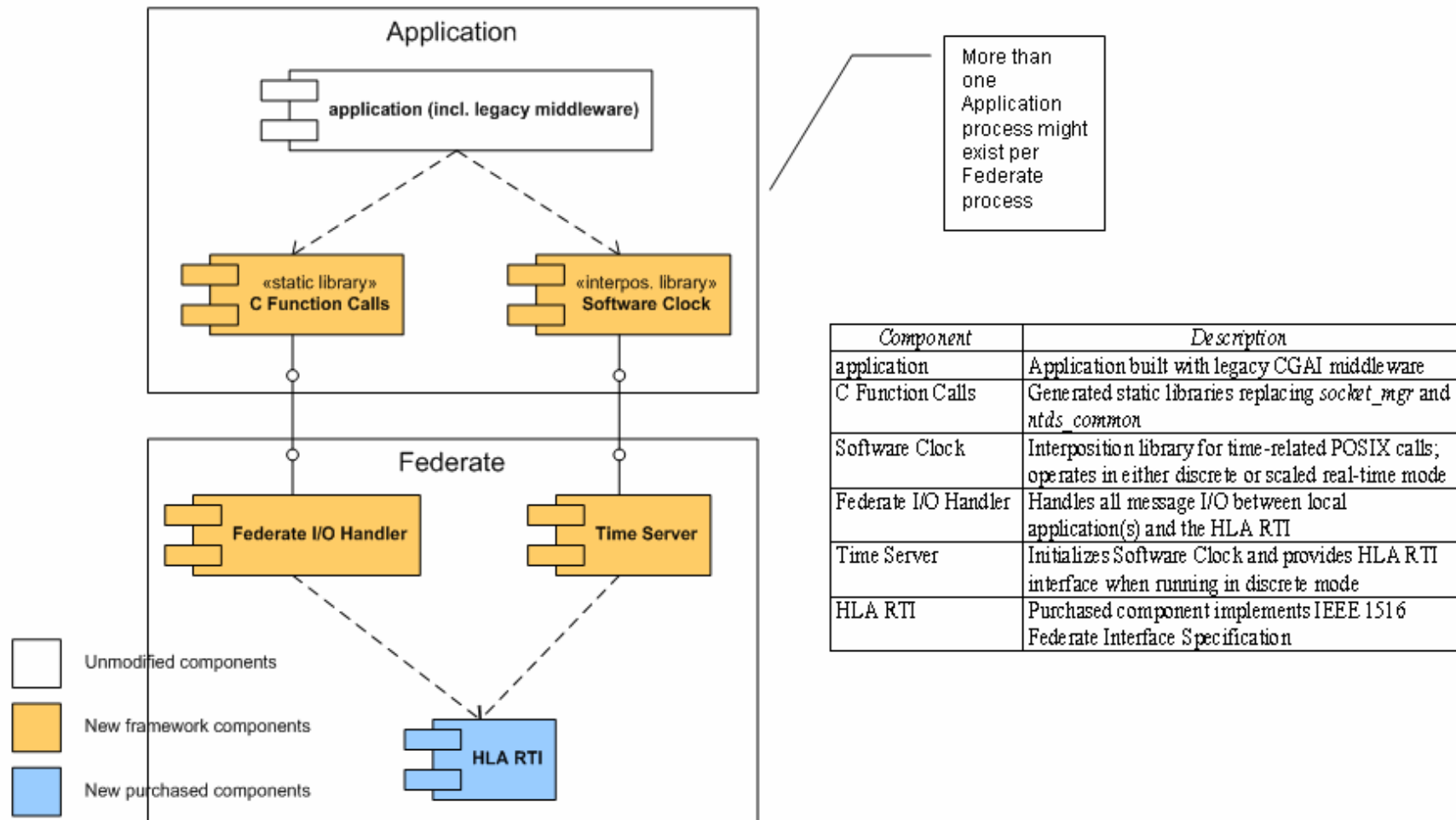


Figure 2 Framework Architecture: I/O and Time


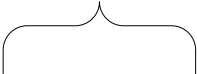
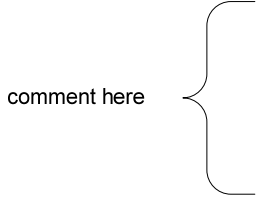
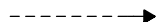

4.2.1.4 HLA-Specific Software

Because HLA figures so prominently in the framework for FY08, a class diagram with all the HLA-specific classes is shown in this design, along with class descriptions.

4.3 Design Notation

4.3.1 Sequence Diagram Legend

The sequence diagrams in this document utilize some notations that are not part of the UML standard for sequence diagrams. These “extra” symbols are described in the following table:

Notation	Description
	A message call with a bold arrow indicates that the message call crosses process boundaries.
	Indicates the process (framework, federate, or application) in which the object resides.
	Indicates the comment to the left of the braces applies to the sequence of events enclosed by the braces. Often used to show a repeated sequence of calls.
	A dashed line with a “create” as the function name indicates a constructor call. Will be bold if the constructor call is triggered from another process.
	A function name in bold indicates that the function call was provided by external middleware, such as the calls provided by the HLA RTI.

4.3.2 Class Stereotypes

Several stereotypes have been defined to tag classes, which indicate that the class has certain characteristics in the design. The stereotypes defined in this design are:

Stereotype	Description
example	This class is shown in the design as an example, and is not expected to appear in the framework software.
external application	A class that represents an application external to the framework, such as a federate application or a user interface application. This is not an actual C++ class in the framework software.
generated	Some or all of the source code for this class can be automatically generated.
generated, not actual class	The “generated” and “not actual class” stereotypes both apply.
interposition library	Represents function calls that have been implemented in interposition libraries.
multiprocess	The class is used by multiple processes, e.g. both the framework and federate processes may use this class.
multiprocess, provided	The “multiprocess” and “provided” stereotypes both apply.
not actual class	The class is a representation of a software artifact that is not actually a C++ class. For example, a collection of C-style functions might be represented in the design by a class with this stereotype.
POSIX	This class is provided by the POSIX specification, and is thus included in any POSIX implementation library.
provided	This class is provided by external software. For example, the HLA RTI provides classes used by the framework.

Stereotype	Description
proxy	This class is used as a proxy for communication between two or more other classes.
static library	Represents a library statically linked with other source code to form an executable. Static libraries cannot be interposed at run-time.
utility	The class offers some sort of generic utility, such as file I/O. In some cases, a new class may not be needed if the C++ standard library, or some other middleware, provides the same functionality. Utilities, by definition, are available to any process in the framework.

4.3.3 Class Diagram Notes

To reduce clutter, associations with a multiplicity of 1 to 1 are shown without the multiplicity indicated.

4.4 Framework Processes

As indicated in the terms and definitions section, and in section 4.2.1.2, a running instance of the framework with executing applications consists of multiple processes, described as follows:

4.4.1 Control Application

There is one controlling application for the entire framework. This application communicates with each instance of the framework process, providing commands for creating federations, federates, and running scenarios. The control application will be driven by some sort of user interface.

4.4.2 Framework Process

One framework process exists on each computer participating in the framework. This process is responsible for communicating with active federate processes, as well as accepting commands from the controlling application.

4.4.3 Federate Process

One federate process exists for each active federate. Each federate process is controlled by the framework process. Once a federate is no longer in use, i.e. not included in the current federation, the federate process is deleted. The federate process is responsible for starting up the application represented by the federate.

4.4.4 Application Process

Each federate application consists of one or more application processes. These are the processes that would exist if the application were running independently of the framework, with the exception of applications that are incorporated into the framework as libraries rather than executables. For those library-only applications (e.g. Aegis C&D) an application process is added for incorporation into the framework.

4.4.5 Interprocess Communication

The following diagram shows the processes, as well as the classes within each process that are responsible for communicating outside the process boundaries.

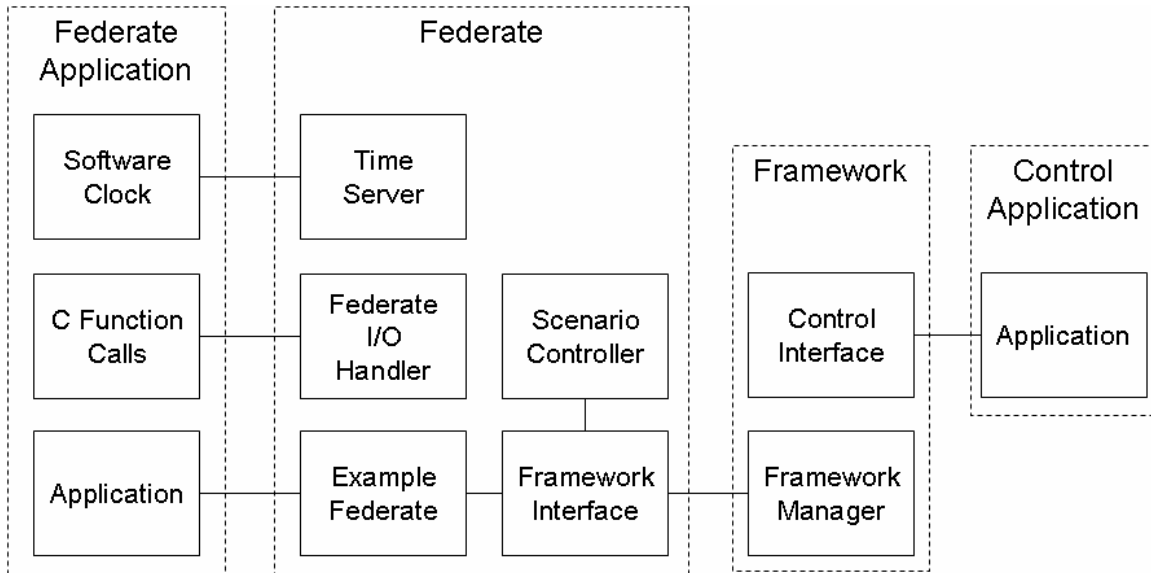


Figure 3 Process Boundaries and Classes

The framework design involves multiple processes: the control application, the framework, the federate, and the actual applications. There must be some sort of inter-process communication (IPC) mechanism to support the process/process interactions shown in many of the sequence diagrams. A straightforward approach, which maps well to the object-oriented design presented in this document, uses a remote procedure call (RPC) mechanism, where a “client” can make a call to a C++ function in a “server,” where the client and server are in two different processes. As a risk-mitigation experiment, an RPC mechanism employing POSIX message queues was developed. This prototype was then used between the Time Server, which will be in the Federate process, and the Software Clock, which will be in the Federate Application process. For the purpose of this experiment, the Time Server and Software Clock from the PRA Pilot were used.

The design of the POSIX queue-based RPC mechanism is shown in the sequence diagram of Figure 4. The client and server each instantiate the FrameworkRpc template class, supplying the types of the input and output classes as template parameters. The client invokes the remote procedure using the invokeFunc method, and the server awaits an incoming call by using the awaitFunc method; when a call is received, a callback function (registered via the FrameworkRpc constructor) is invoked. This prototype was successfully used between the Time Server and Software Clock. There is no dependency on startup order: either the client or server can be instantiated first.

To measure the performance of this RPC implementation, FrameworkRpc was used with the input type being a 32-bit integer and the output type being a structure containing a 64-bit Unix time and a 1000-byte character array. When running on a multi-user Solaris system, the invocation rate was about 10,100 calls/sec. For a Pentium III Linux box, the measured rate was 114,000 calls/sec. Thus, this RPC implementation appears to be more than fast enough for the purposes of this framework.

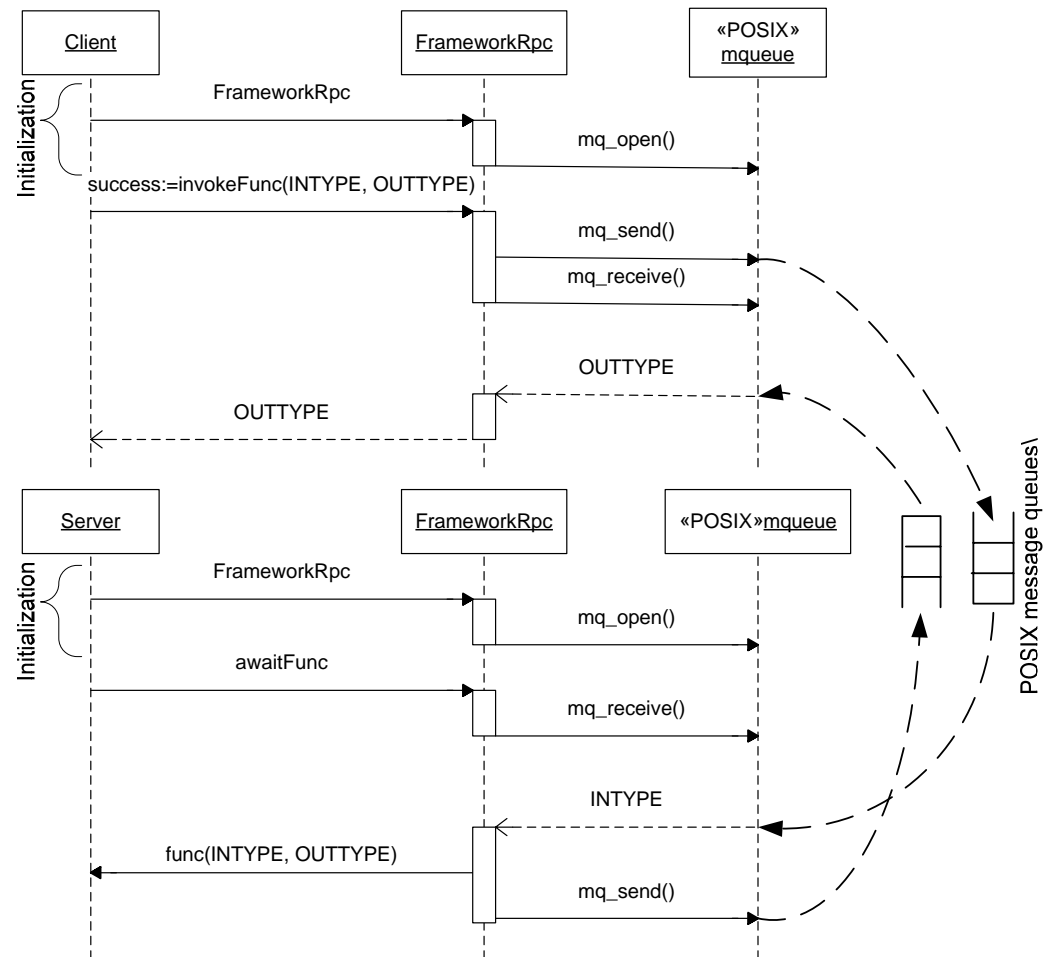
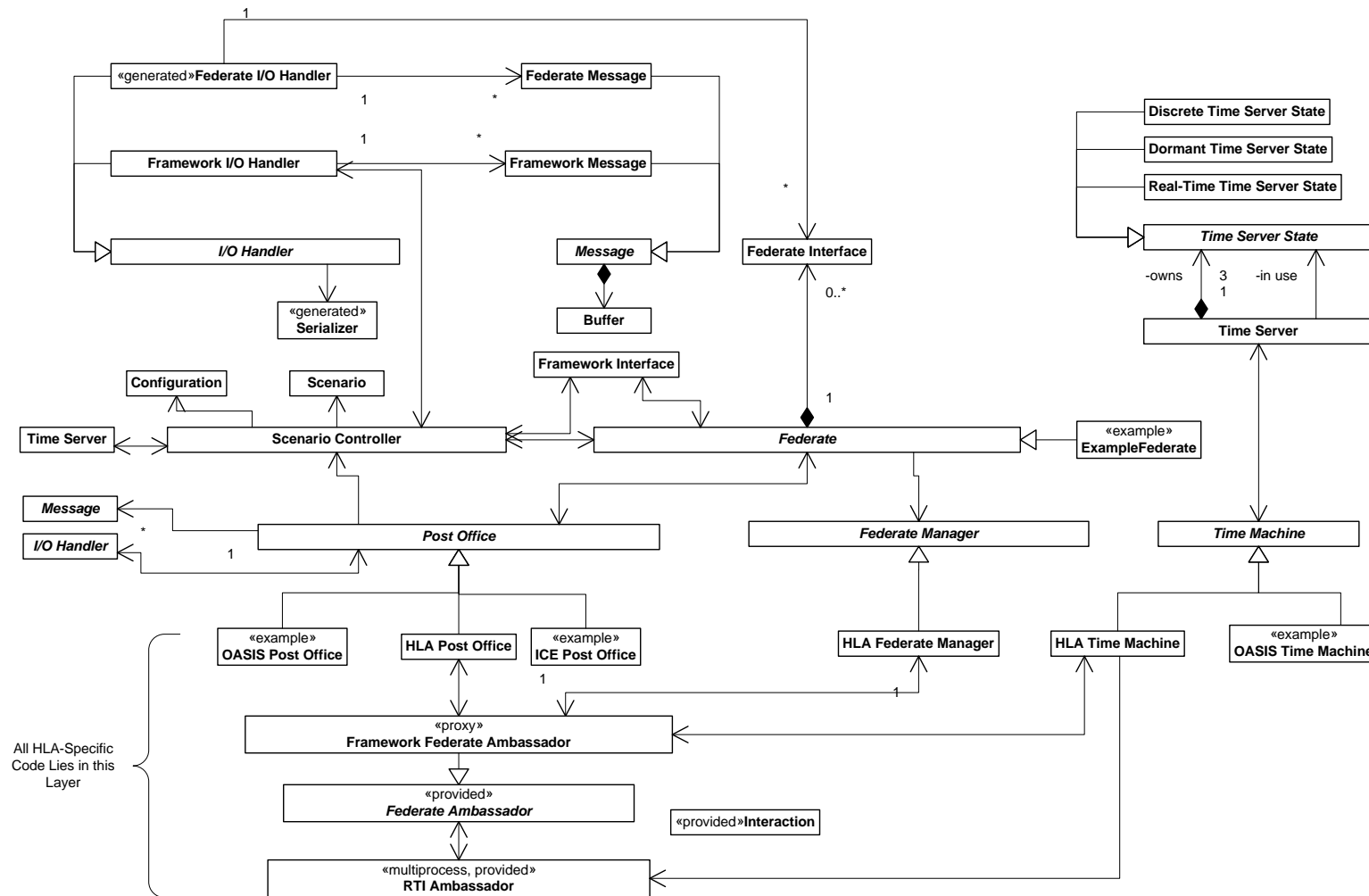


Figure 4: Notional IPC Mechanism

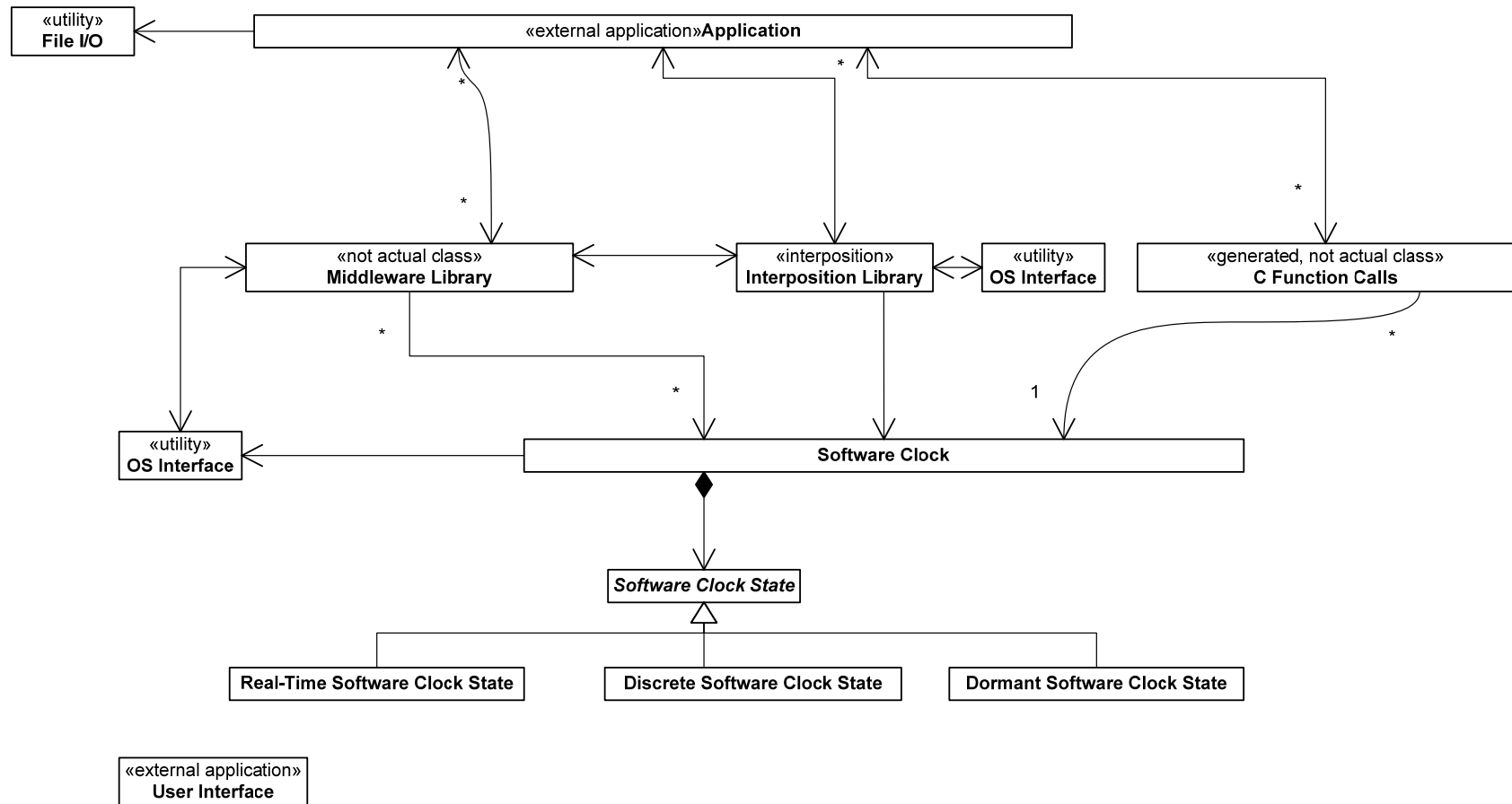
4.4.6 Class Diagrams By Process

The following set of class diagrams show the framework classes that make up each process. For real-estate reasons, method names are omitted.

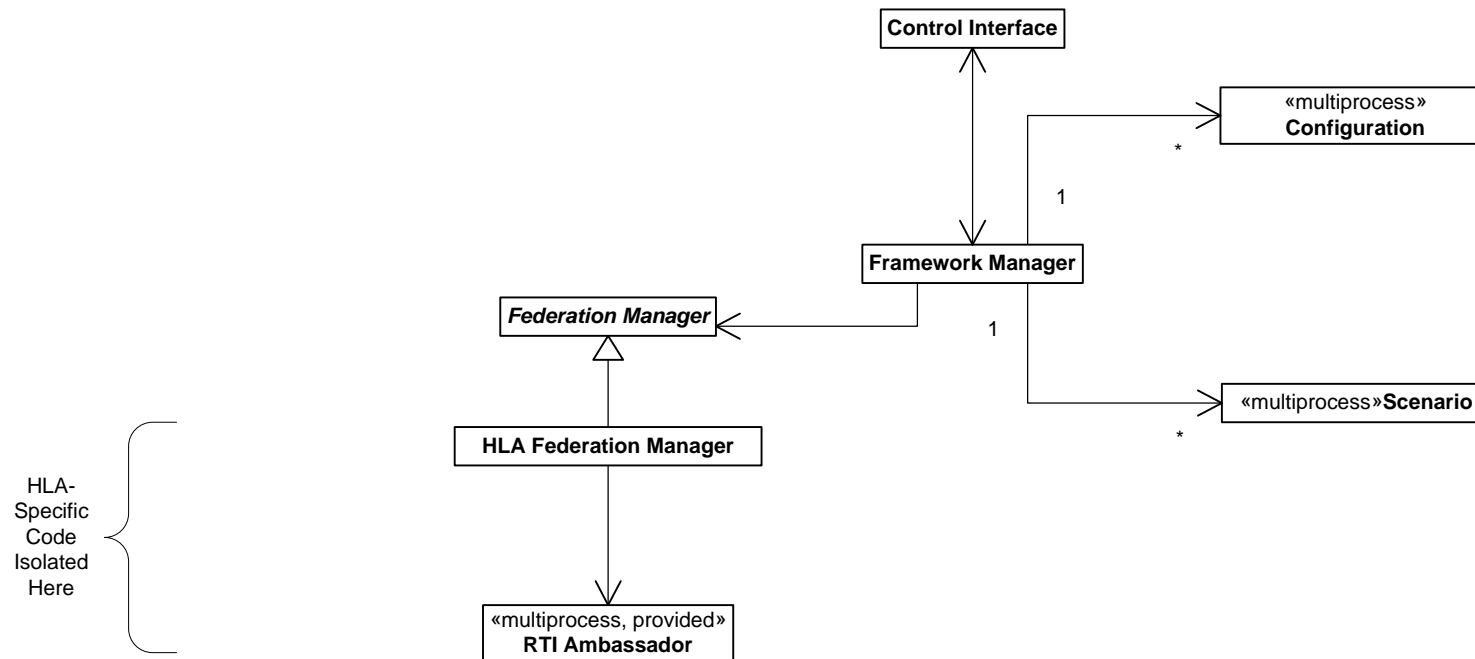
Federate Process Classes



*Application Process
Classes*



*Framework Process
Classes*



4.5 Major Functional Design Areas

4.5.1 Federate Management

The classes that comprise federate management apply to all of the major functional design areas, thus they are described separately in this section. The classes that comprise federate management include an abstract Federate class and subclasses for each specific federate. Federate subclasses are represented in this design by the Example Federate class.

4.5.1.1 Class Descriptions

Federate

Abstract class that serves as a proxy for the actual executing application that this federate represents (e.g. a CWS, SSDSWS, WASP, etc). Subclasses of this class provide federate-type specific methods for starting up, purging data from, and shutting down the federate. Subclasses of this class may also provide functionality used by the Federate I/O Handler to decode I/O calls in order to identify message datatypes so that messages can be serialized and converted to network byte order.

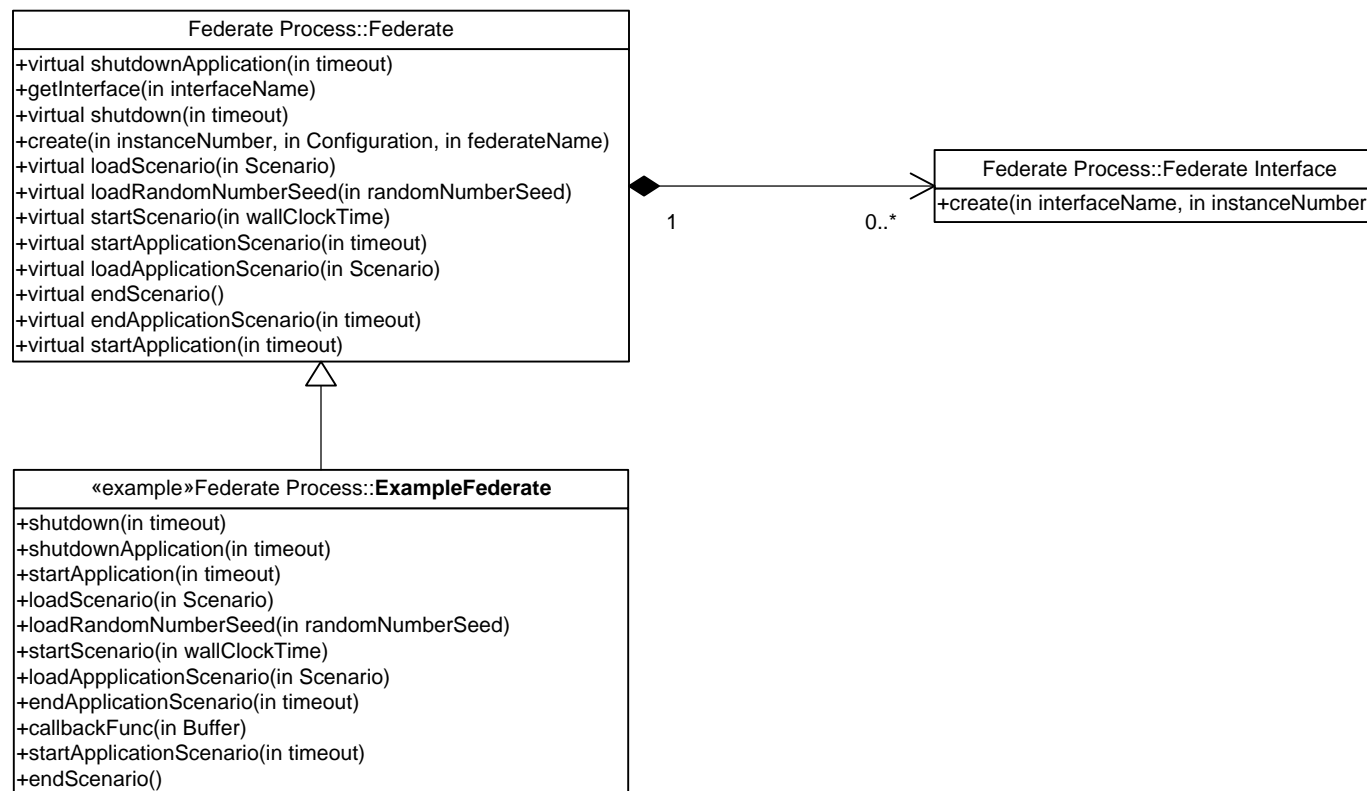
Example Federate

This class is used as a representative concrete Federate implementation in the class and sequence diagrams in this document. For FY08, other concrete Federate implementations will likely be:

- CEP Federate
- SSDS Federate
- Sensor Manager Federate
- Aegis C&D Federate

4.5.1.2 Class Diagram

Federate Classes



4.5.2 Framework Initialization and Scenario Control

The classes in this functional area are responsible for responsible for starting up and shutting down the framework, federates within the framework, and scenarios. They include mechanisms for loading configurations and scenarios, which contain all the information needed to execute a scenario.

4.5.2.1 Class Descriptions

Control Interface

Responsible for catching control commands from an external controlling application (such as a user interface application) and passing them to the Framework Manager. Commands include loading configurations, loading scenarios, and scenario control commands such as start, stop, pause, resume.

Framework Manager

Responsible for creating and destroying federations and federates, including startup and shutdown of federate processes. Also responsible for passing scenario loading and control commands to each Federate instance under its control.

Configuration

Holds configuration information, as described in the “Terms and Definitions” section. This information primarily describes what federates are present, which federates communicate with each other, and which computers each federate is running on.

Scenario

Hold scenario information, as described in the “Terms and Definitions” section. Includes scenario start time, run-time length of scenario, time mode, and any information targeted towards specific federates.

Scenario Controller

Part of the Federate process, this class is responsible for more making calls related to starting, stopping, pausing, and resuming federates. The Scenario Controller generally interacts with Federate and Time Server to accomplish this.

Federate Manager

Abstract class providing an interface for federates to join or resign from federations. Other functions may be provided in the future, as needed.

HLA Federate Manager

HLA-specific implementation of Federate Manager. Allows federates to join or resign from federations that have been established in the HLA RTI.

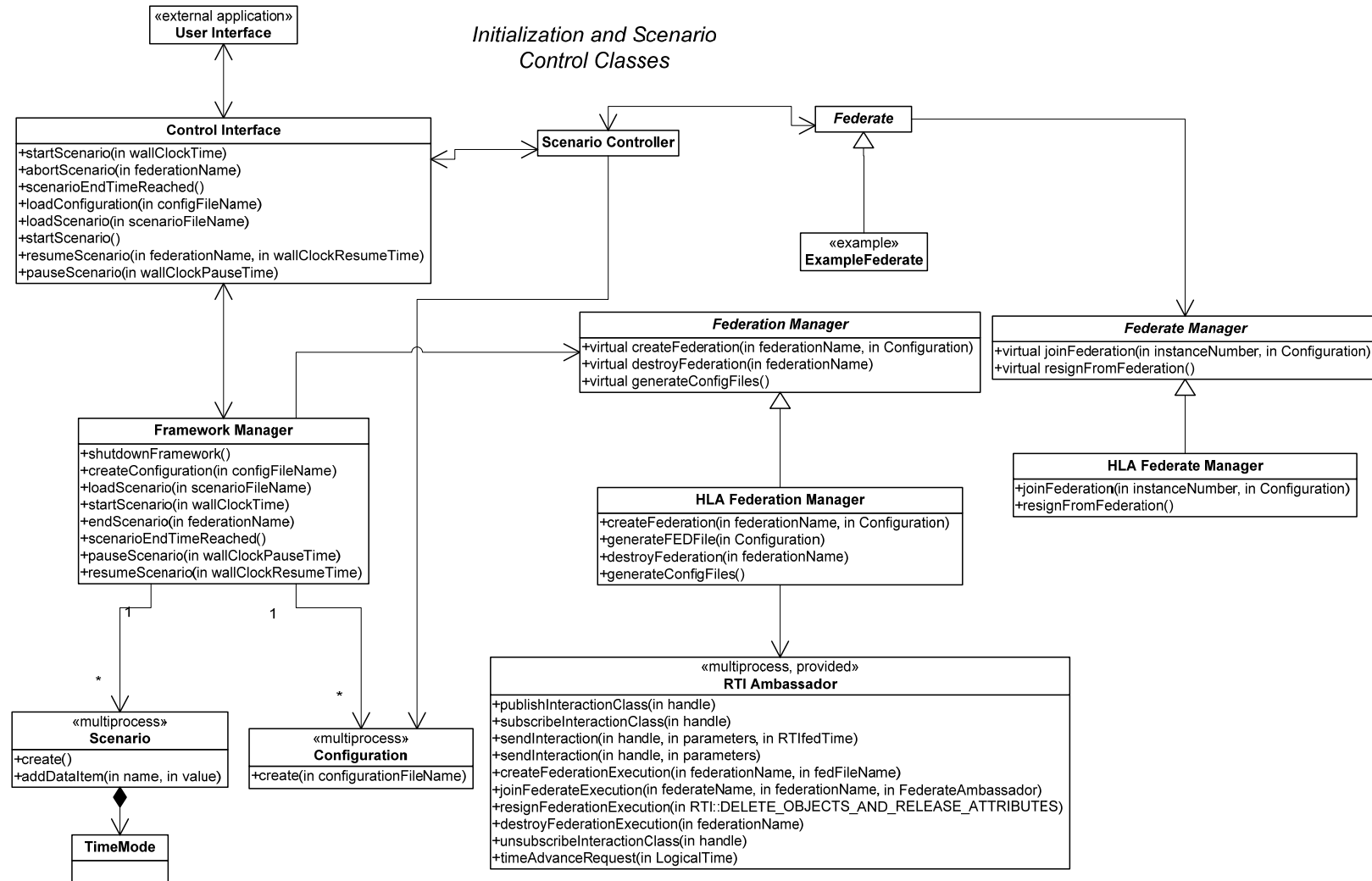
Federation Manager

Abstract class providing an interface for federations to be created and destroyed, as well as generating configuration files needed to establish a federation.

