Componentized Combat System Testbed IR&D: Framework Design Review

Adam Miller
Steve Kahn
15 January 2008

The Johns Hopkins University
APPLIED PHYSICS LABORATORY

IR&D Objectives

1. Establish a framework that allows real-time tactical code and simulation models to operate together.
2. Develop a capability to allow models to run either in real-time or as fast as possible for Monte Carlo analysis.
3. Develop a capability to evaluate networked and interacting sets of combat system components of increasing complexity.

Today's Design Review Scope

- Design of a framework that accomplishes objectives 1 and 2, and partially meets objective 3. Specifically, the framework shall:
  - Provide a means for real-time and discrete applications to interact and execute a common scenario.
  - Execute applications independently of real-time to run fast-as-possible for Monte Carlo analysis.
  - Provide a means to analyze the results.
- Related efforts that are important to meet objective 3, but out of scope for this review include:
  - Aegis C&D Modeling
  - "Sensor Manager" Development (Possible WASP Mods)
- However, comments on incorporating the applications from related efforts into the framework are entirely welcome!

Framework Goals

- 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 execute within the framework with little or no modifications to the source code.
- Provide guidance for…
  - Techniques to easily add additional applications to the framework in the future. We do not want a solution that is only effective for the applications targeted for FY08.

Terms and Definitions (1 of 3)

- Selected terms used throughout this design. The full list, along with detailed descriptions, are in section 2.5 of the paper.
  - *Application* – A software application that has been or could be incorporated into the framework, e.g. CEP, SSDS, WASP, Aegis C&D
  - *Application Process* – Executable process that is part of an application.
  - *Discrete-Time Mode* – A timing mode in which each federate is executed as fast as possible by immediately executing the "next" sequence of events.
  - *Interposition Library* – A unix/linux mechanism for having multiple shared libraries contain implementations of the same function call, along with the ability to call any of those implementations.

Terms and Definitions (2 of 3)

- *Federate* – An application that has been incorporated to execute within the framework, along with application-specific supporting framework code. Multiple instances of a federate can execute within the framework.
- *Federate Process* – An executable process within the framework that manages a specific instance of a Federate.
- *Federation* – A collection of federates that interact with each other to execute a common scenario.
- *Framework* – The software that supports execution of federations and federates.
- *Framework Process* – An executable process that supports execution of the framework and its federates.

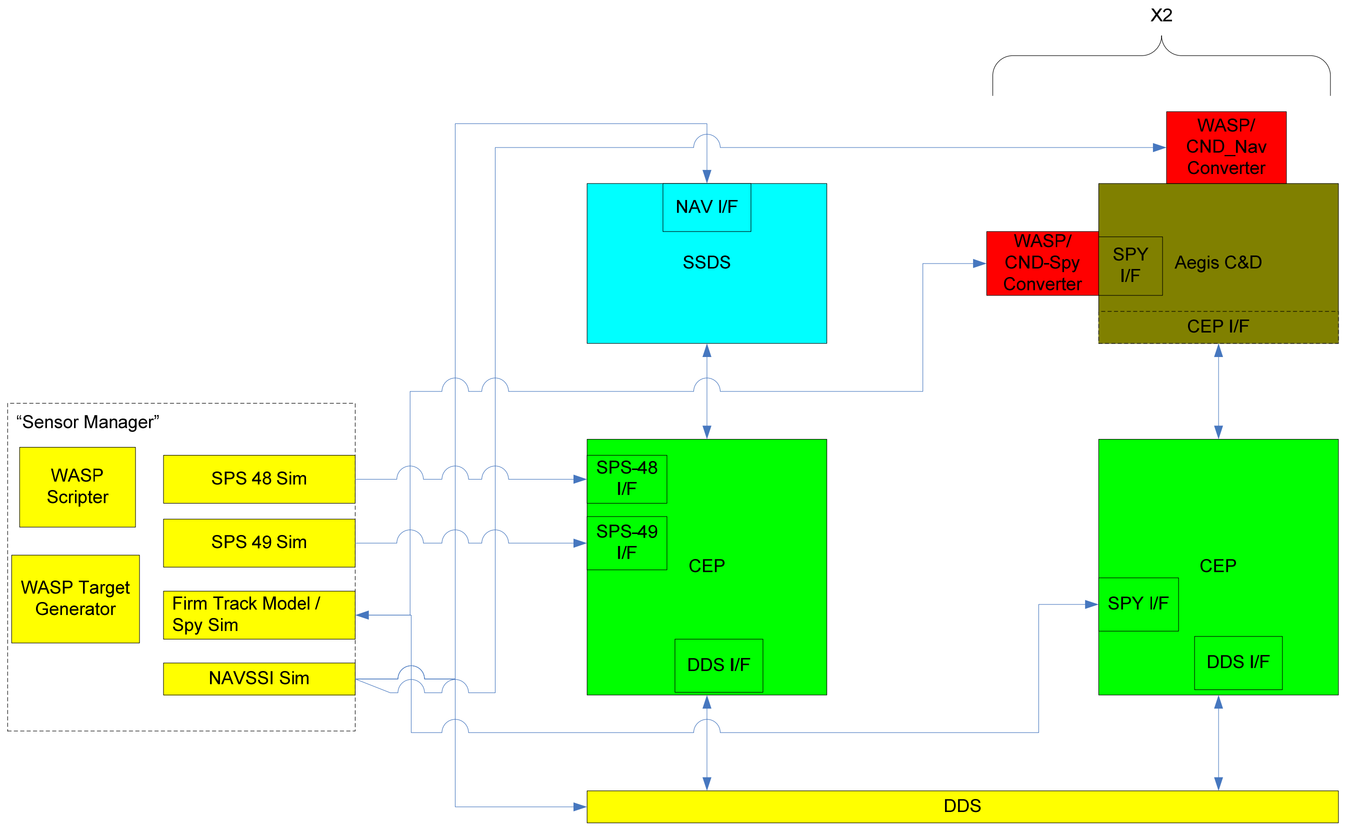
Terms and Definitions (3 of 3)

- ***High Level Architecture (HLA)*** **– An IEEE standard that defines specifications for a simulation interoperability framework.**
- ***Runtime Infrastructure (RTI)*** **– An implementation of the HLA specification.**
- ***Scaled Real-Time Mode*** **– A time mode, similar to real-time, in which time-related calls are scaled before being sent to their intended target. For example, the intent of a time scale of 2.0 would be to make an application execute twice as fast as real-time.**
- ***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 composed of WASP components.**

FY08 Federates

- CEP on a Workstation (CWS)
- SSDS on a Workstation (SSDSWS?)
- Sensor Manager
- Aegis Command and Decision (C&D) Model

FY08 Federates and Interfaces



Similar Efforts at APL

- Though not related programmatically, these efforts share many technical concepts with this IR&D.
  - OASIS / Simitar
    - Simulation framework specification / implementation developed by APL and MIT
  - Total Ship Computing Environment Infrastructure (TSCEI)
    - Raytheon's DDX infrastructure. Effort underway at APL to decouple from real-time.
  - SSDS Software in the Loop (SIL)
    - Execute recent versions of SSDS on a Workstation in real-time.

Framework Requirements (1 of 2)

- Detailed list of requirements is presented in the paper. Here are the key top-level requirements for FY08.
  - Framework must provide a mechanism for scripting active federates, inter-federate communications (which federates talk to each other), and initial scenario data sent to federates.
  - Framework must allow user to start and stop scenarios, as well as script repeated runs for Monte-Carlo analysis. Pause and resume commands are optional.
  - Framework must provide a mechanism for inter-federate input and output (I/O)
    - Note: In order to accomplish this with minimal intrusion to existing source code, framework will have to intercept federate I/O function calls.

Framework Requirements (2 of 2)

- Framework must provide a mechanism for federates to execute in discrete or real-time.
  - Note: In order to accomplish this with minimal intrusion to existing source code, framework must intercept time-related system calls.
- Framework must allow federates to run on distributed computers.
- Framework must support endianness incompatibilities.
- Framework must provide a data extraction (DX) capability.

Use Cases

- Use cases developed for many of the requirements, including text descriptions and sequence diagrams.
- Sequence diagrams shown later in this presentation, along with selected elements of text descriptions. Full text descriptions are in design paper.

Framework Design: Software Environment (1 of 2)

- C++ programming language selected
  - All FY08 federate applications are programmed in C or C++.
  - C++ is a high-performance programming language commonly used for real-time applications.
  - Most if not all HLA RTIs provide a C++ API.
  - Source code from PRA testbed and vxWorks middleware is in C++

Framework Design: Software Environment (2 of 2)

- FY08 Operating Systems
  - Determined by federate applications' supported OS as well as RTI

Federate	Operating System	Comments
CWS	Red Hat Enterprise Linux 4	Also runs on Solaris 10.
SSDSWS	Solaris 8 or 10 on SPARC	Hosting of SSDS tactical code on Solaris 10 is underway.
Sensor Manager (WWS)	Solaris 8 or 10 on SPARC hardware	WWS builds and runs on Solaris 10, though release versions are still built on Solaris 8
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.

Framework Design : Major Components (1 of 3)

- Framework can be looked at in multiple “views”, each view defined by dividing the framework using different criteria.
- Division by functionality:
  - Federate Management*
  - Framework Initialization and Scenario Control
  - Time Management
  - I/O Management
  - Framework Controlling Application
  - Application Capture**
  - Utilities

*Federate Management cuts across multiple areas.

**Application Capture is not a run-time function.

Framework Design : Major Components (2 of 3)

- Framework software could be divided up by executable process:
  - Framework Process (one per computer)
  - Federate Process (one per federate)
  - Application Process (at least one per federate)
- Framework software could be divided up into layers:
  - Federate/Application Specific
  - Generic Framework
  - Framework Middleware (e.g. HLA)
    - For FY08, this layer will be made up of HLA/RTI components.

Framework Design : Major Components (3 of 3)

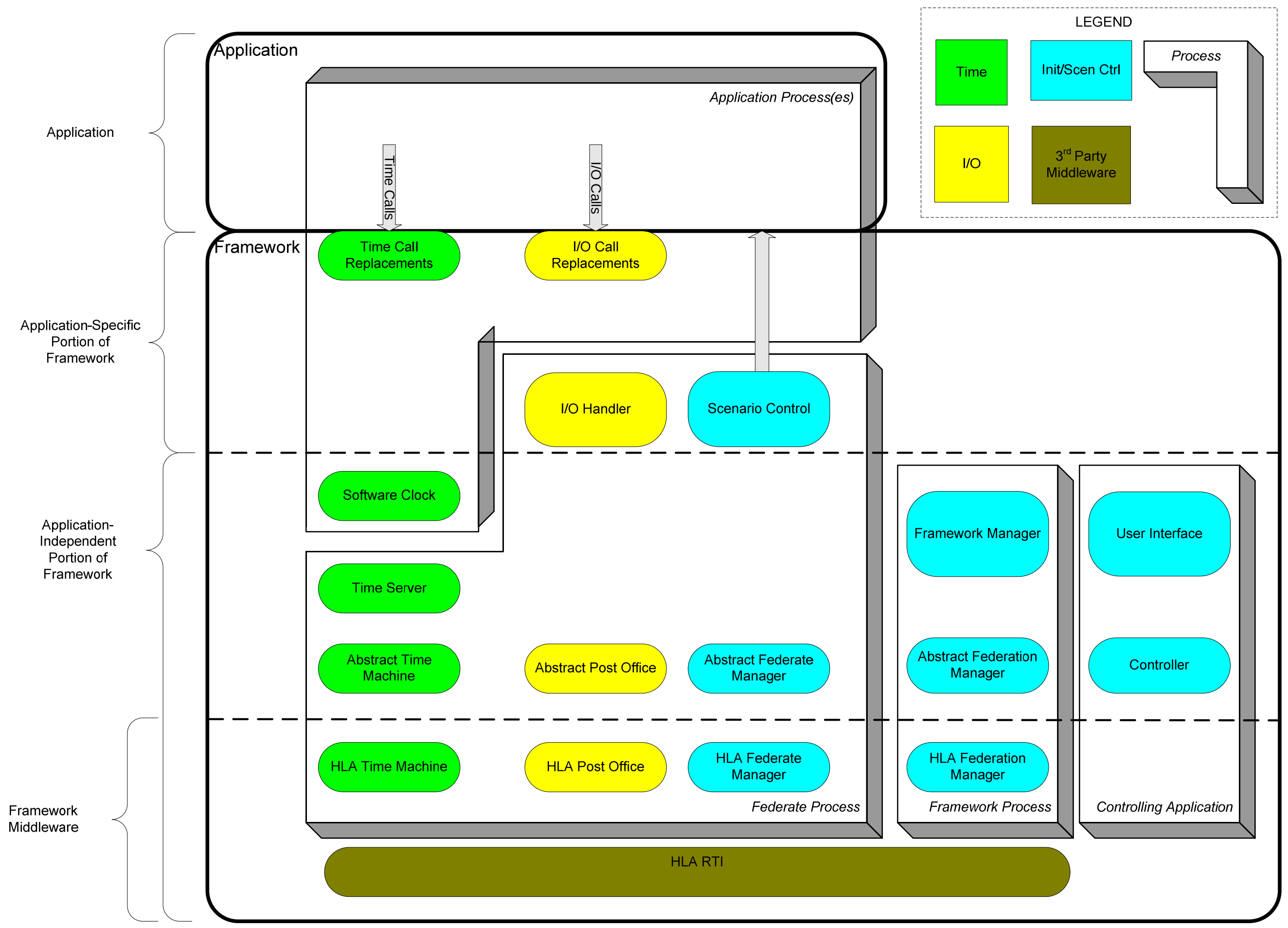


Diagram Notation (1 of 3)

- Our sequence diagrams utilize “extra” notation not part of standard UML sequence diagrams

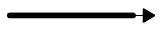
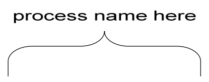
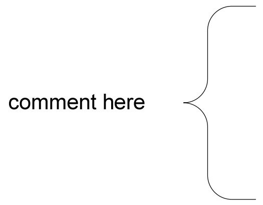
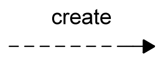
Notation	Description
	A message call with a bold arrow indicates that the message call crosses process boundaries.
	Indicates the process (framework, federate, or application) in which the object resides.
	Indicates the comment to the left of the braces applies to the sequence of events enclosed by the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

Diagram Notation (2 of 2)

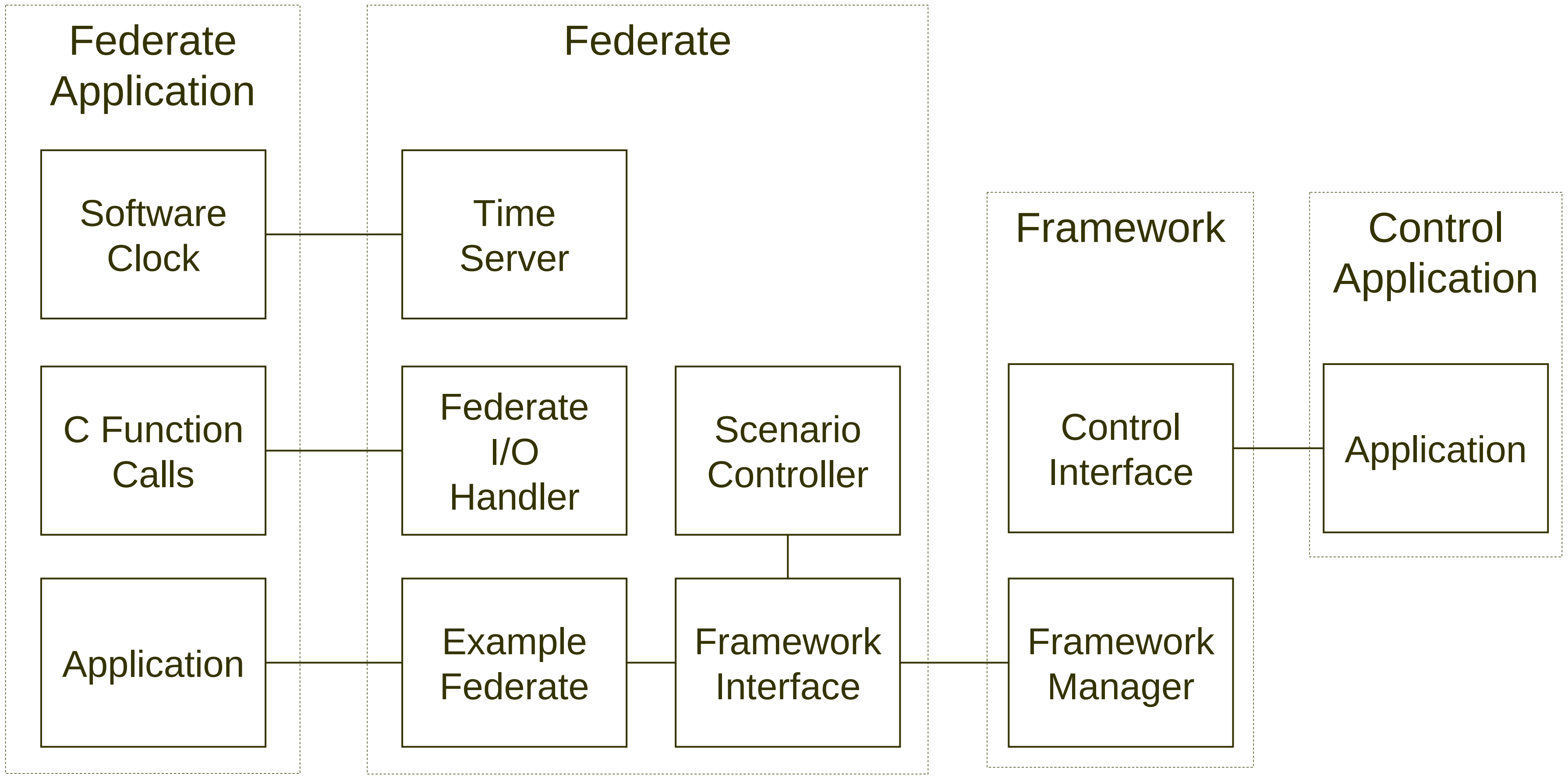
Notation	Description
$\xrightarrow{\textbf{provided\_function\_call()}}$	A function name in bold indicates that the function call was provided by external middleware, such as the calls provided by the HLA RTI.

Diagram Notation (3 of 3)

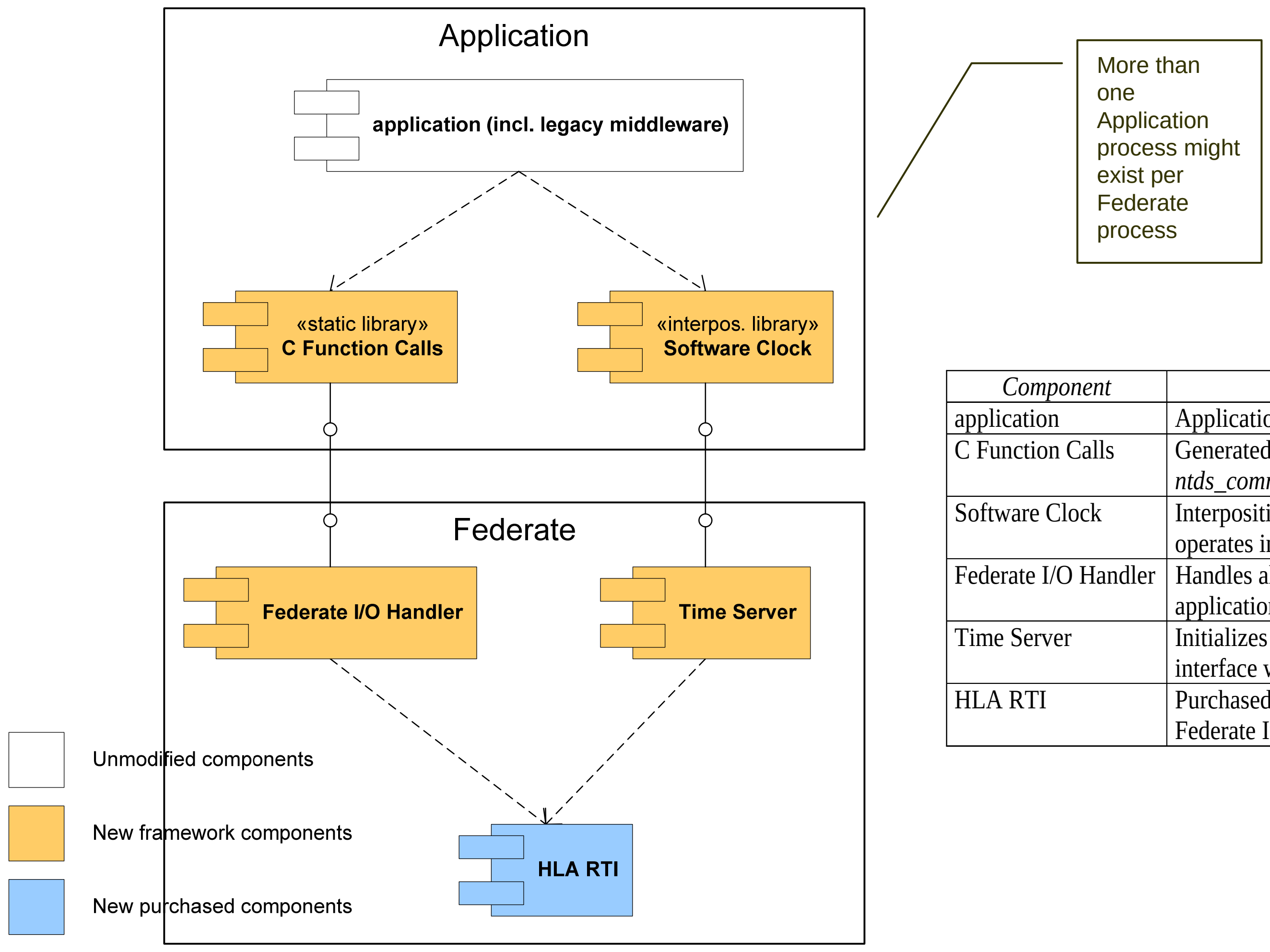
- Several new stereotypes used in class and sequence diagrams. These are probably not critical to understanding the diagrams, thus are not described here. Refer to section 4.3.2 for detailed descriptions of the stereotypes.
- Class diagrams: To reduce clutter, multiplicities of 1 to 1 associations are not shown. An association shown in our class diagrams without multiplicities can be assumed to be 1 to 1.

Framework Processes

Process Boundaries and Classes

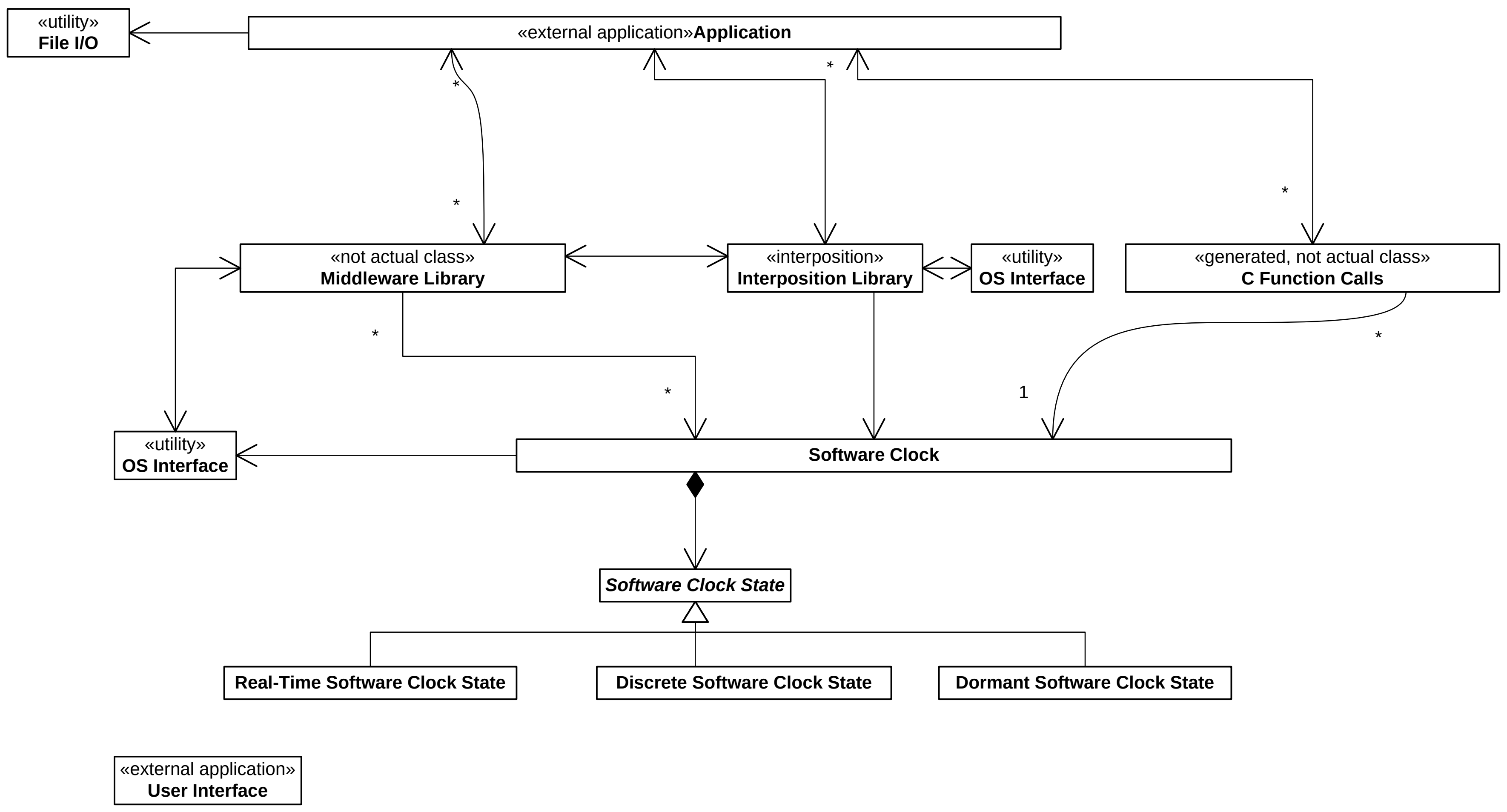


Framework Architecture: I/O and Time

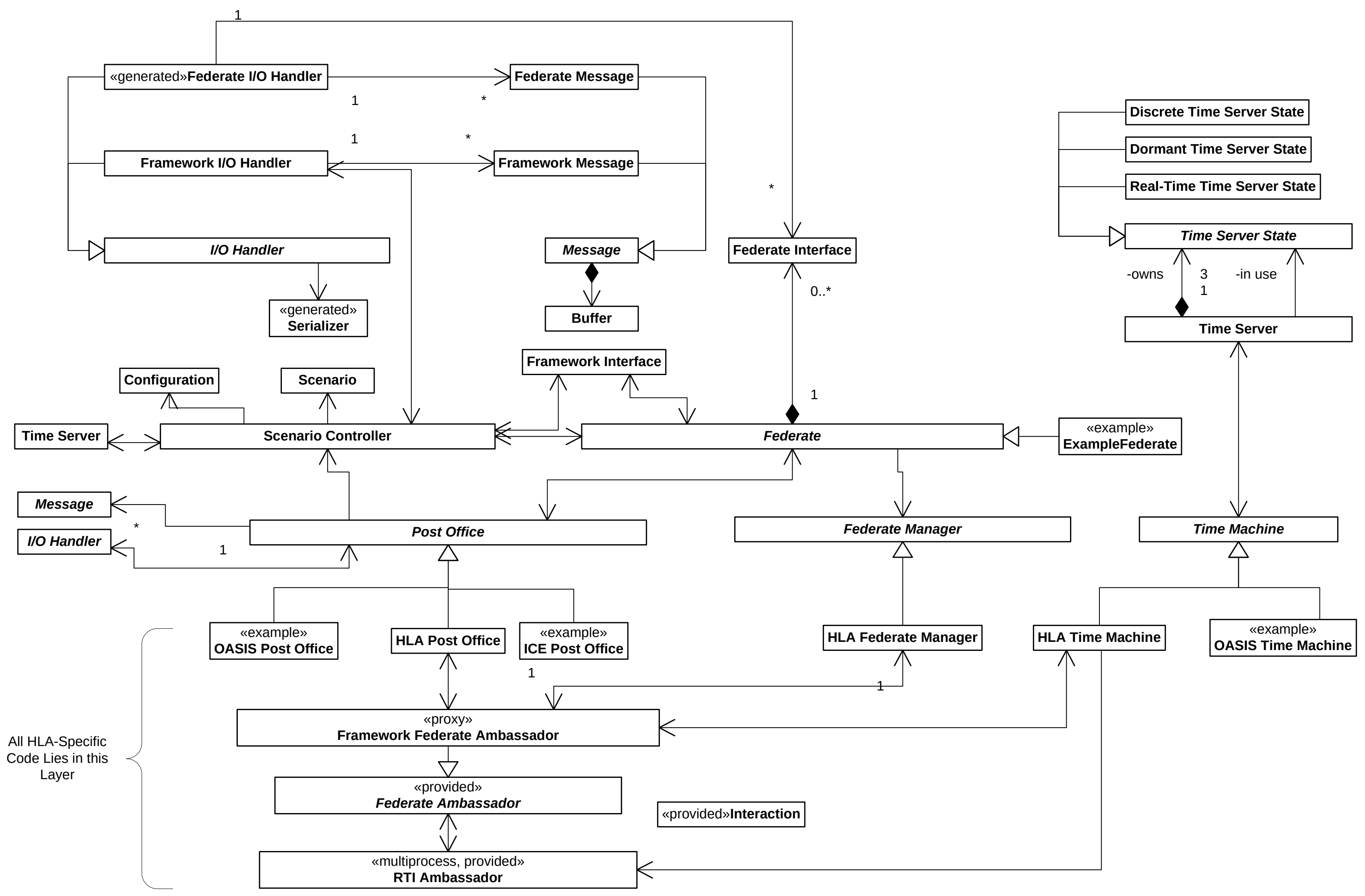

Application
application (incl. legacy middleware)
«static library» C Function Calls
«interpos. library» Software Clock
Federate
Federate I/O Handler
Time Server
HLA RTI
More than one Application process might exist per Federate process
Unmodified components
New framework components
New purchased components

Component	*Description*
application	Application built with legacy CGAI middleware
C Function Calls	Generated static libraries replacing *socket_mgr* and *ntds_common*
Software Clock	Interposition library for time-related POSIX calls; operates in either discrete or scaled real-time mode
Federate I/O Handler	Handles all message I/O between local application(s) and the HLA RTI
Time Server	Initializes Software Clock and provides HLA RTI interface when running in discrete mode
HLA RTI	Purchased component implements IEEE 1516 Federate Interface Specification

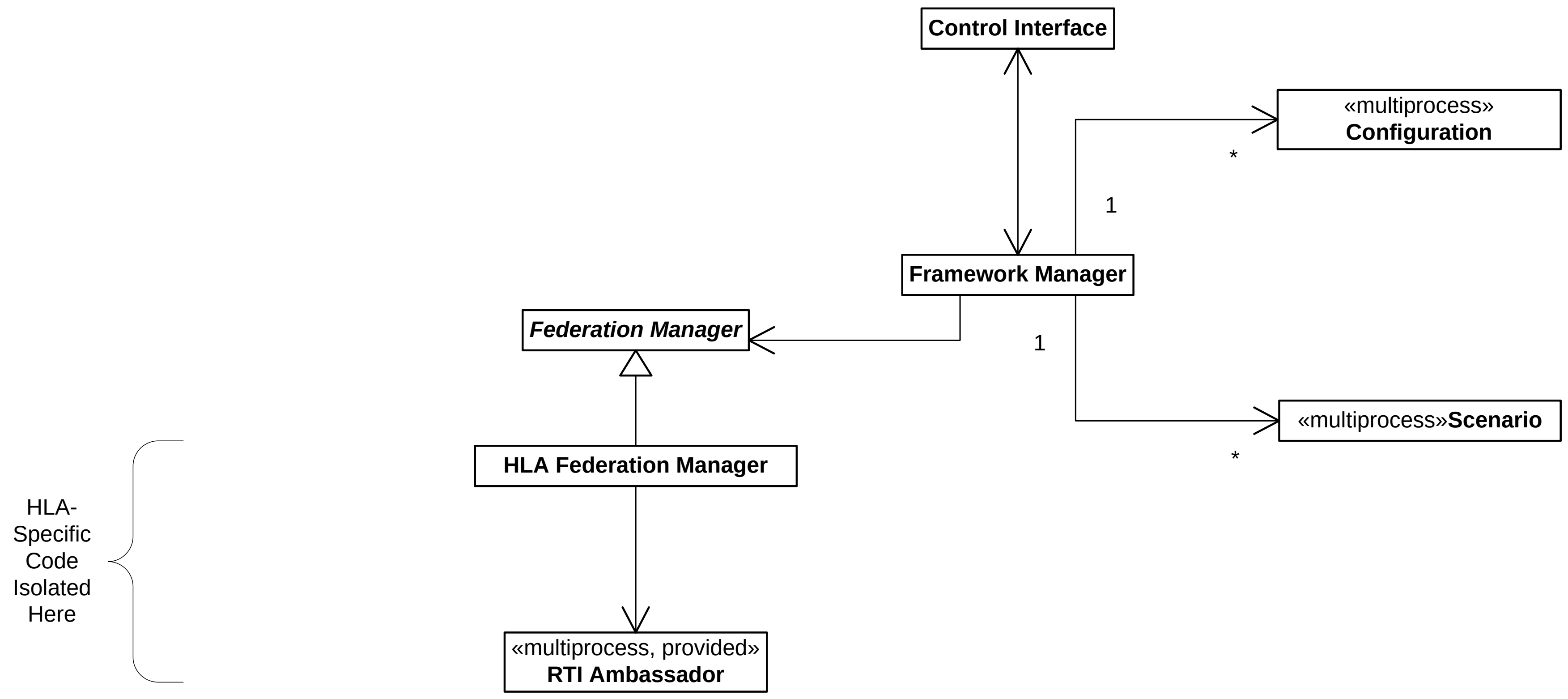
Application Process Classes



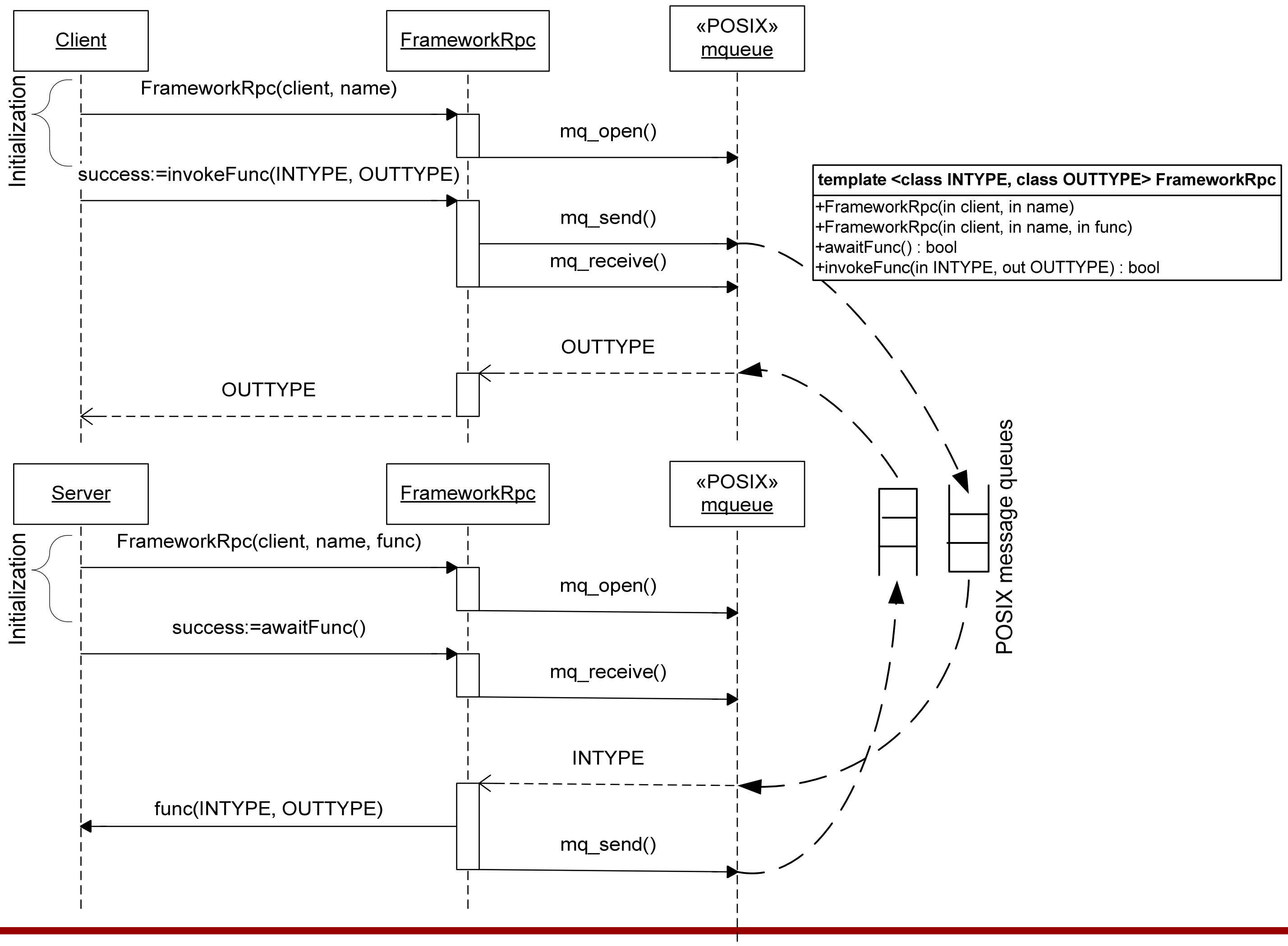
Federate Process Classes



Framework Process Classes



Notional IPC Mechanism



Client
FrameworkRpc
«POSIX» mqueue
Initialization
FrameworkRpc(client, name)
mq_open()
success:=invokeFunc(INTYPE, OUTTYPE)
mq_send()
mq_receive()
OUTTYPE
OUTTYPE
template <class INTYPE, class OUTTYPE> FrameworkRpc
+FrameworkRpc(in client, in name)
+FrameworkRpc(in client, in name, in func)
+awaitFunc() : bool
+invokeFunc(in INTYPE, out OUTTYPE) : bool
Server
FrameworkRpc
«POSIX» mqueue
Initialization
FrameworkRpc(client, name, func)
mq_open()
success:=awaitFunc()
mq_receive()
INTYPE
func(INTYPE, OUTTYPE)
mq_send()
POSIX message queues

Major Functional Design Areas

- The following slides address the significant functional areas of the design:
  - Federate Management
  - Time Management
  - I/O Management
  - Framework Initialization and Scenario Control

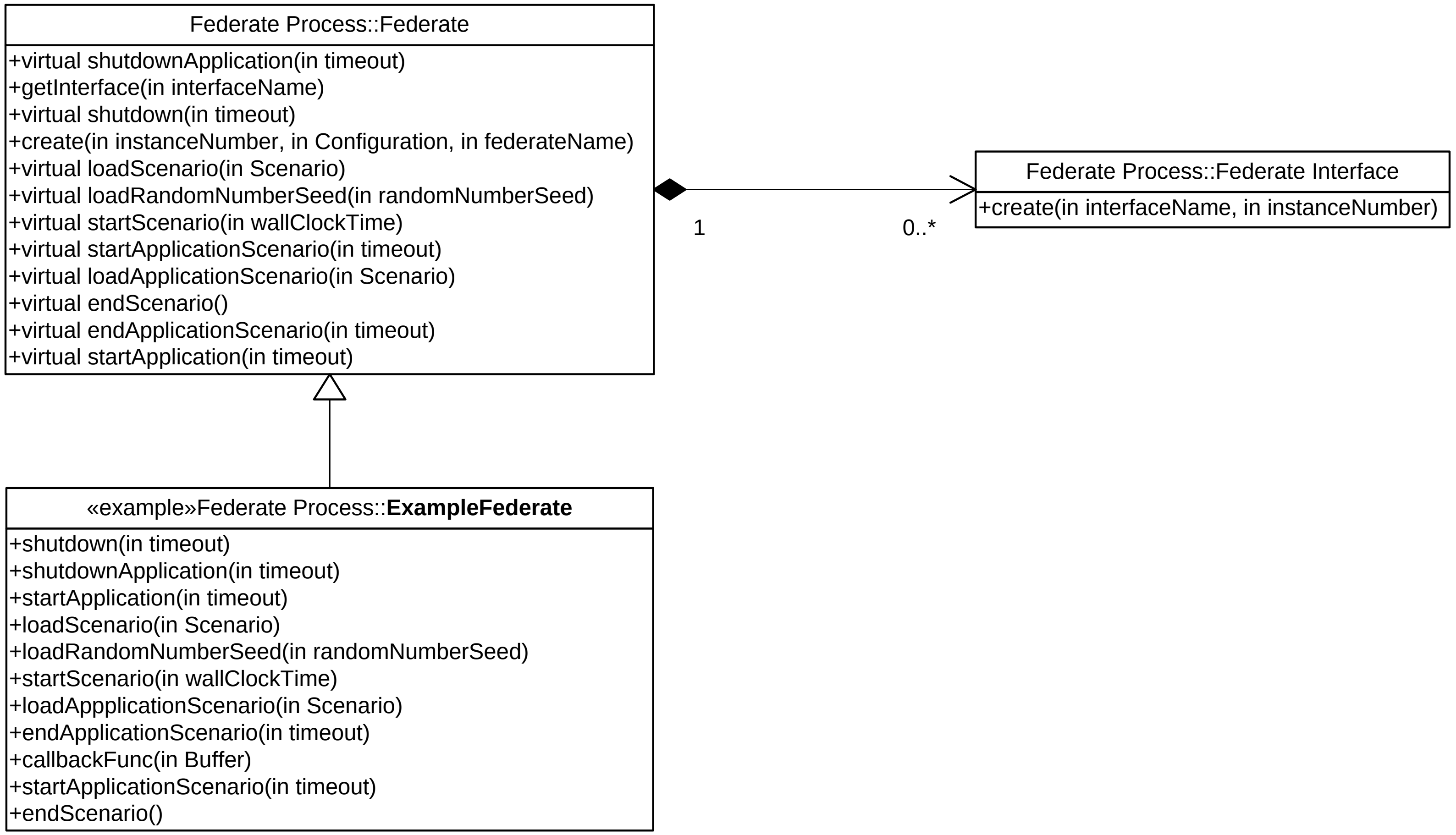
Federate Management

The Johns Hopkins University
APPLIED PHYSICS LABORATORY

Federate Management

- Consists of abstract Federate class along with subclass for each type of Federate. “Example Federate” is used as a representative example in this design.
- Federate & subclasses interact with Time, I/O, Initialization/Scenario Control, thus do not fit in any one functional package.