HLA Federation Manager

HLA-specific implementation of Federation Manager. Creates and destroyed HLA federations by communicating with the HLA RTI. Provides ability to generate FED file required for HLA federation creation.

4.5.2.2 Class Diagram



4.5.2.3 Use Cases

4.5.2.3.1 Use Case 0.1

Summary

Framework is initialized.

Preconditions

No framework executable is running on any of the computers participating in the framework.

Triggers

User starts up the framework.

Basic Course of Events

These two steps can be completed in any order, and must be run on each computer participating in the framework.

- 1a. Run framework executable, providing necessary command line arguments [TBD, re-examine what, if any, command-line arguments are necessary, the other initialization-related use cases are completed.]
- 1b. Run the HLA RTI.

Alternate Paths

None.

Postconditions

A framework process is running on all the computers participating in the framework. Each framework process is in a state in which it is ready to initialize federations, initialize federates, and run scenarios.

Design Notes

Starting up the framework and the RTI could go into a startup script that is run on each framework computer. Additional info is needed on starting up the RTI; this is likely vendor-specific.

There could be an automated way for one framework executable to start up others on other computers. That might be beyond what is necessary for FY08. For FY08, could simply start up each executable one at a time with a startup script. One option to consider is to run the Framework Process on each computer as a daemon process. If that is done, there will need to be a way to shutdown all federates and get back to the starting state.

4.5.2.3.2 Use Case 0.2

Summary

Framework is shut down.

Preconditions

Framework is running. This means that on each framework computer there is/are:

- one framework process running
- zero or more federate processes running, depending on whether a scenario is currently executing or not.
- for each federate process running, the federate application itself may have its own processes executing.

Triggers

Shutdown framework command is received from User Interface (UI)

Basic Course of Events

1. UI tells the Control Interface, for each framework process in the framework, to shut down the framework.
2. Each framework process, upon receiving a shutdown command, must kill every federate process that is running, by calling the *ScenarioController::shutdownFederate* method. The following steps 3-6 are repeated once for each federate process.
3. ScenarioController calls the *shutdown* method on its associated Federate object, passing it a timeout value.
4. The Federate calls the *shutdownApplication* method to kill all processes in its actual application. This method is abstract in the Federate class; the behavior will be Federate specific, e.g. the commands to kill all CEP processes will be different than the commands required to kill all SSDS processes. This method is given a timeout; if the application cannot shut down completely and successfully before the timeout is reached, an exception is thrown.
5. The Federate calls *resignFromFederation* on its associated Federate Manager object. This ultimately results in a call to *RTIAmbassador::resignFederationExecution*.
6. The federate process exits.
7. After shutting down all its federate processes, the framework process calls *destroyFederation* on the Federation Manager. This results in a call to *destroyFederationExecution* on the RTI Ambassador.

8. The framework process then either exits, or remains in a dormant state as a daemon process until the framework is used again.

Alternate Paths

Only alternate path is a design decision: whether UI connects to one framework process and that framework process disperses shutdown message, or whether UI connects to all framework processes and sends them shutdown message simultaneously. The response to the shutdown message should be the same in either case.

Postconditions

All framework and federate processes have been shut down.

Design Notes

The abstract *Federate::shutdownApplication()* method should be required by contract to kill all processes. This method will attempt to shutdown all applications processes and is responsible for determining if shutdown was successful or not.

One design decision is whether the *shutdownApplication* methods should block or not. The current decision is to block, but with a timeout. *shutdownApplication* would then return as soon as all processes for the running application shut themselves down, or the timeout happens, whichever occurs first. If the application times out without shutting down correctly, an exception is thrown.

There is another design decision to make: whether the UI connects to all framework processes on all framework computers, or whether the UI connects to one framework process, which would be responsible for distributing commands to all other federate process. The decision was made for FY08, for simplicity, to have the UI connect to all framework processes.

Note that if inter-positional libraries are used for clock-related calls, this will need to be accounted for when using a timeout to shutdown. One option would be for the federate process to not have a software clock. This is what will be done in FY08 – neither the federate nor framework processes will execute on a software clock; only application processes are bound by software clocks. Another option would be for the federate process to have a software clock that is always running in real-time mode, and scaled to 1.0.

Framework is shut down.

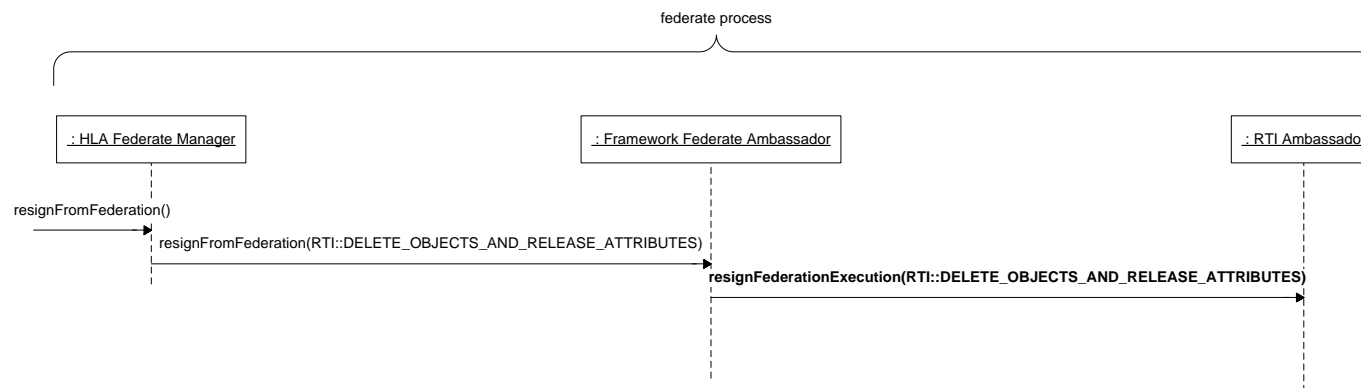
This sequence diagram assumes that there is a Display Interface attached to each Framework Instance. The initial design calls for a single user interface (UI) application to connect to the framework process on each computer running the framework.

```

sequenceDiagram
    participant Framework as framework process
    participant Federate as federate process
    participant Application as application process
    participant Federate2 as federate process

    Framework->>Framework: shutdownFramework()
    Framework->>Federate: shutdownFederate()
    Federate->>Federate: shutdown(timeout)
    Federate->>Application: shutdownApplication(timeout)
    Federate->>Application: shutdown()
    Federate->>Federate2: resignFromFederation()
    Federate2->>Federate2: This call is application-specific, shutdown() is simply an example.
    Federate2->>Federate2: See next page
    Federate2-->>Framework: 
    Federate2->>Federate2: destroyFederation
    Federate2->>Federate2: destroyFederationExecution
  
```

Sequence Diagram for
Use Case 0.2
Framework is shut down.
Diagram 2 of 2



4.5.2.3.3 Use Case 10.1

Summary:

A federation named “Fed1” is established.

Preconditions:

“Fed1” is not already established. The RTI is executing.

Triggers

During a configuration or scenario initialization step, a call to establish a federation is made.

Basic Course of Events

1. Framework Manger calls *createFederation* on the HLA Federation Manager, giving it the federation name, and the Configuration object that describes the federation (see Use Case 10.4 for details on how the Configuration object is instantiated).
2. The HLA Federation Manager uses the Configuration object to generate the FED file required to instantiate the federation.
3. The HLA Federation Manager calls *createFederationExecution* on the RTI Ambassador, passing it the federation name and the FED file name.

Alternate Paths

None

Postconditions:

The federation “Fed1” has been established.

Design Notes:

Creation of a federation is a prerequisite for:

- inter-federate communication
- running a scenario

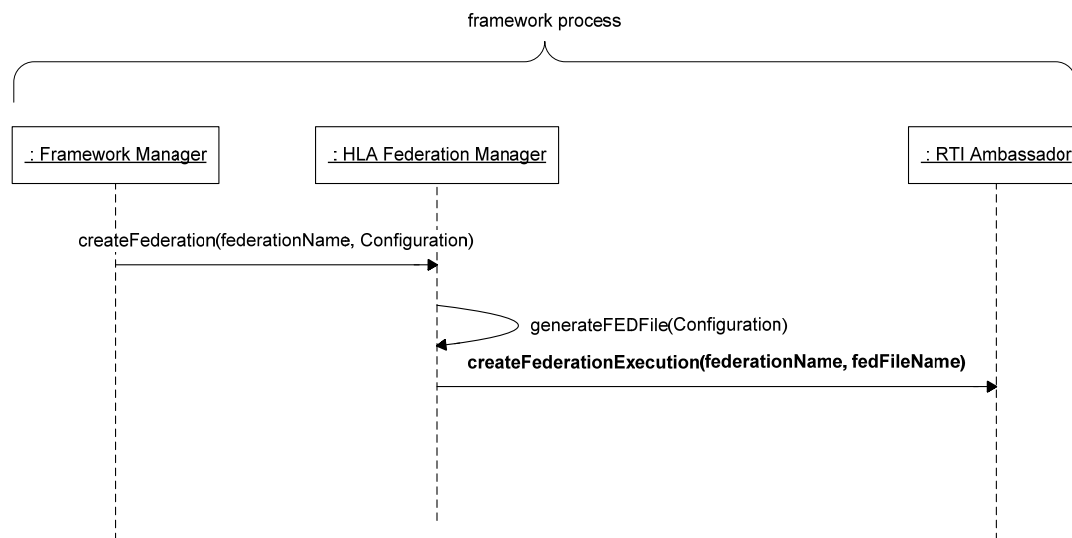
All processing for this use case is done within a framework process.

Among all computers that are going to be running our federation, there only needs to be one call to the RTI Ambassador once to create the federation. However, if the framework tries to create it twice by accident, the RTI should simply throw an *RTI::FederationExecutionAlreadyExists* exception.

We may have to decide whether we want a “master” framework process responsible for creating federations, or have each framework process attempt to create the federation before adding any federates. In the FY08 design, the controlling application would be responsible for triggering the framework process to create a federation.

Sequence Diagram(s)

Sequence Diagram for
Use Case 10.1
A federation named "Fed1" is
established.



4.5.2.3.4 Use Case 10.2

Summary:

User specifies one instance of Federate A and one instance of Federate B to be part of a Federation.

Preconditions:

Name of federates and federation is known. Federation is created.

Triggers

Framework has reached the step in initialization to the point at which federates need to join their respective federations.

Basic Course of Events

The following steps are repeated for each Federate that is going to be created.

1. Framework manager creates an instance of Federate, in a new process, giving the new Federate object its instance number, its name, and a reference to the Configuration object.
2. The Federate object, during construction, calls *joinFederation* on the HLA Federate Manager, which results in a chain of calls leading to calling *joinFederateExecution* on the RTI Ambassador. As parameters to that call, the RTI Ambassador is given the federate name, the federation name, and a pointer to the Framework Federate Ambassador.
3. The Federate calls *startApplication*, which is an abstract method implemented by Federate specific subclasses of Federate, to execute the “real” federate application. If the federate fails to start up correctly before the timeout, an exception is thrown.

Alternate Paths

If an exception is thrown in step 3 because the application fails to start up, the Federate should resign from the federation by calling *resignFromFederation()* on the Federate Manager. The sequence of events for this call is shown in Use Case 0.2.

Postconditions:

Federates A and B are in the federation. Implementation postcondition: a new framework federate process is created for both federates.

Design Notes:

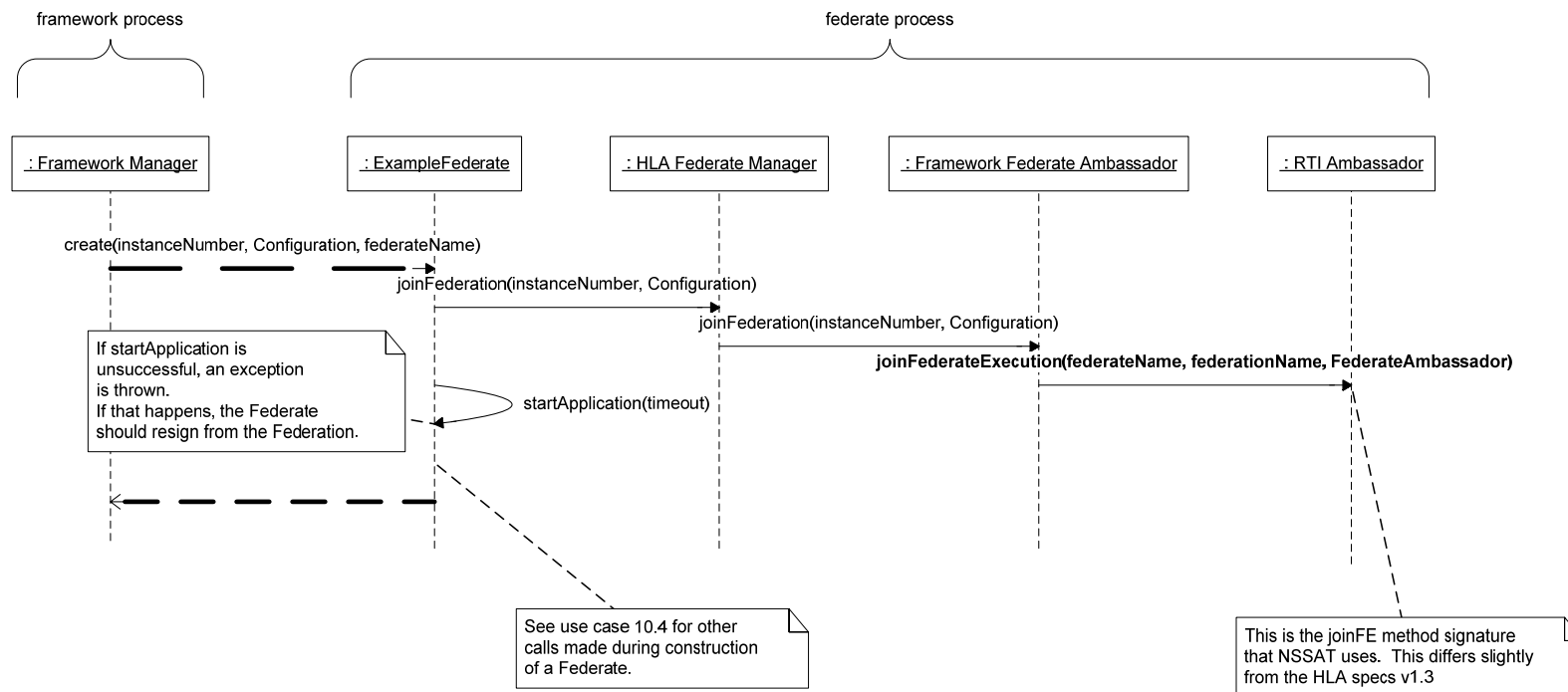
See use case 10.4 for a description of how Configuration objects are filled in.

Federate A and B may or may not be running on separate computers. If the computer running federate A has created the federation, the computer running the framework process of federate B may not be aware of this. For this reason, we could have each framework process attempt to create the federation first (see Use Case 10.1).

The current sequence of events specifies that a number of steps are performed within construction of a Federate. If they fail (most notably the *startApplication* call), an exception may be thrown. Further investigation may be needed to determine whether it is good C++ design to throw exceptions during construction of an object. An alternative would be a simple constructor that doesn't throw exceptions, and a more complex initialization function that does.

Sequence Diagram(s)

Sequence Diagram for
 Use Case 10.2
 User specifies one instance of
 Federate A and one instance of
 Federate B to be part of a
 Federation.



4.5.2.3.5 Use Case 10.4

Summary

A federation exists consisting of two instances of Federate A and two instances of Federate B. Let's call them A.1, A.2, B.1, and B.2. Federate A.1 and B.1 want to subscribe to each other's message, but **not** to identical message types from A.2 and B.2.

Preconditions

Federation and federates are created, but have not announced their subscriptions yet.

Triggers

The initialization step in which federates need to announce their subscriptions has been reached.

Basic Course of Events

This can partially be handled at configuration time¹.

Steps 1 and 2 are performed offline, prior to scenario execution. The other steps are performed at runtime.

1. When creating a configuration of which federates are going to run, give each federate an instance number.
2. Also specify in the file which federates are going to subscribe to the interfaces of the other federates – the federate type and instance number are needed for this.
3. At runtime, during initialization, the configuration file is read into a Configuration object.
4. A federate process is executed and a Federate object is created. These steps are explained in detail in Use Case 10.2. The Configuration object is passed into the Federate object's constructor.
5. The Federate loops over the interfaces (name and instance number) it is supposed to publish to. This information is contained in the Configuration object. For each interface, the Federate:
 - Creates a Federate Interface object giving it the interface name and instance number. The instance number should be the same as instance number of the Federate.
 - Calls the HLA Post Office to announce publication. The instance number for publications should be the same as the instance number of the Federate. See use case 42.1 for the remainder of processing chain.

¹ "Configuration Time" refers to offline steps, prior to scenario execution, in which the federation, federates, interfaces (which become interactions in HLA), etc. are identified, and likely written into some form of configuration file.

6. The Federate loops over the interfaces (name and instance number) it is supposed to subscribe to. For each of those interfaces:
 - If this interface was not one of the published interfaces, a new Federate Interface object is created.
 - Calls the HLA Post Office to announce subscription. See use case 42.2 for remainder of the processing chain.

Alternate Paths

None.

Postconditions

Design Notes

Example potential xml configuration file:

```
<!-- Define federate types -->
<FederateType name="CEP">
    <SupportedInterface name="CEPtoSSDSTcp"/>
    <SupportedInterface name="CEPtoSSDSUdp"/>
</FederateType>
<Federate Type name="SSDS">
    <SupportedInterface name="CEPtoSSDSTcp"/>
    <SupportedInterface name="CEPtoSSDSUdp"/>
</FederateType>

<!--Define Interface Types -->
<Interface name="CEPtoSSDSTcp"/>
<Interface name="CEPtoSSDSUdp"/>

<Federation name="Fed1">

    <!--CEP 1 is defined, and subscribes to the 2 interfaces of SSDS
    1 -->
    <FederateInstance type="CEP" instanceNumber="1">
        <Host name="host1" port="1234"/> <!--run on a computer
        named host1, connect to port 1234 -->
        <Subscribe federateType="SSDS" instanceNumber="1"
        interface="CEPtoSSDSTcp"/>
        <Subscribe federateType="SSDS" instanceNumber="1"
        interface="CEPtoSSDSUdp"/>
    </FederateInstance>

    <!--CEP 2 subscribes to the two interfaces of SSDS 2 -->
    <FederateInstance type="CEP" instanceNumber="2">
        <Host name="host2" port="5678"/> <!--run on a computer
        named host2, connect using port 5678 -->
        <Subscribe federateType="SSDS" instanceNumber="2"
        interface="CEPtoSSDSTcp"/>
```

```
        <Subscribe federateType="SSDS" instanceNumber="2"  
        interface="CEPtoSSDSUdp" />  
    </FederateInstance>
```

```
</Federation>
```

Note that tools can be created to automate construction of configuration files like the one above.

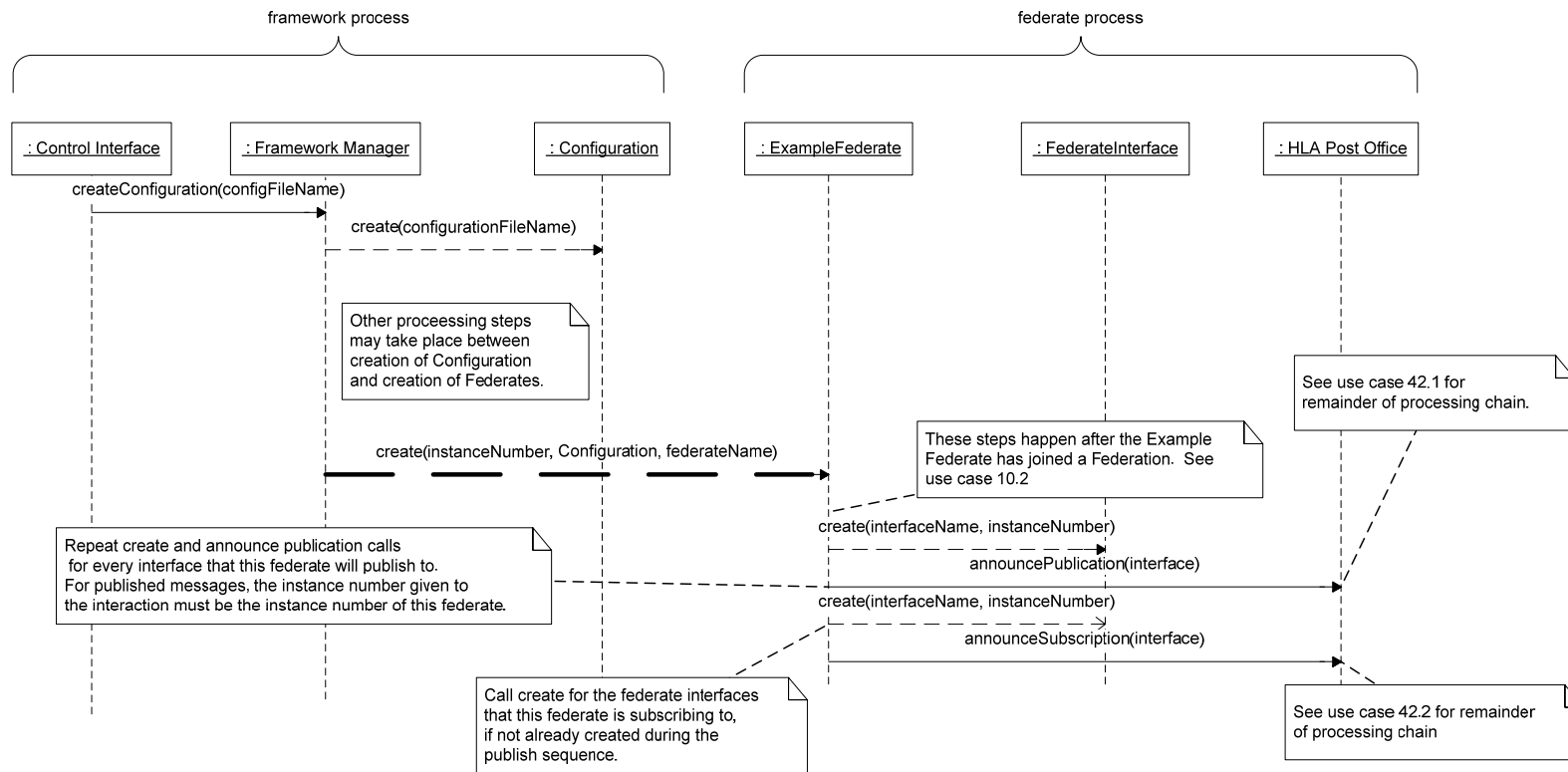
The sample configuration file above could and probably should be split into two files: one for defining the Federate and Federate Interface types, the other for defining the actual Federates and their associated Federate Interfaces for a given configuration/scenario.

Applying the same instance numbers to federate instances and federate interface instances may have pitfalls. The strategy defined here is that each federate can only *publish* to interfaces with the same instance number (e.g. CEP #1 cannot publish to the CEPtoSSDSTcp interface #2). Each federate may *subscribe* to an interface with any instance number.

Sequence Diagram(s)

Sequence Diagram for Use Case 10.4

A federation exists consisting of two instances of Federate A and two instances of Federate B. Let's call them A.1, A.2, B.1, and B.2. Federate A.1 and B.1 want to subscribe to each other's message, but **not** to identical message types from A.2 and B.2.



4.5.2.3.6 Use Case 20.1

Summary

User scripts a scenario. Assume federates have already joined the desired federation. Scenario initial conditions are delivered to all federates. Scenario initial conditions should include a generic enough placeholder so that users of the framework could add references to more data if need be (e.g. references to input files). Design note: This could simply be an abstract class with a basic implementation in the framework code.

Preconditions

To load the scenario into the framework: a scenario file must exist.

To give the scenario data to federates: the Federate objects must have been created.

Triggers

User informs framework that they have finished scripting scenario. (this triggers loading scenario data into the framework – this can be done before or after federates are created)

User is ready to execute scenario – federates must exist and be ready to receive scenario data before the data can be sent to federates.

Basic Course of Events

1. User or automated tool enters federate-specific scenario data into a file (offline, independent of framework)
2. User interface sends load scenario command, along with the scenario file name, to the framework process on each computer running the framework.

The next steps cannot happen until after the federate processes are running and their Federate objects are created.

3. For each federate running, one Scenario object is created, containing only the data targeted towards each federate. (note: an alternative would be to create one scenario object for all federates).
4. Each Scenario object is given to appropriate Scenario Control and Federate objects. If the Federate object cannot load the scenario correctly, an exception is thrown.
5. Each Scenario Control sends its Time Server the Time Mode of the scenario. Time Mode is an object that indicates if the scenario is going to be running in real or discrete time, and if in real-time, what the scaling is.
6. The Time Server tells each Software Clock registered with it what the Time Mode is. When additional software clocks register with the Time Server, they must be

- given this mode as well. Additional software clocks would be created if the Federate application starts up new processes during scenario execution.
7. The Time Server calls *initScenarioEndTimeMonitoring()* on itself.

Alternate Paths

For a given federate, the scenario data cannot be loaded correctly. That federate should be shut down and its process killed. An error message should be logged for the users.

Postconditions

All federates have been informed of scripted scenario parameters.

Design Notes

This use case introduces the concept of a scenario file and corresponding class. The scenario file includes parameters necessary for the framework, and also contains any parameters necessary to supply to federates as initial conditions. The scenario file does not define what federate types or federates are present – those are defined in the configuration file and corresponding class.

Data required that is not specific to any particular federate could include, but is not limited to:

- Scenario Name
- Number of iterations
- Random number seed
- Scenario start time
- Scenario run time – how long, in scenario time, the scenario should run
- Time mode – discrete or real-time.
- If real-time, the scaling factor.

A flexible way to script scenarios is needed. Several options are possible. Regardless of the method chosen, it is required that:

- any data can be inserted into a scenario file
- each federate is responsible for interpreting or ignoring the scenario data.

Option 1: use name value pairs. Example:

```
<Scenario Data>
  <DataItem name="WaspScenarioFile"
    value="someWaspScenarioFileName" />
  <!-- note that this data item refers to an external file.  this
    is ok -->
  <DataItem name="RandomNumberSeed" value="1" />
</Scenario Data>
```

Option 2: create a block of text data for each federate type or even each federate. It's up to the federate type to interpret correctly. The framework would direct each data

```
<Scenario Data>
  <!-- Assume WASP is a federate type -->
  <DataItem federateType="WASP" instanceNumber="1">
    <!-- insert anything directed towards instance number
    1 of the WASP federate here -->
  </DataItem>

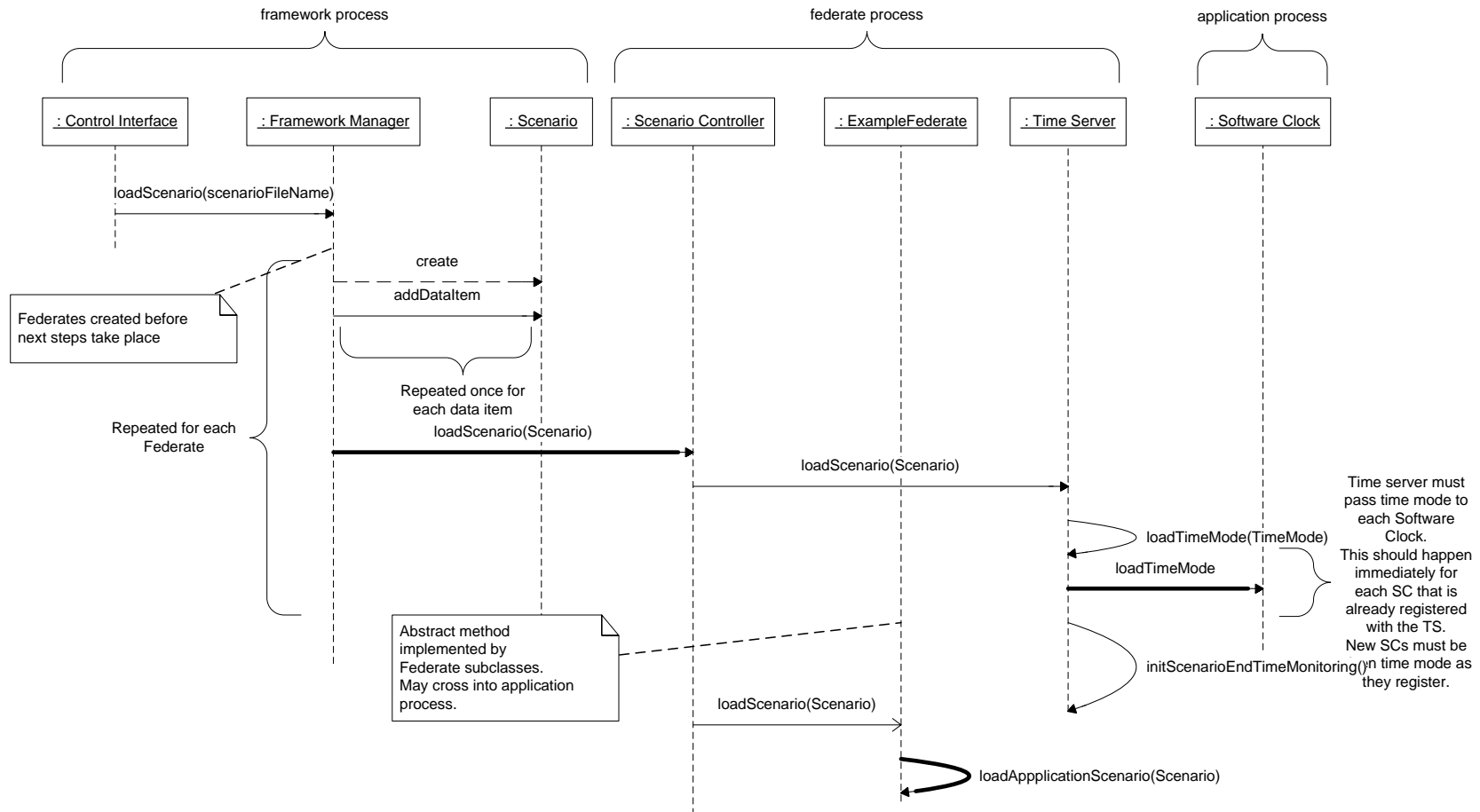
  <!-- Assume CEC is a federate type -->
  <DataItem federateType="CEC" instanceNumber="2">
    <!-- insert anything directed towards instance number
    2 of the CEC federate here -->
  </DataItem>

</Scenario Data>
```

Another class introduced in this Use Case is the Time Mode class. Time Mode contains indications of real or discrete time, and if real-time, what the scaling factor is. More attributes could be added as necessary.

Sequence Diagram(s)

Sequence Diagram for Use Case 20.1
User scripts a scenario...Scenario initial conditions are delivered to all federates.



4.5.2.3.7 Use Case 27.1

Summary

User specifies random number seed. This seed is sent to all federates as part of initial condition data.

Preconditions

Scenario and Federate objects have been created.

Triggers

Scenario data is sent to Federate objects.

Basic Course of Events

1. Federate object sets random number seeds into the Federate application code.
The way this is done would be Federate type-specific.

Alternate Paths

An alternative design decision would be to override the federate application's random number calls; see the design notes below.

Postconditions

All federates are given random number seeds.

Design Notes

It is not clear if random number support is required for FY08.

There are some difficult design issues associated with random number seeds. For FY08, the federates that might be affected would be the WASP radar simulations and the Aegis C&D model.

Issue #1: The way each federate handles random numbers is application-specific, and may vary greatly. This may require code inspection of each potential federate to decide how to handle random number generation, specifically how to “insert” random number seeds into a federate application.

Issue #2: The design has focused on shutting down federates by killing their application processes. Shutting down and then starting up application processes again for the next iteration of a monte-carlo run is the best way to ensure that all application data is purged for a new iteration. However, it is desired for an application to *not* reset its random

number generator (i.e. give it a random number seed) between iterations of monte-carlo runs.

If an application has the capability of resetting without being shut down completely, that approach could be taken between iterations. Then the random number sequences would not be interrupted.