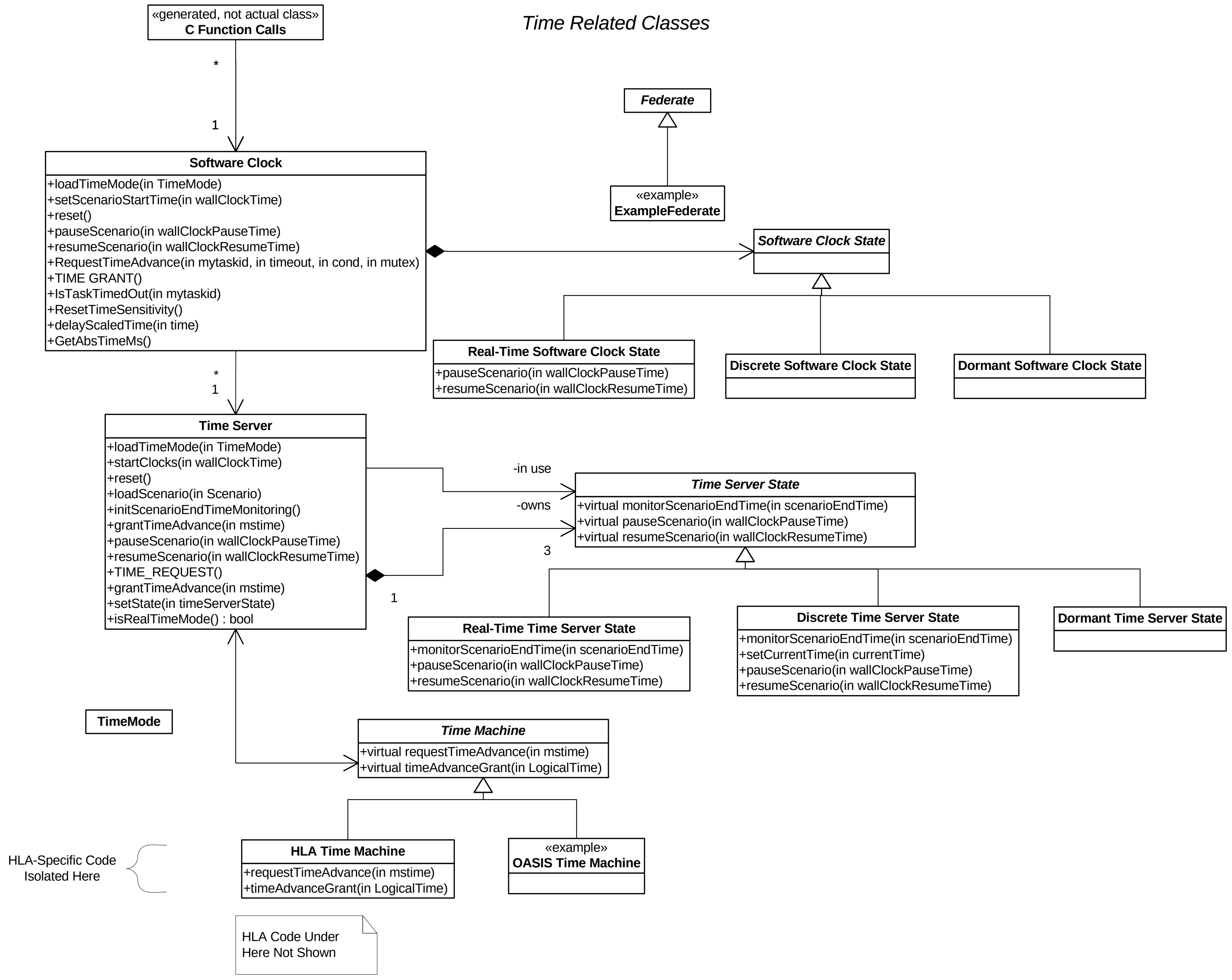
Federate Classes


Federate Process::Federate
+virtual shutdownApplication(in timeout)
+getInterface(in interfaceName)
+virtual shutdown(in timeout)
+create(in instanceNumber, in Configuration, in federateName)
+virtual loadScenario(in Scenario)
+virtual loadRandomNumberSeed(in randomNumberSeed)
+virtual startScenario(in wallClockTime)
+virtual startApplicationScenario(in timeout)
+virtual loadApplicationScenario(in Scenario)
+virtual endScenario()
+virtual endApplicationScenario(in timeout)
+virtual startApplication(in timeout)
1
0..*
Federate Process::Federate Interface
+create(in interfaceName, in instanceNumber)
«example»Federate Process::**ExampleFederate**
+shutdown(in timeout)
+shutdownApplication(in timeout)
+startApplication(in timeout)
+loadScenario(in Scenario)
+loadRandomNumberSeed(in randomNumberSeed)
+startScenario(in wallClockTime)
+loadAppplicationScenario(in Scenario)
+endApplicationScenario(in timeout)
+callbackFunc(in Buffer)
+startApplicationScenario(in timeout)
+endScenario()

Time Management

The Johns Hopkins University
APPLIED PHYSICS LABORATORY

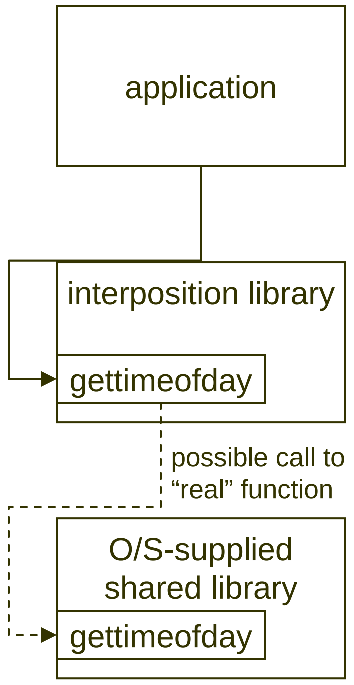
Time Related Classes


«generated, not actual class»
C Function Calls
*
1
Software Clock
+loadTimeMode(in TimeMode)
+setScenarioStartTime(in wallClockTime)
+reset()
+pauseScenario(in wallClockPauseTime)
+resumeScenario(in wallClockResumeTime)
+RequestTimeAdvance(in mytaskid, in timeout, in cond, in mutex)
+TIME GRANT()
+IsTaskTimedOut(in mytaskid)
+ResetTimeSensitivity()
+delayScaledTime(in time)
+GetAbsTimeMs()
Federate
«example»
ExampleFederate
Software Clock State
Real-Time Software Clock State
+pauseScenario(in wallClockPauseTime)
+resumeScenario(in wallClockResumeTime)
Discrete Software Clock State
Dormant Software Clock State
*
1
Time Server
+loadTimeMode(in TimeMode)
+startClocks(in wallClockTime)
+reset()
+loadScenario(in Scenario)
+initScenarioEndTimeMonitoring()
+grantTimeAdvance(in mstime)
+pauseScenario(in wallClockPauseTime)
+resumeScenario(in wallClockResumeTime)
+TIME_REQUEST()
+grantTimeAdvance(in mstime)
+setState(in timeServerState)
+isRealTimeMode() : bool
-in use
-owns
3
1
Time Server State
+virtual monitorScenarioEndTime(in scenarioEndTime)
+virtual pauseScenario(in wallClockPauseTime)
+virtual resumeScenario(in wallClockResumeTime)
Real-Time Time Server State
+monitorScenarioEndTime(in scenarioEndTime)
+pauseScenario(in wallClockPauseTime)
+resumeScenario(in wallClockResumeTime)
Discrete Time Server State
+monitorScenarioEndTime(in scenarioEndTime)
+setCurrentTime(in currentTime)
+pauseScenario(in wallClockPauseTime)
+resumeScenario(in wallClockResumeTime)
Dormant Time Server State
TimeMode
Time Machine
+virtual requestTimeAdvance(in mstime)
+virtual timeAdvanceGrant(in LogicalTime)
HLA-Specific Code Isolated Here
HLA Time Machine
+requestTimeAdvance(in mstime)
+timeAdvanceGrant(in LogicalTime)
«example»
OASIS Time Machine
HLA Code Under Here Not Shown

Library Interposition

- 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

Example: Calling gettimeofday

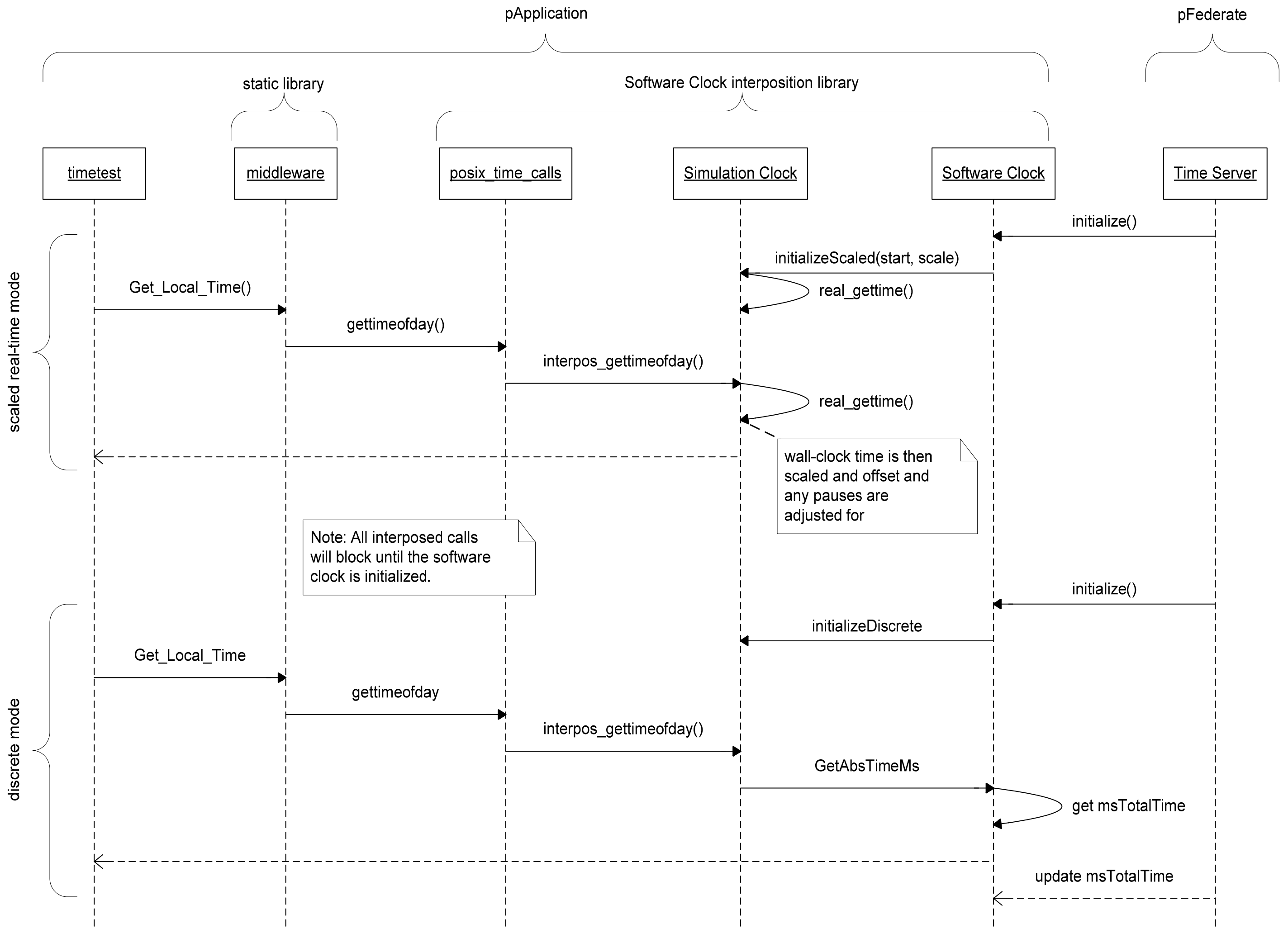


Software Clock Interposition Library (1)

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

    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

Software Clock Interposition Library (2)



Software Clock Interposition Library (3)

- 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

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.

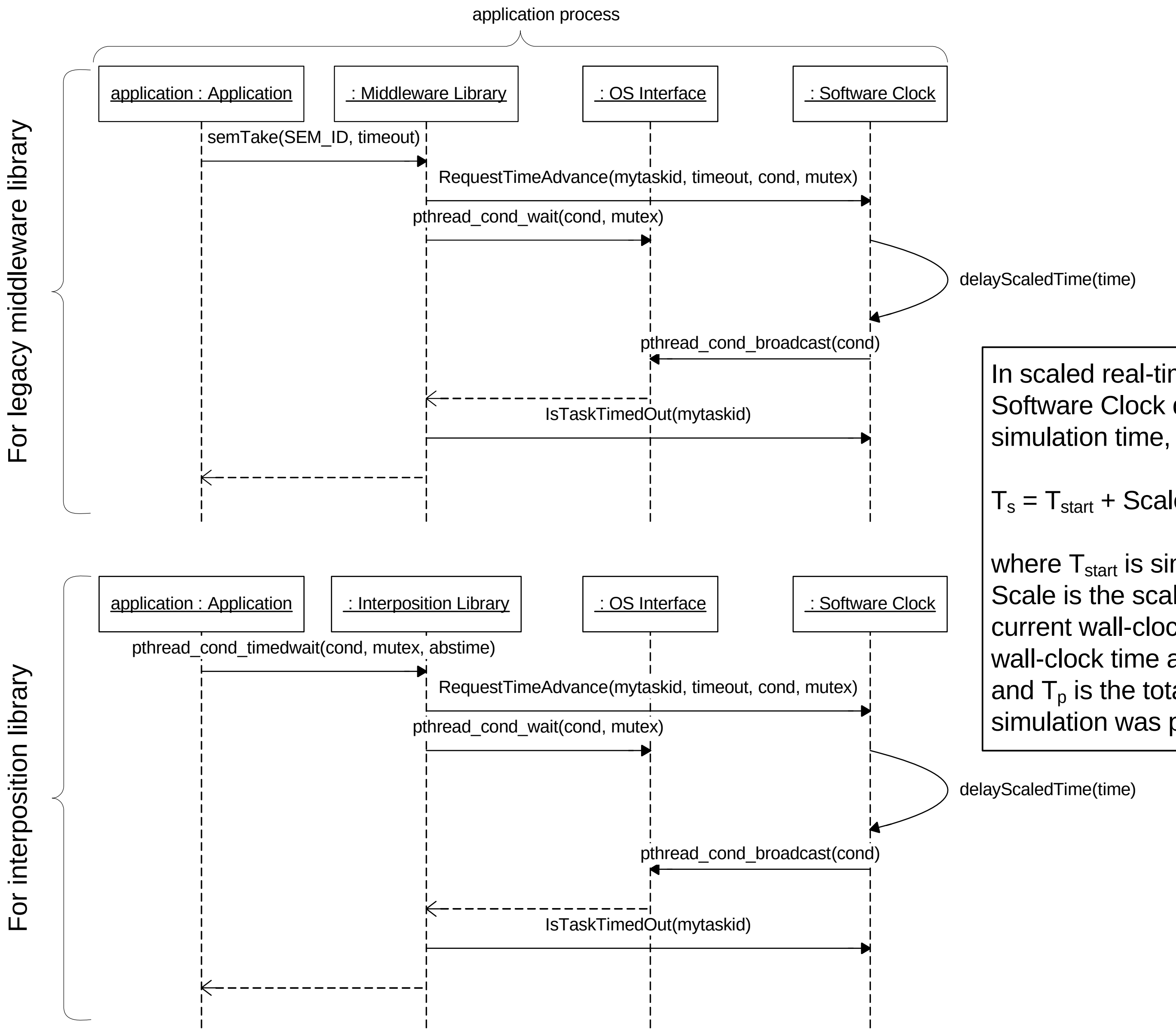
Basic Course of Events

For 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. The following formula relates simulation time, T_s, to the current wall-clock time, T_w, as follows:
   $$T_s = T_{start} + Scale * (T_w – T_{wStart} – T_p)$$
   where T_{start} is 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. 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.

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."


application process
For legacy middleware library
application : Application
: Middleware Library
: OS Interface
: Software Clock
semTake(SEM_ID, timeout)
RequestTimeAdvance(mytaskid, timeout, cond, mutex)
pthread_cond_wait(cond, mutex)
delayScaledTime(time)
pthread_cond_broadcast(cond)
IsTaskTimedOut(mytaskid)
For interposition library
application : Application
: Interposition Library
: OS Interface
: Software Clock
pthread_cond_timedwait(cond, mutex, abstime)
RequestTimeAdvance(mytaskid, timeout, cond, mutex)
pthread_cond_wait(cond, mutex)
delayScaledTime(time)
pthread_cond_broadcast(cond)
IsTaskTimedOut(mytaskid)

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

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

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

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.

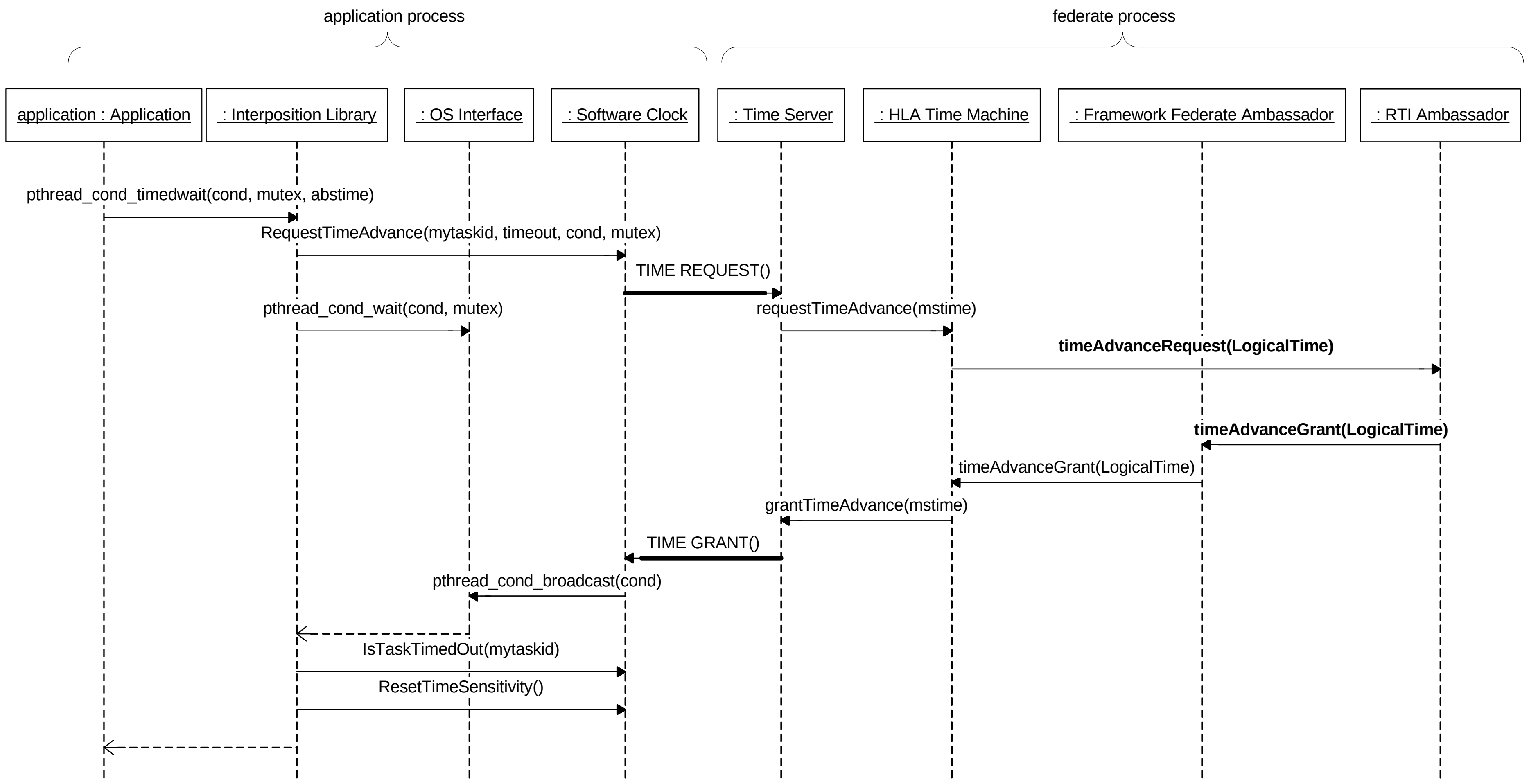
Basic Course of Events

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.

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."


application process
federate process
application : Application
: Interposition Library
: OS Interface
: Software Clock
: Time Server
: HLA Time Machine
: Framework Federate Ambassador
: RTI Ambassador
pthread_cond_timedwait(cond, mutex, abstime)
RequestTimeAdvance(mytaskid, timeout, cond, mutex)
TIME REQUEST()
pthread_cond_wait(cond, mutex)
requestTimeAdvance(mstime)
timeAdvanceRequest(LogicalTime)
timeAdvanceGrant(LogicalTime)
timeAdvanceGrant(LogicalTime)
grantTimeAdvance(mstime)
TIME GRANT()
pthread_cond_broadcast(cond)
IsTaskTimedOut(mytaskid)
ResetTimeSensitivity()

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

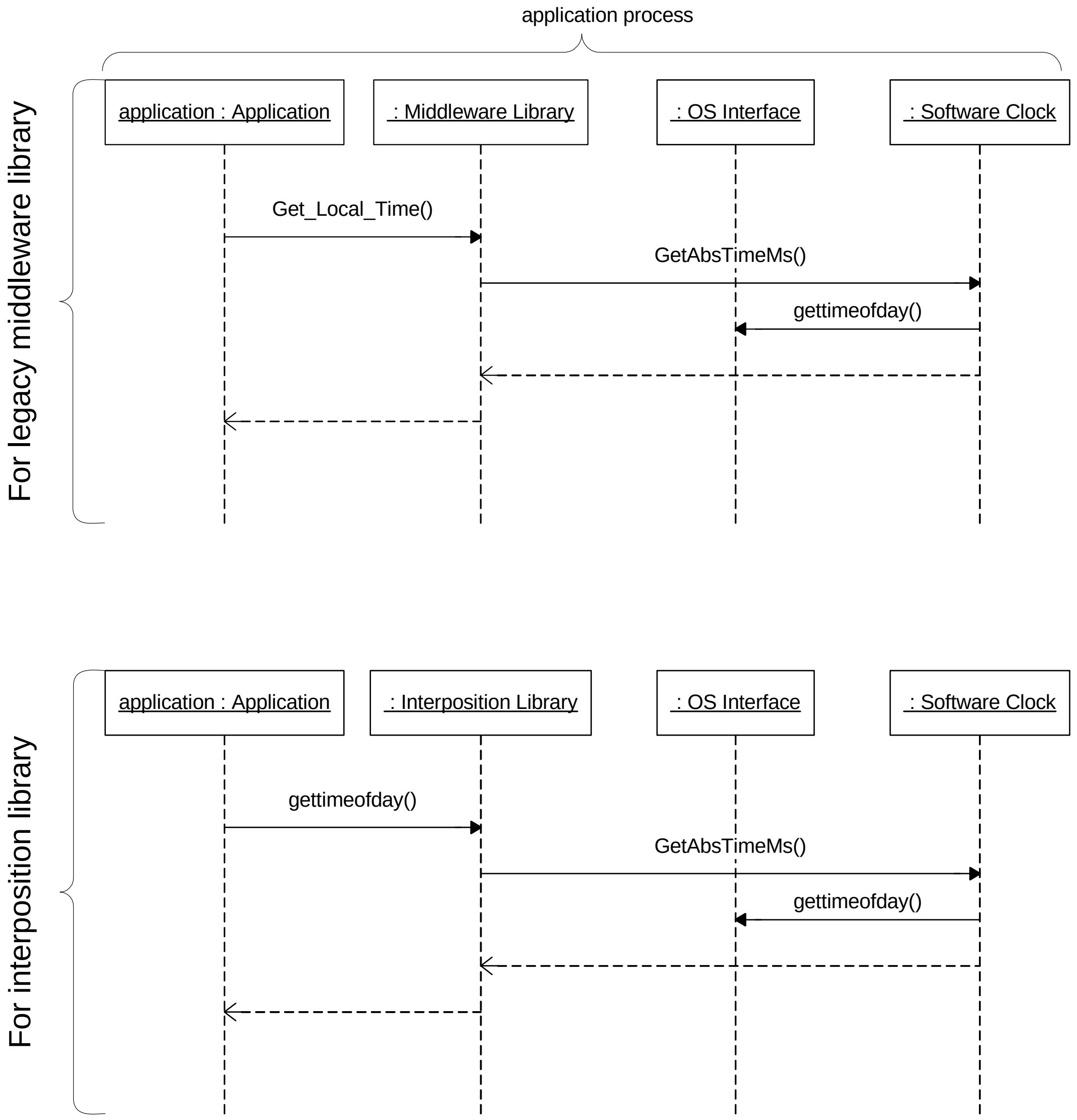
Application makes a call to obtain the current (simulation) time to either a legacy middleware layer or a POSIX system call.

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

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 <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).”



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

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

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

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:

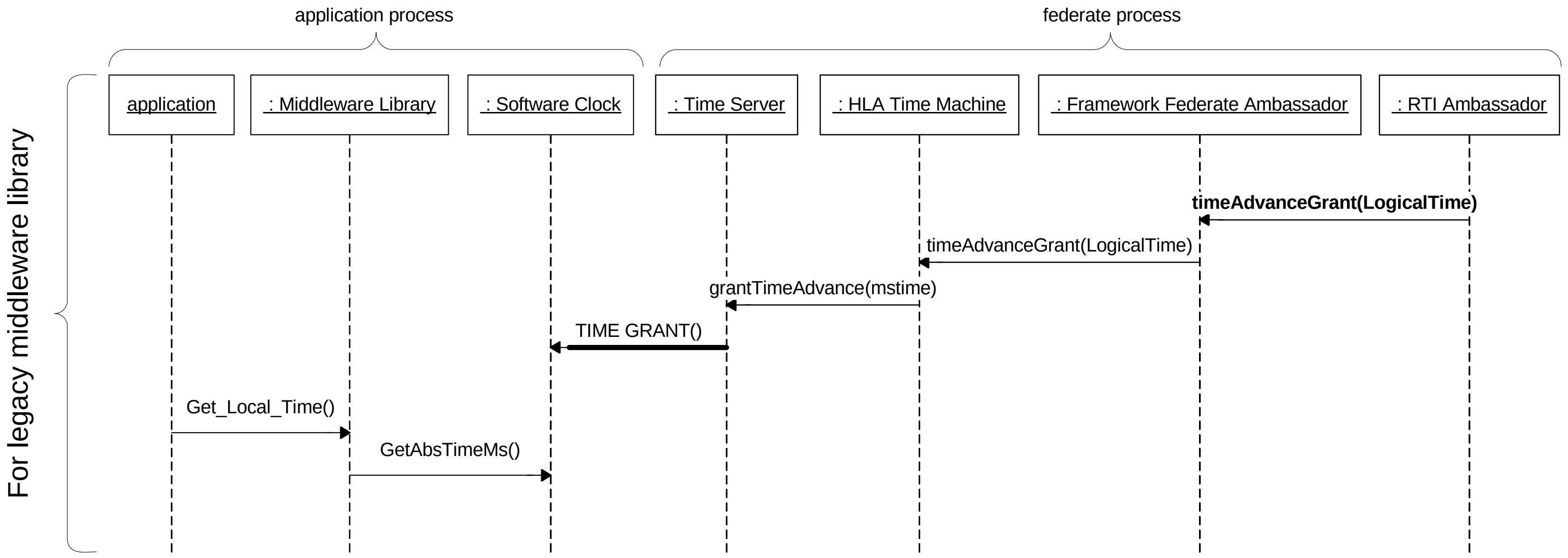
1. The RTI Ambassador advances the time by calling the timeAdvanceGrant method in the Framework Federate Ambassador.
2. The time grant is passed to the timeAdvanceGrant method in the HLA Time Machine.
3. The time grant is passed to the grantTimeAdvance method in the Time Server.
4. The Time Server sends 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.

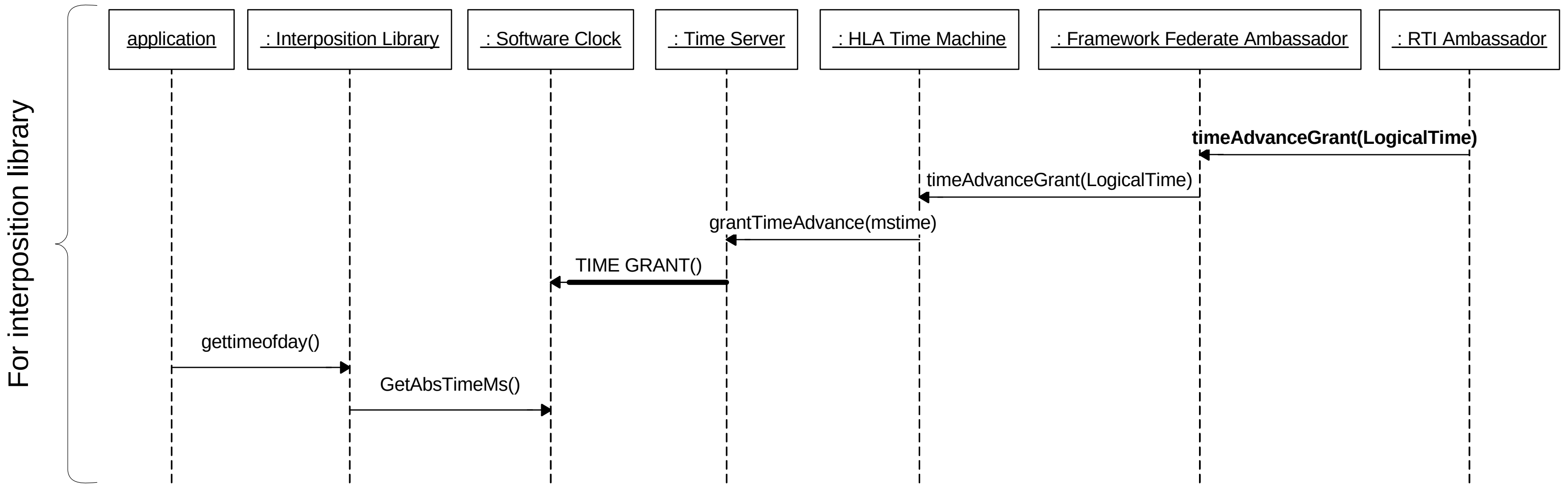
The application actually gets the current value of the simulation time via these steps:

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.

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."





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

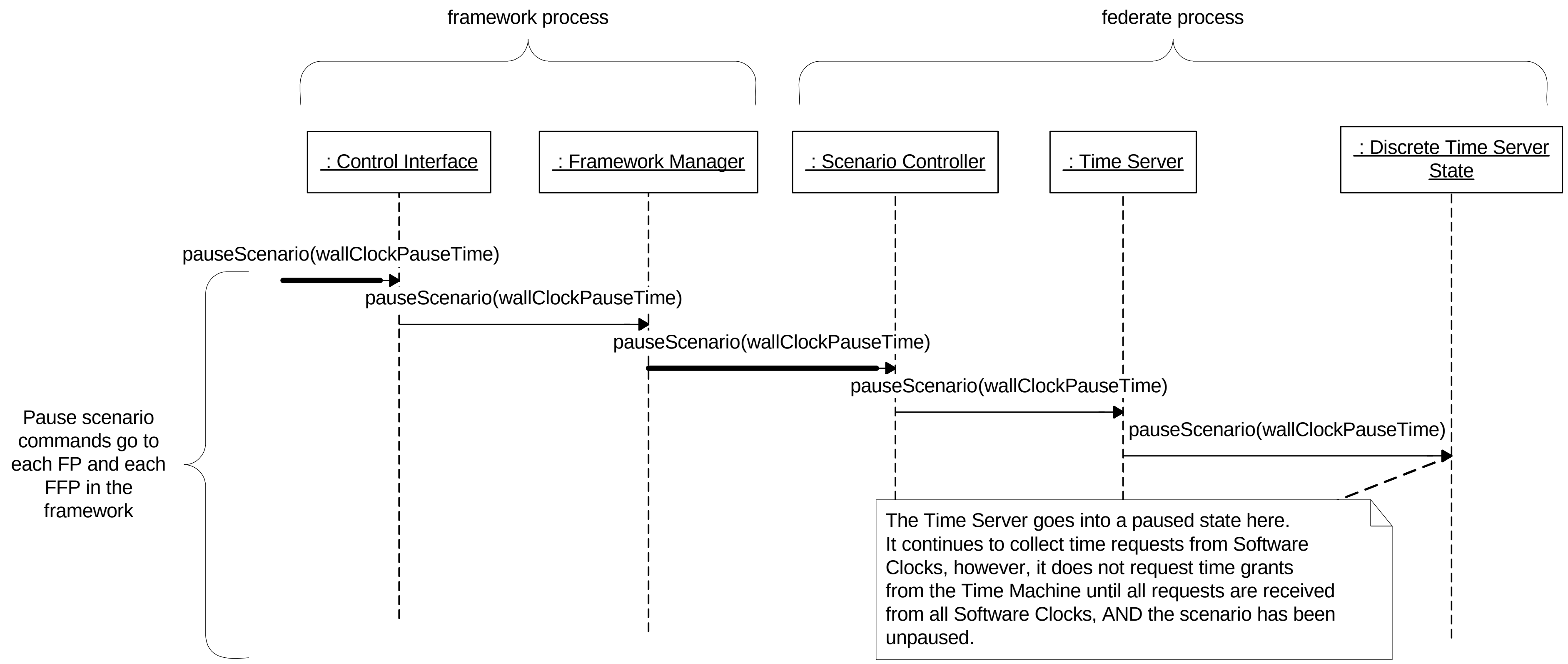
Sub Use Case a: Scenario is running in discrete time mode.

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.

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.


framework process
federate process
: Control Interface
: Framework Manager
: Scenario Controller
: Time Server
: Discrete Time Server State
pauseScenario(wallClockPauseTime)
pauseScenario(wallClockPauseTime)
pauseScenario(wallClockPauseTime)
pauseScenario(wallClockPauseTime)
pauseScenario(wallClockPauseTime)
Pause scenario commands go to each FP and each FFP in the framework

The Time Server goes into a paused state here.
It continues to collect time requests from Software Clocks, however, it does not request time grants from the Time Machine until all requests are received from all Software Clocks, AND the scenario has been unpaused.

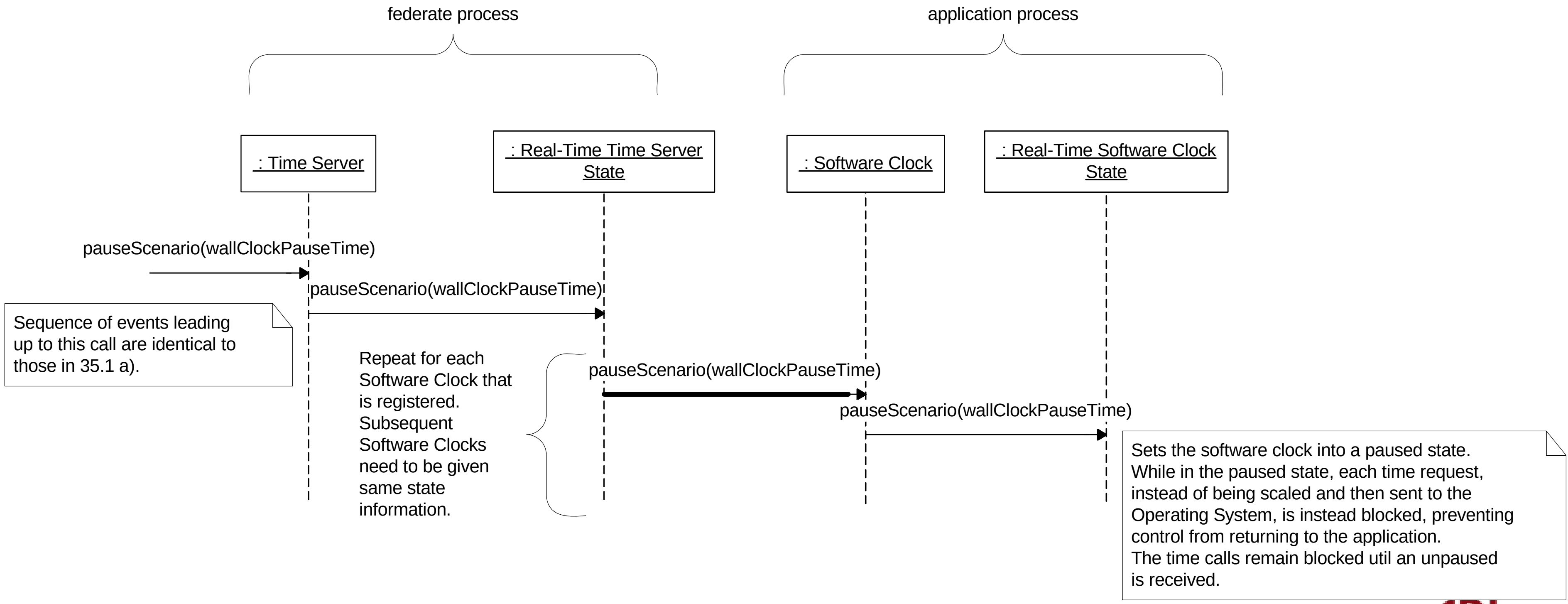
Sub Use Case b: Scenario is running in real-time mode.

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.

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.


federate process
application process
: Time Server
: Real-Time Time Server State
: Software Clock
: Real-Time Software Clock State
pauseScenario(wallClockPauseTime)
pauseScenario(wallClockPauseTime)
Sequence of events leading up to this call are identical to those in 35.1 a).
Repeat for each Software Clock that is registered. Subsequent Software Clocks need to be given same state information.
pauseScenario(wallClockPauseTime)
pauseScenario(wallClockPauseTime)
Sets the software clock into a paused state. While in the paused state, each time request, instead of being scaled and then sent to the Operating System, is instead blocked, preventing control from returning to the application. The time calls remain blocked util an unpaused is received.

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

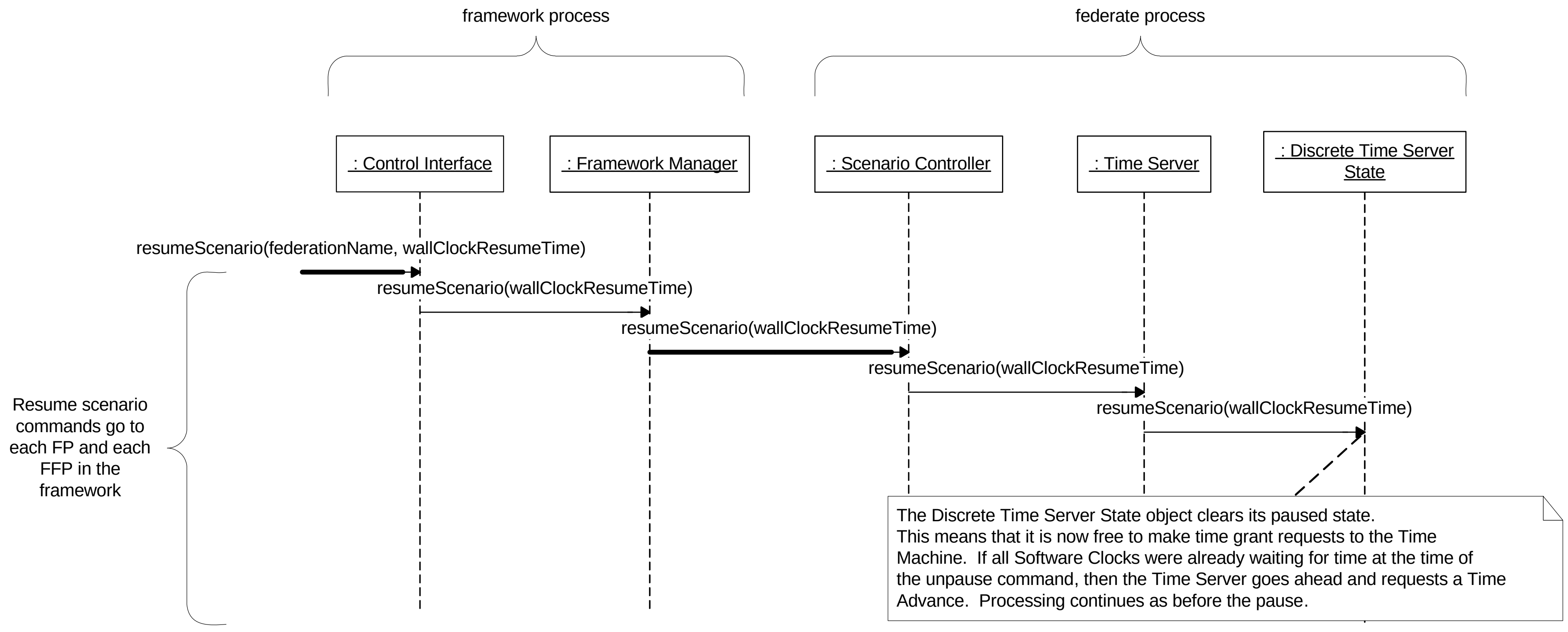
Sub Use Case a: Scenario is running in discrete time mode.

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.

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.



framework process
federate process
: Control Interface
: Framework Manager
: Scenario Controller
: Time Server
: Discrete Time Server State
resumeScenario(federationName, wallClockResumeTime)
resumeScenario(wallClockResumeTime)
resumeScenario(wallClockResumeTime)
resumeScenario(wallClockResumeTime)
resumeScenario(wallClockResumeTime)
Resume scenario commands go to each FP and each FFP in the framework

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.

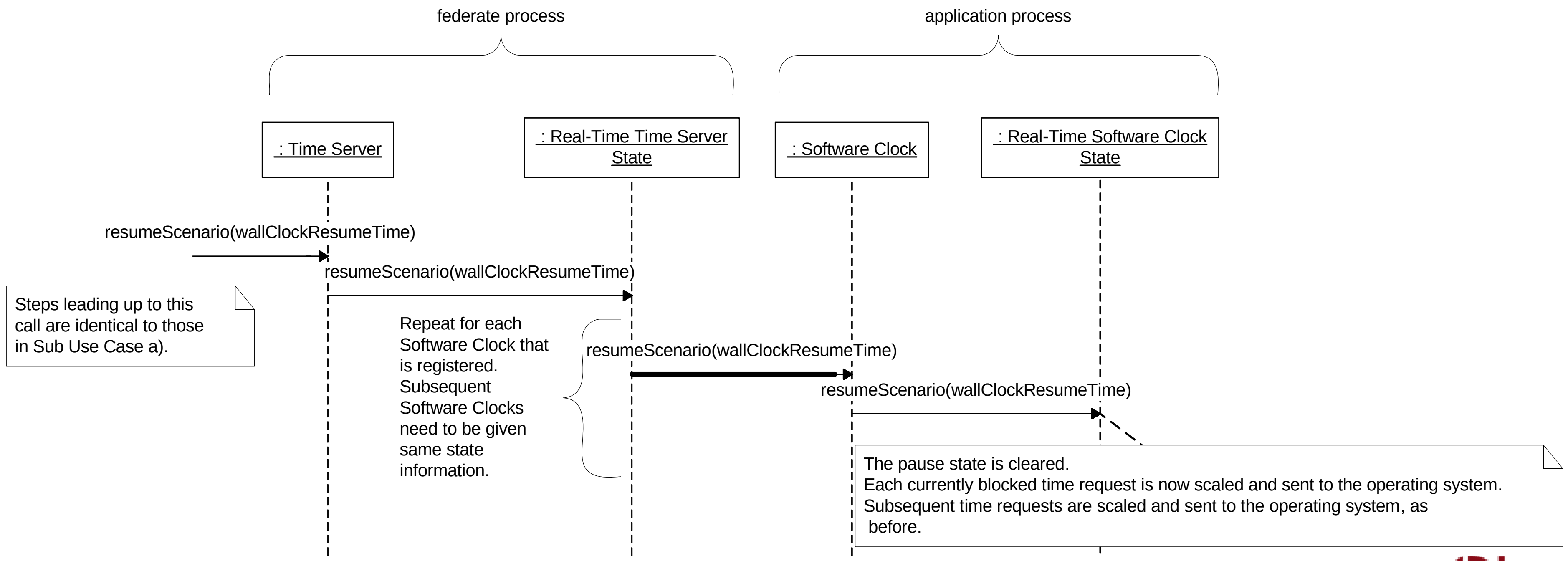
Sub Use Case b: Scenario is running in real-time mode.