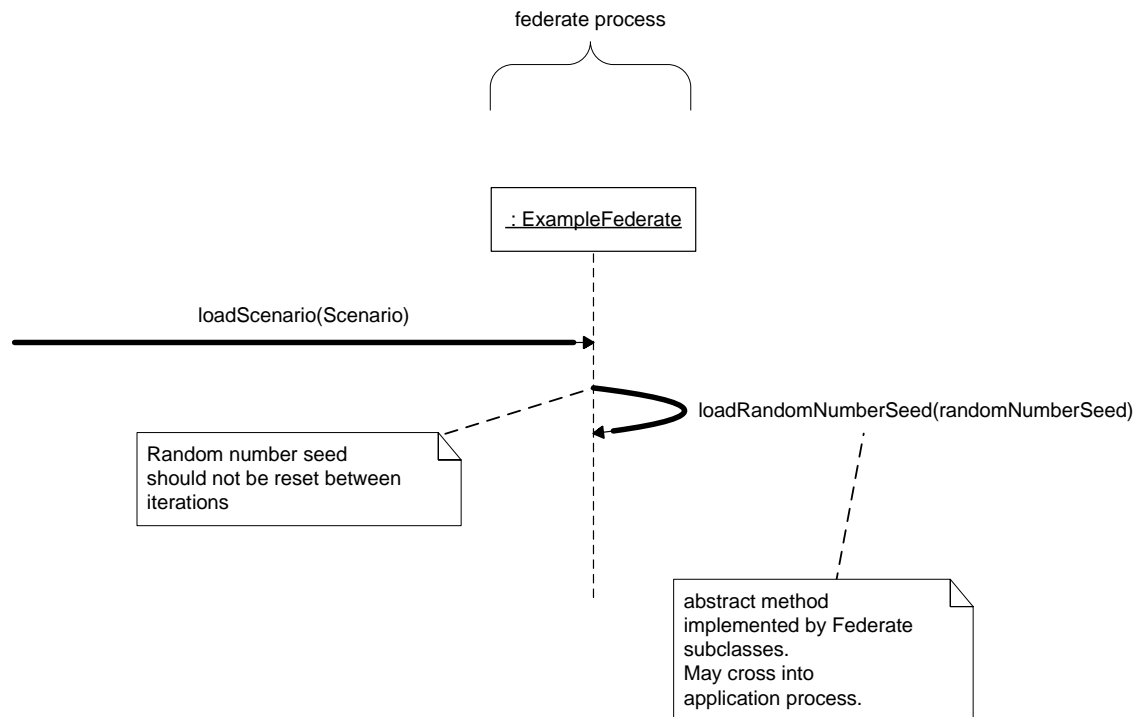
Another potential approach would be for the framework to supply its own random number generation calls, in the same manner in which the framework supplies time and I/O related calls. If that approach is taken, the framework would have ultimate control of random number sequences.

Further investigations into the use of random numbers in existing models, and the priority for FY08 should be taken into account before choosing an approach.

Sequence Diagram(s)

Sequence Diagram for Use Case 27.1

User specifies random number seed. This seed is sent to all federates as part of initial condition data.



4.5.2.3.8 Use Case 31.1

Summary

Control Interface receives start scenario message to start all federations from user interface. Precondition: Use case 30.1 has been executed. Post-condition: all federations are running their scripted scenario.

Preconditions

Use case 20.1 and its preconditions have been executed.

Triggers

Start scenario message comes in.

Basic Course of Events

1. Start scenario command sent to the Framework Manager on each framework process. This call specifies the wall clock time at which to start the scenario.
2. The Framework Manager tells each Scenario Controller to start the scenario.

The rest of the steps are performed by each Federate running in the current scenario.

3. The Scenario Controller tells the Time Server to start its clocks.
4. The Time Server tells each Software Clock registered with it what the starting wall clock time is. Each Software Clock that subsequently registers with the Time Server must be told this starting wall clock time as well.
5. The Scenario Controller tells the Federate to start the scenario. The Federate calls its *startApplicationScenario* method. This is an abstract method that is implemented in a Federate Type-specific manner. For some Federate Types, the method may be null if the application is already able to run a scenario. For others, such as a WASP Federate, the *startApplicationScenario* method would trigger execution of a scenario.

Alternate Paths

If *startApplicationScenario* fails to execute correctly within the given timeout, an exception is thrown. Note that this timeout needs to go to the real OS clock, not the Software Clocks.

Postconditions

All federations are running their scripted scenario.

Design Notes

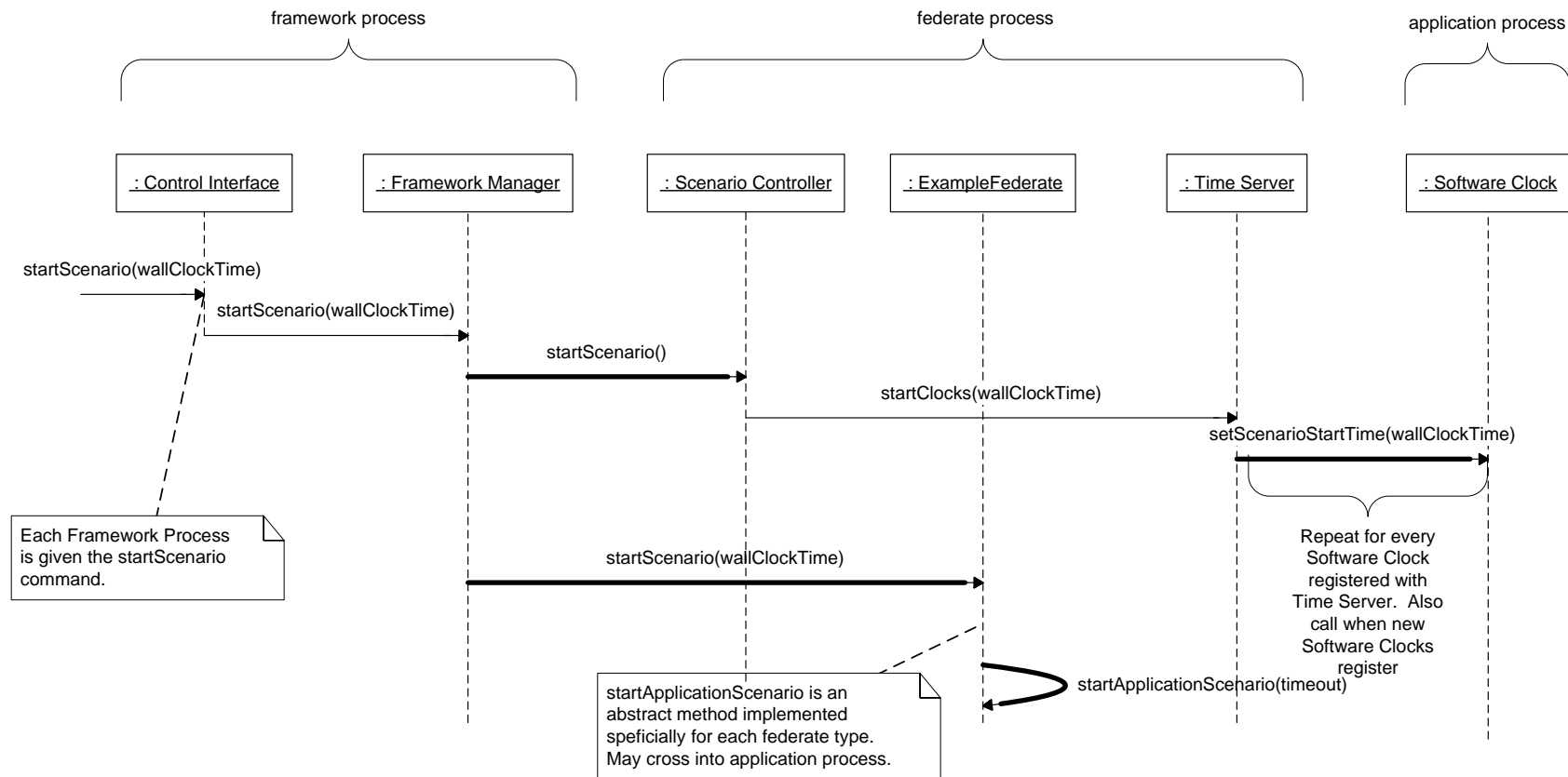
The purpose of specifying the wall clock time at which to start the scenario (step 1 above) is in order to provide all federates a common scenario start time. This is particularly important when running in real-time mode, since the wall clock time at which a scenario is started is used to determine current scenario time.

The wall clock time at which to start the scenario might be a short time in the future, such as 3 seconds from when use hits a start button. The delta between the start command and actually starting a scenario should be long enough to send the start command to all framework processes before the start time is arrived at.

Sequence Diagram(s)

Sequence Diagram for Use Case 31.1

Display interface receives start scenario message to start
all federations from user interface.



4.5.2.3.9 Use Case 32.1

Summary

Control Interface receives abort scenario message for all federations. Every federate stops running and purge its data. Postcondition: no federations are running.

Preconditions

Scenario is currently running.

Triggers

Abort scenario message received from user interface.

Basic Course of Events

1. The Control Interface receives an *abortScenario* command, given the federation name whose scenario is to be aborted.
2. The Control Interface calls the Framework Manager's *endScenario* method.
3. The Framework Manager calls *endScenario* on the Scenario Controller in every federate process that is taking part in the given federation.
4. The Scenario Controller calls *endScenario* on each Federate, which in turn calls *endApplicationScenario* on itself.
5. The Scenario Controller calls *reset()* on the Time Server. The Time Server puts itself into a dormant state.
6. The Time Server calls *reset()* on each of its registered Software Clocks.

Alternate Paths

If step 4 above fails, an *ApplicationScenarioNotEndedException* is thrown.

Postconditions

No federations are running scenarios.

Design Notes

Though we only showed the sequence of events for one particular federation, it should scale easily for multiple federations.

The Federate's *endApplicationScenario(timeout)* method is required to purge all application data. How this is accomplished is specific to each Federate Type. One way would be to simply kill all application processes. Another way would be to call a

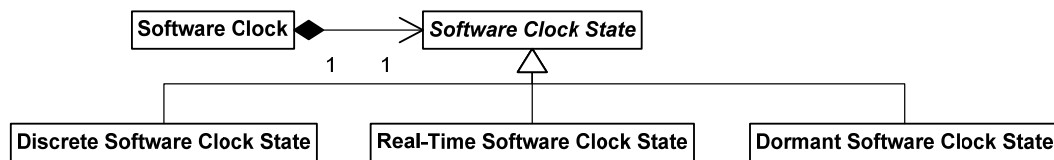
federate application “purge” function, if such a function already exists for that application.

The implementation of *endApplicationScenario(timeout)* is responsible for determining success or failure. If it fails, an exception is thrown.

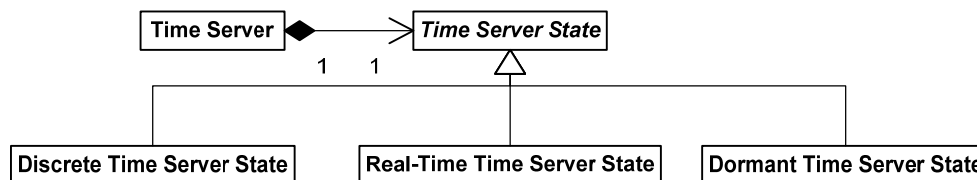
The *reset()* call puts Time Server into a dormant state, and in turn Time Server tells all its software clocks to do the same. This might lead to a third subclass of Time Server and of Software Clock – Dormant Time Server and Dormant Software Clock.

Note that if the Federate’s *endApplicationScenario(timeout)* call kills federate processes, processes that have Software Clocks registered with the Time Server will ungracefully be destroyed. The Time Server currently handles this by checking for a dropped TCP connection with its registered software clocks. All calls from Time Server to Software Clock should also be wrapped in exception handlers, which could disconnect the Software Clock if sending a command fails.

Since Software Clock and Time Server will have to change modes dynamically, the state pattern might be a good choice here. For example:



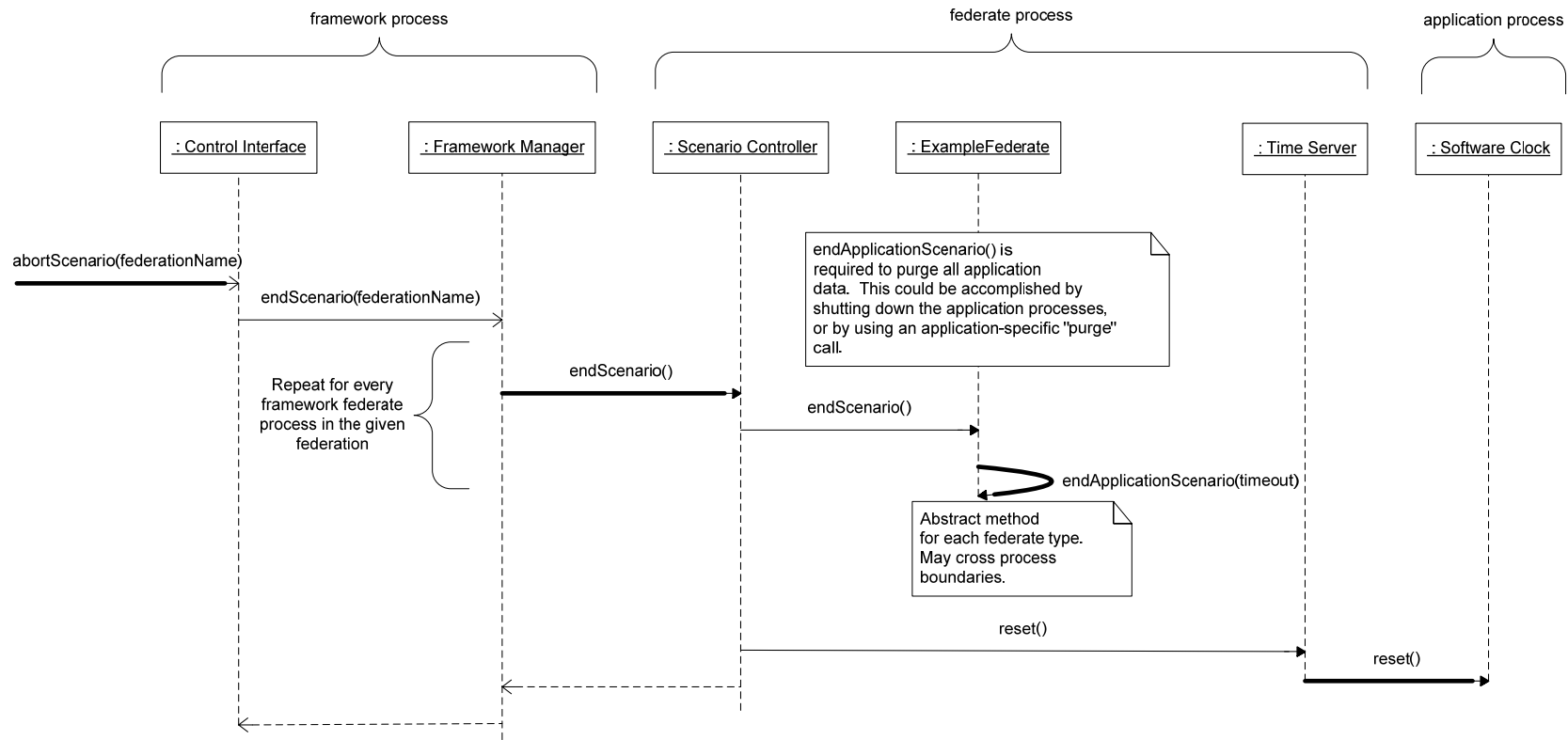
and



Note also that the design calls for the Software Clocks to call the operating system directly when in scaled real-time mode, the Real-Time Time Server State class may not be needed. Or it could simply be a null object.

Sequence Diagram(s)

Sequence Diagram for Use Case 32.1
Display interface receives abort scenario message for all
federations. Every federate stops running and purge its
data.



4.5.2.3.10 Use Case 33.2

Summary

Framework needs to determine whether scenario end time has been reached, for a given federation.

Preconditions

Scenario is running.

Triggers

Triggered by the call to *TimeServer::initScenarioEndTimeMonitoring()* - see Use Case 20.1

Basic Course of Events

Sub Use Case a: Scenario is executing in discrete time.

1. Time Server calls *initScenarioEndTimeMonitoring()* on itself.
2. This calls *monitorScenarioEndTime(scenarioEndTime)* on the active instance of Time Server State, which is Discrete Time Server State in this case.
3. The Discrete Time Server State object stores off the scenario end time.
4. Each time the Time Server gets time granted, current time is passed to the Discrete Time Server state using the *setCurrentTime(currentTime)* call.
5. If the boolean:
$$\text{currentScenarioTime} - \text{scenarioStartTime} \geq \text{scenarioRunTime}$$
is true, then the end of the scenario has been reached. If so, proceed to step 6, if not, this sequence of events ends.
6. The Discrete Time Server State calls *ScenarioController::scenarioEndTimeReached()*. In turn, the Scenario Controller calls *FrameworkManager::scenarioEndTimeReached()*.
7. If multiple Federates are executing on a single computer, the Framework Manager may receive this call multiple times. The first time it is called while a scenario is running, the Framework Manager calls *ControlInterface::scenarioEndTimeReached()*, which in turn is forwarded to the User Interface application.
8. The first time the User Interface gets a *scenarioEndTimeReached()* call while a scenario is running, it calls *abortScenario* on each Framework Process on each computer running the framework. From there, the sequence of events is identical to that of Use Case 32.1.

Sub Use Case b: Scenario is executing in real-time.

1. Time Server calls *initScenarioEndTimeMonitoring()* on itself.
2. This calls *monitorScenarioEndTime(scenarioEndTime)* on the active instance of Time Server State, which is Real-Time Time Server State in this case.
3. The Real-Time Time Server sets up a thread that sleeps for a wall-clock time equal to $\text{scenarioEndTime} / \text{timeScale}$.
4. Once the thread awakes, the scenario end time still may not have been reached, due to pauses. Sleep for the amount of time equal to total pause time. When this thread wakes up, see if there was any more pause time added, or if the scenario is currently in pause mode.
5. Once the boolean check:

$\text{currentWallTime} - \text{scenarioStartTime} - \text{totalPausedTime} \geq \text{scenarioRunTime}$

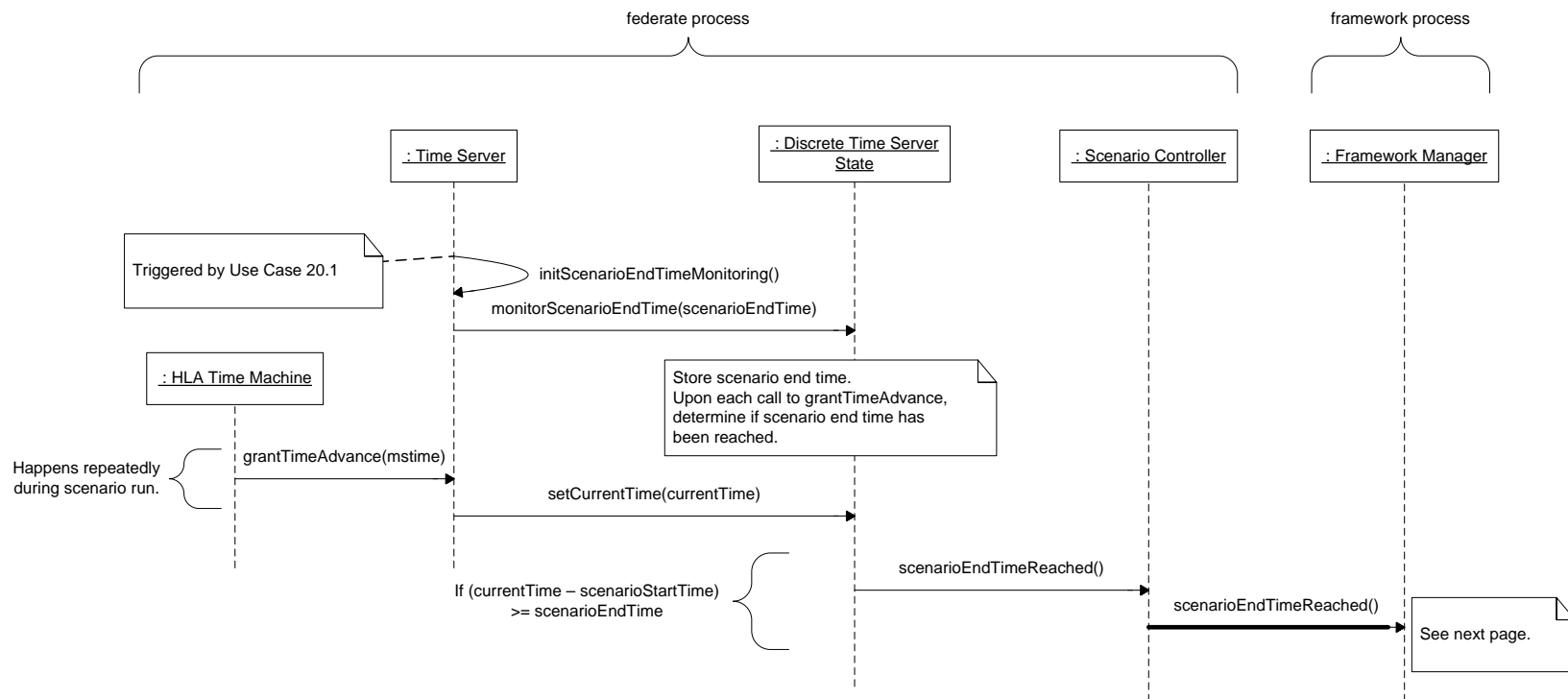
is true, proceed to step 6.

6. Execute step 6 and subsequence steps from sub use case a as described above.

Sequence Diagram for Use Case 33.2

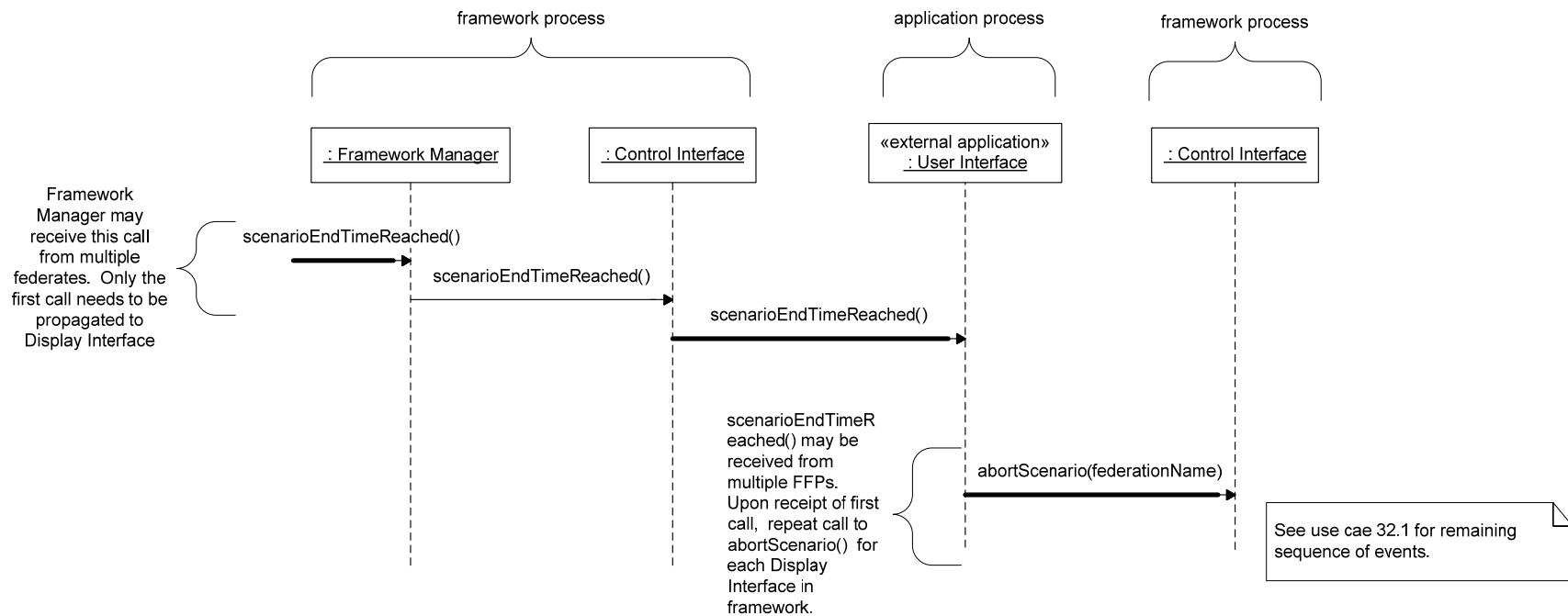
Framework needs to determine whether scenario end time has been reached for a given federation.

Sub-case a: Scenario is executing in discrete time.
Diagram 1 of 2



Sequence Diagram for Use Case 33.2
Framework needs to determine whether scenario end
time has been reached for a given federation.

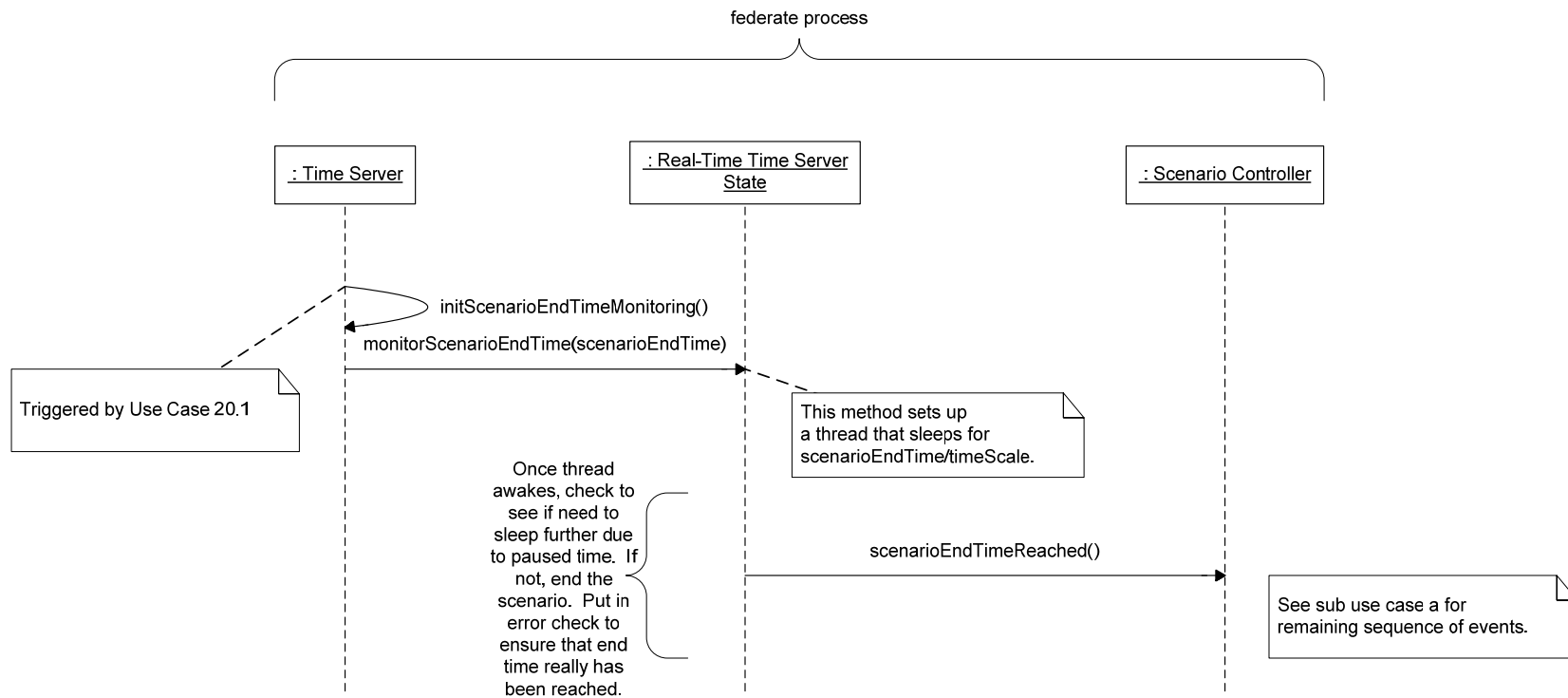
Sub-case a: Scenario is executing in discrete time.
Diagram 2 of 2.



Sequence Diagram for Use Case 33.2

Framework needs to determine whether scenario end time has been reached for a given federation.

Sub-case b: Scenario is executing in real time.



4.5.2.3.11 Use Case 34.1

Summary

User scripts multiple scenarios to be run. This could be the same scenario, multiple runs for monte-carlo, or completely different scenarios.

Preconditions

None

Triggers

N/A

Basic Course of Events

Offline: user scripts batch file (as described in Design Notes below) either manually or through a helper tool.

Online:

1. User Interface (or some sort of controlling application) loads batch file.
2. For the first Run, use information in the ConfigurationFile to identify computers participating in the framework.
3. User Interface calls *ControlInterface::loadConfiguration(configFileName)* on each Framework Process. The subsequent chain of events is documented in Use Case 10.4.
4. User Interface calls *ControlInterface::loadScenario(scenarioFileName)* on each Framework Process. The subsequent chain of events is documented in Use Case 20.1.
5. User Interface calls *ControlInterface::startScenario()* on each Framework Process. The subsequent chain of events is documented in Use Case 31.1
6. The scenario completes, as described in Use Case 33.2.
7. If there are more than one iterations of the scenario scheduled for execution, repeat steps 5 and 6 for each iteration.
8. Repeat steps 2 through 7 for each additional Run in the batch file.

Alternate Paths

None.

Postconditions

Framework has stored up information about each scenario, so that it can run each scenario sequentially without requiring interaction from user.

Design Notes

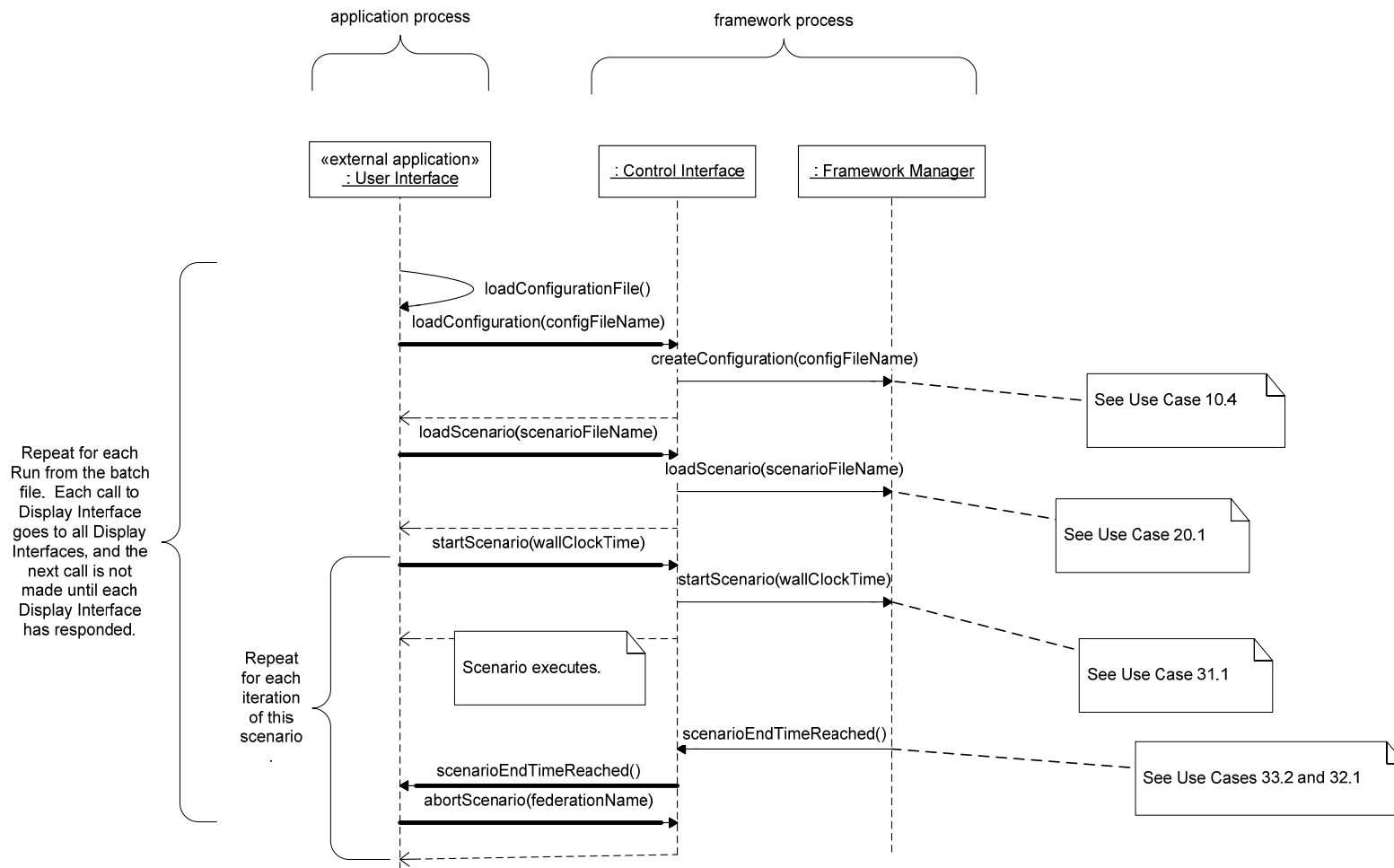
A simple configuration file can be used to script multiple scenarios. Reusing concepts from earlier use cases, in which a Configuration laid out the federate instances and communications between them (see use case 10.4) , and a Scenario which laid out scenario information including number of iterations (see use case 20.1) , a file with information such as this can be used:

```
<Batch File>
  <Run>
    <ConfigurationFile name="configFile1.xml" />
    <ScenarioFileName name="scenarioFile1.xml" />
  </Run>
  <Run>
    <ConfigurationFile name="configFile2.xml" />
    <ScenarioFileName name="scenarioFile2.xml" />
  </Run>
</Batch File>
```

The Framework Manager should have a graceful way of clearing out old Configurations and Scenario objects when a new one starts. Also, if a Run uses the same Configuration file as a previous Run, or different but identical files, the steps of creating a new Configuration can be avoided – just use the old one. The same can be done for the Scenario objects, but only if the Configuration is also identical.