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.

Sequence Diagram for Use Case 35.2

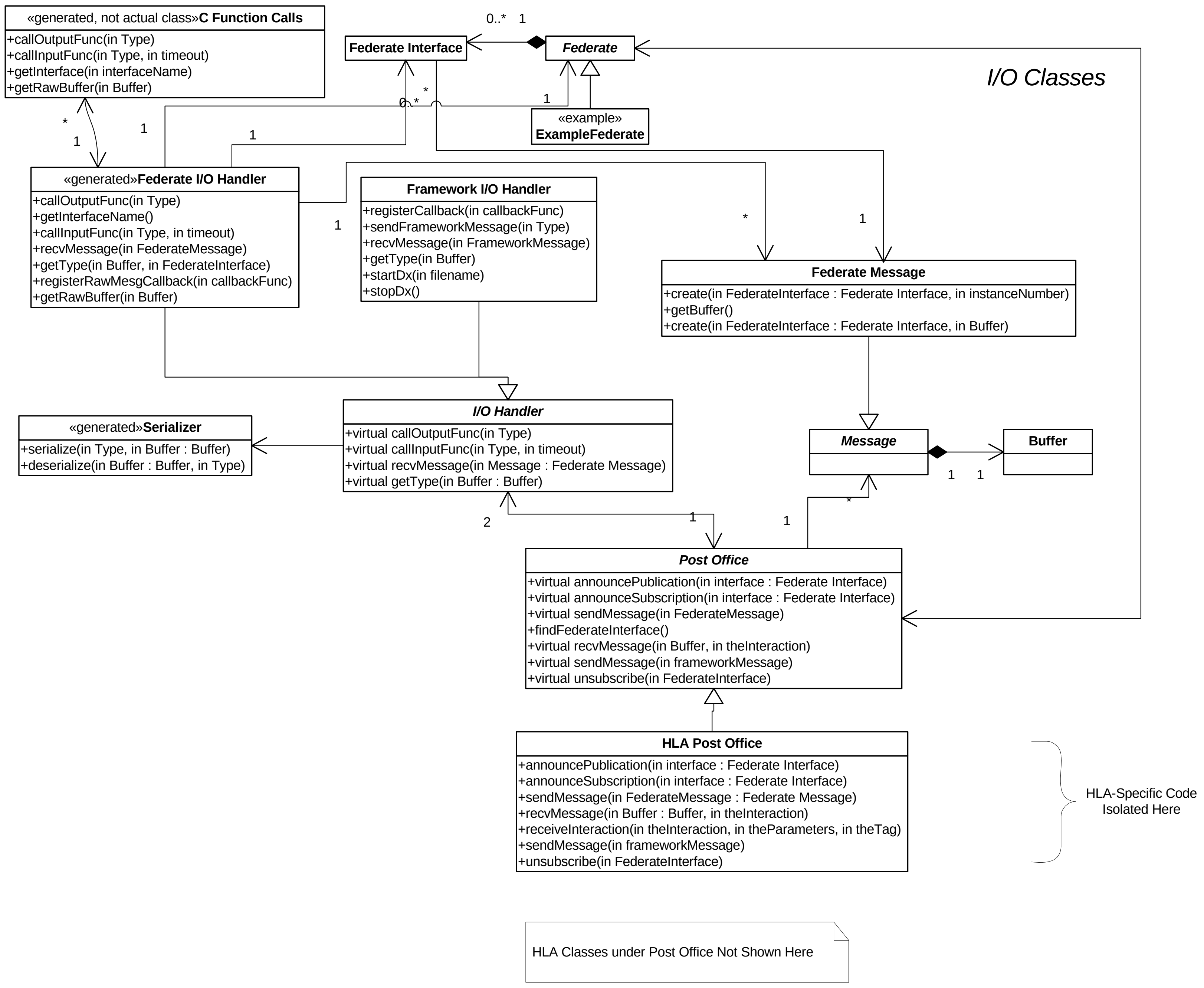
A “resume” command is received from the user interface, resulting in a paused scenario executing again where it left off.

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


federate process
application process
: Time Server
: Real-Time Time Server State
: Software Clock
: Real-Time Software Clock State
resumeScenario(wallClockResumeTime)
resumeScenario(wallClockResumeTime)
Steps leading up to this call are identical to those in Sub Use Case a).
Repeat for each Software Clock that is registered. Subsequent Software Clocks need to be given same state information.
resumeScenario(wallClockResumeTime)
resumeScenario(wallClockResumeTime)
The pause state is cleared.
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.

I/O Management

The Johns Hopkins University
APPLIED PHYSICS LABORATORY


I/O Classes
«generated, not actual class»C Function Calls
+callOutputFunc(in Type)
+callInputFunc(in Type, in timeout)
+getInterface(in interfaceName)
+getRawBuffer(in Buffer)
Federate Interface
Federate
0..*
1
«example»
ExampleFederate
«generated»Federate I/O Handler
+callOutputFunc(in Type)
+getInterfaceName()
+callInputFunc(in Type, in timeout)
+recvMessage(in FederateMessage)
+getType(in Buffer, in FederateInterface)
+registerRawMesgCallback(in callbackFunc)
+getRawBuffer(in Buffer)
Framework I/O Handler
+registerCallback(in callbackFunc)
+sendFrameworkMessage(in Type)
+recvMessage(in FrameworkMessage)
+getType(in Buffer)
+startDx(in filename)
+stopDx()
Federate Message
+create(in FederateInterface : Federate Interface, in instanceNumber)
+getBuffer()
+create(in FederateInterface : Federate Interface, in Buffer)
I/O Handler
+virtual callOutputFunc(in Type)
+virtual callInputFunc(in Type, in timeout)
+virtual recvMessage(in Message : Federate Message)
+virtual getType(in Buffer : Buffer)
«generated»Serializer
+serialize(in Type, in Buffer : Buffer)
+deserialize(in Buffer : Buffer, in Type)
Message
Buffer
Post Office
+virtual announcePublication(in interface : Federate Interface)
+virtual announceSubscription(in interface : Federate Interface)
+virtual sendMessage(in FederateMessage)
+findFederateInterface()
+virtual recvMessage(in Buffer, in theInteraction)
+virtual sendMessage(in frameworkMessage)
+virtual unsubscribe(in FederateInterface)
HLA Post Office
+announcePublication(in interface : Federate Interface)
+announceSubscription(in interface : Federate Interface)
+sendMessage(in FederateMessage : Federate Message)
+recvMessage(in Buffer : Buffer, in theInteraction)
+receiveInteraction(in theInteraction, in theParameters, in theTag)
+sendMessage(in frameworkMessage)
+unsubscribe(in FederateInterface)
HLA-Specific Code Isolated Here
HLA Classes under Post Office Not Shown Here

“C Function Calls” Static Library (1)

- 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

“C Function Calls” Static Library (2)

- 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_Que
  - Send_Socket_Msg
  - Recv_Socket_Msg

Use Case 42.1

Summary

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

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.

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.

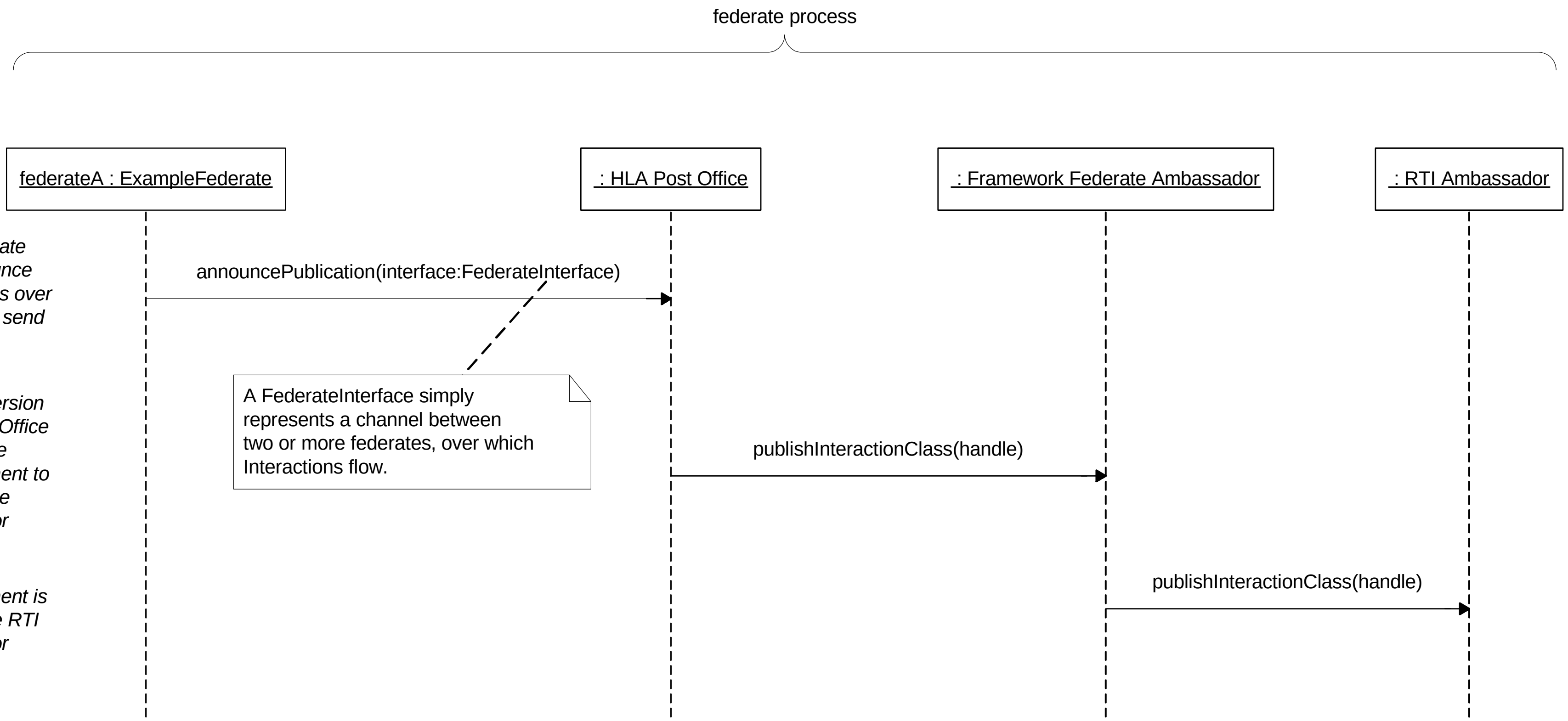
Sequence Diagram for Use Case 42.1

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

Every federate must announce all interfaces over which it will send messages

The HLA version of the Post Office forwards the announcement to the Federate Ambassador

Finally, the announcement is made to the RTI Ambassador


federate process
federateA : ExampleFederate
: HLA Post Office
: Framework Federate Ambassador
: RTI Ambassador
announcePublication(interface:FederateInterface)
A FederateInterface simply represents a channel between two or more federates, over which Interactions flow.
publishInteractionClass(handle)
publishInteractionClass(handle)

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.

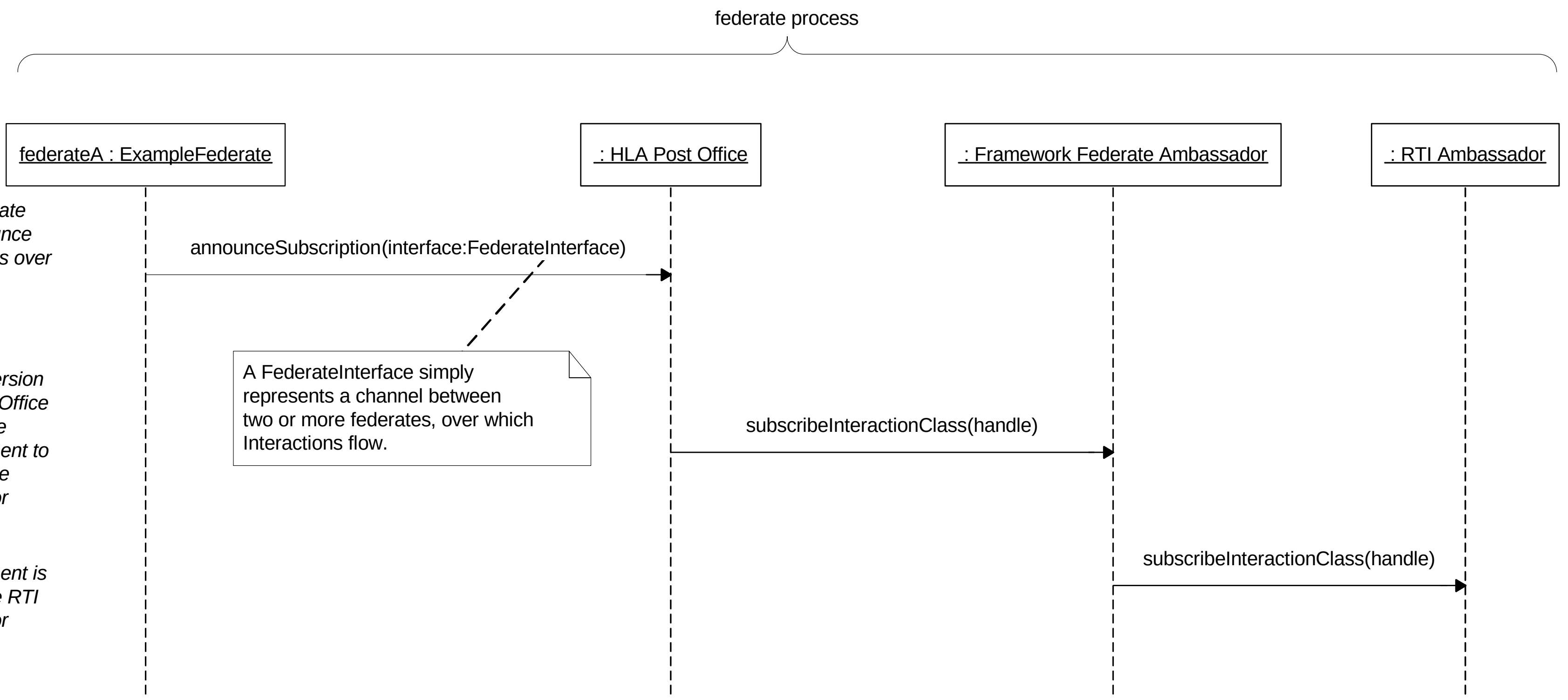
Sequence Diagram for Use Case 42.2

“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.”

Every federate must announce all interfaces over which it will receive messages

The HLA version of the Post Office forwards the announcement to the Federate Ambassador

Finally, the announcement is made to the RTI Ambassador


federate process
federateA : ExampleFederate
: HLA Post Office
: Framework Federate Ambassador
: RTI Ambassador
announceSubscription(interface:FederateInterface)
subscribeInteractionClass(handle)
subscribeInteractionClass(handle)

A FederateInterface simply represents a channel between two or more federates, over which Interactions flow.

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 Federal 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.

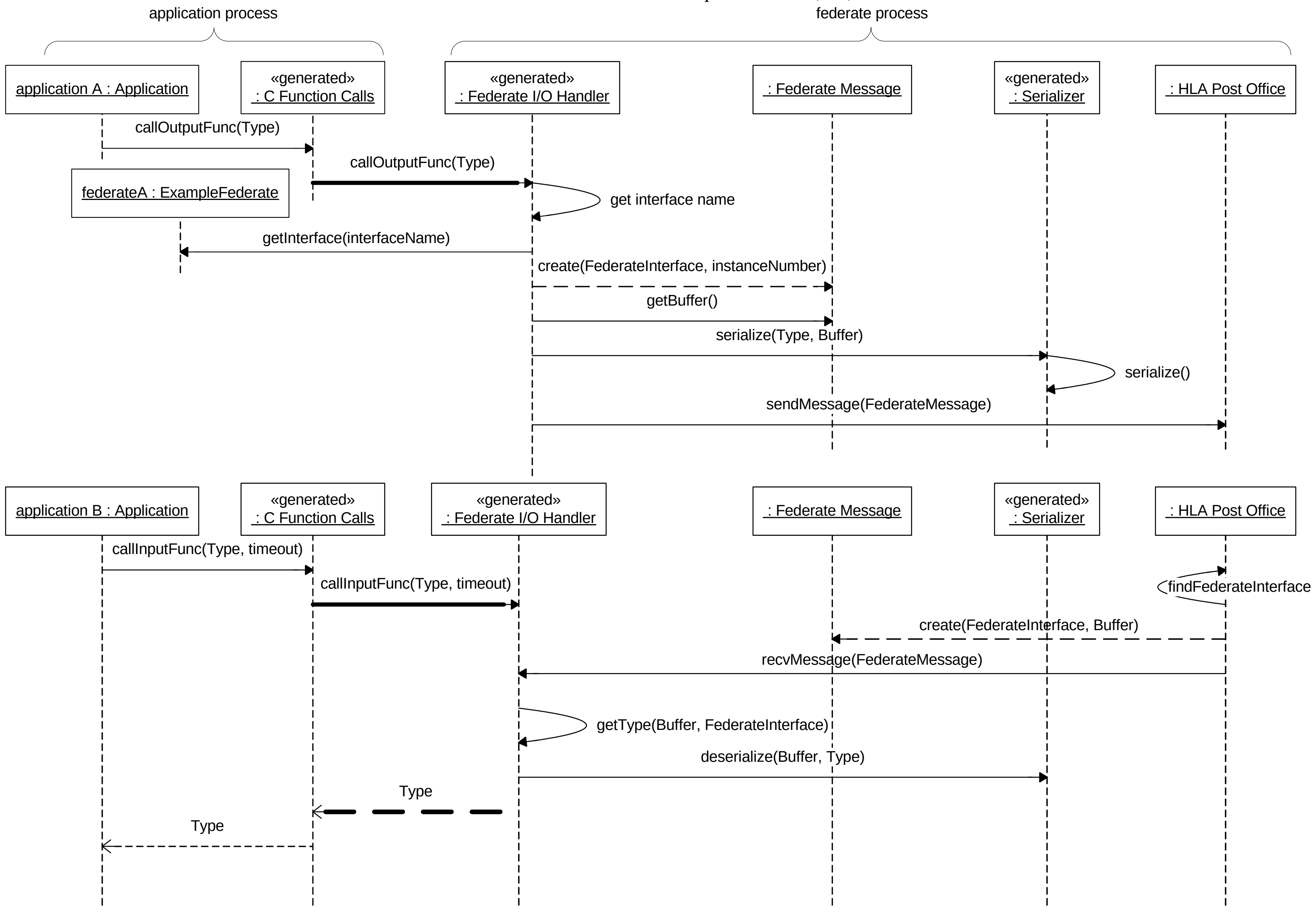
3. Now that the message has been serialized into the Federate Message, this message is passed to the HLA Post Office using its sendMessage method.
4. The HLA Post Office first determines the mode from the Scenario Controller using its isRealTimeMode method. Then,
   a. Sub Use Case A (Real-Time Mode): 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.