Sequence Diagram(s)

Sequence Diagram for Use Case 34.1
User scripts multiple scenarios to be run. This could be
the same scenario, multiple runs for monte-carlo, or
completely different scenarios.



4.5.2.3.12 *Use Case 51.2*

Summary

A scenario is started using distributed federates. The framework code associated with each federate needs to hold the same scenario start time (from the OS time) when running in a real-time or scaled real-time mode.

Design Notes

The sequence of events for starting a scenario is already covered by Use Case 31.1, which highlights the need to synchronize scenario start time among federates.

The mechanism selected for synchronizing the start time is to have the controlling application (i.e. the user interface) select a wall clock time in the near future – close enough in the future that the end user won't get impatient waiting for the scenario to start, but far enough in the future that each federate can receive the start time in advance, and prepare to start at exactly that time.

The start time is not the only time attribute that needs to be synchronized when running in real-time. The list of attributes that need to be synchronized includes:

- Scenario Start Time
- Scenario Pause Time (for a single pause command)
- Scenario Resume Time (for a single resume command)
- Total Scenario Pause Time

Each of the above attributes can be synchronized using the same technique.

There are no new sequences of events here.

4.5.2.3.13 *Use Case 90.1*

Summary

Framework should have interface to display that accommodates:

- all scenario controls (start, stop, pause, change time scale).
- scenario scripting – unless we just want users to script through files at the beginning. If nothing else, the name of the file we’re going to use.
- a generic mechanism to support new data fields as new federates are added in the future would be nice.

Design Notes

No new requirements or sequences of events are introduced through this use case. The function calls into the Control Interface class should be an indicator of what the display (and its interface) should accommodate. Use cases 10.4, 20.1, and 34.1 describe potential configuration files and their data.

Sequence Diagram(s)

N/A.

4.5.2.3.14 *Use Case 90.2*

Summary

If the framework is running in distributed mode, there should be one display for the whole framework. Perhaps there should be some sort of framework configuration file that identifies one of the distributed framework instances to be the controller and talk to the UI. Or maybe the UI connects to every framework instance.

Design Notes

An approach was decided on for FY08 that would have one controlling application (which may double as the user interface application) that would talk to the Control Interface object in each framework process.

The configuration file, described in 10.4, would need to identify which computers each Federate instance is running on. Refer to use case 10.4, which has been updated with those details.

Sequence Diagram(s)

N/A.

4.5.2.3.15 *Use Case 91*

Summary

Create a Framework Display.

Design Notes

Non-critical design, deferred until a later date. If there is a well-defined interface to a display, a simple display can be developed later with minimal risk to the rest of the framework, hopefully fairly quickly. If the display doubles as a controlling application some safety mechanisms need to be coded.

Sequence Diagrams

No new sequence diagrams. Some of the User Interface application's sequences of events are described in the other sequence diagrams.

Alternate Paths

Postconditions

Postcondition: If scenario end time hasn't been reached, scenario keeps running. If scenario end time has been reached, framework is ready to end scenario.

Design Notes

Each state of the time server has responsibility for determining when scenario end time has been reached. This is because the mechanism for determining this would be different for different time modes.

This current design calls for the Real-Time Time Server to have no need to know current scenario time, and takes it on faith that the scenarioStartTime plus the totalPausedTime equals the equivalent actual time that the scenario has been running. This is determined this way so that the Time Server does not have to contact any Software Clocks in order to determine current scenario time. Note that the pause feature is not critical for real-time mode.

Sequence Diagram(s)

N/A

4.5.3 Time Management

The following sections describe the classes and use cases associated with Time Management. The basic idea is that the Federate Applications run in simulation time, as distinct from “wall clock” (or “real”) time. Whenever they request the current time or make any call that involves time (e.g., wait for an incoming message with a timeout), the framework must somehow make sure that simulation time is used, even if the application calls an operating system service. In addition, the framework supports two types of time: discrete and scaled real-time, as described in the next section.

In order to accomplish these goals, the design calls for use of an “interposition library”: basically a shared library that contains functions that override the operating system’s time-related services (see Figure 5). This library is built using two classes: the Simulation Clock and the Software Clock. During initialization, the Time Server initializes the Simulation Clock with its mode and parameters (discrete or scaled real-time). Before this initialization, time cannot advance, thus forcing the applications to wait. After initialization, the interposition library vectors each time-related call to the Simulation Clock, which, depending on the mode, either invokes the Software Clock (for discrete mode) or offsets and scales the wall clock time appropriately (for scaled real-time). For an overview of the classes and interactions involved in Time Management, refer to Figure 6.

4.5.3.1 Class Descriptions

Software Clock

Handles time-related calls from applications. Behavior depends on the time mode the clock is operating in: scaled real-time or discrete. Differences in behavior are handed by subclasses of the Software Clock State. One instance of the software clock exists for each application process.

Software Clock State

Abstract base class for the various states that a software clock can operate in. Current states are scaled real-time, discrete time, and a null state when no simulation is running.

Discrete Software Clock State

A state of the software clock in which application time-related calls are intercepted, treated as requests for time advances, and the lowest time advance request is forwarded on to the Time Server.

Dormant Software Clock State

A state of the software clock in which no time-related behavior is performed. This is essentially a null object, and is used when no scenario is running.

Real-Time Software Clock State

A state of the software clock in which application time-related calls are intercepted, their time values scaled according to the scenario's time-scale factor, and then sent to their intended destination (likely a POSIX or operating system call).

Time Server

Responsible for handling time-related functionality for a given federate, i.e. one Time Server is created for each federate. Responsible for coordinating with the software clocks to start, pause, resume, and end scenarios. Differences in behavior between various time modes are handled by subclasses of the Time Server State

Time Server State

Abstract base class for the various states that a time server can operate in. Current states are real-time, discrete-time, and a null state when no simulation is running.

Discrete Time Server State

Handles Time Server behavior when running in discrete-time mode. In this mode, time advance requests are collected from the associated federate application's software clocks, and the smallest request is forwarded on to the Time Machine. Additionally, scenario pauses and resumes are implemented within in this class, by withholding or releasing, respectively, requests to the Time Machine. The behavior in this class is based off of the PRA test-bed's Time Server.

Dormant Time Server State

A state of the time server in which no time-related behavior is performed. This is essentially a null object, and is used when no scenario is running.

Real-Time Time Server State

A state of the Time Server used when running in a real-time mode. When in this state, the Software Clocks are responsible for scaling time and redirecting calls without the time server's involvement, however control commands (start, stop, pause, resume) are still sent to the Software Clocks.

Time Machine

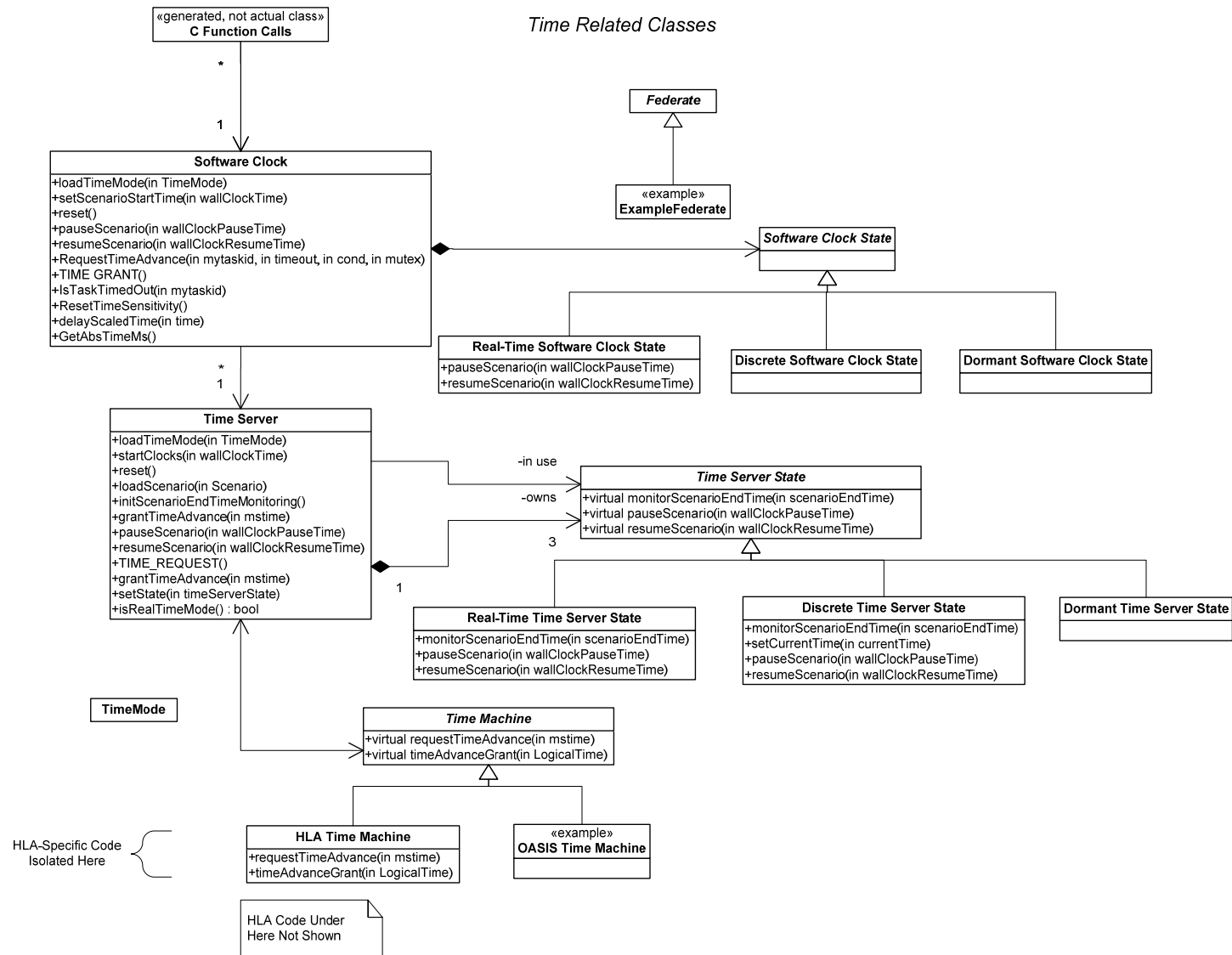
Abstract class that serves as an interface for handling time requests and time grants from the Time Server. This is primarily intended when operating in a discrete time mode.

HLA Time Machine

Implementation of the Time Machine when using HLA/RTI as the middleware for advancing time. Responsible for making time requests to the RTI, and delivering time grants from the RTI back to the Time Server. This class will likely contain functionality similar to that of the “RTI Converter” from the PRA test-bed.

Time Mode

Indicates a real-time or discrete time mode.



4.5.3.2 Library Interposition

The following lists the general features and advantages of using interposition libraries:

- Used to provide different behavior for O/S functions
- Applications use O/S shared libraries and dynamic linking
- Can intercept any function calls an application makes to a shared library
- An interposed function can do anything, even calling the “real” function
- No application changes are required—works even if only an executable is available
- Focus here is on POSIX time-related calls

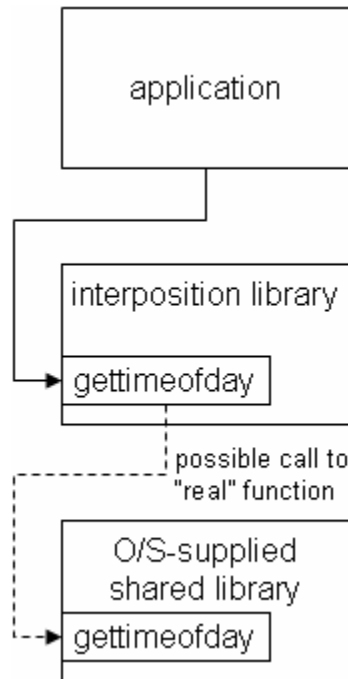


Figure 5: Interposition Example of Calling “gettimeofday”

The following describes the use of the Software Clock interposition library for intercepting and modifying the relevant time-related calls:

- Overrides time-related POSIX calls
- IPC to Time Server in Federate process
- Initialized from time server (e.g., mode, simulation start time, and scale factor)
- Can be in one of two modes:
 - Scaled real-time:
$$T_s = T_{\text{start}} + \text{Scale} * (T_w - T_{w\text{Start}} - T_p)$$
where T_s is simulation time, T_{start} is simulation start time, Scale is the scale

factor, T_w is the current wall-clock time (from NTP synchronized computer clock), T_{wStart} is the wall-clock time at simulation start, and T_p is the total wall-clock time the simulation was paused.

- Discrete: simulation time from Time Server and ultimately HLA RTI

Following is the current list of interposed calls, as required for CEC CGAI middleware:

- clock_gettime
- connect
- fgets
- gettimeofday
- nanosleep
- pause
- pthread_cancel
- pthread_cond_wait
- pthread_cond_timedwait
- pthread_exit

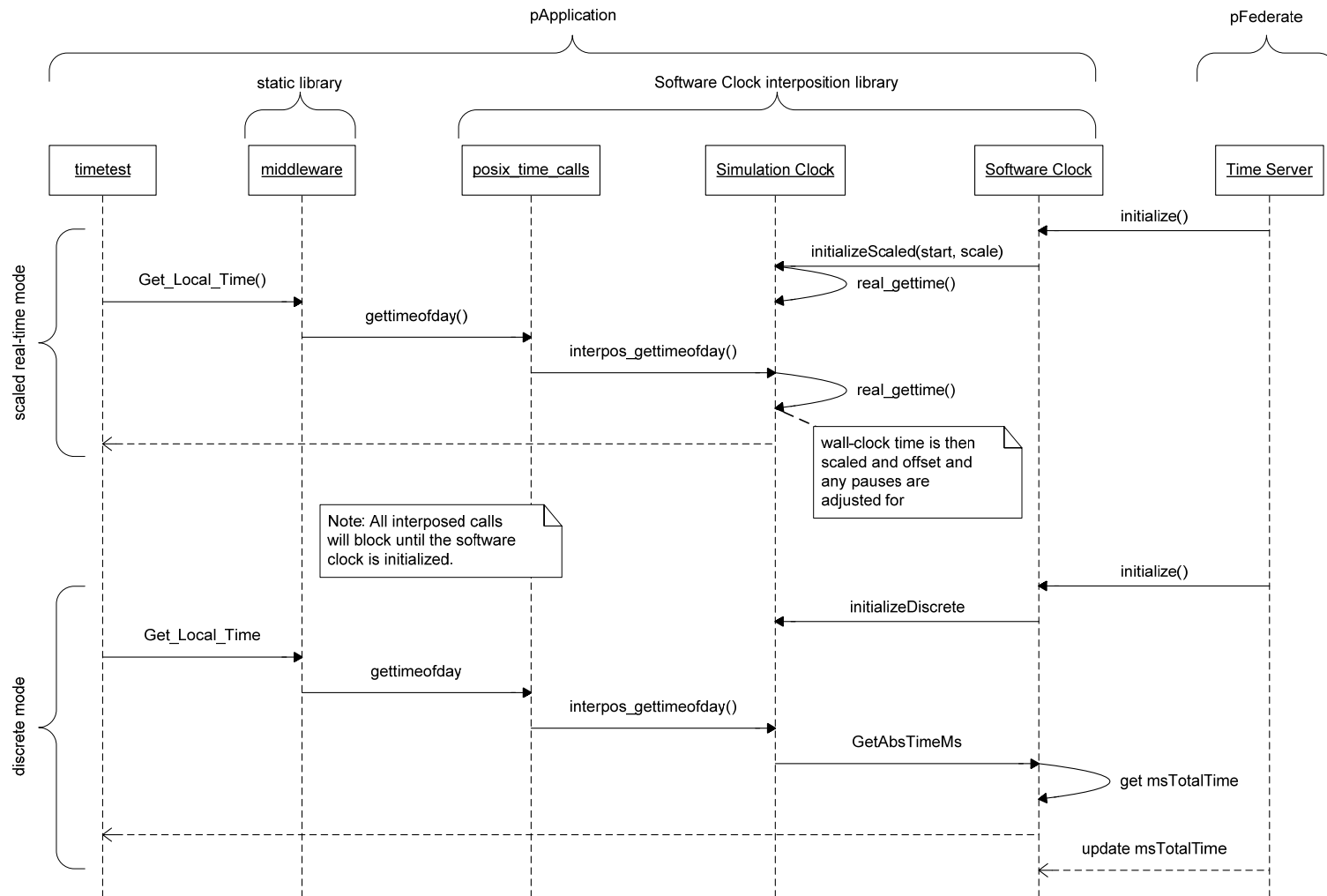


Figure 6: Sequence Diagram Example for Software Clock Interposition Library

4.5.3.3 Use Cases

4.5.3.3.1 Use Case 35.1

Summary

Pause command is received from the user interface, resulting in software clock being paused.

Preconditions

Scenario is currently running.

Triggers

Pause command received through User Interface / Control Interface.

Basic Course of Events

Sub Use Case a: Scenario is running in discrete time mode.

1. A *pauseScenario(wallClockPauseTime)* command is given to each framework process.
2. Each Framework Manager propagates the *pauseScenario(wallClockPauseTime)* command to the Scenario Controller in each federate process.
3. The Scenario Controller calls *pauseScenario(wallClockPauseTime)* on the Time Server, which subsequently calls *pauseScenario(wallClockPauseTime)* on the Discrete Time Server State object.
4. The Discrete Time Server State object goes into a “paused” state. This means that it will continue to collect time requests from software clocks, but it will not request time grants from the Time Machine until both of these conditions have been met:
 - Requests from all Software Clocks have been received.
 - The scenario is unpaused.

Sub Use Case b: Scenario is running in real-time mode.

1. A *pauseScenario(wallClockPauseTime)* command is given to each framework process.
2. Each Framework Manager propagates the *pauseScenario(wallClockPauseTime)* command to the Scenario Controller in each federate process.
3. The Scenario Controller calls *pauseScenario(wallClockPauseTime)* on the Time Server, which subsequently calls *pauseScenario(wallClockPauseTime)* on the Real-Time Server State object.

4. The wall-clock time of the pause command is stored off. This is so that total pause time can be recorded. Note: this needs to be synchronized between all Federates.
5. A *pauseScenario(wallClockPauseTime)* command is sent to each Software Clock that is registered with the Time Server. Software Clocks that subsequently register with the Time Server are also put into a paused state, assuming that the scenario is still paused.
6. The Software Clock passes the call to the Real-Time Software Clock State object. This sets the Software Clock into a paused state. Each time request, instead of being scaled and sent to the operating system, is blocked, preventing the application thread from regaining control. The time requests remain blocked until unpaused.

Alternate Paths

None.

Postconditions

Each federate is not currently executing the scenario, but is capable of picking up where it left off once the system clock unblocks.

Design Notes

Synchronizing the pause time between all federates can be accomplished in a similar model to what is described for scenario start time, as described in use case 31.1. For example, a user hits the “pause” button on the display, and a wall clock time three seconds from now is passed to each framework process as the pause time. The three second delay would be used to help insure that all federates are able to set the pause time before it happens.

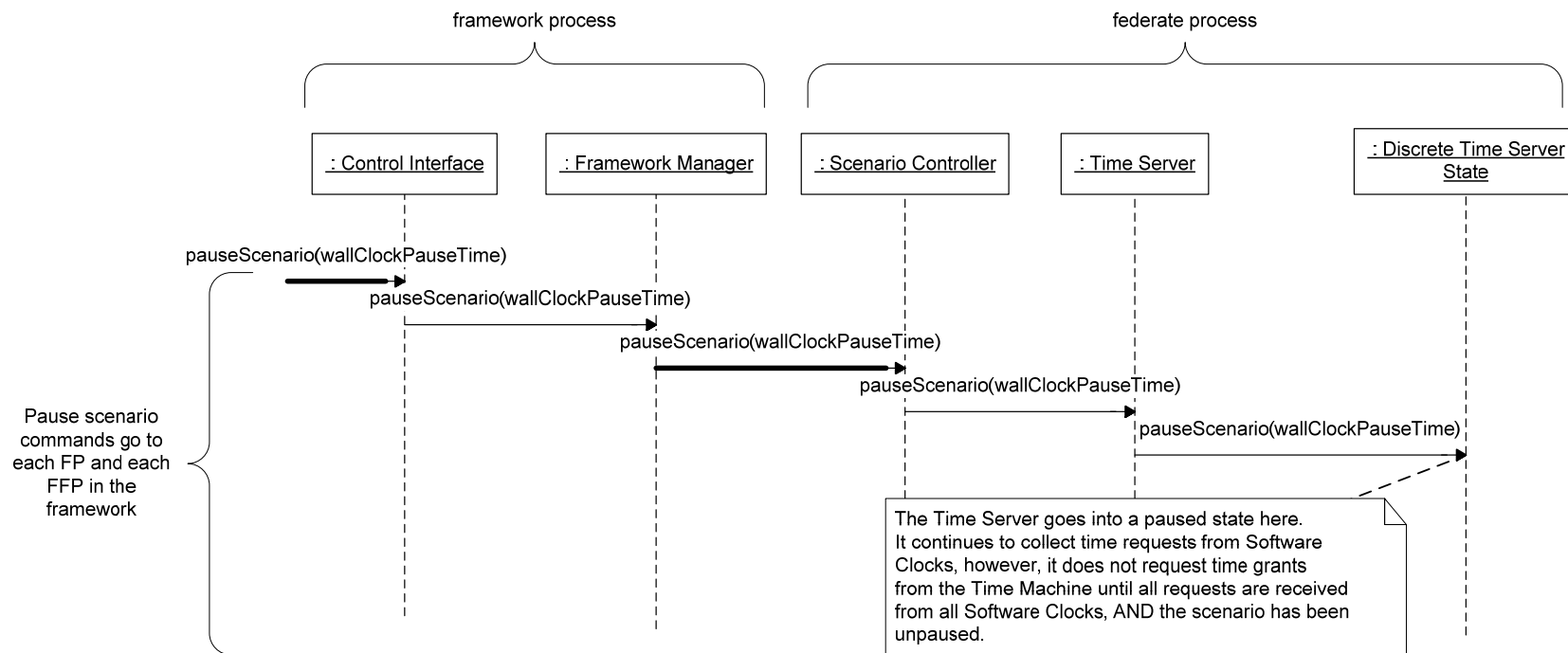
When running in a real-time mode, and a pause command is received while some application threads are sleeping, or are in some other blocking call, is the start time of the pause going to be when the pause command is received, or is it going to be after current blocking calls are returned? This is not a critical issue as pausing a scenario is not a critical feature when in real-time mode.

Sequence Diagram(s)

Sequence Diagram for Use Case 35.1

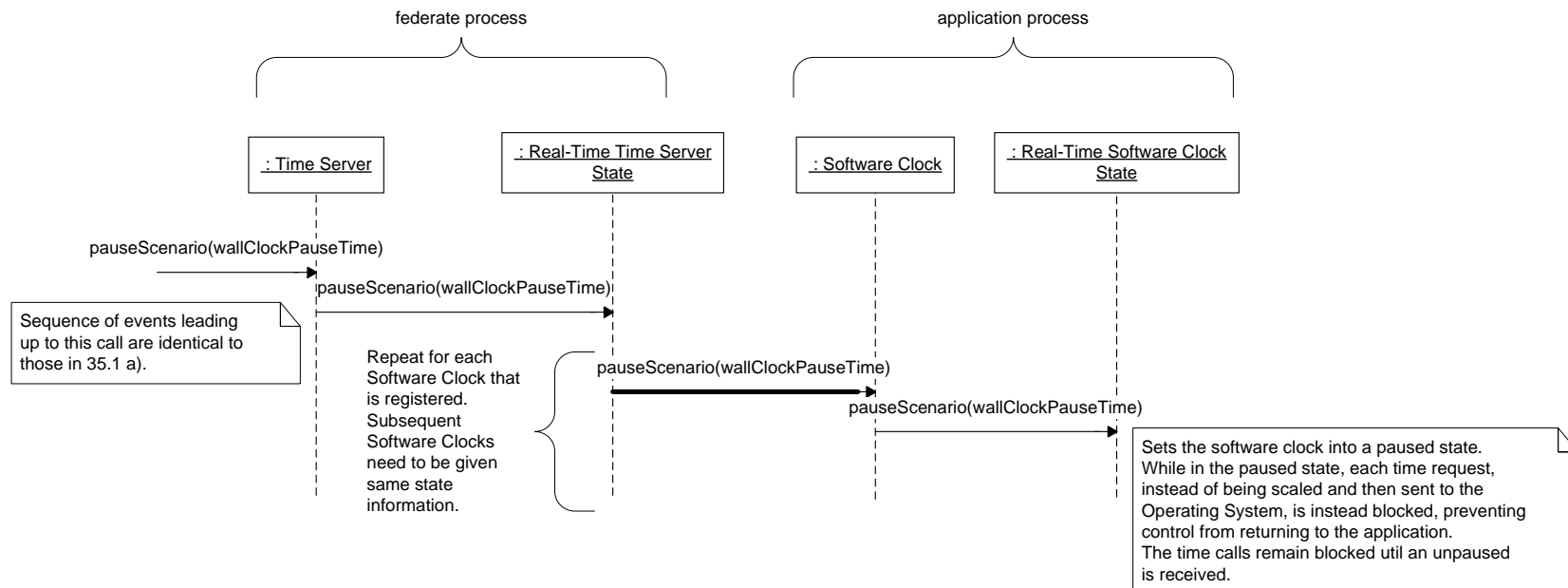
Pause command is received from the user interface,
resulting in software clock being paused.

Sub Use Case a: Scenario is running in discrete time
mode.



Sequence Diagram for Use Case 35.1
Pause command is received from the user interface,
resulting in software clock being paused.

Sub Use Case b: Scenario is running in real-time mode.



4.5.3.3.2 Use Case 35.2

Summary

A “resume” command is received from the user interface, resulting in a paused scenario executing again where it left off.

Preconditions

A previously running scenario is paused.

Triggers

Resume command received through User Interface / Control Interface.

Basic Course of Events

Sub Use Case a: Scenario is running in discrete time mode.

1. A *resumeScenario(wallClockResumeTime)* command is given to each framework process.
2. Each Framework Manager propagates the *resumeScenario(wallClockResumeTime)* command to the Scenario Controller in each federate process.
3. The Scenario Controller calls *resumeScenario(wallClockResumeTime)* on the Time Server, which subsequently calls *resumeScenario(wallClockResumeTime)* on the Discrete Time Server State object.
4. The Discrete Time Server State object clears its paused state. This means that it is now free to make time grant requests to the Time Machine. If all Software Clocks were already waiting for time at the time of the unpause command, then the Time Server goes ahead and requests a Time Advance. Processing continues as before the pause.

Sub Use Case b: Scenario is running in real-time mode.

1. A *resumeScenario(wallClockResumeTime)* command is given to each Framework Process.
2. Each Framework Manager propagates the *resumeScenario(wallClockResumeTime)* command to the Scenario Controller in each federate process.
3. The Scenario Controller calls *resumeScenario(wallClockResumeTime)* on the Time Server, which subsequently calls *resumeScenario(wallClockResumeTime)* on the Real-Time Server State object.

4. The wall-clock time of the resume command is used to determine the total pause time, for this pause command. The pause time of this command is added to the total pause time. The Note: this needs to be synchronized between all Federates.
5. A *resumeScenario(wallClockResumeTime)* command is sent to each Software Clock that is registered with the Time Server.
6. The Software Clock passes the call to the Real-Time Software Clock State object. This clears the Software Clock from its paused state. Each currently blocked time request is now scaled and sent to the operating system. Subsequent time requests are scaled and sent to the operating system, as before.

Alternate Paths

If the scenario is not currently paused, the Scenario Controller discards the resume command at step 2.

Postconditions

The scenario is no longer paused, and all federates are running.

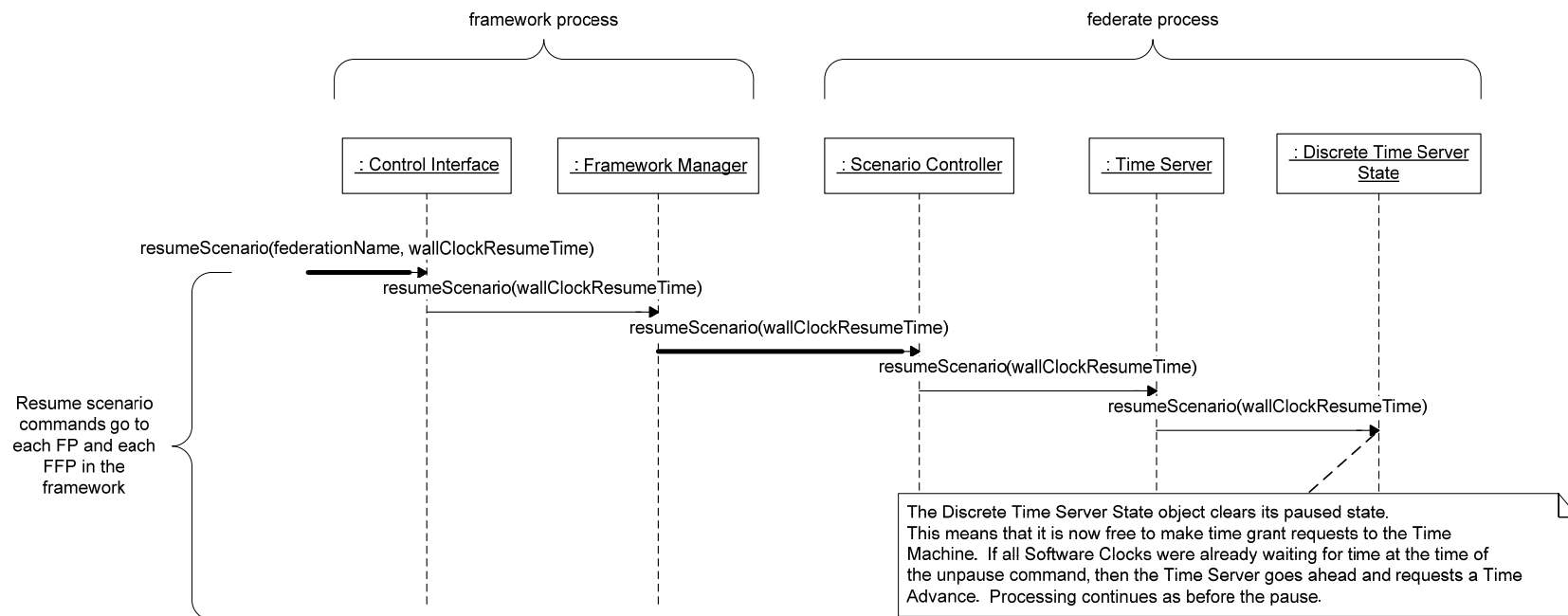
Design Notes

Synchronizing the resume time among all framework processes can be done in the same matter as synchronizing the scenario start and pause times (see Use Cases 31.1 and 35.1).

Sequence Diagrams.

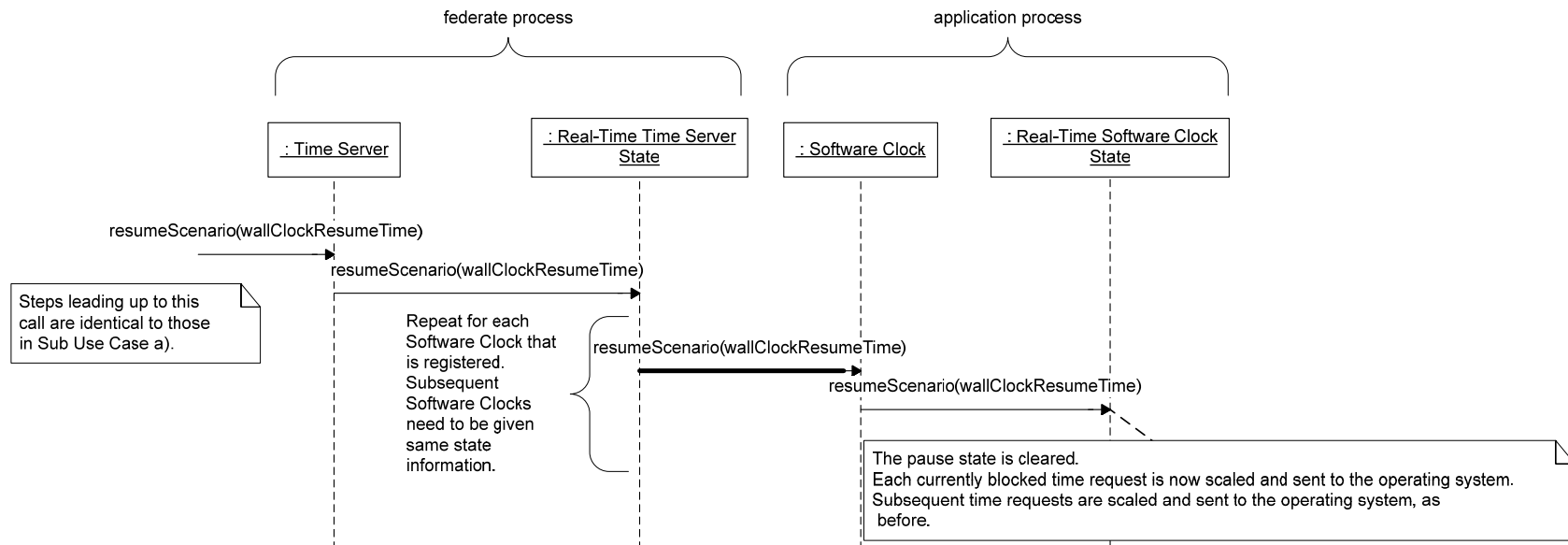
Sequence Diagram for Use Case 35.2
A "resume" command is received from the user
interface, resulting in a paused scenario executing again
where it left off.

Sub Use Case a: Scenario is running in discrete time
mode.



Sequence Diagram for Use Case 35.2
A "resume" command is received from the user interface, resulting in a paused scenario executing again where it left off.

Sub Use Case b: Scenario is running in real-time mode.



4.5.3.3.3 Use Case 45.2

Summary

A process in a federate makes a blocking time-related call. That call is scaled and then routed to the operating system's clock. Note that we are going to have to identify every time related call that an application could make, in order to turn that application into a federate.

Preconditions

The Software Clock is initialized to operate in scaled real-time mode. The simulation start time and scale factor must have already been supplied.

Triggers

Application makes a blocking time-related call to either a legacy middleware layer or a POSIX system call.

Basic Course of Events

For legacy middleware library: representative example is semTake

1. Application invokes semTake with a finite (non-zero) timeout.
2. Middleware library issues RequestTimeAdvance to Software Clock, and then issues the blocking pthread_cond_wait POSIX call, awaiting either a semGive to be issued or a timeout to occur.
3. The Software Clock delays for the requested simulation time. The following formula relates simulation time, T_s , to the current wall-clock time, T_w , as follows:
$$T_s = T_{\text{start}} + \text{Scale} * (T_w - T_{w\text{Start}} - T_p)$$
where T_{start} is simulation start time, Scale is the scale factor, $T_{w\text{Start}}$ is the wall-clock time at simulation start, and T_p is the total wall-clock time the simulation was paused.
4. When the timeout expires, the Software Clock issues the pthread_cond_broadcast POSIX call (unless semGive had already given the semaphore; see Alternate Paths below).
5. The Middleware Library then calls the IsTaskTimedOut method in the Software Clock to determine if a time-out has occurred.
6. Since the timeout has occurred, the semTake method returns a timeout status.

For interposition library: representative example is pthread_cond_timedwait

1. Application invokes pthread_cond_timedwait with a time in the future.

2. The interposition library issues RequestTimeAdvance to Software Clock, and then issues the blocking pthread_cond_wait POSIX call, awaiting either a pthread_cond_broadcast to be issued or a timeout to occur.
3. The Software Clock delays for the requested simulation time (see step 3 above).
4. When the timeout expires, the Software Clock issues the pthread_cond_broadcast POSIX call (unless another application thread had already called pthread_cond_broadcast; see Alternate Paths below).
5. The Middleware Library then calls the IsTaskTimedOut method in the Software Clock to determine if a time-out has occurred.
6. Since the timeout has occurred, the pthread_cond_timedwait method returns a timeout status.

Alternate Paths

If semGive or pthread_cond_broadcast was called by an application thread before the timeout had occurred, the Middleware or Interposition Library, respectively, wakes up and then issues the IsTaskTimedOut call to the Software Clock. The effect of issuing this call is to cancel the pending timeout.

Postconditions

None.

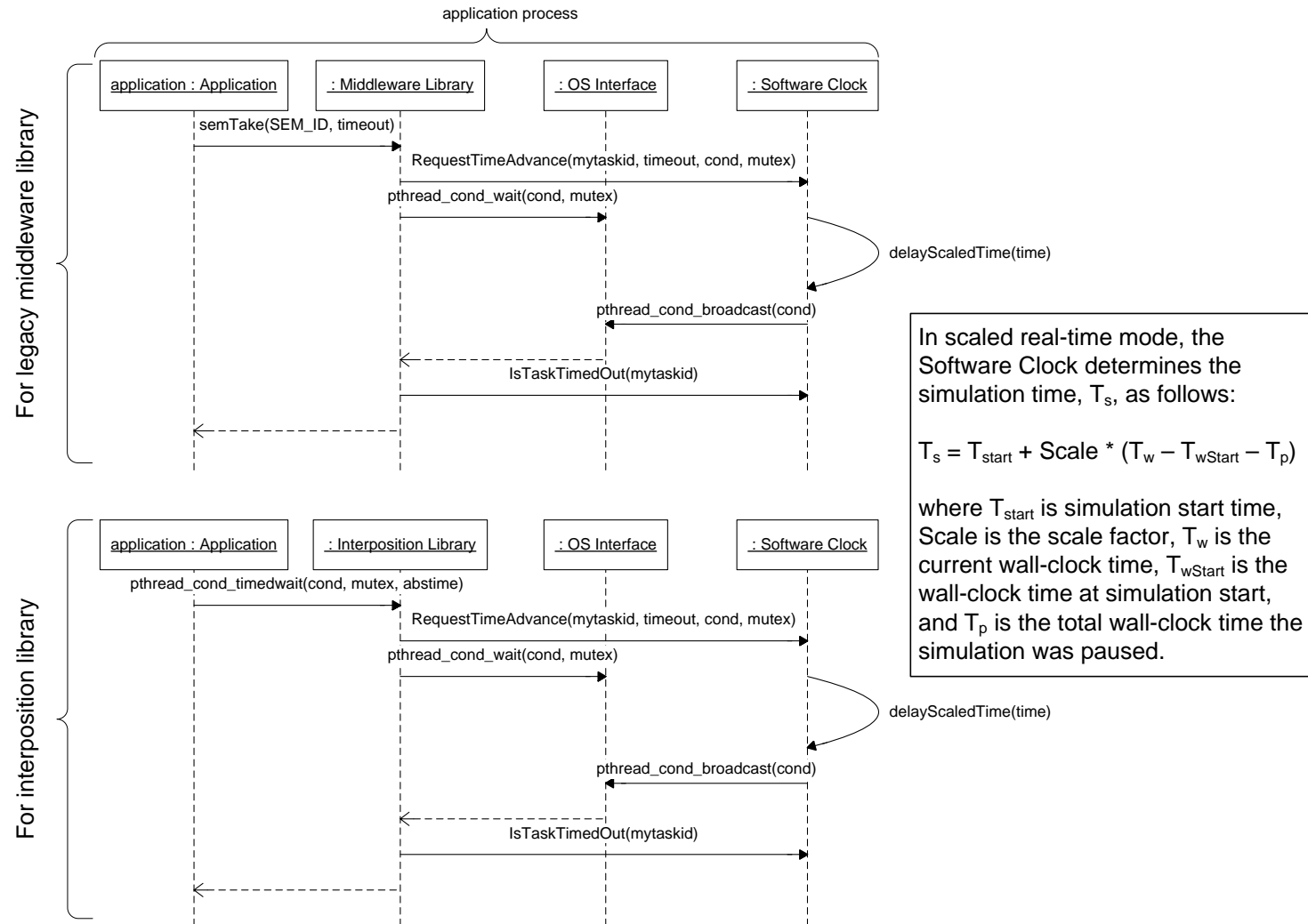
Design Notes

Use of the interposition library is probably a better approach, since it doesn't require that the applications use the Middleware Library for all time-related calls. Initially, however, the Middleware Library will probably be used for legacy applications, such as CEC, to simplify development.

Sequence Diagram(s)

Sequence Diagram for Use Case 45.2

“A process in a federate makes a blocking time-related call. That call is scaled and then routed to the operating system's clock. Note that we are going to have to identify every time related call that an application could make, in order to turn that application into a federate.”



4.5.3.3.4 Use Cases 47.1 through 47.4

Summary

47.1: Within a given federate, one thread of one process goes to sleep. Other threads/processes are still running.

47.2: Within a given federate, all threads of one process have gone to sleep. Other processes in that federate have not gone to sleep completely.

47.3: Within a given federate, all threads of all processes have gone to sleep.
Postcondition: the federate does not execute until time advanced.

47.4: All federates have gone to sleep. The framework advances to the next lowest time advance value, and wakes up only the thread(s) that have requested that time advance.

Note: Only one use case description and one sequence diagram are needed for these four use cases, since the hierarchical method for time advance works for single or multiple threads in one process (via the Software Clock), one or more processes in a federate awaiting a time advance (via the Time Server) and one or more federates awaiting a time advance (via the RTI).

Preconditions

The Software Clock is initialized to operate in discrete time mode.

Triggers

One or more threads in one or more processes in the federation issue blocking time-related calls.

Basic Course of Events

1. Application invokes a call such as semTake (when using the Middleware Library) or pthread_cond_timedwait (when using the Interposition Library) with a finite (non-zero) timeout.
2. The library issues RequestTimeAdvance to the Software Clock, and then issues the blocking pthread_cond_wait POSIX call, which then awaits for either the semGive or pthread_cond_broadcast to be issued or a timeout to occur.
3. The Software Clock sends a TIME REQUEST message, containing the smallest time advance for all the time-sensitive threads in the application process, via inter-process communication to the Time Server in the framework federate process.

4. The Time Server determines the smallest time advance for all the Software Clocks in the federate, and then invokes requestTimeAdvance in the HLA Time Machine.
5. The HLA Time Machine invokes the RTI method timeAdvanceRequest method in the RTI Ambassador.
6. When the RTI advances the time to the smallest request among all the federates, it invokes the RTI callback method timeAdvanceGrant in the Framework Federate Ambassador.
7. The Framework Federate Ambassador calls timeAdvanceGrant in the HLA Time Machine.
8. The HLA Time Machine calls grantTimeAdvance in the Time Server.
9. The Time Server sends a TIME GRANT message via inter-process communication to the Software Clock in the application processes.
10. The Software Clock issues the pthread_cond_broadcast POSIX call (unless semGive or pthread_cond_broadcast had already given the semaphore; see Alternate Paths below).
11. The Middleware Library then calls the IsTaskTimedOut method in the Software Clock to determine if a time-out has occurred.
12. The Middleware Library then calls the ResetTimeSensitivity method in the Software Clock to let it know that it is now a time-sensitive thread again (i.e., the Software Clock must wait until it has reached the next blocking call).
13. Since the timeout has occurred, the semTake/pthread_cond_timedwait method returns a timeout status.

Alternate Paths

If semGive or pthread_cond_broadcast was called by an application thread before the timeout had occurred, the Middleware or Interposition Library, respectively, wakes up and then issues the ResetTimeSensitivity call to the Software Clock. The effect of issuing this call is to cancel the pending timeout.

Postconditions

None.

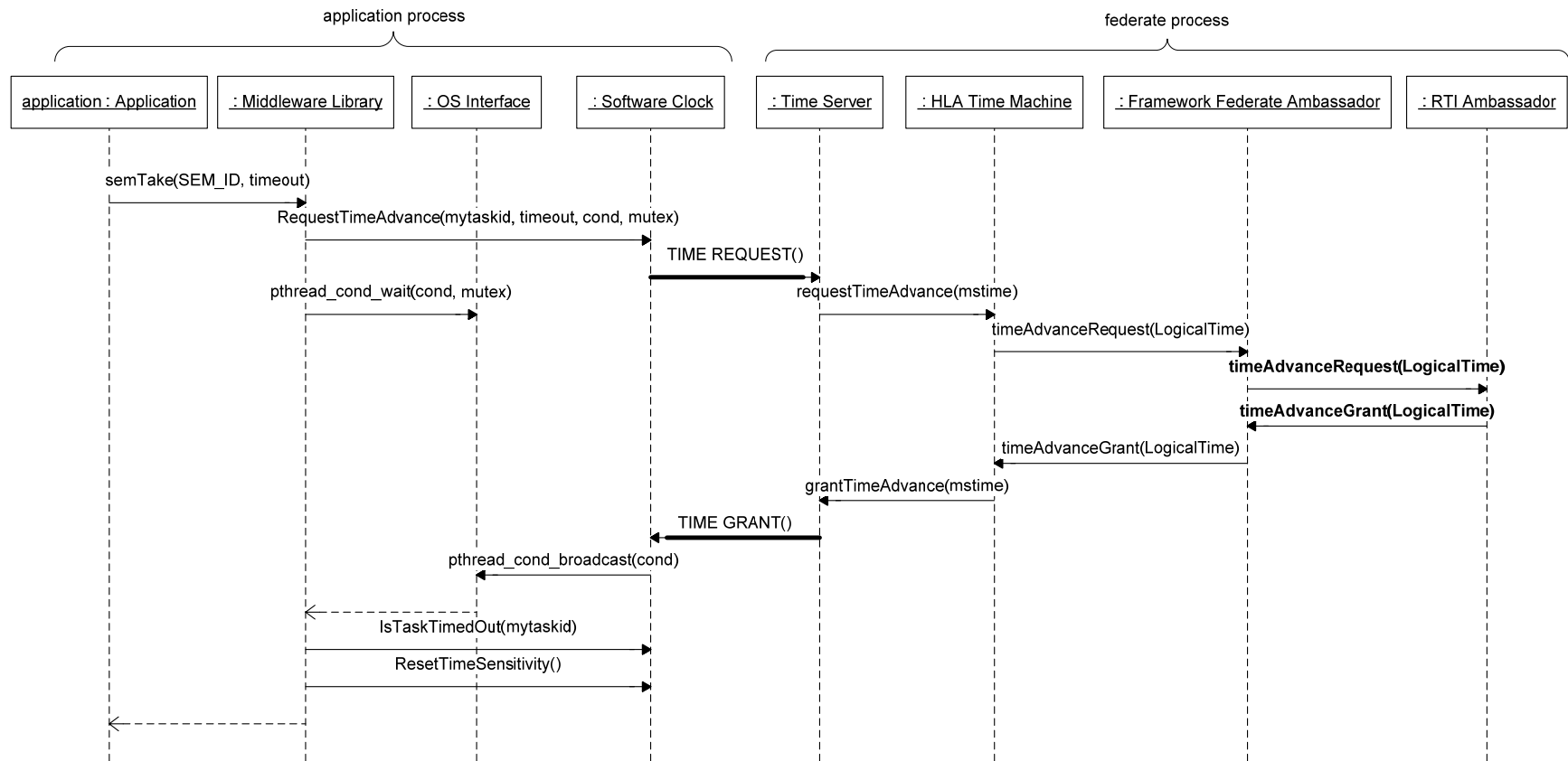
Design Notes

Use of the interposition library is probably a better approach, since it doesn't require that the applications use the Middleware Library for all time-related calls. Initially, however, the Middleware Library will probably be used for legacy applications, such as CEC, to simplify development.

Sequence Diagram(s)

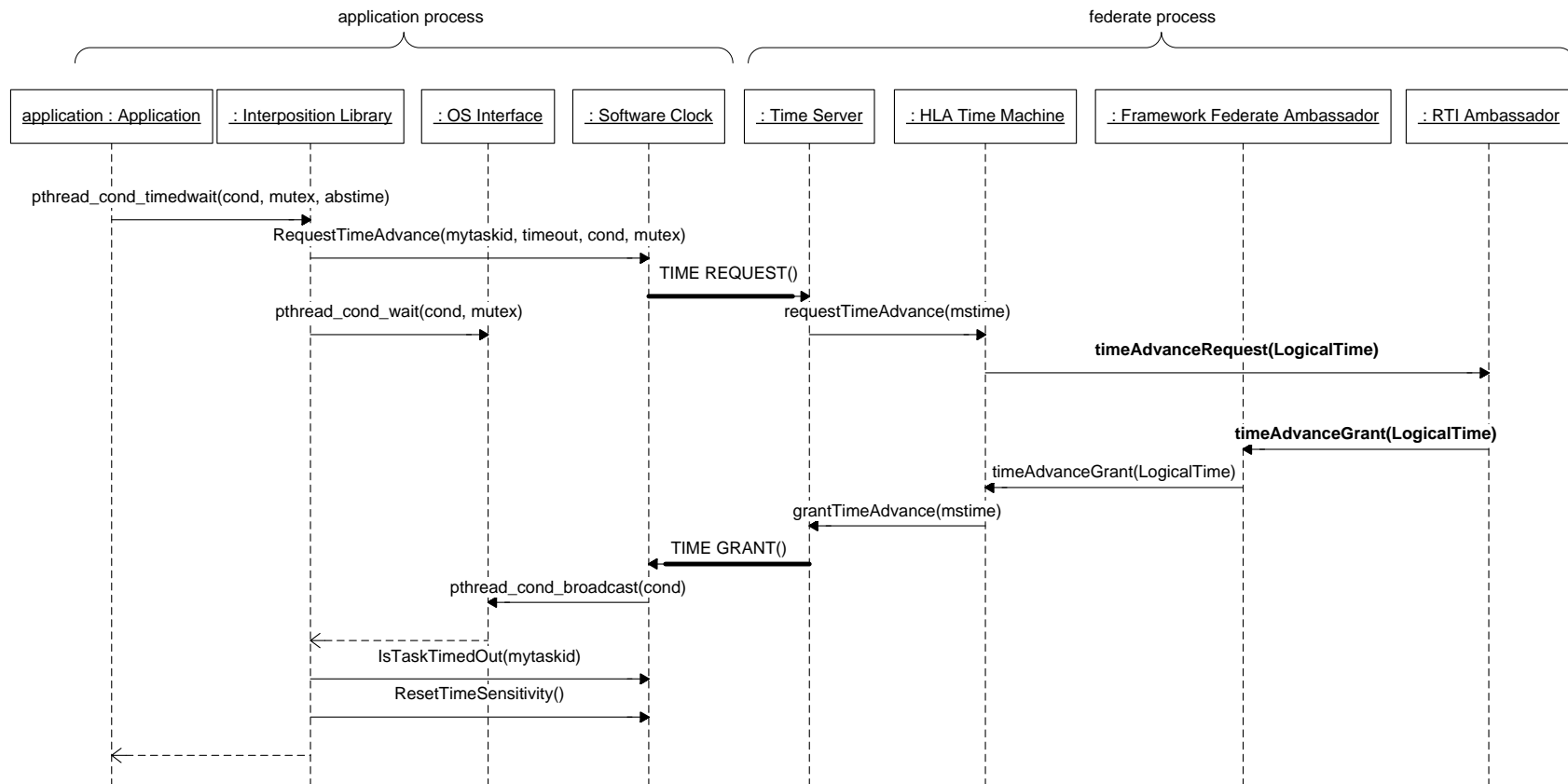
Sequence Diagram for Use Cases 47.1 through 47.4: Legacy Middleware Library

- “47.1 Within a given federate, one thread of one process goes to sleep. Other threads/processes are still running.
47.2 Within a given federate, all threads of one process have gone to sleep. Other processes in that federate have not gone to sleep completely.
47.3 Within a given federate, all threads of all processes have gone to sleep. Postcondition: the federate does not execute until time advanced.
47.4 All federates have gone to sleep. The framework advances to the next lowest time advance value, and wakes up only the thread(s) that have requested that time advance.”



Sequence Diagram for Use Cases 47.1 through 47.4: Interposition Library

- “47.1 Within a given federate, one thread of one process goes to sleep. Other threads/processes are still running.
 47.2 Within a given federate, all threads of one process have gone to sleep. Other processes in that federate have not gone to sleep completely.
 47.3 Within a given federate, all threads of all processes have gone to sleep. Postcondition: the federate does not execute until time advanced.
 47.4 All federates have gone to sleep. The framework advances to the next lowest time advance value, and wakes up only the thread(s) that have requested that time advance.”



4.5.3.3.5 Use Case 49.1

Summary

Model code makes some sort of "getTime()" call while running in a scaled real-time mode. Framework returns a time value of <start_scenario_time> + <time_since_start>*<time_scale> - <total_paused_time>. (Or we could subtract out the starting scenario time and each model would "see" a starting time of zero).

Preconditions

The Software Clock is initialized to operate in scaled real-time mode. The simulation start time (including the associated wall-clock time) and scale factor must have already been supplied.

Triggers

Application makes a call to obtain the current (simulation) time to either a legacy middleware layer or a POSIX system call.

Basic Course of Events

1. The application makes a call to get the current time to either the Middleware or Interposition library.
2. The library invokes the GetAbsTimeMs method in the Software Clock to obtain the simulation time.
3. The Software Clock first calls gettimeofday in the OS Interface to obtain the current wall-clock time, T_w . Then, the following formula is used to obtain the current simulation time, T_s :

$$T_s = T_{start} + Scale * (T_w - T_{wStart} - T_p)$$

where T_{start} is the simulation start time, $Scale$ is the scale factor, T_{wStart} is the wall-clock time at simulation start, and T_p is the total wall-clock time the simulation was paused.

4. The simulation time is then returned to the application via the Middleware or Interposition library.

Alternate Paths

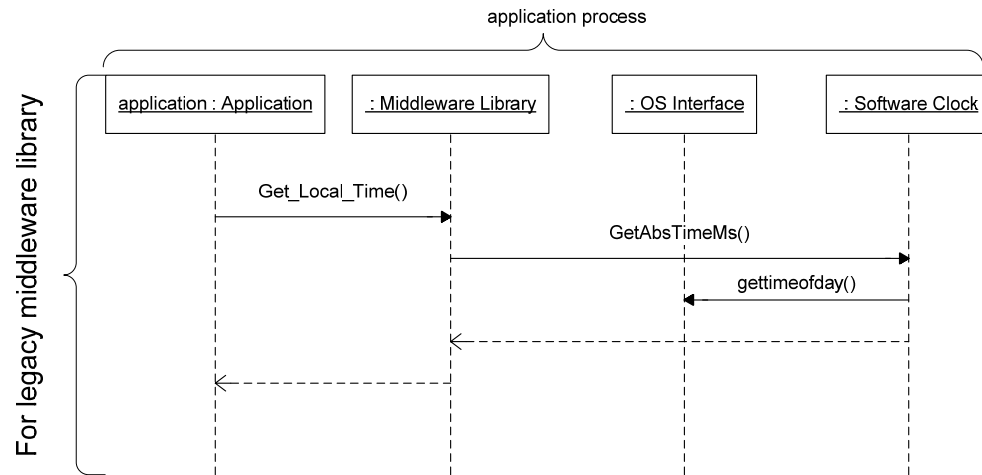
Postconditions

Design Notes

Sequence Diagram(s)

Sequence Diagram for Use Case 49.1

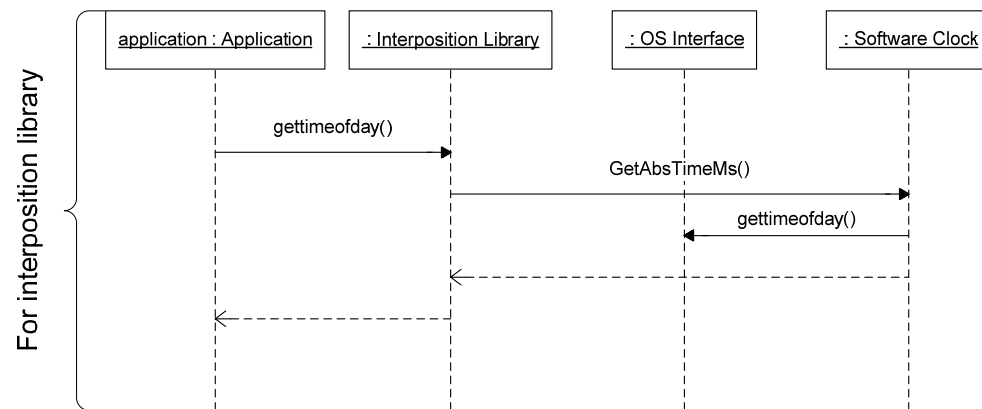
“Model code makes some sort of "getTime()" call while running in a scaled real-time mode. Framework returns a time value of $\langle \text{start_scenario_time} \rangle + \langle \text{time_since_start} \rangle * \langle \text{time_scale} \rangle - \langle \text{total_paused_time} \rangle$. (Or we could subtract out the starting scenario time and each model would "see" a starting time of zero).”



In scaled real-time mode, the Software Clock determines the simulation time, T_s , as follows:

$$T_s = T_{\text{start}} + \text{Scale} * (T_w - T_{w\text{Start}} - T_p)$$

where T_{start} is simulation start time, Scale is the scale factor, T_w is the current wall-clock time, $T_{w\text{Start}}$ is the wall-clock time at simulation start, and T_p is the total wall-clock time the simulation was paused.



4.5.3.3.6 Use Case 49.2

Summary

Model code makes same "getTime()" call while running in a discrete mode. Framework returns the time that the requesting federate is running at. The actual time should exist at a low level in our framework; when running in HLA mode, the RTI should maintain control of that the model time is.

Preconditions

The Software Clock is initialized to operate in discrete time mode.

Triggers

Application makes a call to obtain the current (simulation) time to either a legacy middleware layer or a POSIX system call.

Basic Course of Events

Time advancement is occurring automatically via the following steps:

1. The RTI Ambassador advances the time by calling the timeAdvanceGrant method in the Framework Federate Ambassador.
2. The time grant is passed to the timeAdvanceGrant method in the HLA Time Machine.
3. The time grant is passed to the grantTimeAdvance method in the Time Server.
4. The Time Server sends a TIME GRANT message via inter-process communication from the framework federate process to the application processes, where the Software Clock updates its current simulation time value.

The application actually gets the current value of the simulation time via these steps:

1. The application makes a call to get the current time to either the Middleware or Interposition library.
2. The library invokes the GetAbsTimeMs method in the Software Clock to obtain the simulation time.

3. The simulation time, which is the time in the last received TIME GRANT message (see step 4 above), is then returned to the application via the Middleware or Interposition library.

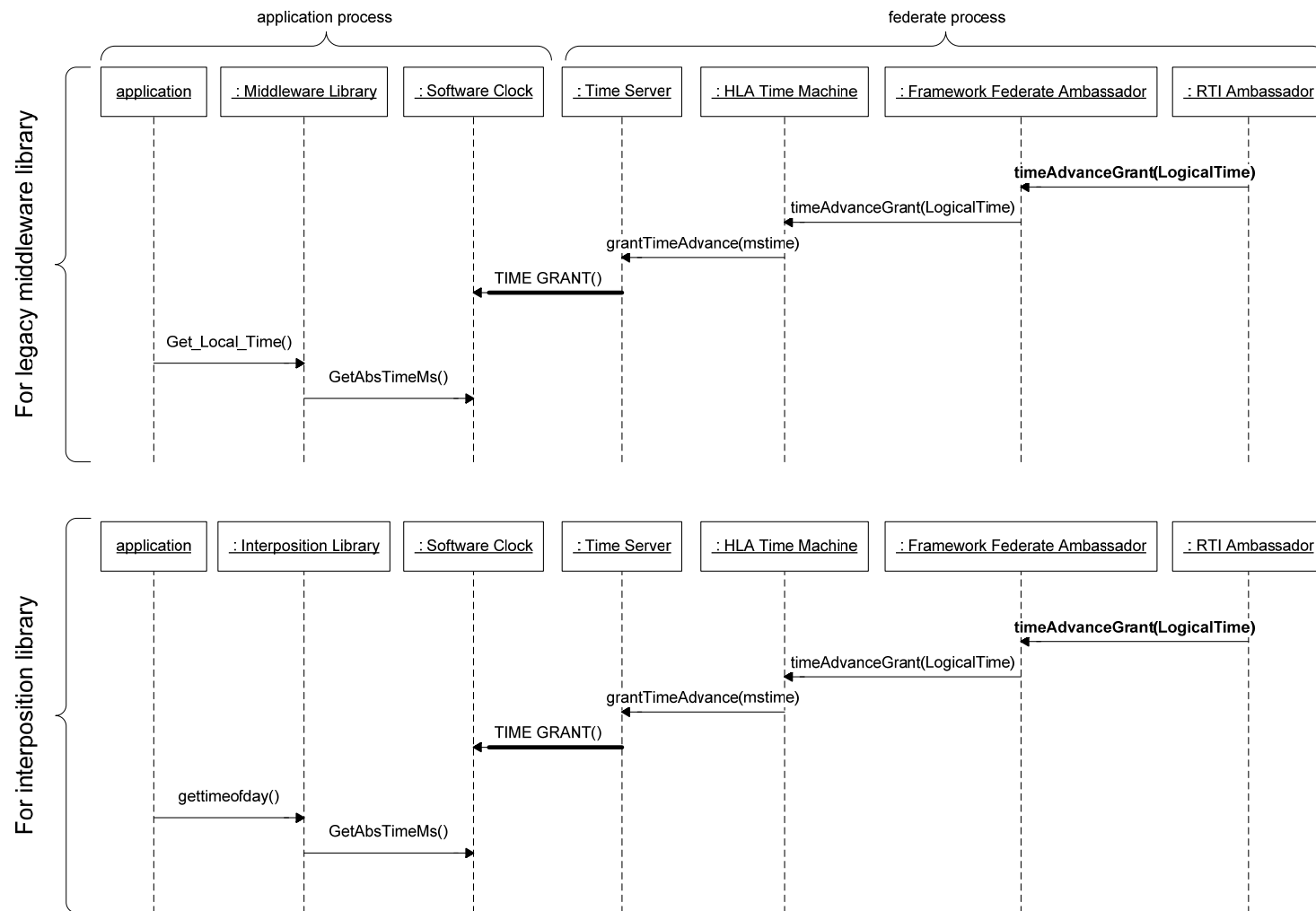
Alternate Paths

Postconditions

Design Notes

Sequence Diagram for Use Case 49.2

“Model code makes same "getTime()" call while running in a discrete mode. Framework returns the time that the requesting federate is running at. The actual time should exist at a low level in our framework; when running in HLA mode, the RTI should maintain control of that the model time is.”



4.5.4 I/O Management

The following sections describe the classes and use cases associated with I/O Management. As shown in the class diagram in the following section, an existing application uses the C Function Calls generated classes to perform all I/O operations, that is, those actions needed to exchange messages with other Federate Applications. As an example, the CEP application uses two static libraries to communicate with other federates: an NTDS library, used to communicate with NTDS devices such as sensors and weapons, and a socket library, used to communicate with the Data Distribution System. Thus, there would be two generated libraries that would replace these existing libraries.

One important part of I/O Management is to handle serialization and de-serialization: that is, the conversion of data structures between the form used by a C++ program and a stream of bytes that can be exchanged over a network. The Serializer class with this responsibility will be automatically generated using message descriptions from the legacy application software.

Finally, for this framework, the HLA RTI will be used to exchange interactions over a network. I/O Management identifies the HLA Post Office as the class with this responsibility.

4.5.4.1 I/O Class Descriptions

I/O Handler

Abstract class for I/O processing. “I/O processing” includes sending/receiving/processing messages sent by federate applications, as well as control messages internal to the framework.

Federate I/O Handler

Process I/O related calls from the federate (e.g. Send_NTDS_Mesg or Send_Socket_Mesg). Responsible for serializing input message data, including endian conversion (requires identifying exact contents of messages, which may require application-specific code), then passing messages to Post Office for delivery. Output messages are deserialized and passed to the destination application.

Framework I/O Handler

Intended to process framework-generated messages as they pass through the framework. The design evolution resulted in this class not being necessary for FY08 implementation, but could be useful in the future.

Framework Interface

Proxy class residing in the Federate Process that is used to handle communications with the Framework Process. For simplicity, this class is not shown on sequence diagrams, but all communications between one class in the Framework Process and another class in the Federate Process go through this proxy.

Message

An abstract class that represents any message that is sent between framework instances on distributed computers, or between federates via the framework. Holds the message data using an instance of Buffer.

Buffer

Class that contains a “raw” buffer of data. Used primarily to hold message data.

Federate Interface

Represents a specific interface supported by a federate. In this context, “supported” means the federate is capable of sending and/or receiving data on that interface.

Federate Message

Represents a message sent from one federate to one or more receiving federates. Holds a reference to the sending or receiving Federate Interface.

Framework Message

Represents a message sent within the framework. The design evolution resulted in this class not being necessary for FY08 implementation, but could be useful in the future.

Post Office

Abstract class that provides an interface for a message sending and receiving service. Any Message object can be sent via the Post Office. The Post Office uses information contained in the Message (such as the Federate Interface for Federate Messages) to decide how to route the message.