Sequence Diagram for Use Case 42.3, Part 1

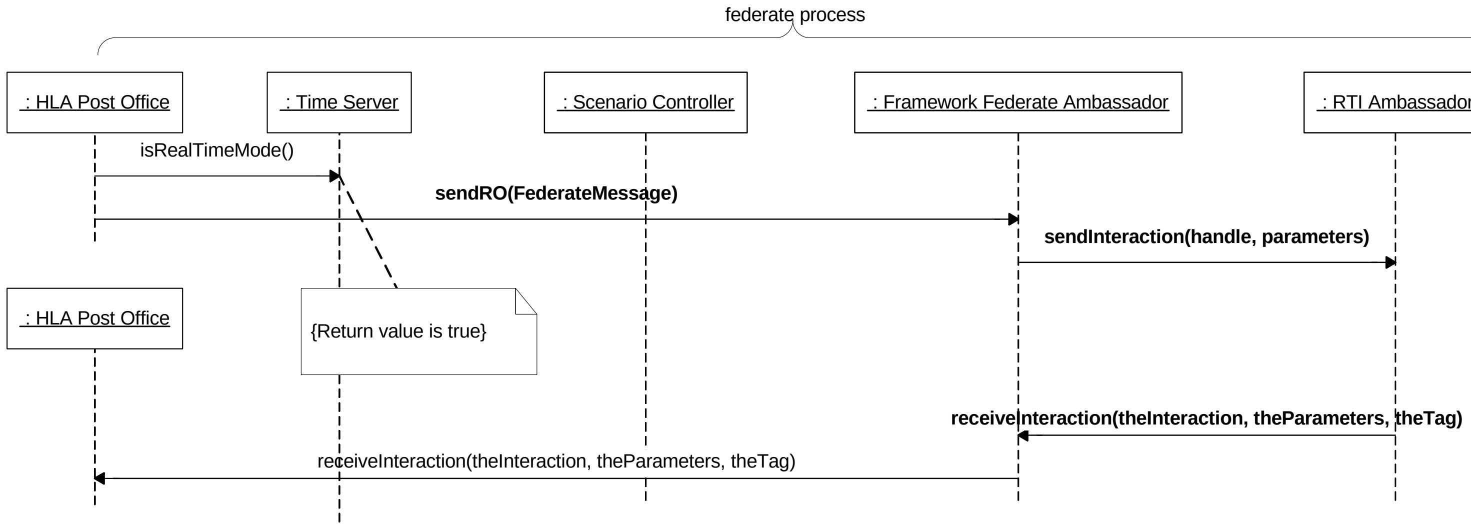
"Federate sends a message. The federates who have subscribed to that message from that federate receive the message. No other federates in the federation, and no federates outside of the federation, receive the message. Note: message sending mechanism may be different depending on real-time or discrete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


application process
federate process
application A : Application
«generated» : C Function Calls
«generated» : Federate I/O Handler
: Federate Message
«generated» : Serializer
: HLA Post Office
callOutputFunc(Type)
callOutputFunc(Type)
federateA : ExampleFederate
get interface name
getInterface(interfaceName)
create(FederateInterface, instanceNumber)
getBuffer()
serialize(Type, Buffer)
serialize()
sendMessage(FederateMessage)
application B : Application
«generated» : C Function Calls
«generated» : Federate I/O Handler
: Federate Message
«generated» : Serializer
: HLA Post Office
callInputFunc(Type, timeout)
callInputFunc(Type, timeout)
findFederateInterface
create(FederateInterface, Buffer)
recvMessage(FederateMessage)
getType(Buffer, FederateInterface)
deserialize(Buffer, Type)
Type
Type

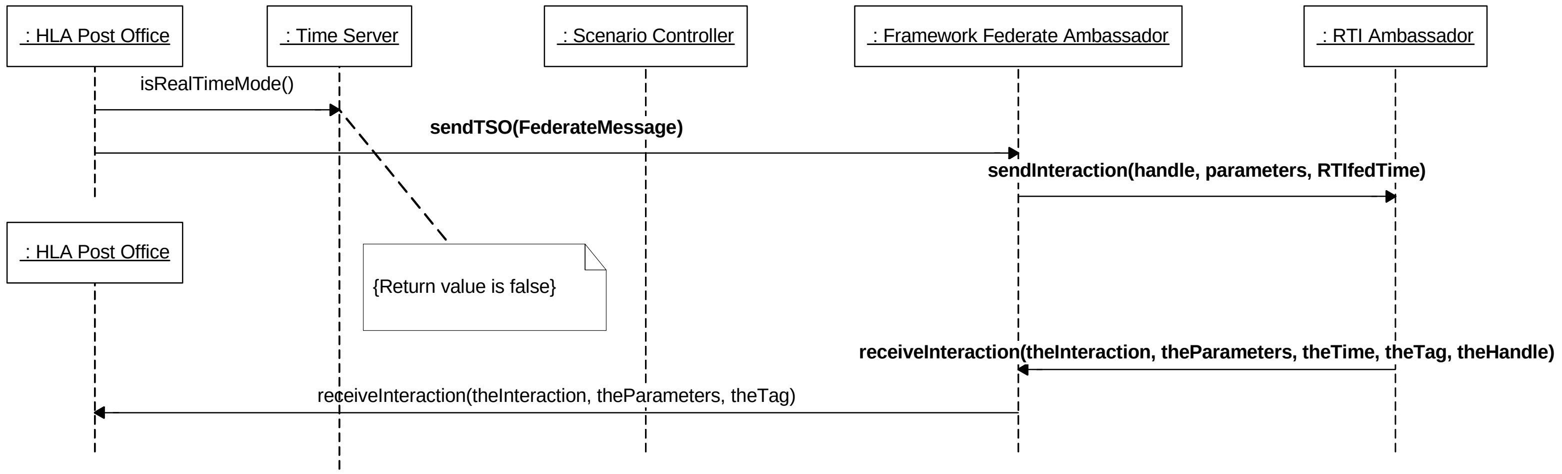
Sequence Diagram for Use Case 42.3, Part 2

"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


federate process
: HLA Post Office
: Time Server
: Scenario Controller
: Framework Federate Ambassador
: RTI Ambassador
isRealTimeMode()
sendRO(FederateMessage)
sendInteraction(handle, parameters)
: HLA Post Office
{Return value is true}
receiveInteraction(theInteraction, theParameters, theTag)
receiveInteraction(theInteraction, theParameters, theTag)

Sub Use Case B: Discrete Mode


: HLA Post Office
: Time Server
: Scenario Controller
: Framework Federate Ambassador
: RTI Ambassador
isRealTimeMode()
sendTSO(FederateMessage)
sendInteraction(handle, parameters, RTIfedTime)
: HLA Post Office
{Return value is false}
receiveInteraction(theInteraction, theParameters, theTime, theTag, theHandle)
receiveInteraction(theInteraction, theParameters, theTag)

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

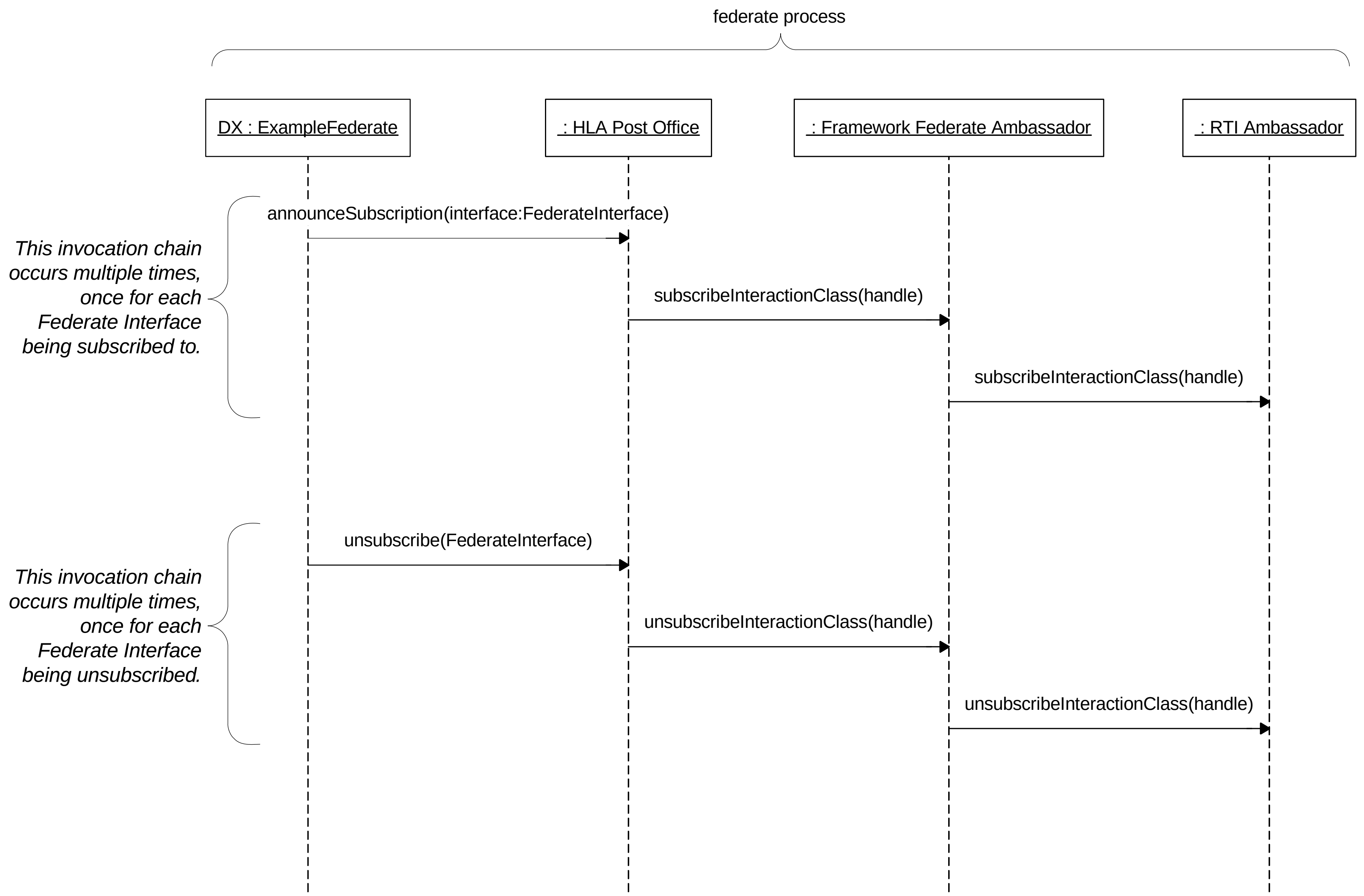
1. The DX federate must invoke the announceSubscription method in the Post Office for all Federate Interfaces it has been configured with (at application capture time).
2. The HLA version of the Post Office forwards the announcement to the Federate Ambassador.
3. Finally, the announcement is made to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

Sequence Diagram for Use Case 64.1

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



federate process
DX : ExampleFederate
: HLA Post Office
: Framework Federate Ambassador
: RTI Ambassador
announceSubscription(interface:FederateInterface)
subscribeInteractionClass(handle)
subscribeInteractionClass(handle)
This invocation chain occurs multiple times, once for each Federate Interface being subscribed to.
unsubscribe(FederateInterface)
unsubscribeInteractionClass(handle)
unsubscribeInteractionClass(handle)
This invocation chain occurs multiple times, once for each Federate Interface being unsubscribed.

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

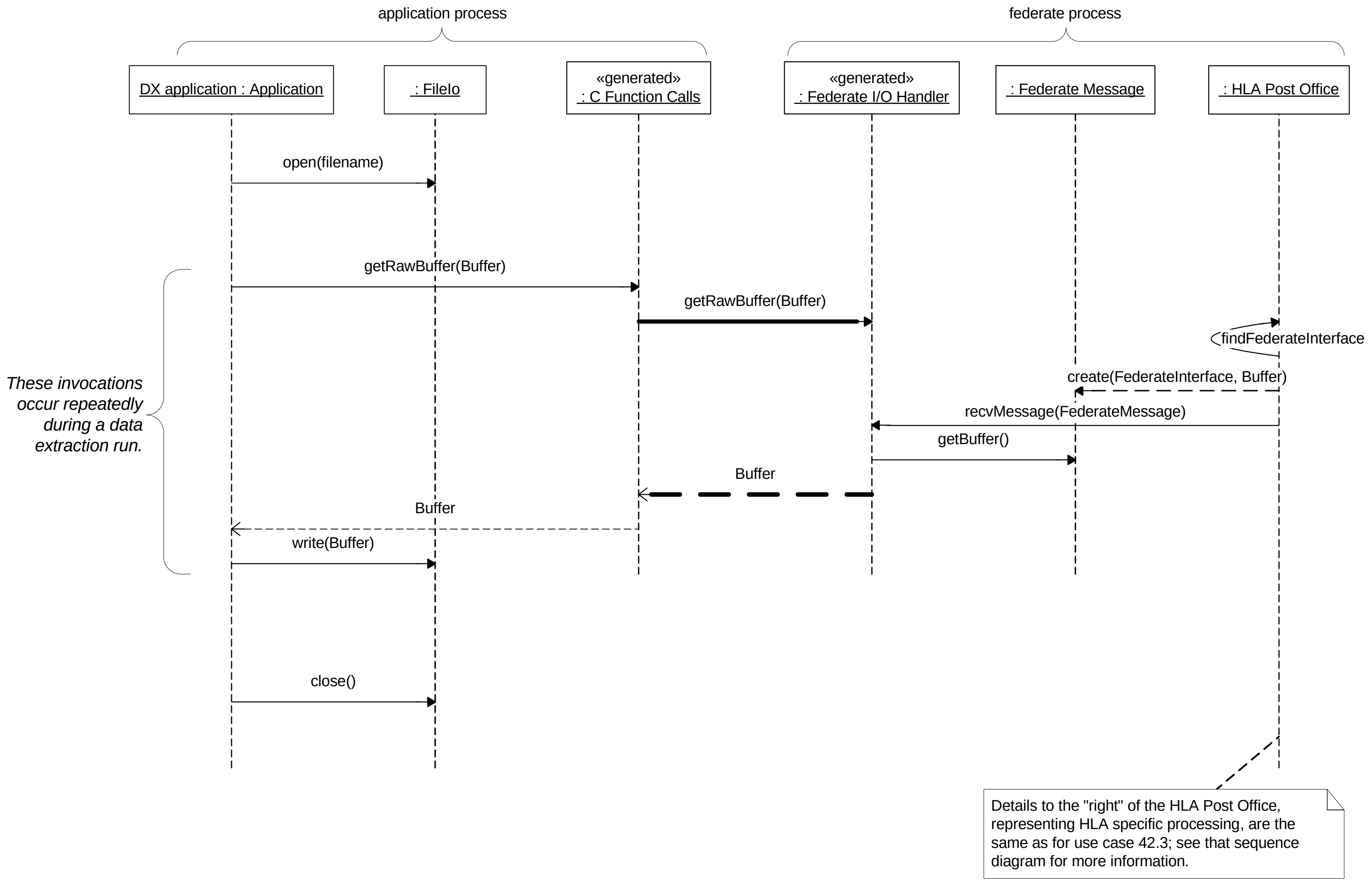
A unique filename for the extracted messages must first be obtained (see Use Case 66.1).

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.

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)."


application process
federate process
DX application : Application
: FileIo
«generated» : C Function Calls
«generated» : Federate I/O Handler
: Federate Message
: HLA Post Office
open(filename)
getRawBuffer(Buffer)
getRawBuffer(Buffer)
findFederateInterface
create(FederateInterface, Buffer)
recvMessage(FederateMessage)
getBuffer()
Buffer
Buffer
write(Buffer)
close()
These invocations occur repeatedly during a data extraction run.
Details to the "right" of the HLA Post Office, representing HLA specific processing, are the same as for use case 42.3; see that sequence diagram for more information.

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.

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.

Design Notes

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

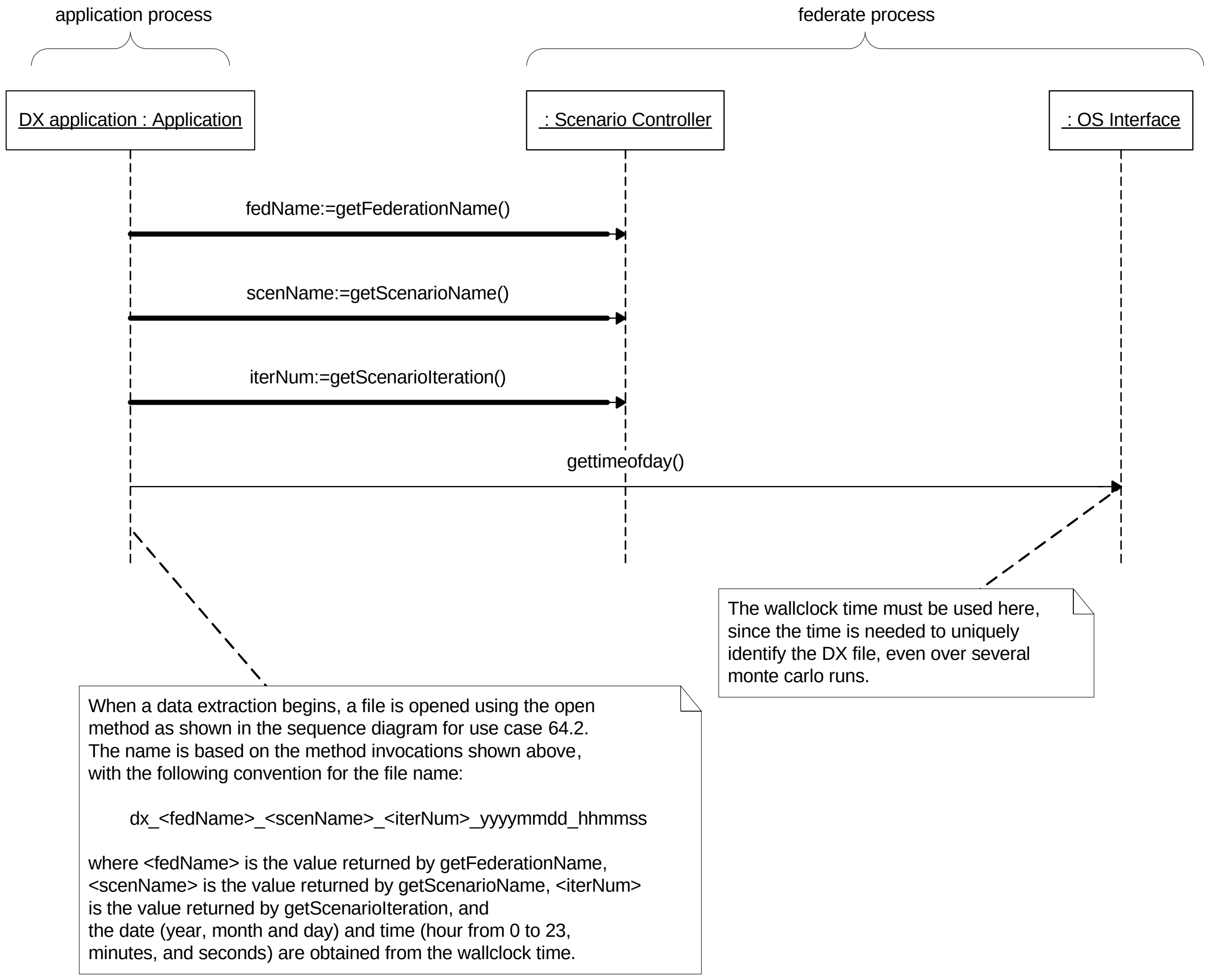
```
dx_<fedName>_<scenName>_<iterNum>_yyyymmdd_hhmmss
```

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 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."


application process
federate process
DX application : Application
: Scenario Controller
: OS Interface
fedName:=getFederationName()
scenName:=getScenarioName()
iterNum:=getScenarioIteration()
gettimeofday()

The wallclock time must be used here, since the time is needed to uniquely identify the DX file, even over several monte carlo runs.

When a data extraction begins, a file is opened using the open method as shown in the sequence diagram for use case 64.2. The name is based on the method invocations shown above, with the following convention for the file name:

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 wallclock time.

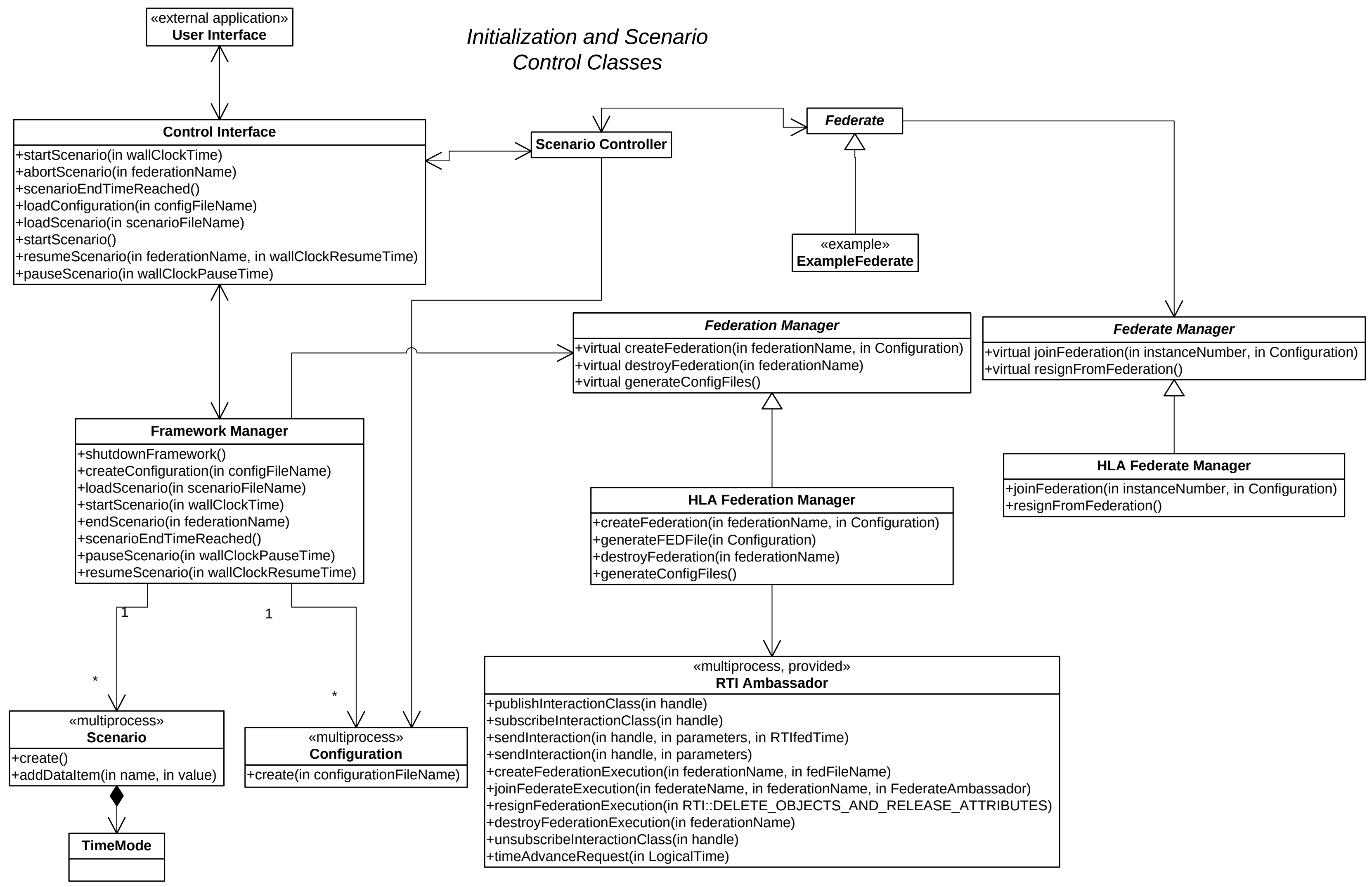
Framework Initialization and Scenario Control

The Johns Hopkins University
APPLIED PHYSICS LABORATORY

Framework Init / Scenario Control Responsibilities

- **Startup & shutdown framework.**
- **Create/destroy federations and federate instances.**
- **Startup, shutdown, pause, resume scenarios.**
- **Load scripted configurations and scenarios into framework, including delivering all necessary scenario initial conditions to federates.**

Initialization and Scenario Control Classes



«external application»
User Interface
Control Interface
+startScenario(in wallClockTime)
+abortScenario(in federationName)
+scenarioEndTimeReached()
+loadConfiguration(in configFileName)
+loadScenario(in scenarioFileName)
+startScenario()
+resumeScenario(in federationName, in wallClockResumeTime)
+pauseScenario(in wallClockPauseTime)
Scenario Controller
Federate
«example»
ExampleFederate
Federation Manager
+virtual createFederation(in federationName, in Configuration)
+virtual destroyFederation(in federationName)
+virtual generateConfigFiles()
Federate Manager
+virtual joinFederation(in instanceNumber, in Configuration)
+virtual resignFromFederation()
Framework Manager
+shutdownFramework()
+createConfiguration(in configFileName)
+loadScenario(in scenarioFileName)
+startScenario(in wallClockTime)
+endScenario(in federationName)
+scenarioEndTimeReached()
+pauseScenario(in wallClockPauseTime)
+resumeScenario(in wallClockResumeTime)
HLA Federation Manager
+createFederation(in federationName, in Configuration)
+generateFEDFile(in Configuration)
+destroyFederation(in federationName)
+generateConfigFiles()
HLA Federate Manager
+joinFederation(in instanceNumber, in Configuration)
+resignFromFederation()
1
1
*
*
«multiprocess»
Scenario
+create()
+addDataItem(in name, in value)
«multiprocess»
Configuration
+create(in configurationFileName)
«multiprocess, provided»
RTI Ambassador
+publishInteractionClass(in handle)
+subscribeInteractionClass(in handle)
+sendInteraction(in handle, in parameters, in RTIfedTime)
+sendInteraction(in handle, in parameters)
+createFederationExecution(in federationName, in fedFileName)
+joinFederateExecution(in federateName, in federationName, in FederateAmbassador)
+resignFederationExecution(in RTI::DELETE_OBJECTS_AND_RELEASE_ATTRIBUTES)
+destroyFederationExecution(in federationName)
+unsubscribeInteractionClass(in handle)
+timeAdvanceRequest(in LogicalTime)
TimeMode

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.

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)

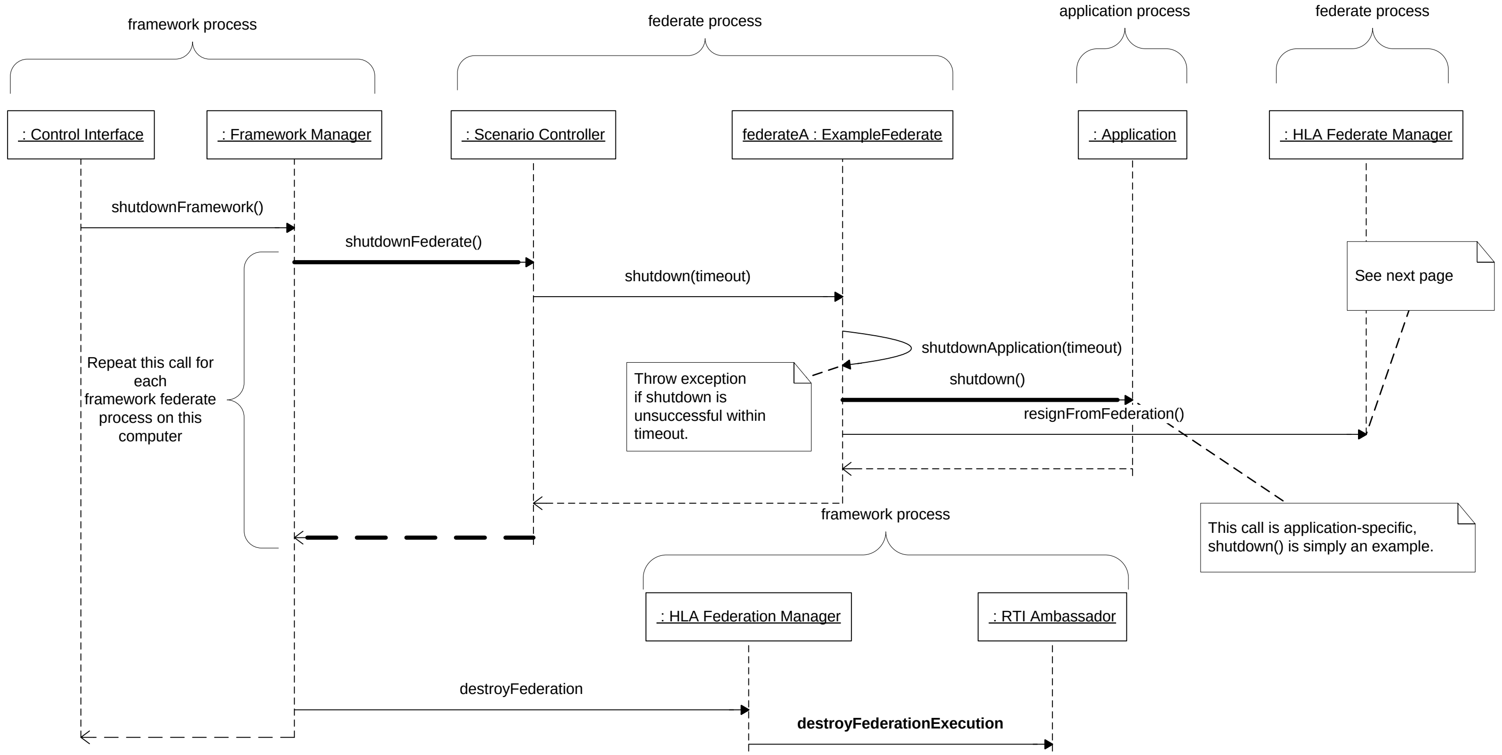
Basic Course of Events

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.

Sequence Diagram for
Use Case 0.2
Framework is shut down.
Diagram 1 of 2

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.

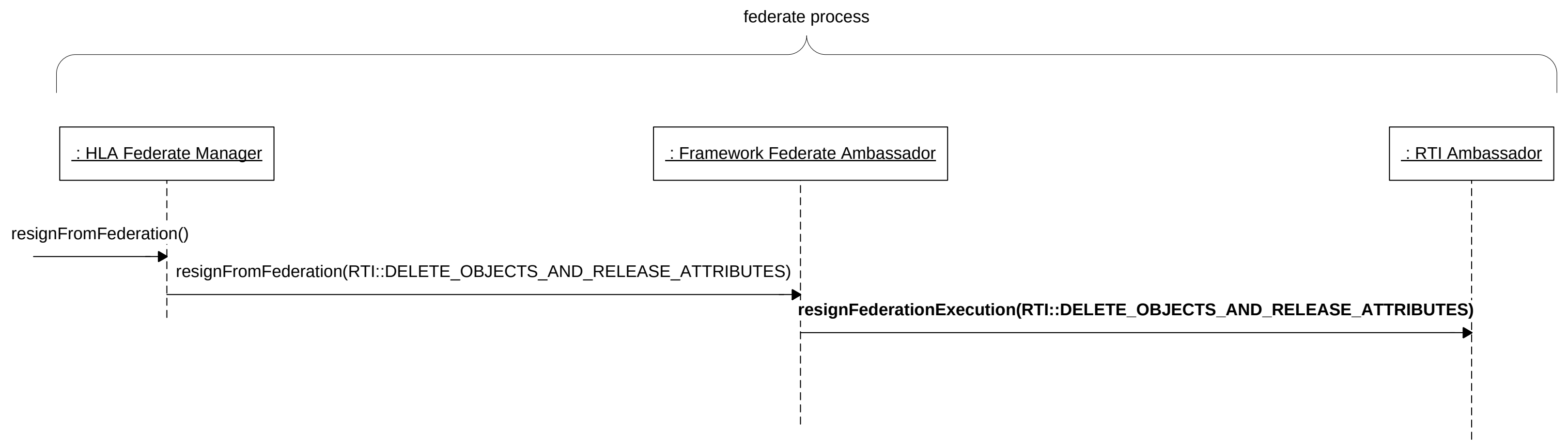
The alternative design would be to pass a shutdown message to all federates as a Framework Message



Sequence Diagram for Use Case 0.2

Framework is shut down.

Diagram 2 of 2


federate process
: HLA Federate Manager
: Framework Federate Ambassador
: RTI Ambassador
resignFromFederation()
resignFromFederation(RTI::DELETE_OBJECTS_AND_RELEASE_ATTRIBUTES)
resignFederationExecution(RTI::DELETE_OBJECTS_AND_RELEASE_ATTRIBUTES)

Use Case 10.1

Summary:

A federation named “Fed1” is established.

Preconditions:

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

Triggers

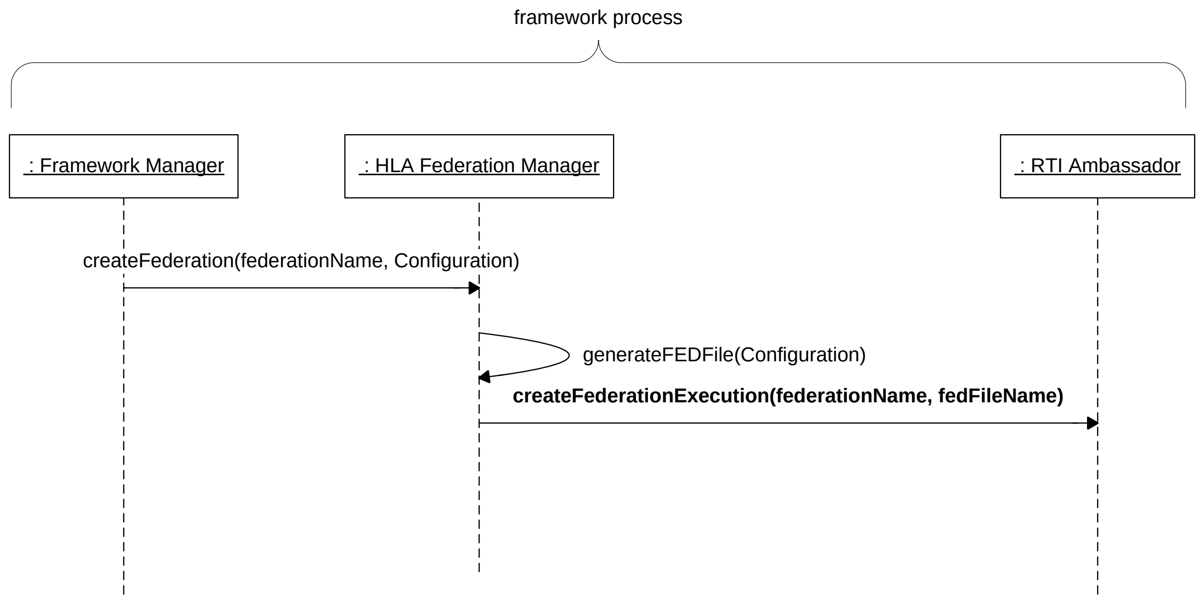
During a configuration or scenario initialization step, a call to establish a federation is made.

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.

Sequence Diagram for Use Case 10.1

A federation named “Fed1” is established.


framework process
: Framework Manager
: HLA Federation Manager
: RTI Ambassador
createFederation(federationName, Configuration)
generateFEDFile(Configuration)
createFederationExecution(federationName, fedFileName)

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.

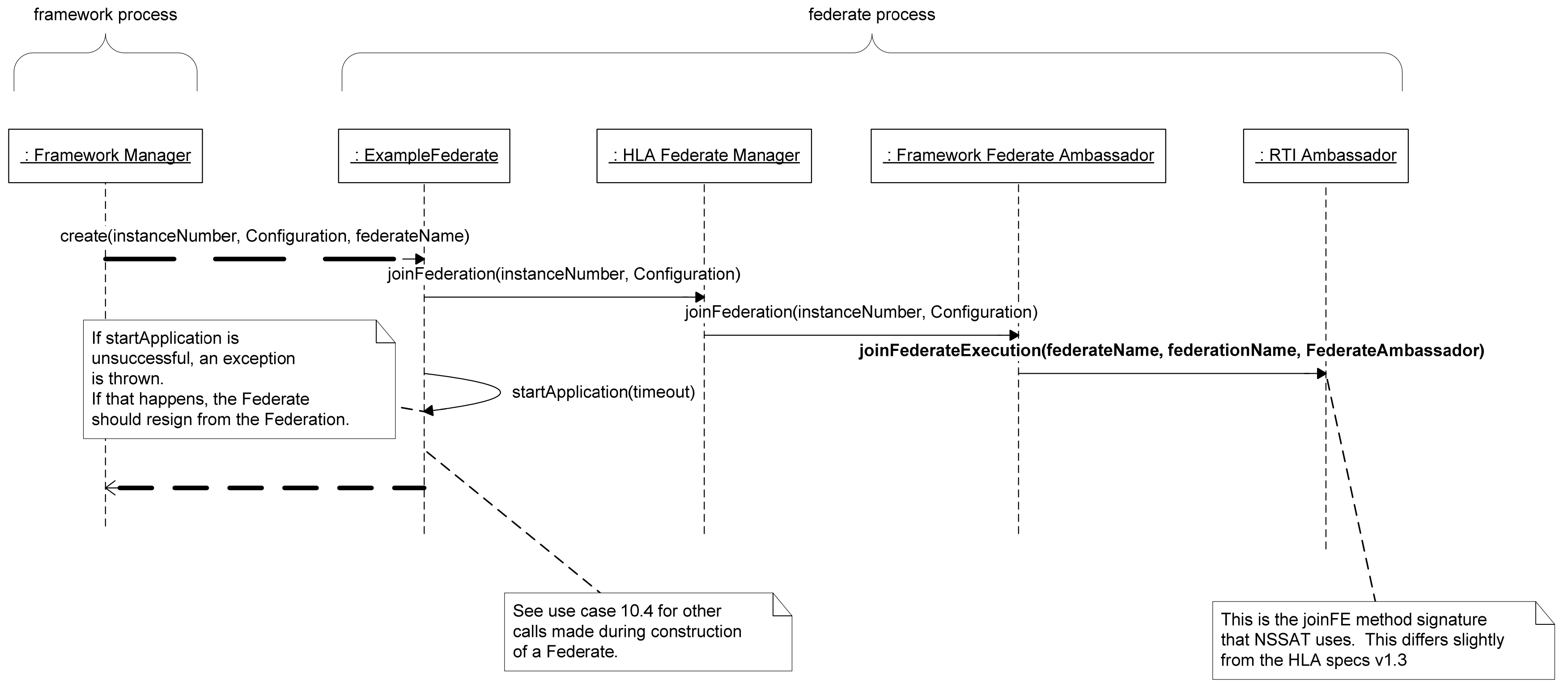
Basic Course of Events

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.

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.


framework process
federate process
: Framework Manager
: ExampleFederate
: HLA Federate Manager
: Framework Federate Ambassador
: RTI Ambassador
create(instanceNumber, Configuration, federateName)
joinFederation(instanceNumber, Configuration)
joinFederation(instanceNumber, Configuration)
joinFederateExecution(federateName, federationName, FederateAmbassador)
startApplication(timeout)
If startApplication is unsuccessful, an exception is thrown. If that happens, the Federate should resign from the Federation.
See use case 10.4 for other calls made during construction of a Federate.
This is the joinFE method signature that NSSAT uses. This differs slightly from the HLA specs v1.3

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

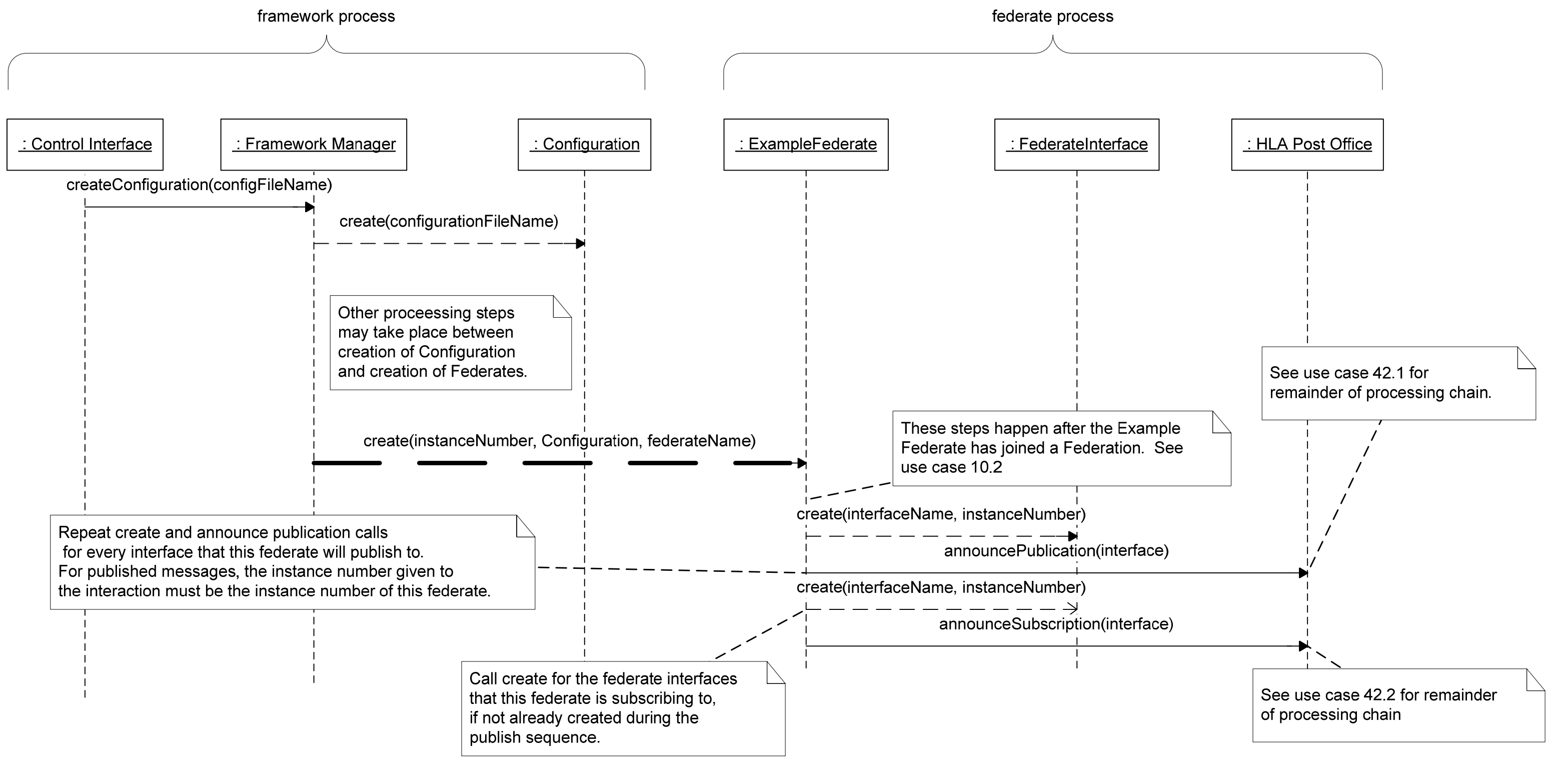
This can partially be handled at configuration time[1].
Steps 1 and 2 are performed offline, prior to scenario execution. The other steps are performed at runtime.

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:

[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.