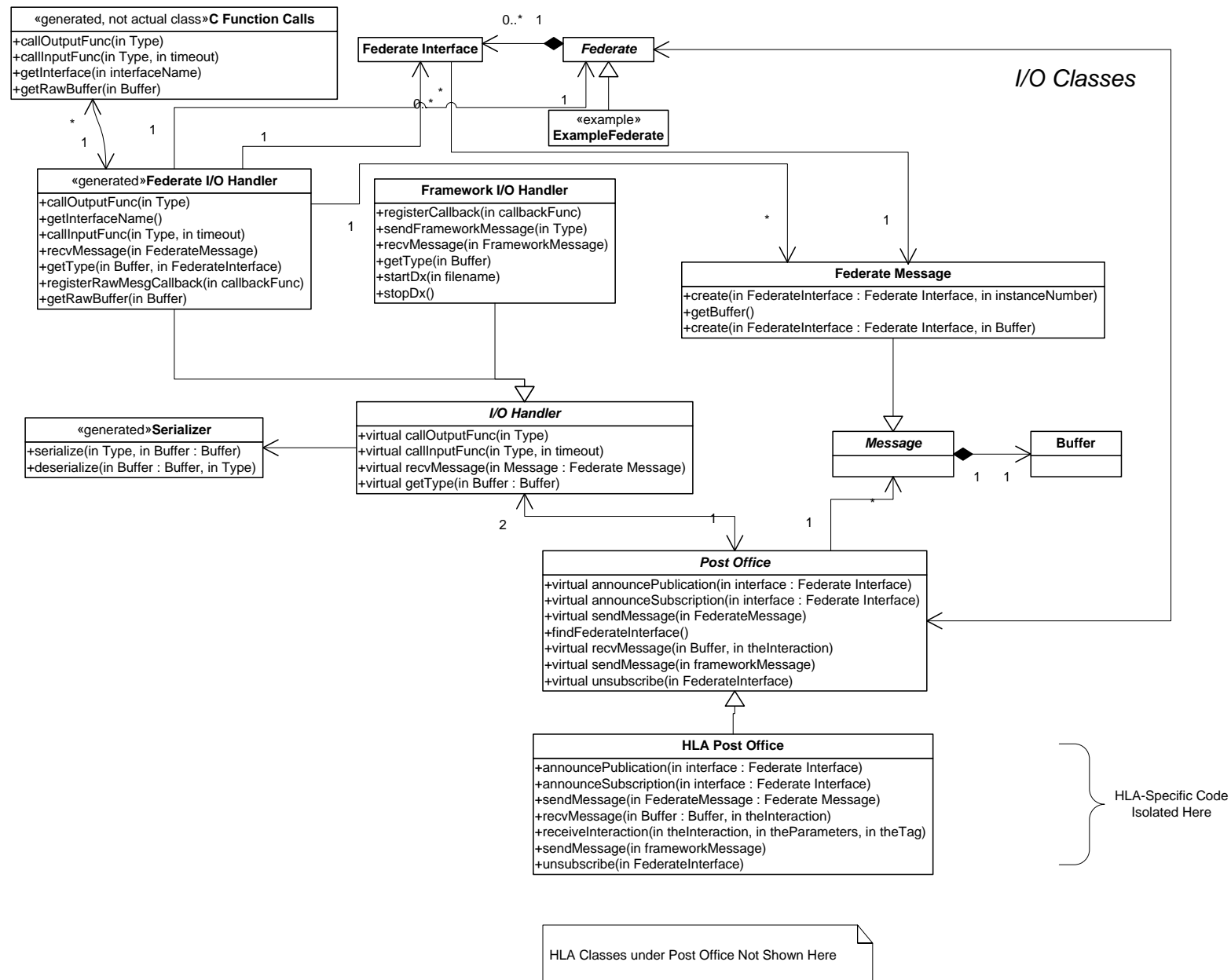
HLA Post Office

An implementation of the Post Office backed by HLA. Responsible for using the RTI to send/receive messages between federates, which includes determining the name of

the HLA interaction to use, as well as determining if the messages are sent with time-stamps or asynchronously.

Serializer

Auto-generated class responsible for serializing/deserializing each application data type and inserting it into or removing it from a Buffer. Data types range from native (e.g. int) to application-defined data structures.



4.5.4.2 Use of the “C Function Calls” Static Libraries:

- Linked in before middleware library
- Replaces two middleware library modules:
 - ntds_common (used to talk to NTDS devices)
 - socket_mgr (used for network socket communication, e.g., by DDS interface)
- IPC to Federate I/O Handler in Federate process, which uses HLA RTI to send/receive messages
- Typical ntds_common functions:
 - Configure_NTDS_Device
 - Create_NTDS_Input_Q
 - Send_NTDS_Mesg
 - Recv_NTDS_Mesg
- Typical socket_mgr functions:
 - Init_Socket_Mgr
 - Create_TCP_Client_Socket
 - Create_UDP_Client_Socket
 - Create_Input_Socket_Queue
 - Send_Socket_Msg
 - Recv_Socket_Msg

4.5.4.3 Use Cases

4.5.4.3.1 Use Case 42.1

Summary

Federate publishes what messages it's capable of sending. In HLA these would be interactions.

Preconditions

None.

Triggers

Federate initialization, during which the scenario controller tells the federate to announce all interfaces it will send messages to.

Basic Course of Events

1. Every federate must announce all interfaces, over which it will send messages, to the Post Office.
2. The HLA version of the Post Office forwards the announcement to the Federate Ambassador.
3. Finally, the announcement is made to the RTI ambassador.

Alternate Paths

None.

Postconditions

None.

Design Notes

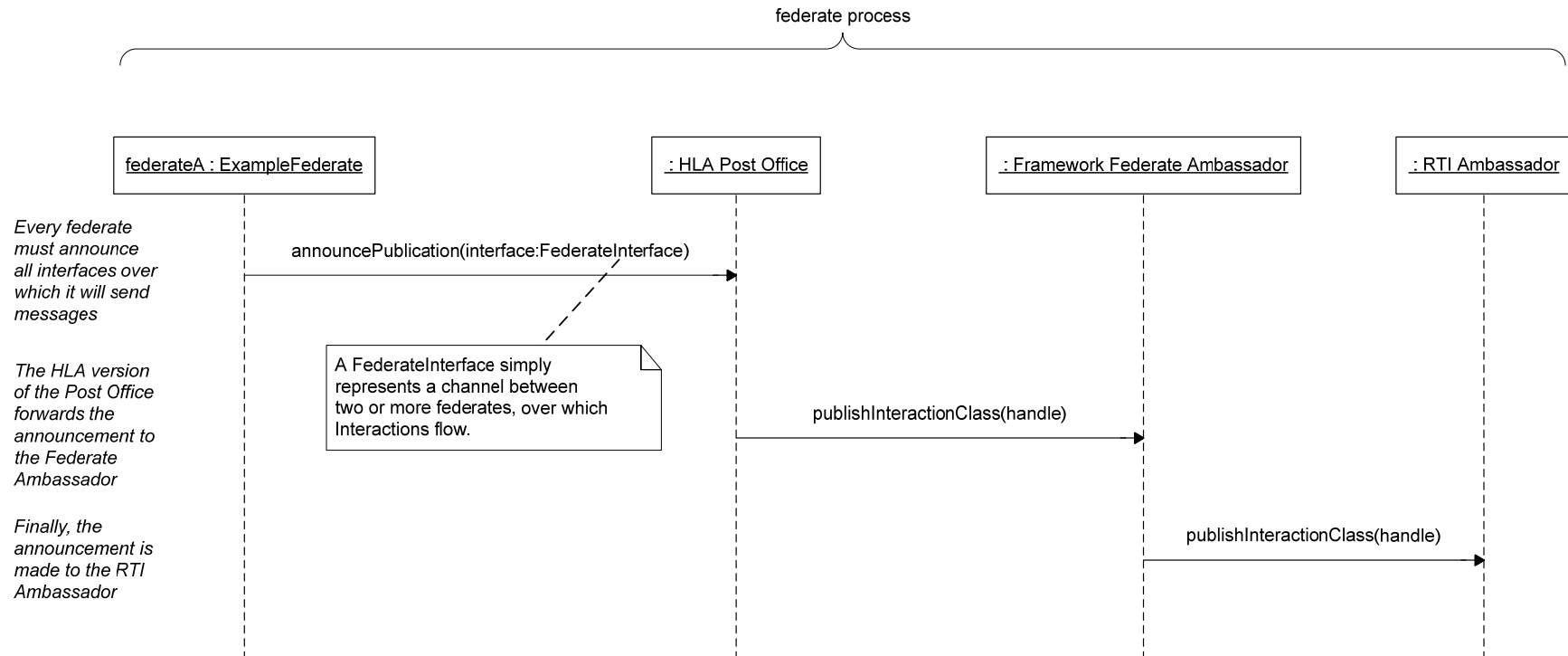
The RTI Ambassador is part of the commercial RTI package. The Framework Federate Ambassador is an implementation of the Federate Ambassador abstract class, also provided by the commercial RTI package, along with additional functions called by the Post Office. Note that although an HLA Post Office is shown here, nothing in the Federate implementation should depend on which version of the Post Office is being used (e.g., an OASIS Post Office could be an alternative).

Note that the announcement of an “interface” gives rise to an announcement of an Interaction that is sent from this federate to a set of one or more other federates. For example, if federate A.1 has messages to send to federates B.1, B.2, and C.1, it would use an Interaction that is registered for by these federates. The Interaction contains an opaque buffer that can be used to contain one or more serialized messages.

Sequence Diagram(s)

Sequence Diagram for Use Case 42.1

“Federate publishes what messages it’s capable of sending. In HLA these would be interactions.”



4.5.4.3.2 Use Case 42.2

Summary

Federate indicates what messages it wants to subscribe to, and which federates it wants to receive data from. This is related to use case 10.4.

Preconditions

None.

Triggers

Federate initialization, during which the scenario controller tells the federate to announce all interfaces it will receive messages from.

Basic Course of Events

1. Every federate must announce all interfaces, over which it will receive messages, to the Post Office.
2. The HLA version of the Post Office forwards the announcement to the Federate Ambassador.
3. Finally, the announcement is made to the RTI ambassador.

Alternate Paths

None.

Postconditions

None.

Design Notes

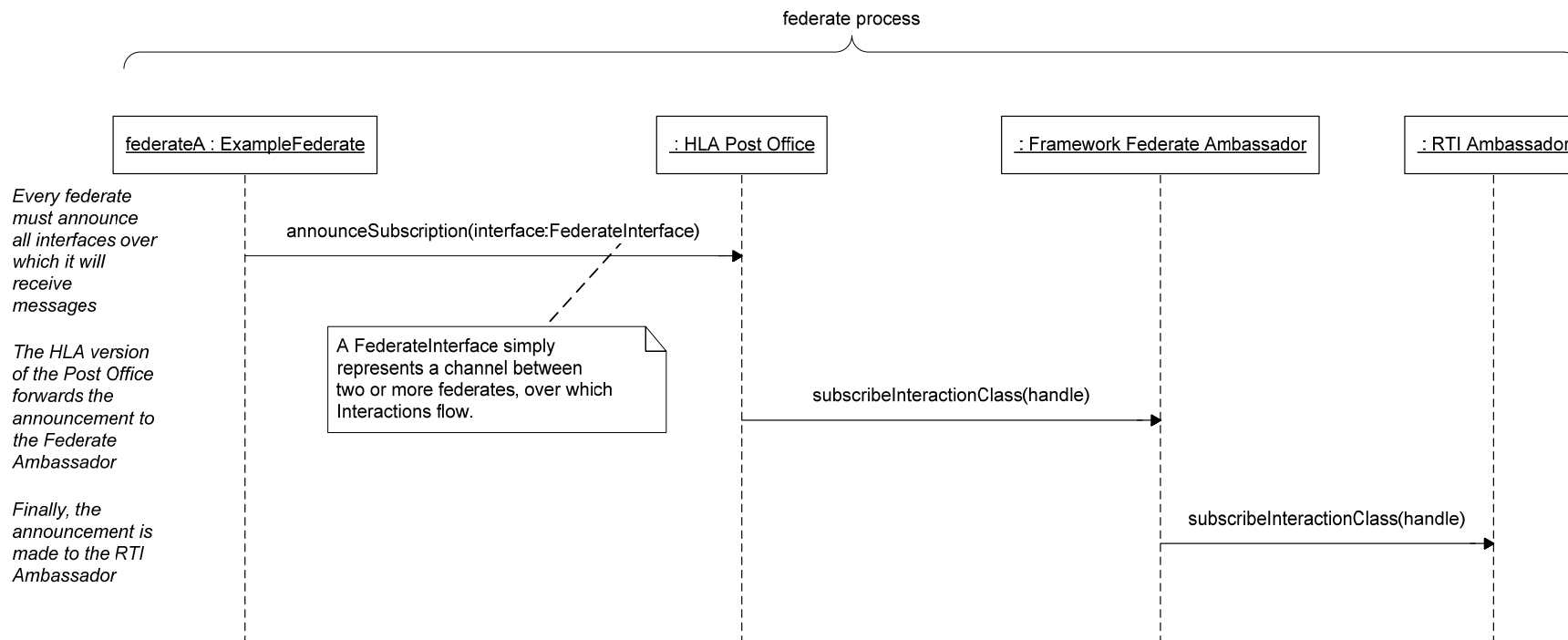
The RTI Ambassador is part of the commercial RTI package. The Framework Federate Ambassador is an implementation of the Federate Ambassador abstract class, also provided by the commercial RTI package, along with additional functions called by the Post Office. Note that although an HLA Post Office is shown here, nothing in the Federate implementation should depend on which version of the Post Office is being used (e.g., an OASIS Post Office could be an alternative).

Note that the announcement of an “interface” gives rise to an announcement of an Interaction that this federate is expecting from a set of one or more other federates. For example, if federate A.1 needs to receive messages from federates B.1, B.2, and C.1, it would subscribe to Interactions that are published by these federates. The Interaction contains an opaque buffer that can be used to contain one or more serialized messages.

Sequence Diagram(s)

Sequence Diagram for Use Case 42.2

“Federate indicates what messages it wants to subscribe to, and which federates it wants to receive data from. This is related to use case 10.4.”



4.5.4.3.3 Use Case 42.3

Summary

Federate sends a message. The federates who have subscribed to that message from that federate receive the message. No other federates in the federation, and no federates outside of the federation, receive the message.

Note: message sending mechanism may be different depending on real-time or discrete modes. See requirements 44, 45, and 47.

Preconditions

The federate must have first created the Federate Interfaces over which messages are to be sent or received, and it must have announced the publications and/or subscriptions for these interfaces.

Triggers

Federate A (for example) sends a message on an interface; federate B waits to receive that message on an interface.

Basic Course of Events

The following description applies to both use case 42.3a (real-time mode) and use case 42.3b (discrete mode), although there are separate sequence diagrams for each. The few differences between these two modes are indicated by bold method names in the sequence diagrams and are explicitly mentioned below.

Message Sending

1. The federate passes an application-specific object to the appropriate output function in C Function Calls, which represents a library that provides the required I/O functions. For example, in the legacy CEP on a Workstation, this could be the `Send_NTDS_Mesg` function in the `ntds_common` library.
2. A call is made to a method with the same signature in the generated Federate I/O Handler, which does the following:
 - a. It obtains the previously-configured interface name to be used for I/O.
 - b. Using this name, it invokes the `getInterface` method in the federate to get a Federate Interface object.
 - c. Passing this Federate Interface object along with an instance number, it creates a Federate Message, obtains the byte-array Buffer from it, and calls the `serialize` method in `Serializer` (passing the application-specific Type and the Buffer) to serialize the message into the byte array.

3. Now that the message has been serialized into the Federate Message, this message is passed to the HLA Post Office using its `sendMessage` method.
4. The HLA Post Office first determines the mode from the Scenario Controller using its `isRealTimeMode` method. Then,
 - a. Sub Use Case A (Real-Time Mode): The Federate Message is sent to the Framework Federate Ambassador using its `sendRO` method.
 - b. Sub Use Case B (Discrete Mode): The Federate Message is sent to the Framework Federate Ambassador using its `sendTSO` method.
5. The Framework Federate Ambassador invokes the overloaded `sendInteraction` method in the RTI Ambassador to actually send the message.

Message Reception

1. The federate passes a reference to an application-specific object, along with a possible timeout, to the appropriate input function in C Function Calls, which represents a library that provides the required I/O functions. For example, in the legacy CEP on a Workstation, this could be the `Recv_NTDS_Mesg` function in the `ntds_common` library.
2. A call is made to a method with the same signature in the generated Federate I/O Handler, which then waits for the next incoming message of the specified type. If a timeout expires before a message is received, an exception is raised.
3. When a message is received by the RTI Ambassador, one of two overloaded callback methods (named `receiveInteraction`) in the Framework Federate Ambassador is called, depending on the mode (real-time or discrete).
4. The interaction is forwarded to the HLA Post Office via its `recvInteraction` method.
5. The HLA Post Office finds the corresponding Federate Interface from the interaction handle, and then uses this interface along with the Buffer to create a Federate Message object.
6. The HLA Post Office then passes the Federate Message to the Federate I/O Handler via its `recvMessage` method.
7. The Federate I/O Handler gets the type of this Buffer and then calls the `deserialize` method in `Serializer` to de-serialize the data into an application-specific object.
8. Finally, this application-specific object is passed as an out parameter back through C Function Calls and to the federate, thus completing the method invocation begun in step 1.

Alternate Paths

As described above for message reception, if a timeout occurs when calling the input method in the I/O Handler, an exception propagates back to C Function Calls, which notifies the invoking application through whatever mechanism is defined by the API (e.g., an error return code).

Postconditions

None.

Design Notes

The Federate is intended to be implemented as a separate process from the processes that make up the application. Therefore, some kind of inter-process communications mechanism (such as POSIX message queues) must be employed. Note that in the sequence diagram (Part 1), inter-process invocations between C Function Calls and I/O Handler are indicated with a thick line.

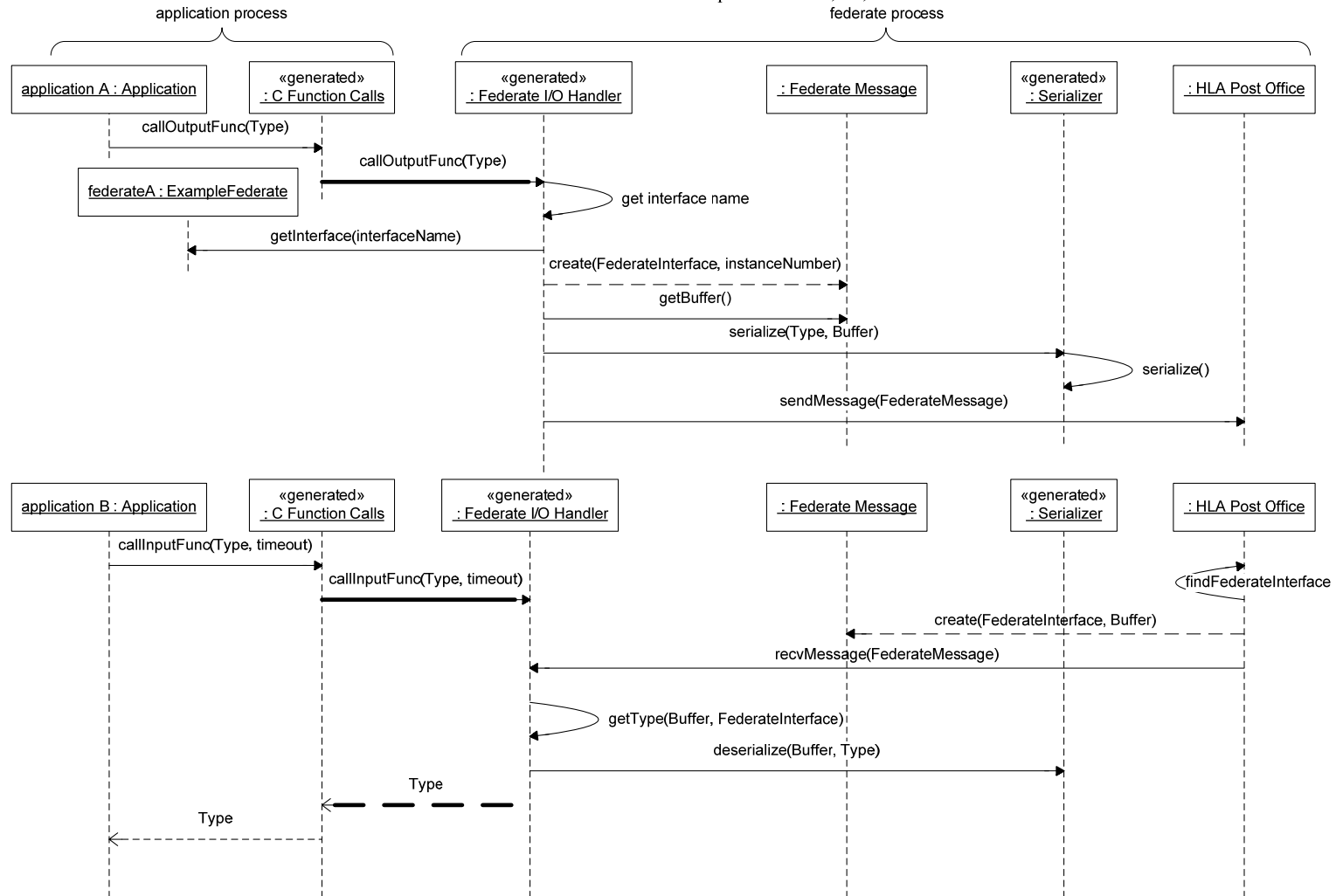
Some sort of message buffering might need to be implemented. For example, if messages are arriving so fast that the application isn't calling the input function in C Function Calls fast enough to process each message in real time, one can either discard the message (usually undesirable) or buffer it in a queue. The location of this input queue (for example, in the Federate I/O Handler) and the length of this queue are TBD. Furthermore, there may need to be multiple queues, one for each incoming type.

Sequence Diagram(s)

There are actually two sequence diagrams. Part 1 covers those interactions from the Example Federate to the HLA Post Office, and Part 2 covers those interactions from the HLA Post Office to the RTI Ambassador. Furthermore, Part 2 is divided into two sub-use cases: A (real-time mode) and B (discrete mode).

Sequence Diagram for Use Case 42.3, Part 1

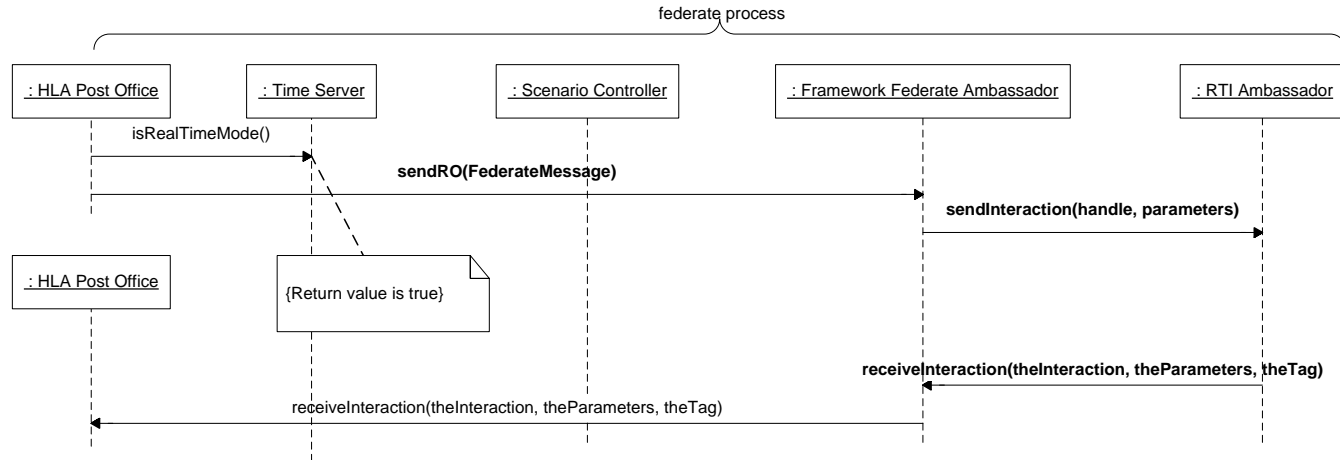
“Federate sends a message. The federates who have subscribed to that message from that federate receive the message. No other federates in the federation, and no federates outside of the federation, receive the message. Note: message sending mechanism may be different depending on real-time or discrete modes. See requirements 44, 45, and 47.”



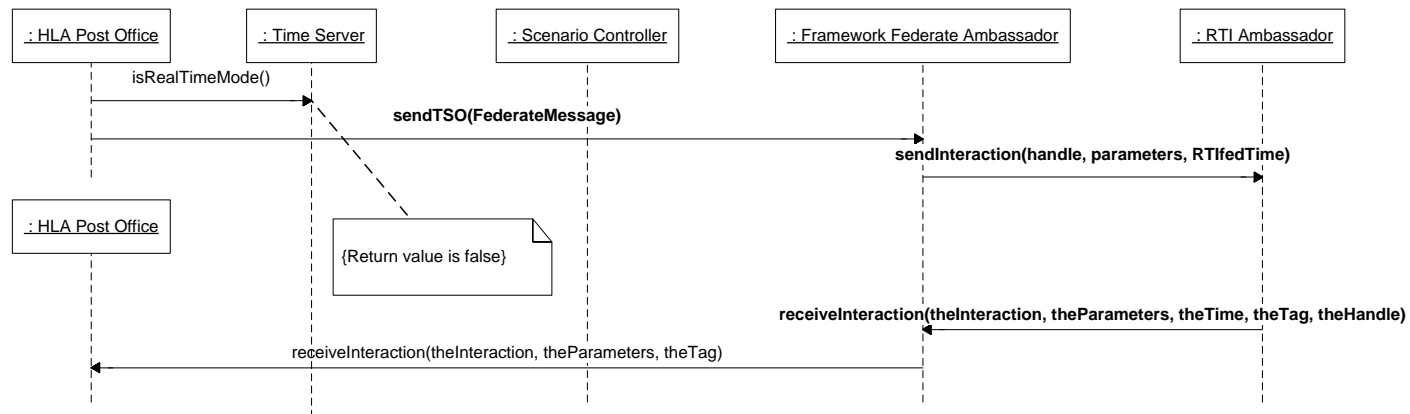
Sequence Diagram for Use Case 42.3, Part 2

“Federate sends a message. The federates who have subscribed to that message from that federate receive the message. No other federates in the federation, and no federates outside of the federation, receive the message. Note: message sending mechanism may be different depending on real-time or discrete modes. See requirements 44, 45, and 47.”

Sub Use Case A: Real-Time Mode



Sub Use Case B: Discrete Mode



4.5.4.3.4 Use Case 64.1

Summary

Provide mechanism for a federate to automatically subscribe to and unsubscribe from every message from all other federates.

Preconditions

At application capture time, the DX federate (as distinct from the DX application itself) must be provided with a list of all the Federate Interfaces in the federation. That is, it must know about all possible interfaces to subscribe to in order to capture all inter-federate messages.

Triggers

When a data extraction begins or ends.

Basic Course of Events

Subscribing to all messages from all federates

1. The DX federate must invoke the announceSubscription method in the Post Office for all Federate Interfaces it has been configured with (at application capture time).
2. The HLA version of the Post Office forwards the announcement to the Federate Ambassador.
3. Finally, the announcement is made to the RTI ambassador.

Un-subscribing from all messages

1. The DX federate must invoke the unsubscribe method in the Post Office for all Federate Interfaces it had previously subscribed to.
2. The HLA version of the Post Office forwards an unsubscribe request to the Federate Ambassador.
3. Finally, an unsubscribe request is made to the RTI ambassador.

Alternate Paths

If an error occurs during the subscribe or un-subscribe process, an exception propagates up to the DX federate (as distinct from the DX application).

Postconditions

None.

Design Notes

In a more sophisticated version of DX, one might wish to subscribe to only a subset of the Federate Interfaces in a federation.

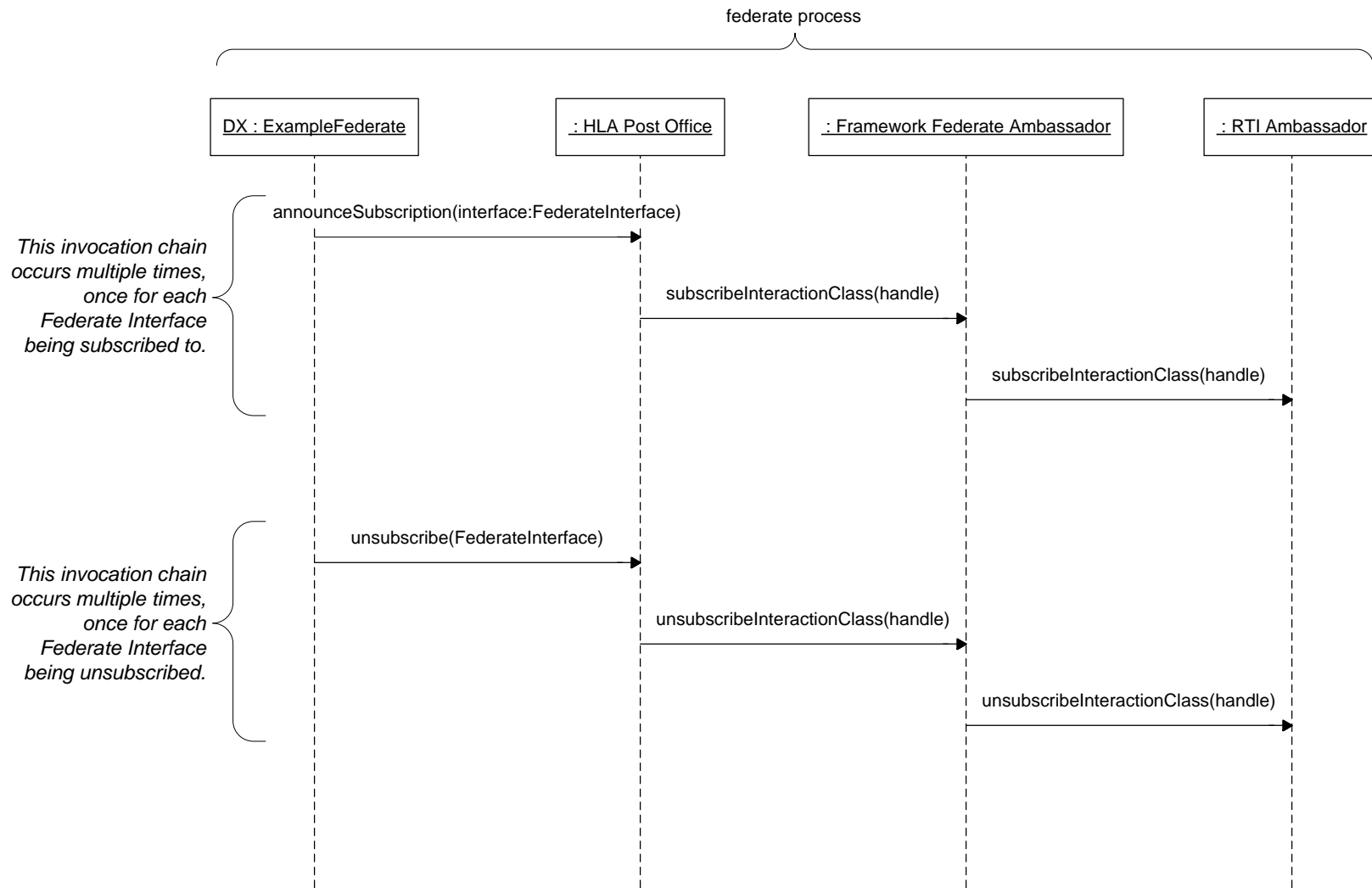
Framework Messages are OBE for FY08.

The control for starting and stopping DX is described by Use Case 64.3. A simplification might be to automatically subscribe at program initialization and never to unsubscribe (i.e., use only the first group of actions in Basic Course of Events, above).

Sequence Diagram(s)

Sequence Diagram for Use Case 64.1

“Provide mechanism for a federate to automatically subscribe to every message from all other federates.”



4.5.4.3.5 Use Case 64.2

Summary

Write all of those messages out to files. Note that it's a design question whether to deserialize or not. I recommend not, that way: all the data remains in network byte order; if our serialization code packs data fields into the next largest byte-aligned data type, we may already have a tool that could read in the data for DR (we would, however, have to create a config file that defined all the messages for that tool to read in, but that may not be too much work).

Preconditions

A unique filename for the extracted messages must first be obtained (see Use Case 66.1).

Triggers

When a data extraction begins or ends.

Basic Course of Events

1. At the beginning of a data extraction, the open method in FileIo is called to create the DX file. The name supplied to the open method comes from Use Case 66.1.
2. To receive the next raw input buffer, the application calls the getRawBuffer method in C Function Calls, supplying the Buffer as an out parameter.
3. A call is made to the getRawBuffer method in the Federate I/O Handler, which then waits for the next incoming message buffer.
4. Processing of incoming messages up to the Post Office is identical to steps 3 and 4 in the Message Reception section of Use Case 42.3, and won't be repeated here.
5. The Post Office finds which Federate Interface the message arrived on and then creates a Federate Message from the received Buffer.
6. The Federate Message is passed to the Federate I/O Handler via its recvMessage method.
7. The Federate I/O Handler extracts the Buffer from the Federate Message and passes it as an out parameter back through C Function Calls and to the DX application, thus completing the method invocation begun in step 2.
8. Now that the DX application has a raw message buffer, it can write it to the DX file using the write method in FileIo.
9. When DX is terminated, the application calls the close method in FileIo to close the DX file.

Alternate Paths

If an error occurs during calls to open, getRawBuffer, write, or close, an exception propagates up to the DX application.

Postconditions

None.

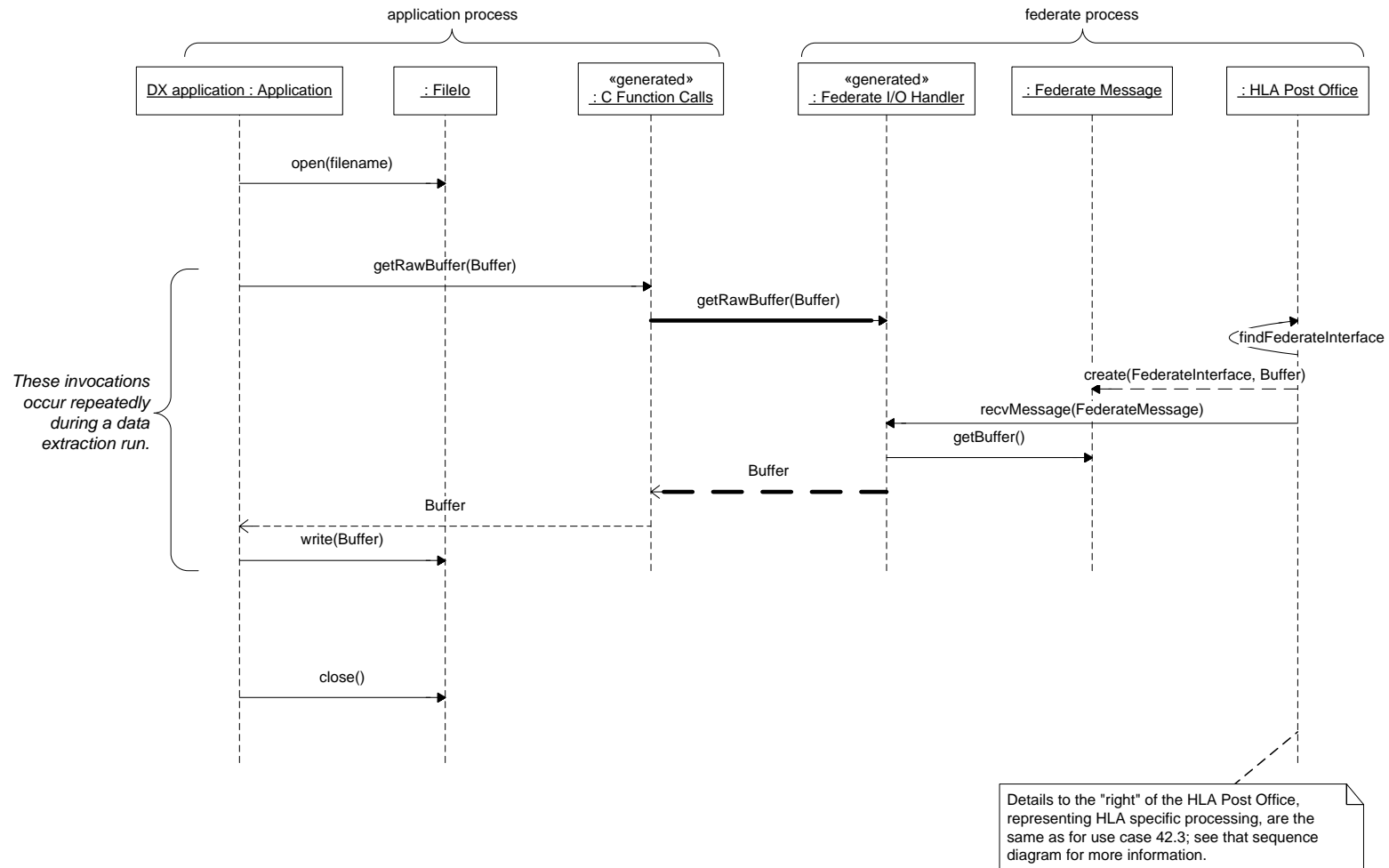
Design Notes

The control for starting and stopping DX is described by Use Case 64.3.

Sequence Diagram(s)

Sequence Diagram for Use Case 64.2

“Write all of those messages out to files. Note that it's a design question whether to deserialize or not. I recommend not, that way: all the data remains in network byte order; if our serialization code packs data fields into the next largest byte-aligned data type, we may already have a tool that could read in the data for DR (we would, however, have to create a config file that defined all the messages for that tool to read in, but that may not be too much work).”



4.5.4.3.6 Use Case 66.1

Summary

DX federate should save recorded data with unique indicators in file of executing federation name, scenario name or # (if in monte-carlo runs) and any other identifiers necessary. Also have a safety mechanism that if there are already files with the names we end up wanting to save as, we add on a timestamp or something else to ensure that we don't overwrite any data.

Preconditions

None.

Triggers

File name is needed by DX when starting a new extraction.

Basic Course of Events

1. The DX application calls the `getFederationName` method in the Scenario Controller to get the federation name in the form of a string.
2. The DX application calls the `getScenarioName` method in the Scenario Controller to get the scenario name in the form of a string.
3. The DX application calls the `getScenarioIteration` method in the Scenario Controller to get the iteration number of the scenario run.
4. The DX application calls the `gettimeofday` method via the OS Interface to get the current wall-clock time.

Alternate Paths

None.

Postconditions

None.

Design Notes

Given the four pieces of information discussed above, a unique DX filename can be constructed as follows:

`dx_<fedName>_<scenName>_<iterNum>_yyyymmdd_hhmmss`

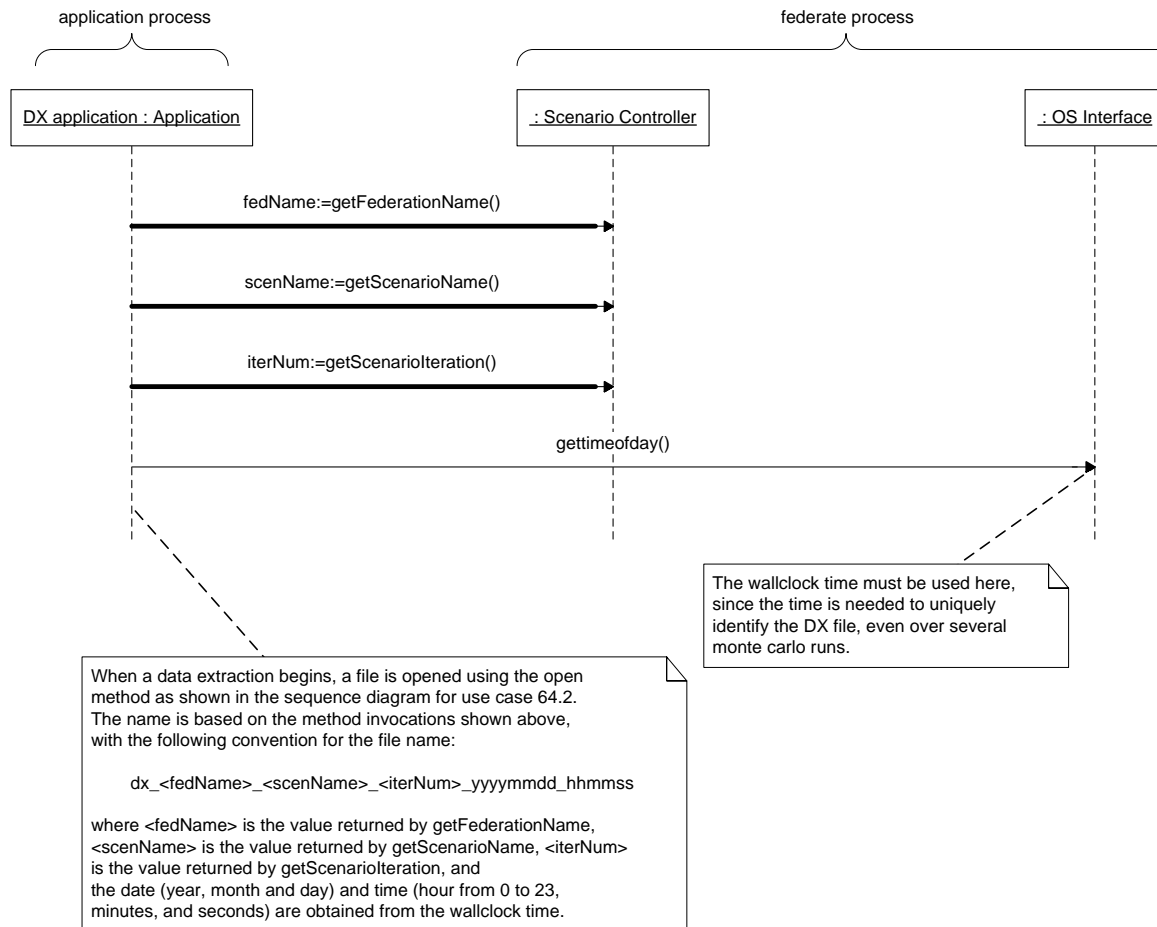
where `<fedName>` is the value returned by `getFederationName`, `<scenName>` is the value returned by `getScenarioName`, `<iterNum>` is the value returned by `getScenarioIteration`, and the date (year, month and day) and time (hour from 0 to 23, minutes, and seconds) are obtained from the wall-clock time.

The user might determine whether a separate scenario file is created (and named) for each scenario iteration, or whether all iterations are placed in one file. It also may be desirable to include the iteration number in each federate message header.

Sequence Diagram(s)

Sequence Diagram for Use Case 66.1

“DX federate should save recorded data with unique indicators in file of executing federation name, scenario name or # (if in monte-carlo runs) and any other identifiers necessary. Also have a safety mechanism that if there are already files with the names we end up wanting to save as, we add on a timestamp or something else to ensure that we don't overwrite any data.”



4.6 HLA Middleware

This section describes the features of the HLA middleware, upon which the framework will be heavily dependent on in FY08. Use or disuse of HLA in future years will depend on lessons learned in FY08, and on the maturity of other middleware options, such as OASIS.

4.6.1 RTI Implementation

There are several options for RTI implementations to use in the framework. These include freeware versions such as the Defense Modeling and Simulation Office (DMSO), and commercial versions provided by vendors including MÄK Technologies and Pitch. Based on a survey of APLers with experience in HLA, it was decided that a commercial RTI is the best way to go, and that the two best candidates were Pitch and MÄK. MÄK has been selected as the best candidate because:

- Other AMDD projects (ANTARES and NSSAT) also use MÄK
- MÄK executes on Solaris 10; support for Pitch on Solaris 10 is uncertain
- MÄK provides several configuration options to tune performance.
- MÄK is implemented in C++, whereas Pitch is implemented in Java, meaning MÄK is likely to perform better.

We intend to run some basic performance tests using the free evaluation version of the MÄK RTI before making a final purchasing decision. The evaluation version only runs on Windows XP and only allows one federation with two federates; other than those constraints it is believed to be identical to the purchased versions.

One slight concern with the MÄK RTI is compiler interoperability. Due to MÄK's API being C++-based, the code that calls the RTI must be compiled with the same compiler as the RTI was compiled with. On Solaris platforms, this is Sun's C++ compiler, which was unlikely to have been used for any federate applications. This means that the code for the federate and framework processes will either both have to be compiled with Sun's compiler, or calls to the RTI would have to be wrapped in a C layer (binaries compiled with C linkage *are* compatible between compilers).

4.6.2 HLA Functions

The use of HLA/RTI in the framework is expected to provide the following services.

4.6.2.1 Time

HLA is capable of maintaining simulation time, and granting time requests and advances to all federates within a federation. This is important when running in a discrete-time mode.

4.6.2.2 I/O

HLA provides I/O services for federates to exchange data with each other. The framework will use HLA “interactions” as the mechanism for exchanging data between federates (as opposed to object attributes). Messages can be timestamped, which will be used when running in discrete-time mode, or delivered asynchronously, when running

4.6.2.3 Federate Management

HLA allows federates to join and resign from federations.

4.6.2.4 Federation Management

HLA allows federations to be created and destroyed.

4.6.3 Class Descriptions

The HLA-specific classes are as follows:

HLA Federation Manager

See description in “Initialization/Scenario Control” section.

HLA Federate Manager

See description in “Initialization/Scenario Control” section.

HLA Post Office

See description in “I/O Classes” section.

HLA Time Machine

See description in “Time-Related Classes” section.

Federate Ambassador

Abstract class provided by RTI implementation that provides callbacks to handle commands and communications from RTI. HLA Federates are required to implement a subclass of Federate Ambassador

Framework Federate Ambassador

The framework’s subclass of Federate Ambassador. Callbacks from the RTI are handled by this class, which in turn passes control to the appropriate class (e.g. I/O-related calls from the RTI are passed to the HLA Post Office, time-related calls are passed to the HLA Time Machine, etc). For symmetry purposes, this class also serves as a proxy for calls going from the framework to the RTI Ambassador. All communications between the Federate Process and the RTI go through the Framework Federate Ambassador.

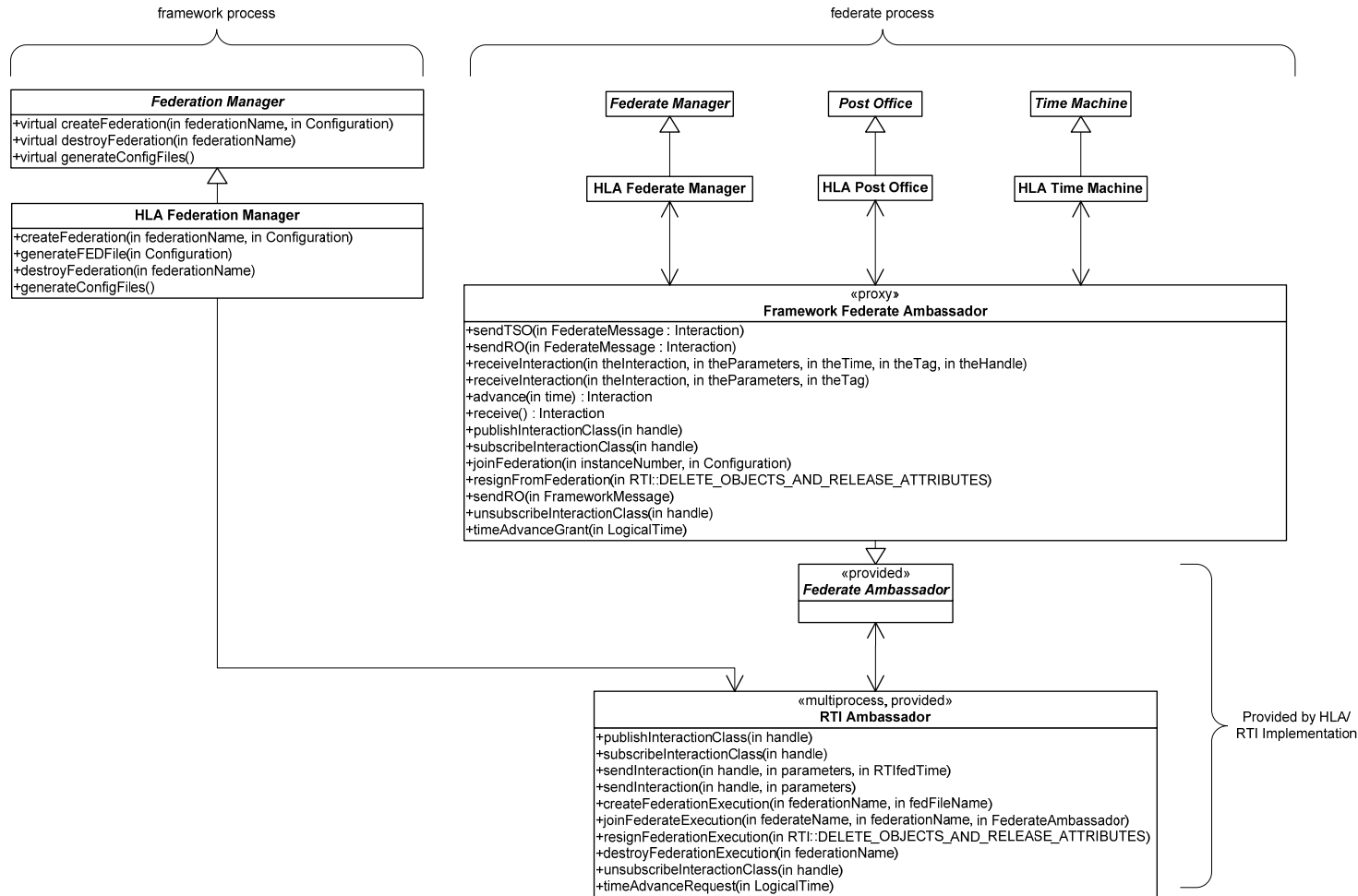
RTI Ambassador

Concrete class provided by the RTI, instances of this class are used by applications to communicate with the RTI. Within a federate process, only the Framework Federate Ambassador communicates with the RTI Ambassador. In the framework process, the HLA Federation Manager communicates with the RTI Ambassador.

4.6.4 Class Diagram

The following diagram shows the HLA-specific classes in the framework. Note that for real-estate reasons, some of the classes have their method names omitted, however, those method names appear in other class diagrams in this document.

*HLA-Specific Classes
(and implemented
interfaces)*



4.7 Application Capture into Framework

Among the goals of this project are development and documenting a method of incorporating applications into the framework as easily as possible. Towards that end, there are plans to automate this process as much as possible, and there are several aspects of the framework where this may apply. This has been touched on briefly in the class designs, using the stereotype <generated> to indicate classes for which the source code can be partially or even fully generated automatically. This section describes the areas of the framework in which automation is an option.

4.7.1 Message Serialization

The framework design calls for federate messages to be serialized into buffers for transport through the post office. Because this potentially involves endian conversion to the framework's network byte order (this could be little or big endian), message fields must be inserted into the buffer one at a time. This requires the framework source code to have knowledge of the fields in each message. Scripts already exist that are capable of parsing C-style data structures in order to do endian conversions. Those scripts should be easily modifiable to generate code that, given a data structure, iterates over that structure, and copies the fields into the structure, performing endian conversion as necessary. The opposite should be doable for pulling messages out of buffers. In the class design, this generated code would likely go into the Serializer class.

4.7.2 Application I/O Function Calls

As discussed previously, the framework is going to intercept each I/O related call that a federate application makes, by providing its own implementation of that function. The framework's function then sends the parameters into an equivalent function in the Federate I/O Handler, and it is ultimately sent out. This is the sequence of events described in Use Case 42.3. Given an application function definition, e.g. "void sendMessage(int data)", it is possible to automatically generate the replacement function as well as the equivalent I/O Handler function.

This approach was prototyped using a simple test application. The test application included an output function call defined as:

```
void callOne(MSG_HDR& data);
```

Based on this function prototype, the following code was generated:

```
// replacement function call
void callOne(MSG_HDR& data) {
cout<<"In Call void callOne(MSG_HDR& data) "<<endl;
```

```
// Insert process-boundary crossing mechanism here.
IOHandler::getInstance().callOne( data);
}

// Call in Federate I/O Handler
void FederateIOHandler::callOne(MSG_HDR& data) {

    std::string interfaceName;

    ///APPLICATION-SPECIFIC CODE TO GET INTERFACE
    ///NAME BASED ON DATA PARAMETERS
    switch (data.x) {
        case 0: interfaceName="CECToSSDSTcp"; break;
        case 1: interfaceName="CECToSSDSUdp"; break;
        case 2: interfaceName="SSDStoSLQ32"; break;
    };
    ///END OF APPLICATION-SPECIFIC CODE

    FederateInterface& fedIF = getFederate()
        ->getInterface(interfaceName);
    std::cout<<"In the IO Handler"<<std::endl;
    FederateInterface& federateInterface =
        this->federate->getInterface(interfaceName);
    std::cout<<"Interface name is "<<interfaceName<<std::endl;

    FederateMessage message(fedIF, getFederate()
        ->getInstanceNumber());
    Buffer& buffer = message.getBuffer();

    ///APPLICATION-SPECIFIC CODE TO SERIALIZE DATA
    Serializer::serialize(data, buffer);
    ///END OF APPLICATION-SPECIFIC CODE

    PostOffice::getInstance().sendMessage(message);
}
```

Note that there are two items in the I/O Handler that are application-specific and cannot be automatically generated:

- The code that identifies the interface that the I/O call applies to.
- The code that identifies the data type(s) contained in the message that need to be serialized (in the above example no extra code was needed; this may not always be the case)

For each function call, the source code to accomplish those two tasks could be placed into a file that the code generate reads and then inserts into the generated code in the appropriate place.

It should be possible to handle application input calls in a similar way, for applications that call functions that wait on input and retrieve. For applications whose

input is provided by a call into the application, another approach will have to be taken, possibly using an application-specific wrapper.

As applications are incorporated into the framework in FY08, the lessons learned about exactly what is required to bring an application into the framework will help to prepare automated tools for incorporation of future applications.

4.7.3 Intercepting Time or Other Application Calls

A similar approach could be taken for auto-generating code to intercept time-related calls, or any other types of calls for which the framework needs to provide replacement functionality, possibly including application calls to random number generators. The challenge with time calls in particular, is that the code to handle them may be unique to the call, unlike the boilerplated I/O approach.

4.7.4 Automated Code Searches

Automated tools could be developed to scan application code for calls that might need to be replaced. This could augment a manual eyeball search for calls. For example, a script could be written to search for anything that might be related to time, using a list of keywords. This might aid in identifying time-related calls that need to be replaced.

4.7.5 Automatic Application Code Modification

An ideal situation is that applications are incorporated into the framework without modifying the original source code. This may not be possible in all cases; there may be times in which it is necessary to alter the application code to operate within the framework. In those cases, it would be preferable that the modifications be automated, so that they can be repeated without error, to maintain records about what in an application was modified, and even to reverse those modifications when necessary.

4.7.6 Advantages to Automatic Application Incorporation

There are several potential advantages in automating as much as possible when incorporating applications. These advantages include:

- Reducing the amount of coding that has to be done manually, saving development and maintenance costs.
- Bookkeeping of exactly what was done to each application in order to incorporate it into the framework. This information could be used to help incorporate future versions of the same application, or other applications. This could also be used to reverse any changes made to applications.

- A GUI application could be developed as a front-end to the automated tools, making it easier on the end user to incorporate applications into the framework. For example, a user could enter a list of function definitions that were I/O related and thus needed to be replaced.

5 Risk Areas

These are the biggest technical risk areas that are envisioned for FY08 framework development:

- The viability of using interposition libraries to intercept time-related calls made by the CWS, WWS, and SSDSWS is still being investigated. If this does not work out, it might not be possible to incorporate those federates without changing application source code (specifically including the VxWorks middleware libraries).
 - This could be mitigated by future work to investigate interception of time calls at a lower level, such as modifying a Linux kernel to utilize a software clock, effectively putting an entire computer on a software clock.
- HLA RTI execution performance turns out to not be suited for faster-than-real-time application execution.
 - This is mitigated by a design which allows HLA components to be swapped out for other middleware in the future, with minimal disruption to the rest of the framework. Federate application software would not know the difference.
- The time required to study application source code to determine what I/O and time calls need to be intercepted, and then the testing of that capability with the actual federates, may turn out to be longer than expected.
 - The only mitigations available for this are more development and testing time, and more consultation with application experts.
 -
- Dependence on the MAK RTI entails some risk. These are the related issues that still need to be investigated:
 - Run-time performance
 - MAK requires use of the Sun C++ compiler to compile code that calls the RTI. This code cannot be linked with libraries built with other compilers, e.g. g++. Workarounds do exist, such as wrapping calls to the RTI in a C layer.
 - If Solaris 8 is used for either the WASP or SSDS federates, the latest version of the MAK RTI cannot be used, and the Solaris 8 version (MAK 3.1.2) is only expected to be supported for one more year.

- Compiler incompatibilities may restrict what can be communicated via the RPC mechanism described earlier. For example, C++ object files compiled with different compilers, or sometimes even different compiler versions, cannot be linked together. This can be mitigated by using simple data structures across process boundaries where possible (though nothing can be done about existing message structs), or possibly compiling all intra-process objects with C linkage, which is compatible. More research is required to fully determine the risk. It should be noted that IPC between applications using different compilers is something that has been successful on numerous applications, and should not be a problem if C-style data structures are used as the primary I/O mechanism.

6 Potential Future Work

There are several opportunities for future work with this framework. Those options include but are not limited to:

- Integration with OASIS. As OASIS matures, it may prove to be a better middleware option than HLA.
- Support for additional programming languages, such as MATLAB and Java. Both of these languages are capable of calling C/C++ code, so a thin layer on top of the framework developed in either of those languages should be doable, and allow a wider range of applications to be incorporated into the framework.
- Support for additional federates. There is interest at APL in modifying TSCEI to run independently of real-time; the framework may be a useful way to accomplish this. TSCEI presents unique challenges in that it is itself a framework rather than an application. Another candidate federate is the Single Integrated Air Picture (SIAP) Integrated Architecture Behavior Model (IABM). There is much analysis work to be done to determine the behavior of the IABM out in the “real world,” and there is also much interest in studying how the IABM interacts with existing systems. The IABM also presents interesting software engineering challenges in that it is developed using the Model-Driven Architecture approach using the Rhapsody toolset.
- Additional federate behavior (tactical data links, more engagement support, additional threats, combat identification)
- Some of the advanced features as described in the requirements.

7 Related Resources

The APL Sharepoint website at <http://aplteam/jhuapl/ADSD/projects/CSSF/> contains a number of useful memoranda and other documents, including the documents prepared as part of this IR&D. Especially useful memoranda that describe the Software Clock, Time Server, and SSDS Federate can be found in the *Shared Documents/PRA* folder. OASIS documentation can be found in the *Shared Documents/OASIS Framework* folder. Documentation created for this IR&D can be found in the *Shared Documents/Tiger Team* folder. Please contact Richard Bourgeois, Adam Miller, or Mark Schmid to request access.

The IEEE 1516 family of specifications comprise the HLA specification. They can be downloaded for free from APL computers at <http://ieeexplore.ieee.org/xpl/standards.jsp>.

The [Wikipedia entry for UML](#) is a good starting point for UML research and related topics.

8 Appendix

8.1 *Additional Use Case Details*

These use cases were included in the original list of use cases, and were investigated at a detailed level, however as the design evolved they either became redundant with other use cases, or in a few instances are not applicable for FY08. They are included here for reference.

Use Case 10.3

Summary

User specifies two instances of Federate A and one instance of Federate B to be part of a Federation.

Preconditions

See use case 10.2

Triggers

See use case 10.2

Basic Course of Events

Same course of events as use case 10.2 applies. Each instance of Federate A is given a different instance number. This leads to a different federate name (The federate name passed into the RTI is a string consisting of the federate type and instance number). This instance number is also used to distinguish between the two instances of Federate A.

Alternate Paths

See use case 10.2

Postconditions

See use case 10.2

Design Notes

See use case 10.2

Sequence Diagram(s)

See use case 10.2

Use Case 10.5

Summary

An existing federation named “Fed1” is deleted. The federation still has federates within it.

Preconditions

Federation “Fed1” exists, and contains at least one active federate.

Triggers

This could be part of a framework shutdown sequence, or the user has changed the configuration of which federates/federations are going to execute.

Basic Course of Events

See use case 0.2 for the sequence of events.

Alternate Paths

See use case 2.

Postconditions

Design Notes

This could be considered a sub-use case of 0.2, “Framework is shut down.”

Sequence Diagram(s)

The sequence of events for this use case is embedded within Use Case 0.2.

Use Cases 11.1 through 11.4

Summary

11.1 Two separate Federations “Fed1” and “Fed2” are instantiated.

11.2 One instance of Federation A joins “Fed1.” Another instance of Federation A joins “Fed2”.

11.3 Fed1 starts a scenario. Fed2 has not yet started a scenario. (Related use case 31.X for starting a scenario).

11.4 Fed1 and Fed2 are both running scenarios. Fed1 finishes while Fed2 is still running. (Related use case 32.X or 33.X for ending a scenario).

Design Notes

These use cases should be accounted for by other use cases, and a scalable design. For example, if all commands from the User Interface (or other controlling application) include either the federation name, or some other unique identifier of a federation then each command can be addressed to the proper framework process (one framework process per federate, rather than one per computer, would probably be required). Any strings used to initialize IPC, or any other resources, would need to be uniquely identified per federation.

These use cases are beyond scope for FY08.

Sequence Diagram(s)

N/A.

Use Case 30.1

Summary

Control Interface receives load scenario command from user interface. This happens after the user has entered/configured their scenario. This load command triggers related use cases for initializing one or more federates, and sending all federates scenario data (use cases for requirements 10, 11, and 20).

The sequence of events here is already covered by use case 20.1.

Use Case 30.2

Summary

User hits load scenario for a new scenario. Federates have the same names and are in same federations as the previous scripted scenario. Ensure that no new federations/federates are created, and that federates don't need to join or resign from federations.

Preconditions

One scenario has already started and finished, and the federates are still members of their respective federations....meaning the Configuration hasn't changed. See use case 20.1 for other prerequisites.

Triggers

See use case 20.1

Basic Course of Events

See use case 20.1

Alternate Paths

See use case 20.1

Postconditions

See use case 20.1.

Design Notes

After this use case was written, the design has evolved such that a Configuration object holds data about which federates are active, and which federates talk to each other and which don't. The Scenario object holds data specific to a given scenario. If the scenario data changes, but the Configuration doesn't, the sequence of events in use case 20.1 is simply repeated.

Sequence Diagram(s)

None.

Use Case 30.3

Summary

User hits load scenario for a new scenario. The same federation(s) as the previous scenario exist, and the same federates exist, but one of the federates has a different name.

Preconditions

One scenario has already started and finished, and the federates are still members of their respective federations.

Triggers

N/A

Basic Course of Events

N/A

Alternate Paths

N/A

Postconditions

One federate has been renamed.

Design Notes

Design decisions subsequent to this use case being written have rendered it moot. Federate names are based on the Federate type and the instance number – the user has no control over federate naming. If federates are added or removed from a Configuration, the Configuration would simply be reloaded – see use case 10.4

Sequence Diagram(s)

N/A

Use Case 30.4

Summary

User hits load scenario, and the federation and federates have different names than before.

Preconditions

One scenario has already started and finished, and the federates are still members of their respective federations.

Triggers

Basic Course of Events

Alternate Paths

Postconditions

The properly named federations and federates are established.

Design Notes

Federate names are derived from Federate Types and instance numbers, and thus are beyond control of the user. The Federation name would be given in the Configuration, and would be initialized when a new Configuration object is created (see use case 10.4 and 34.1 for initializing Configurations, and handling changes).

Sequence Diagram(s)

Use Case 31.2

Summary

Two federations have been scripted, Fed1 and Fed2. Control Interface receives start scenario command for Fed1, but not Fed2.

Design Notes

Multiple simultaneous federations are beyond scope for FY08. This feature can also be achieved by designing for scalability, in this case by having the startScenario command include the federation name, or some other data uniquely identifying the federation.

Use Case 32.2

Summary

Control Interface receives abort scenario message for only one federation. Every federate in that federation stops running and purges its data. Postcondition: the specified federation has stopped running.

Design Notes

The sequence of events in this use case have been covered by Use Case 32.1.

Use Case 33.1

Summary

When scripting a scenario, user scripts scenario end time. Postcondition: scenario end time is stored, so that framework knows when to end scenario. May be related to use case 20.1.

Preconditions

None.

Triggers

See design notes.

Basic Course of Events

See design notes.

Alternate Paths

None.

Postconditions

Scenario end time is stored.

Design Notes

Scenario end time is stored as one of the attributes in the Scenario object, as described in use case 20.1.

Sequence Diagram(s)

None.

Use Case 33.3

Summary

Once scenario end time has been reached, all federates in that federation stop executing.

Preconditions

Framework has determined that scenario end time has been reached, for a given federation.

Triggers

See use case 33.2.

Basic Course of Events

See use case 33.2.

Alternate Paths

See use case 33.2.

Postconditions

All federates in that federation are no longer running scenarios.

Design Notes

Use case 33.2 covers the chain of events related to this use case.

Sequence Diagram(s)

None.

Use Case 34.2

Summary

User has scripted batch file of scenarios, and the first scenario has finished execution. The next scenario from the batch file is read and executed.

Preconditions

Out of a batch file of scenarios to run, the first scenario has finished execution.

Triggers

Framework is ready to run the next scenario.

Basic Course of Events

See Use Case 34.1 for the sequence of events.

Alternate Paths

See Use Case 34.1

Postconditions

See Use Case 34.1

Design Notes

See Use Case 34.1

Sequence Diagram(s)

None. Sequence diagram for Use Case 34.1 covers this Use Case.

Use Case 40.1

Summary

Given some sort of configuration indicating which federates must be in a given federation, the framework puts each federate into an executing state.

This use case is already covered by use cases 10.X and 20.1.

Use Case 47.6

Summary

While the framework is running in discrete-time mode, we will still need to send some control messages out immediately (i.e. scenario control messages). Framework component sends such a message. Our lower-level message control needs to ensure that the message is sent immediately.

Preconditions

None.

Triggers

Either a framework component (such as the Scenario Controller) is ready to send a framework message, or an incoming framework message has been received.

Basic Course of Events

Message Sending

1. When the Scenario Controller (or any other framework component) is ready to send a message, it invokes the `sendFrameworkMessage` method in the Framework I/O Handler.
2. The Framework I/O Handler first creates a Framework Message and obtains a reference to its contained byte buffer.
3. The Serializer class is used to serialize the framework message into the byte buffer.
4. The Framework Message is then passed to the `sendMessage` method in the HLA Post Office.
5. The HLA Post Office invokes the real-time `sendRO` method in the Framework Federate Ambassador, passing it the Framework Message.
6. Finally, the byte array representing the serialized message is passed to the `sendInteraction` method in the RTI Ambassador.

Message Reception

1. Before message reception of framework messages can occur, the Scenario Controller (or any other framework component) must first invoke the `registerCallback` method in the Framework I/O Handler to notify it of the callback method that should be invoked upon message reception.
2. When the RTI Ambassador receives an interaction, it passes the byte array to the Framework Federate Ambassador via its `receiveInteraction` method.
3. The Framework Federate Ambassador passes the interaction to the `recvInteraction` method in the HLA Post Office.
4. Based on information in the interaction, if the HLA Post Office determines that the contained message is a framework message, it creates a Framework Message object from the Buffer and passes this message object to the Framework I/O Handler via its `recvMessage` method.
5. The Framework I/O Handler de-serializes the message by first getting the type contained within the Buffer and then invoking the `deserialize` method in Serializer.
6. Finally, the de-serialized object is passed up to the Scenario Controller (or any other framework component) using the callback function that was previously registered in step 1 above.

Alternate Paths

If an error occurs during message sending (via the `sendFrameworkMessage` method) or an error occurs during the callback registration (via the `registerCallback` method), an exception is raised and propagated up to the invoking framework component (such as the Scenario Controller).

Postconditions

None.

Design Notes

In the registerCallback method, the argument should probably be a class that contains the callback function rather than the function pointer itself.

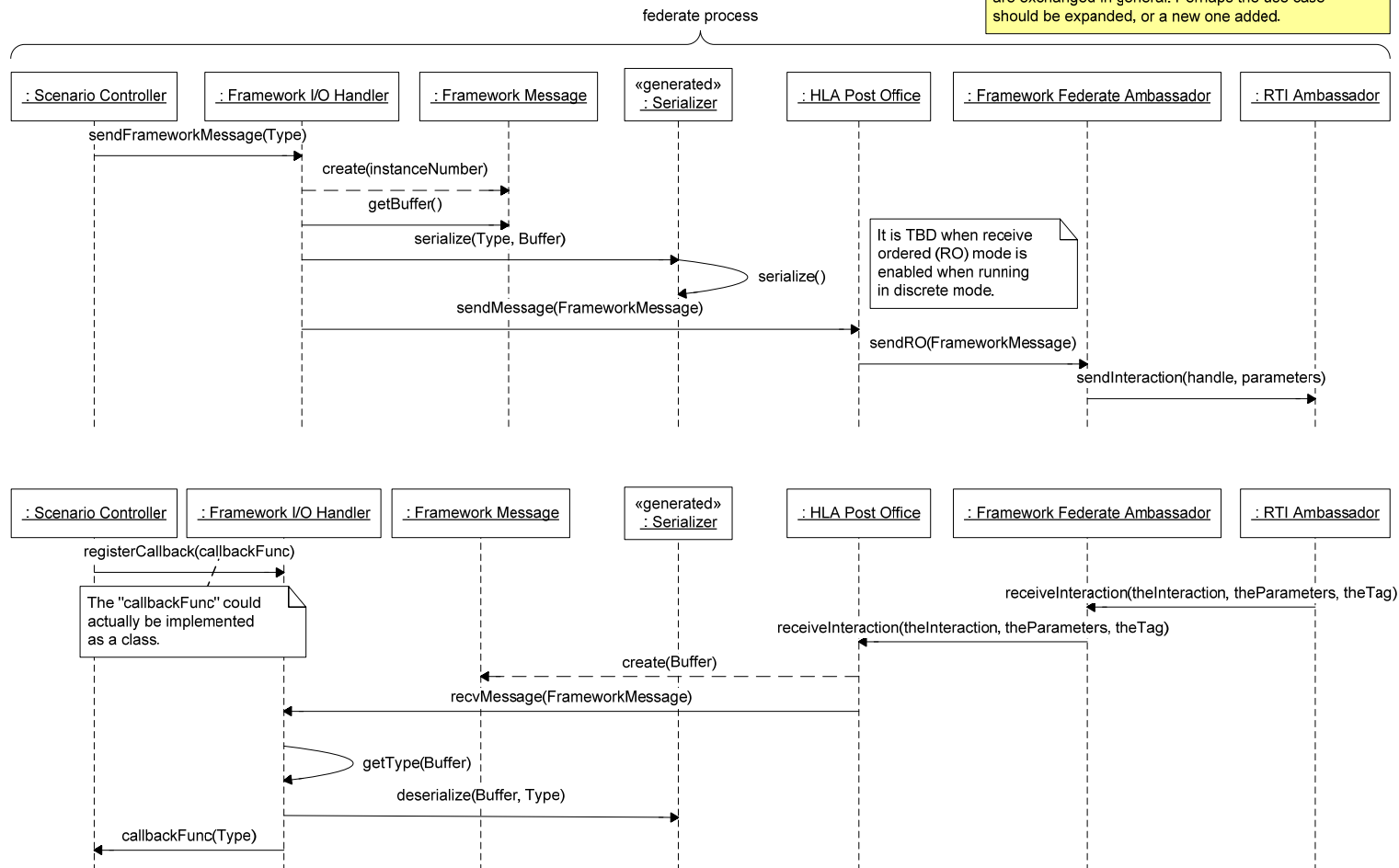
When in discrete mode, it is currently unclear when RO mode must be enabled for receiving framework messages.

Sequence Diagram(s)

Sequence Diagram for Use Case 47.6

“While the framework is running in discrete-time mode, we will still need to send some control messages out immediately (i.e. scenario control messages). Framework component sends such a message. Our lower-level message control needs to ensure that the message is sent immediately.”

This sequence diagram really shows more than just this use case: it shows how framework messages are exchanged in general. Perhaps the use case should be expanded, or a new one added.



Use Case 49.3

Summary

Special case of 49.1. If multiple federations are being executed simultaneously, each federation will have a different start_scenario_time.

Design Notes

Multiple simultaneous federations are beyond scope for FY08. This feature can also be achieved by designing for scalability, in this case by having the startScenario command include the federation name, or some other data uniquely identifying the federation. Thus only one particular federation is started, and its start time is only held within that federation.

Use Case 64.3

Summary

Write all framework messages to a data extraction file for possible use in framework debugging.

Preconditions

None.

Triggers

Some controlling application starts DX or stops DX (see below).

Basic Course of Events

1. The controlling application begins DX of framework messages by calling the startDx method in the Framework I/O Handler. A unique filename must be provided.
2. The Framework I/O Handler creates the DX disk file by calling the open method in FileIo.
3. Whenever the Post Office receives an incoming buffer (as described in Use Case 42.3), and it is determined to contain a framework message, it creates a Framework Message object from the incoming Buffer.
4. The Framework Message is passed to the Framework I/O Handler via its recvMessage method.

5. Obtaining the Buffer from the Framework Message using the `getBuffer` method, the Framework I/O Handler writes the buffer to the DX disk file using the `write` method in `FileIo`.
6. When the controlling application stops DX by calling the `stopDx` method in the Framework I/O Handler, a call is made to the `close` method in `FileIo`.

Alternate Paths

None.

Postconditions

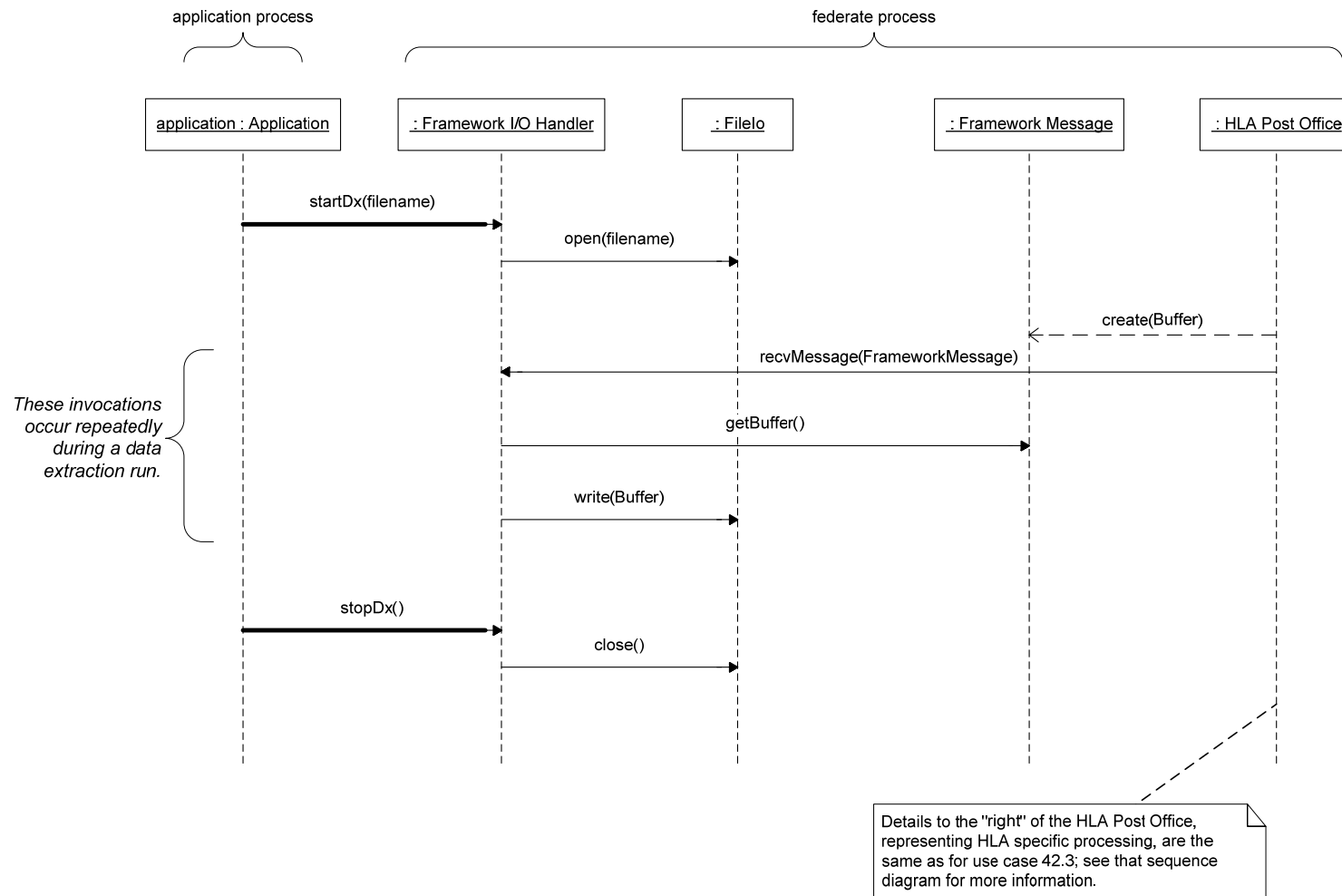
None.

Design Notes

It is still TBD how the control for starting and stopping DX will occur.

Sequence Diagram(s)

Sequence Diagram for Use Case 64.3
“Write all framework messages to a data extraction file for possible use in framework debugging.”



Use Case 75.1

Summary

Framework finishes scenario execution, makes a call to federates to purge their data. This could be considered part of an “end scenario” call. Up to the federate to decide how to purge their data; if the federate has a purge function already, great, otherwise we may need to kill and restart the federate processes. This is related to the use cases for requirement 33.

Design Notes

This is redundant with use cases 32 and 33. No new sequences of events are introduced. Purging federate data is also related to random number seeds; see use case 27.1.

8.2 *Complete List of Use Cases*

The following table presents the complete list of use cases identified from the requirements.

Rqmnt #	Requirement	Use Case
10	Specify which components are going to be executed, who interacts with whom, etc	<p>10.1 A federation named “Fed1” is established.</p> <p>10.2. User specifies one instance of Federate A and one instance of Federate B to be part of a Federation.</p> <p>10.3. User specifies two instances of Federate A and one instance of Federate B to be part of a Federation.</p> <p>10.4. A federation exists consisting of two instances of Federate A and two instances of Federate B. Let’s call them A.1, A.2, B.1, B.2. Federate A.1 and B.1 want to subscribe to each other’s messages, but not to identical message types from A.2 and B.2.</p> <p>10.5 An existing federation named “Fed1” is deleted. The federation still has federates within it.</p> <p>Note: These use cases will help us define what needs to be in whatever we use for a configuration/scenario definition file.</p>
11	Run multiple scenarios independently and concurrently, if hardware permits.	<p>11.1 Two separate Federations “Fed1” and “Fed2” are instantiated.</p> <p>11.2 One instance of Federation A joins “Fed1.” Another instance of Federation A joins “Fed2”.</p> <p>11.3 Fed1 starts a scenario. Fed2 has not yet started a scenario. (Related use case 31.X for starting a scenario).</p> <p>11.4 Fed1 and Fed2 are both running scenarios. Fed1 finishes while Fed2 is still running. (Related use case 32.X or 33.X for ending a scenario).</p>
20	All components must support a common scenario (received via framework)	<p>20.1 User scripts a scenario. Assume federates have already joined the desired federation. Scenario initial conditions are delivered to all federates. Scenario initial conditions should include a generic enough placeholder so that users of the framework could add references to more data if need be (e.g. references to input files). Design note: This could simply be an abstract class with a basic implementation in the framework code.</p>

Rqmnt #	Requirement	Use Case
21	Unit groundtruth data scripted into scenario	None? Use case 20.1 accounts for any type of data. If we are using WASP to drive other federates, will not need a new scripting mechanism for unit groundtruth data.
22	Target groundtruth data scripted into scenario	None? Use case 20.1 accounts for any type of data. If we are using WASP to drive other federates, will not need a new scripting mechanism for target groundtruth data.
23	Radar contact data scripted into scenario.	None? Use case 20.1 accounts for any type of data. If we are using WASP to drive other federates, will not need a new scripting mechanism for target radar data.
24	Navigation data scripted into scenario.	None? Use case 20.1 accounts for any type of data. If we are using WASP to drive other federates, will not need a new scripting mechanism for nav data.
25	ID data scripted into scenario.	None? Use case 20.1 accounts for any type of data. If we are using WASP to drive other federates, will not need a new scripting mechanism for ID data.
26	IFF data scripted into scenario. H	None? Use case 20.1 accounts for any type of data. If we are using WASP to drive other federates, will not need a new scripting mechanism for ID data.
27	Add noise/randomness for monte-carlo runs.	27.1 User specifies random number seed. This seed is sent to all federates as part of initial condition data. Note: If the random number seeds can be scripted through the WASP, does this use case still make sense?

Rqmnt #	Requirement	Use Case
30	Capability to load a scenario. H	<p>30.1 Control Interface receives load scenario command from user interface. This happens after the user has entered/configured their scenario. This load command triggers related use cases for initializing one or more federates, and sending all federates scenario data (use cases for requirements 10, 11, and 20). Design Notes: Postcondition: every federate is in their respective federation, and has received their initial condition data. The involved federation(s) are ready to execute the scripted scenario.</p> <p>30.2 User hits load scenario for a new scenario. Federates have the same names and are in same federations as the previous scripted scenario. Ensure that no new federations/federates are created, and that federates don't need to join or resign from federations. Design Notes: Precondition: One scenario has already started and finished, and the federates are still members of their respective federations.</p> <p>30.3 User hits load scenario for a new scenario. The same federation(s) as the previous scenario exist, and the same federates exist, but one of the federates has a different name. Design Notes: Precondition: One scenario has already started and finished, and the federates are still members of their respective federations. Postcondition: One federate has been renamed. Note: Can we avoid the need for this use case if we don't require user to name federates? Or does the user need to be able to name federates so that we can figure out which federates receive which messages (and other things)? Can we keep the federates in the federation and simply rename it? Design idea: if we have to resign and then join again, could hide that behind a rename function call that we define.</p> <p>30.4. User hits load scenario, and the federation and federates have different names than before. Design Notes: Precondition: One scenario has already started and finished, and the federates are still members of their respective federations. Postcondition: The properly named federations and federates are established.</p>

Rqmnt #	Requirement	Use Case
31	Capability to start a scenario.	<p>31.1 Control Interface receives start scenario message to start all federations from user interface. Precondition: Use case 30.1 has been executed. Post-condition: all federations are running their scripted scenario.</p> <p>31.2 Two federations have been scripted, Fed1 and Fed2. Control Interface receives start scenario command for Fed1, but not Fed2.</p> <p>Design notes: Precondition: Use case 30.1 has been executed. Postcondition: Fed1 is running scenario, Fed2 is not.</p>
32	Capability to abort a scenario gracefully.	<p>32.1 Control Interface receives abort scenario message for all federations. Every federate stops running and purge its data. Postcondition: no federations are running.</p> <p>32.2 Control Interface receives abort scenario message for only one federation. Every federate in that federation stops running and purges its data. Postcondition: the specified federation has stopped running.</p>

Rqmnt #	Requirement	Use Case
33	Scenario ends at its scripted time.	<p>33.1 When scripting a scenario, user scripts scenario end time. Postcondition: scenario end time is stored, so that framework knows when to end scenario. May be related to use case 20.1.</p> <p>33.2 Framework needs to determine whether scenario time has been reached, for a given federation. Design Notes: Precondition: Scenario is running. Postcondition: If scenario end time hasn't been reached, scenario keeps running. If scenario end time has been reached, framework is ready to end scenario.</p> <p>33.3 Once scenario end time has been reached, all federates in that federation stop executing. Design notes: Precondition: Framework has determined that scenario end time has been reached, for a given federation. Post condition: All federates in that federation are no longer running scenarios.</p>
34	Run multiple scenarios from a batch file	<p>34.1 User scripts multiple scenarios to be run. This could be the same scenario, multiple runs for monte-carlo, or completely different scenarios. Design Note: Postcondition: Framework has stored up information about each scenario, so that it can run each scenario sequentially without requiring interaction from user.</p> <p>34.2 User has scripted batch file of scenarios, and the first scenario has finished execution. The next scenario from the batch file is read and executed.</p>

Rqmnt #	Requirement	Use Case
35	Capability of pausing a running scenario.	<p>35.1 Pause command is received from the user interface, resulting in software clock being paused. Design Notes: Precondition: scenario is currently running. Postcondition: each federate is not currently executing the scenario, but is capable of picking up where it left off once the system clock unblocks.</p> <p>35.2 A “resume” command is received from the user interface, resulting in a paused scenario executing again where it left off. Design Notes: Precondition: a previously running scenario is paused. Postcondition: scenario is no longer paused; all federates are running.</p> <p>Note: these will be related to clock-related use cases for requirements 44, 45 and 47.</p>

Rqmnt #	Requirement	Use Case
40	Execute all federates.	<p>May be related to requirement 31, starting a scenario. We should try to distinguish starting a scenario between starting execution of a federate. For example, some federates we might want to kill their processes entirely and start them up again before the next run, as a means for purging their data. Other federates may have a “purge” call already built in, thus we can simply call that at the end of a scenario, and leave components running.</p> <p>40.1 Given some sort of configuration indicating which federates must be in a given federation, the framework puts each federate into an executing state.</p> <p>Design Notes: Precondition: Federates are not executing. Postcondition: Federates are executing.</p> <p>We may need to handle cases in which multiple instances of a federate can exist, and are in various states.</p>
41	Support scenario control: Initialize/shutdown federates. H	<p>This may be redundant with other use cases, and related to the use cases for requirement 40. From a design perspective, I think what we really need to do is figure out what states a federate can go through in its life cycle. For example:</p> <ul style="list-style-type: none"> • Not executing. • Executing, not running scenario, does not have initial scenario data. • Executing, has any necessary scenario data, is not executing scenario. • Executing scenario.

Rqmnt #	Requirement	Use Case
42	Message passing between federates.	<p>42.1 Federate publishes what messages it's capable of sending. In HLA these would be interactions.</p> <p>42.2 Federate indicates what messages it wants to subscribe to, and which federates it wants to receive data from. This is related to use case 10.4.</p> <p>42.3 Federate sends a message. The federates who have subscribed to that message from that federate receive the message. No other federates in the federation, and no federates outside of the federation, receive the message.</p> <p>Note: message sending mechanism may be different depending on real-time or discrete modes. See requirements 44, 45, and 47.</p>
43	Provide mechanism to exchange any data among framework and application components.	<p>This is somewhat related to the use cases for requirement 42. We should ensure that the use cases in 42 are implemented such that a message can contain anything that a federate wants it to be.</p> <p>43.1 An arbitrary framework component needs to synchronize with its counterpart on another process or another computer. Our message passing mechanism needs to be flexible enough to handle that.</p>
44	Flexible to support various modes of operation: Real-time	See use cases for requirement 45. Real-time is a special case of scaled real-time, in which the scaling factor is 1.0.

Rqmnt #	Requirement	Use Case
45	Flexible to support various modes of operation: Scaled real-time.	<p>45.1 One federate sends a message to another (see requirement 42). This message is sent & received as fast as our middleware can do it. Design note: from the sender's perspective, it shouldn't matter whether the receiving federate(s) are distributed or not.</p> <p>45.2 A process in a federate makes a blocking time-related call. That call is scaled and then routed to the operating system's clock. Note that we are going to have to identify every time related call that a application could make, in order to turn that application into a federate.</p>

Rqmnt #	Requirement	Use Case
47	Flexible to support various modes of operation: Discrete time.	<p>Note: many of these use cases will involve incorporating the Software Clock and Time Server from the existing middleware into our framework. In these cases, we can consider the Software Clock and the Time Server to each be classes in our class design.</p> <p>47.1 Within a given federate, one thread of one process goes to sleep. Other threads/processes are still running.</p> <p>47.2 Within a given federate, all threads of one process have gone to sleep. Other processes in that federate have not gone to sleep completely.</p> <p>47.3 Within a given federate, all threads of all processes have gone to sleep. Postcondition: the federate does not execute until “woken up”, i.e. time is advanced.</p> <p>47.4 All federates have gone to sleep. The framework advances to the next lowest time advance value, and wakes up only the thread(s) that have requested that time advance.</p> <p>47.5 A federate sends a message out. Subscribing federates process that message only when time has advanced to the point at which the processing thread is supposed to wake up again. Design question: From an HLA perspective, what time stamp should federates put on that message, current time, current time plus lookahead (if we use lookahead) or a future time that represents a delay in getting the message out.</p> <p>Design note: If a federate sends a message to another federate, that gets queued up by the receiving federate, we need to make sure the receiving federate does not process that message until the responsible thread is woken up. We need to make sure that the federates do not immediately pull messages off of the queue to process them. I don’t think this can be achieved by having the sender time-stamp the message for future processing, because the sender can’t know when the receiver is supposed to be woken up.</p> <p>47.6 While the framework is running in discrete-time mode, we will still need to send some control messages out immediately (i.e. scenario control messages). Framework component sends such a message. Our lower-level message control needs to ensure that the message is sent immediately.</p>

Rqmnt #	Requirement	Use Case
46	Fast as possible: components execute simultaneously.	<p>This may depend on underlying middleware and its ability to pass messages to multiple federates without blocking them.</p> <p>46.1 Framework distributes a message to two different federates. The framework does not block on the first receiving federate.</p> <p>Design note: in some cases our replacement call scheme might take care of this on its own. For example, if a thread is looping and periodically checking a queue for messages, that thread would already be under our time control, real time or discrete. The framework would simply dump the message into the queue and then go on its merry way. Hopefully the middleware implementations on distributed systems would be able to send the message out simultaneously.</p> <p>Note: HLA users at APL have stated that even the DMSO RTI allows multiple federates to process simultaneously (they might be processing at different timestamps when running in discrete mode, but that is ok).</p>
48	Support components on distributed computers.	<p>Not sure if there are specific use cases for this requirement. All use cases that represent specific chains of events should be developed with a distributed framework and distributed federates in mind.</p>

Rqmnt #	Requirement	Use Case
49	Provide current time to components.	<p>49.1 Application code makes some sort of “getTime()” call while running in a scaled real-time mode. Framework returns a time value of $\langle \text{start_scenario_time} \rangle + \langle \text{time_since_start} \rangle * \langle \text{time_scale} \rangle - \langle \text{total_paused_time} \rangle$. (Or we could subtract out the starting scenario time and each application would “see” a starting time of zero).</p> <p>49.2 Application code makes same “getTime()” call while running in a discrete mode. Framework returns the time that the requesting federate is running at. The actual time should exist at a low level in our framework; when running in HLA mode, the RTI should maintain control of that the application time is.</p> <p>49.3 Special case of 49.1. If multiple federations are being executed simultaneously, each federation will have a different start_scenario_time.</p>

Rqmnt #	Requirement	Use Case
50	Need to intercept all calls to system clock	<p>Note: this requirement is from the “Framework Execution” section. May be related to requirement 70 which is from the “Component Requirements: Converting Real-Time to Discrete” section.</p> <p>These use cases would apply at application capture time rather than framework execution time.</p> <p>50.1 At application capture time, somehow obtain list of time related calls that a application makes. Make sure all those calls are managed by our framework’s time library. Note: may not be a priority for FY08 since this has already been done for CECWS, SSDSWS, and an older version of WWS. If we want to upgrade to newer WWS, it is possible there are calls not handled by current software clock middleware (or would those be updated as WWS middleware is updated?)</p> <p>50.2 Either include as part of the build, or autogenerate, or a mixture of both, our replacement calls for the time related calls.</p>

Rqmnt #	Requirement	Use Case
53	In addition to system clock calls, need to determine what other system calls to intercept and how to handle them	<p>When given a new federate, this will require a fair amount of research.</p> <p>53.1 At application capture time, somehow obtain list of calls that a application makes that we need to intercept. These would include, but not are limited to:</p> <ul style="list-style-type: none"> • time-related calls. • I/O calls. • scenario control calls <p>The uses cases for requirement 50 represent a specific subset of these cases.</p>
51	Framework must keep components in sync	<p>51.1 When running in real-time or scaled real-time, OS clocks need to be in synch.</p> <p>51.2 A scenario is started using distributed federates. The framework code associated with each federate needs to hold the same scenario start time (from the OS time) when running in a real-time or scaled real-time mode.</p>

Rqmnt #	Requirement	Use Case
52	Support endianness incompatibilities	<p>52.1 At application capture time, framework needs to generate code to serialize, including endian conversion, all data structures that are part of our federate's I/O.</p> <p>52.2 If message data is packed in a generic form, receiving portion of the framework needs identifier of what deserializer/endian converter to apply to deserialize the data before sending up to the federate.</p> <p>52.3 At run time, serialize, including endian conversion, any extra data we send along with a message, such as sending federate, ID of the handler used to deserialize, and anything else we send along.</p>
60	Determine when components "fall behind"	Probably beyond scope for FY08. Should be easy to determine in FY08 federates by message queue overload.
61	Measure performance of components.	Probably beyond scope for FY08. Not critical for design phase; as design develops, good ways to do this may become more apparent.
62	Ideally try to measure an optimum speed for a given scenario.	Probably beyond scope for FY08.
63	Ideally determine at run time if a component is processing when it shouldn't be	Probably beyond scope for FY08.

Rqmnt #	Requirement	Use Case
64	Extract all data that passes through framework.	<p>64.1 Provide mechanism for federate to automatically subscribe to and unsubscribe from every message from all other federates.</p> <p>64.2 Write all of those messages out to files. Note that it's a design question whether to deserialize or not. I recommend not, that way:</p> <ul style="list-style-type: none"> • all the data remains in network byte order • if our serialization code packs data fields into the next largest byte-aligned data type, we may already have a tool that could read in the data for DR (we would, however, have to create a config file that defined all the messages for that tool to read in, but that may not be too much work). <p>64.3 Write all framework messages to a data extraction file for possible use in framework debugging.</p>
66	File bookkeeping for multiple runs	<p>66.1 DX federate should save recorded data with unique indicators in file of executing federation name, scenario name or # (if in monte-carlo runs) and any other identifiers necessary. Also have a safety mechanism that if there are already files with the names we end up wanting to save as, we add on a timestamp or something else to ensure that we don't overwrite any data.</p>
67	Use existing DX capabilities.	<p>No use cases here, just a lab setup issue. Bring in any existing DX tools into our lab. Note: much preferable to have tools located in our lab than have to walk to another lab to do analysis, since there may be times when existing DX capabilities may be used for testing of our framework.</p>
70	No direct calls to system clock - all system clock calls need to go through framework	<p>Note: this is in the "Component Requirements: Converting Real-Time to Discrete" grouping of requirements, these use cases are be redundant with those for requirement 50.</p>

Rqmnt #	Requirement	Use Case
71	No infinite loops H	Note: This is one code construct that we need to figure out how to handle in a converted application. See requirement 72.
72	Identify programming constructs that, if left alone, may cause application to keep processing when it shouldn't, i.e. continuous processing when framework is running in a discrete mode.	This is application capture time case. For FY08, CECWS, WWS, and SSDS, this has probably already been done. Since C&D is a discrete event simulation, this shouldn't be an issue. So there's hopefully not much to do for this year.

Rqmnt #	Requirement	Use Case
73	All inputs received through framework	<p>Application capture time cases....</p> <p>73.1 Given a application, identify the mechanisms used to send input into a application. These could be C style function calls, object methods, data streams, etc.</p> <p>73.2 Either manually or automatically, generate replacements for the calls that the application would have made to receive input. If the application has functions that are called from outside to send them input, we need to create some sort of wrapper function that captures messages from the framework, converts them into what the application is expecting, and send them to the application.</p> <p>73.3 Do bookkeeping of which calls we have captured, for re-creation and documentation, or testing purposes (testing would be involved if an intercepted call might be causing defects)</p> <p>73.4 For any application code that gets manipulated, have a reverse capability.</p> <p>73.5 Ensure that calls that have been replaced do not get used at run time. This could be accomplished by interposition libraries if the application code is dynamically linked in. If the application code is statically linked in, the first alternative would be to not compile the file containing the function we need to replace. If that function is in a file with other functions that we do need, we probably want to automatically comment out, or remove altogether, that function.</p>

Rqmnt #	Requirement	Use Case
74	All outputs go to framework	Same use cases for requirement 73, application input, also apply to application output.
75	Purge all data in response to framework	75.1 Framework finishes scenario execution, makes a call to federates to purge their data. This could be considered part of an “end scenario” call. Up to the federate to decide how to purge their data; if the federate has a purge function already, great, otherwise we may need to kill and restart the federate processes. This is related to the use cases for requirement 33.
76	Common way for each component to define its message specs	76.1 At application capture time, have a way to list what each federate’s inputs and outputs are, e.g. what data structures they are using. This would lead to auto-generation of serialization code, see use cases for requirement 52. 76.2 Could have a way for application developer to enter through a GUI what the inputs and outputs are. Probably a future capability.

Rqmnt #	Requirement	Use Case
80	Discovery	<p>Design note: somewhere in our frameworks API should be the ability to discover what the other federates in the federation are. This should definitely be in the abstract layer that wraps the HLA interface, and possibly should be accessible at a higher level, perhaps even available as close to the application as possible, if we were to build in a capability for applications to dynamically discover the services they need.</p> <p>80.1 Federate A needs to subscribe to data from a particular instance of Federate B. Does Federate A need to discover B to subscribe to its messages? I'm not sure, more research may be required. Related to use cases for requirement 10.</p>
81	Dynamic data semantics incompatibility resolution	Beyond scope for FY08.
83	Automated process to modify existing applications to conform to framework.	Related to requirements 73 and 74, the same process to identify replacement calls and ensure that we run our calls instead of existing calls. I'm not sure how much design work we can do on this requirement yet. What we might want to do is start by manually incorporating our FY08 federates into the framework. Then, see what we could have done automate, partially automate, or at least capture information about what we did in case we need to recreate it.
84	Automagically support endianness incompatibilities.	See requirement 52. This is probably a duplicate requirement.

Rqmnt #	Requirement	Use Case
85	Create a library of components	85.1 This has been mentioned in other places, once we have an automated process to convert components, or at least capture what was done manually, we can store off those changes along with the application that has been converted into a federate, as well as any framework to application adaptation layers. The more automated the process is, the more flexible we can be for incorporating future versions of those applications. Not sure if this is a real use case.
86	Create a library of reusable software functions	Doesn't really apply to FY08 since we're not going to be doing a whole lot of new development. If we need some sort of data wrapper layer to get data from the framework into Aegis C&D model, maybe there would be code developed for that which would be a candidate for reuse.
87	Ensure repeatable results across monte-carlo runs.	We can't do much at design time for this, and it's probably beyond scope for FY08 anyways. Once we have a system running, we can investigate how close to repeatable results we are and possibly how to resolve cases in which our results are not repeatable.

Rqmnt #	Requirement	Use Case
90	Define interface between framework and display	<p>90.1 Framework should have interface to display that accommodates:</p> <ul style="list-style-type: none"> all scenario controls (start, stop, pause, change time scale). scenario scripting – unless users are required to manually script files at the beginning. If nothing else, the name of the file we’re going to use. a generic mechanism to support new data fields as new federates are added in the future would be nice. <p>90.2 If the framework is running in distributed mode, there should be one display for the whole framework. Perhaps there should be some sort of framework configuration file that identifies one of the distributed framework instances to be the controller and talk to the GUI. Or maybe the GUI connects to every framework instance.</p>
91	Create framework display	<p>91.1 Create a framework display.</p> <p>Design Notes: Non-critical design, deferred until a later date. If there is a well-defined interface to a display, a simple display can be developed later with minimal risk to the rest of the framework, hopefully fairly quickly. If the display doubles as a controlling application some safety mechanisms need to be coded.</p>
92	Incorporate available legacy displays	<p>More of a lab setup issue than anything else. Legacy applications will carry their Control Interfaces with them. Those interfaces shouldn’t need to run through the framework. We should make sure to include display machines in our lab setup requirements.</p>

Rqmnt #	Requirement	Use Case
100	Integrate with OASIS.	Beyond scope for FY08. Design note: wrap all HLA calls under abstract classes, so that in the future we could wrap OASIS (or TENA, or some other middleware's) calls under the same layer.
101	Integrate with HLA.	By definition in our design.
102	Integrate with NSSAT.	Beyond scope for FY08.
103	Integrate with ARTEMIS	Beyond scope for FY08. Integration with HLA should help with this.
104	Integrate with ANTARES	Probably beyond scope for FY08. Integration with HLA should help with this.
110	Maintain paper trail of modifications to existing code to conform to framework.	Not sure if direct use cases fall out of this. If we can use our meta-model idea to incorporate applications into our framework, we can capture information. Perhaps there is a use case. 110.1 Application-incorporation code should record, in log files, or through on-screen GUI, what modifications or replacement libraries are being used. 110.2 Record changes to our configuration so that they are repeatable.
111	Justification of why OASIS/Simular isn't sufficient on its own.	Not a software development requirement.
112	Justification of why HLA is not sufficient on its own.	Not a software development requirement.

Rqmnt #	Requirement	Use Case
113	Create and document approach for incorporating future applications	Not a software development requirement. Requirement 110 will help us gather this information to guide future application developers.
N/A	Non-functional requirement.	0.1 Framework is initialized.
N/A	Non-functional requirement	0.2 Framework is shut down.