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.



framework process
federate process
: Control Interface
: Framework Manager
: Configuration
: ExampleFederate
: FederateInterface
: HLA Post Office
createConfiguration(configFileName)
create(configurationFileName)
Other proceessing steps may take place between creation of Configuration and creation of Federates.
create(instanceNumber, Configuration, federateName)
These steps happen after the Example Federate has joined a Federation. See use case 10.2
See use case 42.1 for remainder of processing chain.
create(interfaceName, instanceNumber)
announcePublication(interface)
Repeat create and announce publication calls for every interface that this federate will publish to. For published messages, the instance number given to the interaction must be the instance number of this federate.
create(interfaceName, instanceNumber)
announceSubscription(interface)
Call create for the federate interfaces that this federate is subscribing to, if not already created during the publish sequence.
See use case 42.2 for remainder of processing chain

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

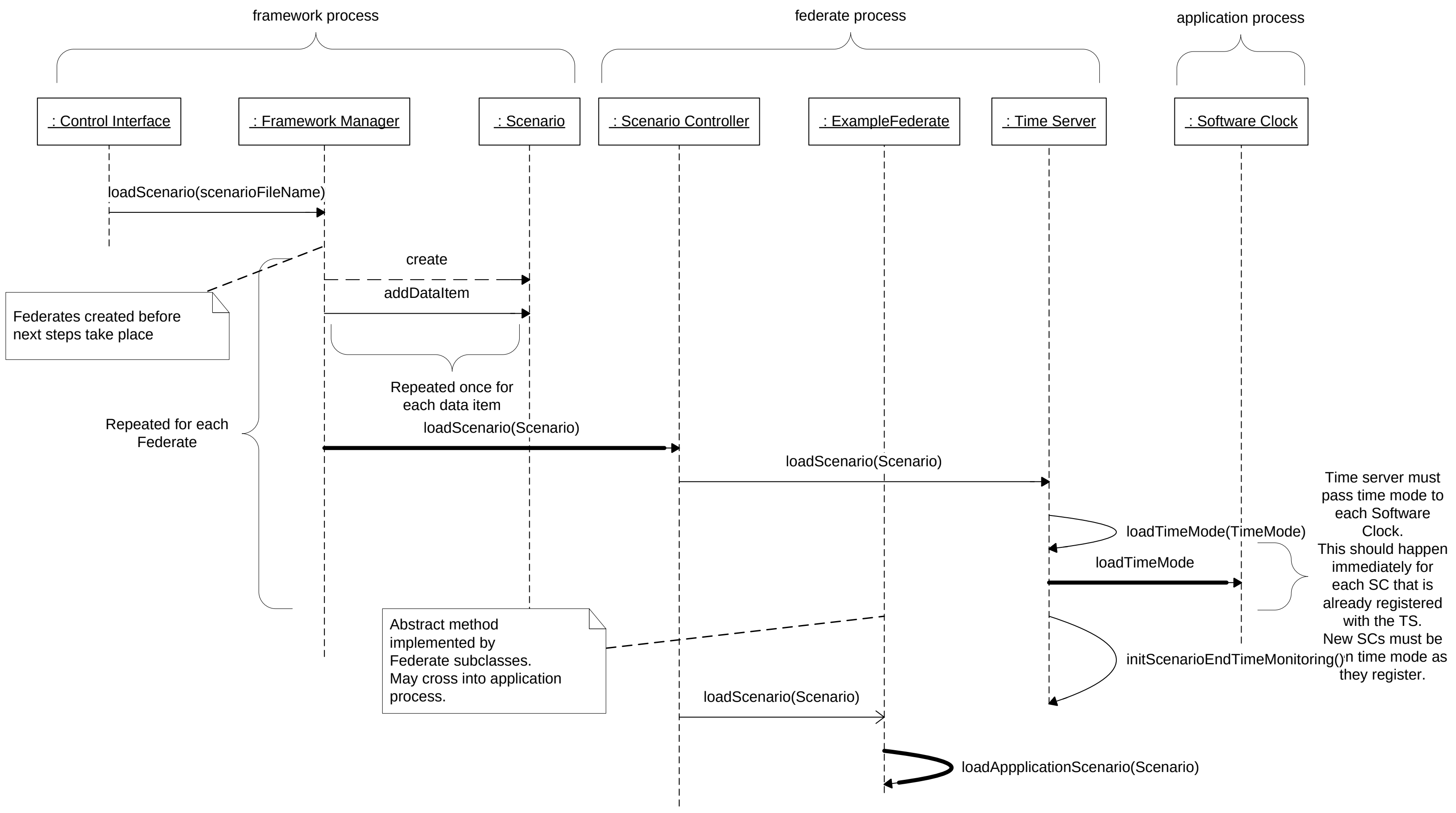
1. User or automated tool enters federate-specific scenario data into a file (offline, independent of framework)
2. User interface sends load scenario command, along with the scenario file name, to the framework process on each computer running the framework.

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.

Sequence Diagram for Use Case 20.1

User scripts a scenario...Scenario initial conditions are delivered to all federates.



framework process
federate process
application process
: Control Interface
: Framework Manager
: Scenario
: Scenario Controller
: ExampleFederate
: Time Server
: Software Clock
loadScenario(scenarioFileName)
create
addDataItem
Federates created before next steps take place
Repeated once for each data item
Repeated for each Federate
loadScenario(Scenario)
loadScenario(Scenario)
loadTimeMode(TimeMode)
loadTimeMode
Time server must pass time mode to each Software Clock. This should happen immediately for each SC that is already registered with the TS. New SCs must be given time mode as they register.
initScenarioEndTimeMonitoring()
Abstract method implemented by Federate subclasses. May cross into application process.
loadScenario(Scenario)
loadAppplicationScenario(Scenario)

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

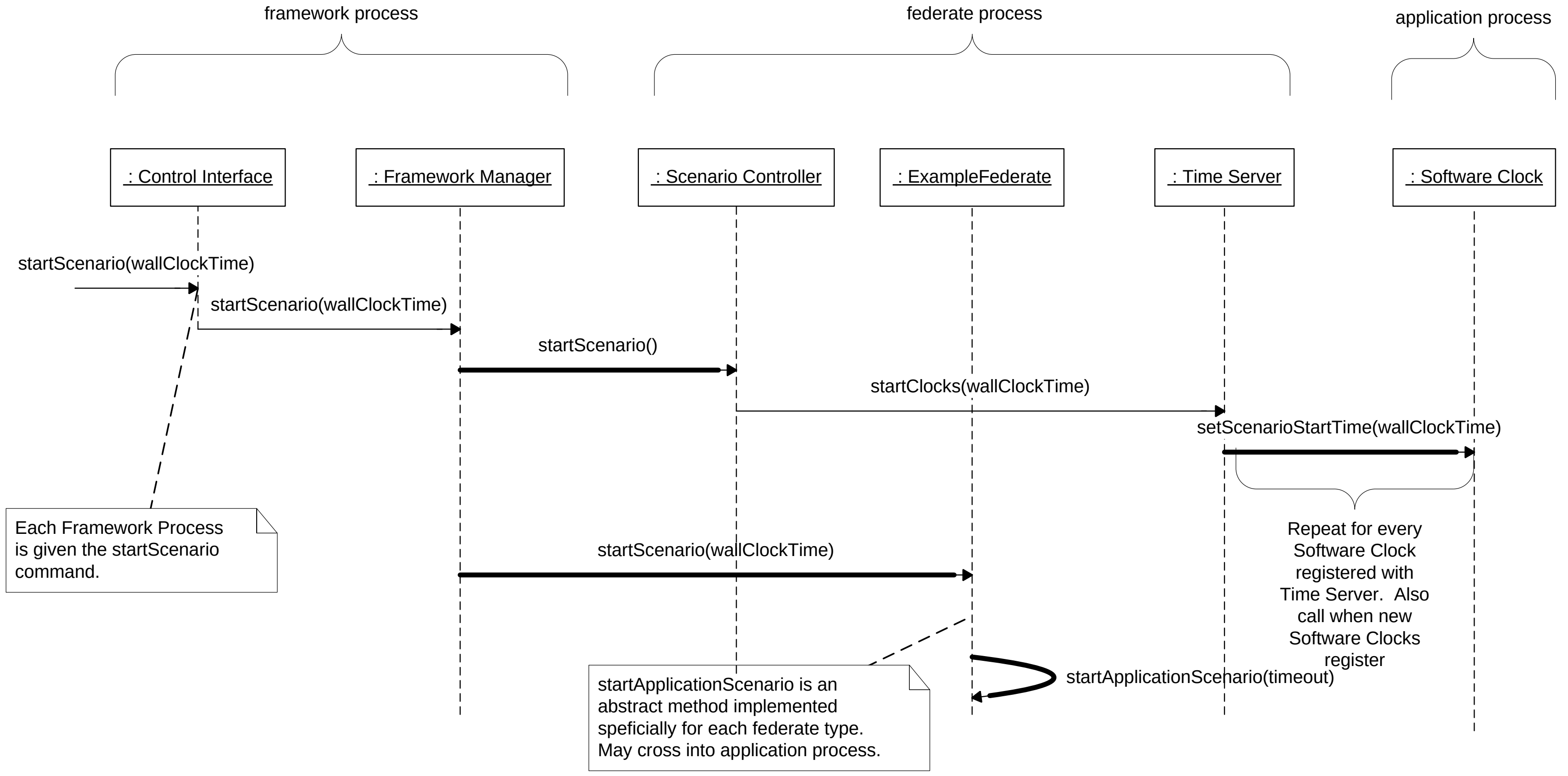
1. Start scenario command sent to the Framework Manager on each framework process. This call specifies the wall clock time at which to start the scenario.
2. The Framework Manager tells each Scenario Controller to start the scenario.

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.

Sequence Diagram for Use Case 31.1

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


framework process
federate process
application process
: Control Interface
: Framework Manager
: Scenario Controller
: ExampleFederate
: Time Server
: Software Clock
startScenario(wallClockTime)
startScenario(wallClockTime)
startScenario()
startClocks(wallClockTime)
setScenarioStartTime(wallClockTime)
Repeat for every Software Clock registered with Time Server. Also call when new Software Clocks register
startScenario(wallClockTime)
startApplicationScenario(timeout)
Each Framework Process is given the startScenario command.
startApplicationScenario is an abstract method implemented speficially for each federate type. May cross into application process.

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

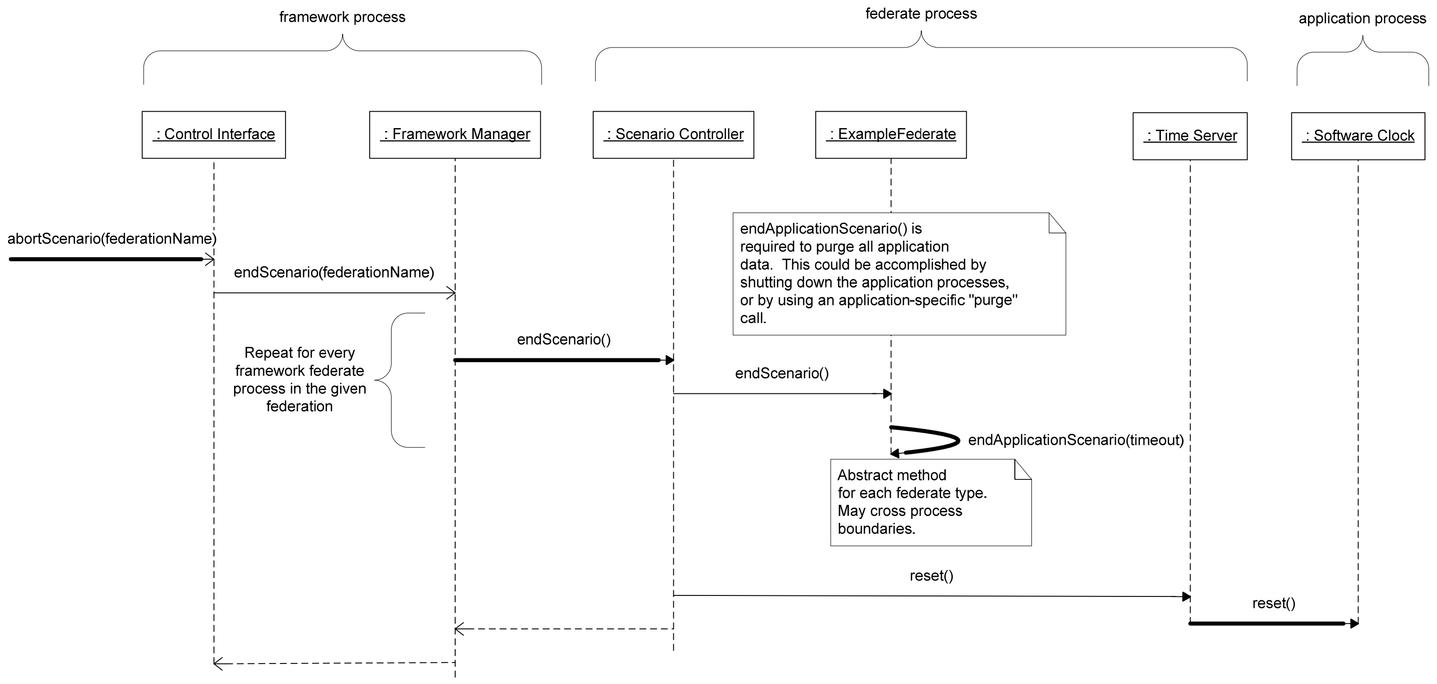
Abort scenario message received from user interface.

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.

Sequence Diagram for Use Case 32.1

Display interface receives abort scenario message for all federations. Every federate stops running and purge its data.


framework process
federate process
application process
: Control Interface
: Framework Manager
: Scenario Controller
: ExampleFederate
: Time Server
: Software Clock
abortScenario(federationName)
endScenario(federationName)
endApplicationScenario() is required to purge all application data. This could be accomplished by shutting down the application processes, or by using an application-specific "purge" call.
Repeat for every framework federate process in the given federation
endScenario()
endScenario()
endApplicationScenario(timeout)
Abstract method for each federate type. May cross process boundaries.
reset()
reset()

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

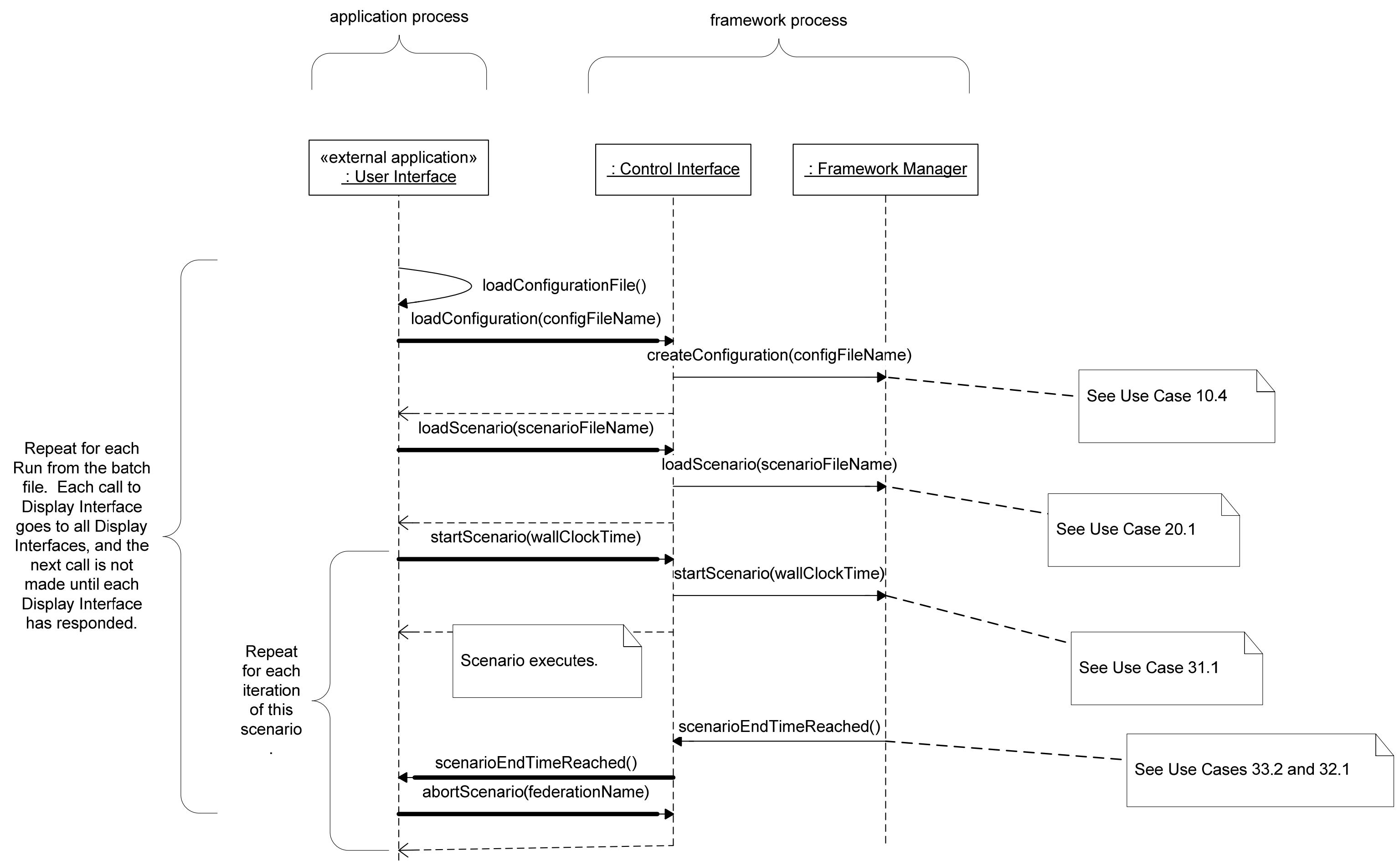
Offline: user scripts batch file (as described in Design Notes below) either manually or through a helper tool.

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.

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.


application process
framework process
«external application» : User Interface
: Control Interface
: Framework Manager
loadConfigurationFile()
loadConfiguration(configFileName)
createConfiguration(configFileName)
See Use Case 10.4
loadScenario(scenarioFileName)
loadScenario(scenarioFileName)
See Use Case 20.1
startScenario(wallClockTime)
startScenario(wallClockTime)
See Use Case 31.1
Scenario executes.
scenarioEndTimeReached()
See Use Cases 33.2 and 32.1
scenarioEndTimeReached()
abortScenario(federationName)
Repeat for each Run from the batch file. Each call to Display Interface goes to all Display Interfaces, and the next call is not made until each Display Interface has responded.
Repeat for each iteration of this scenario

Use Case 27.1

Summary

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

Preconditions

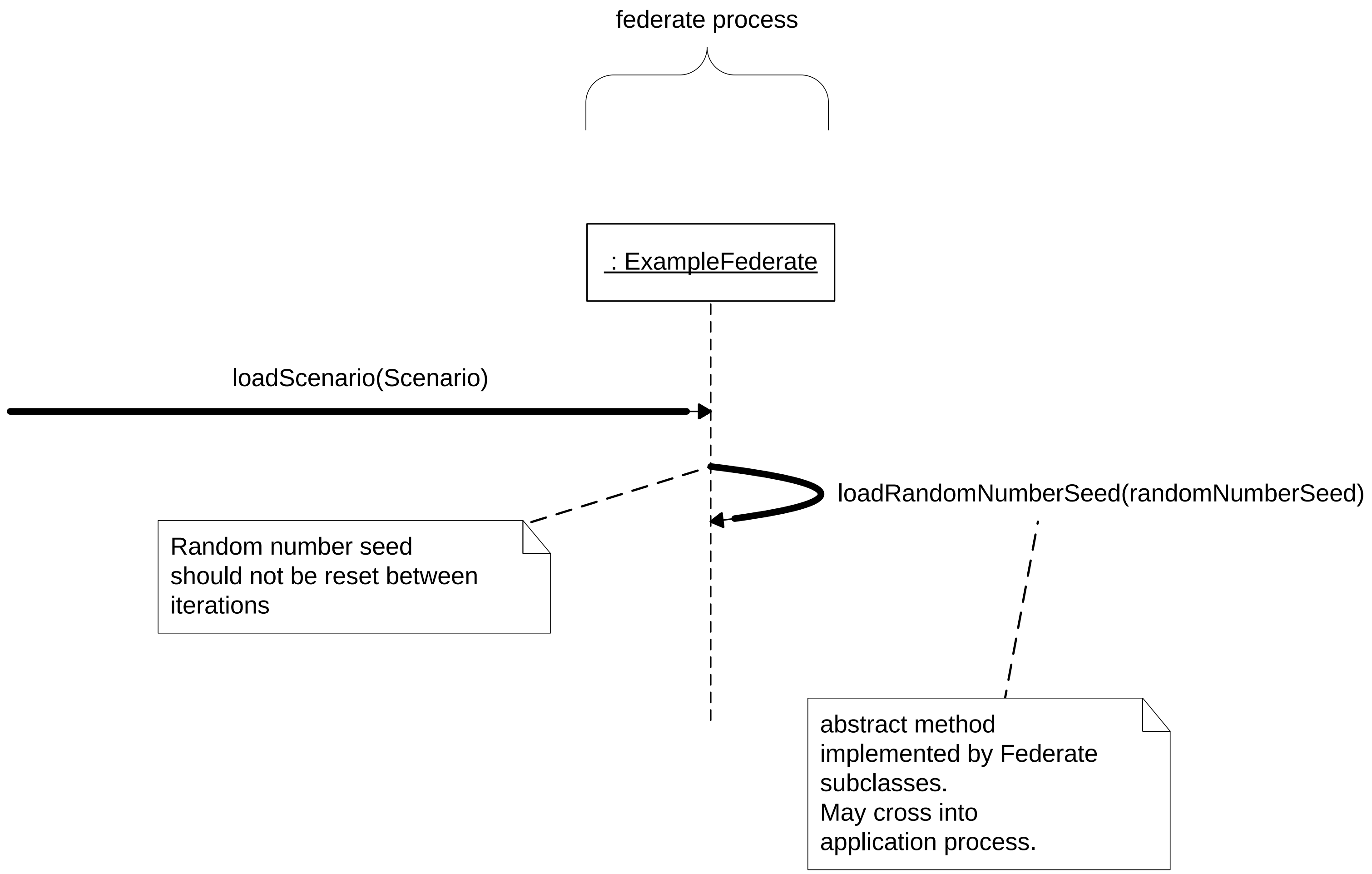
Scenario and Federate objects have been created.

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.


federate process
: ExampleFederate
loadScenario(Scenario)
loadRandomNumberSeed(randomNumberSeed)
Random number seed should not be reset between iterations
abstract method implemented by Federate subclasses. May cross into application process.

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:

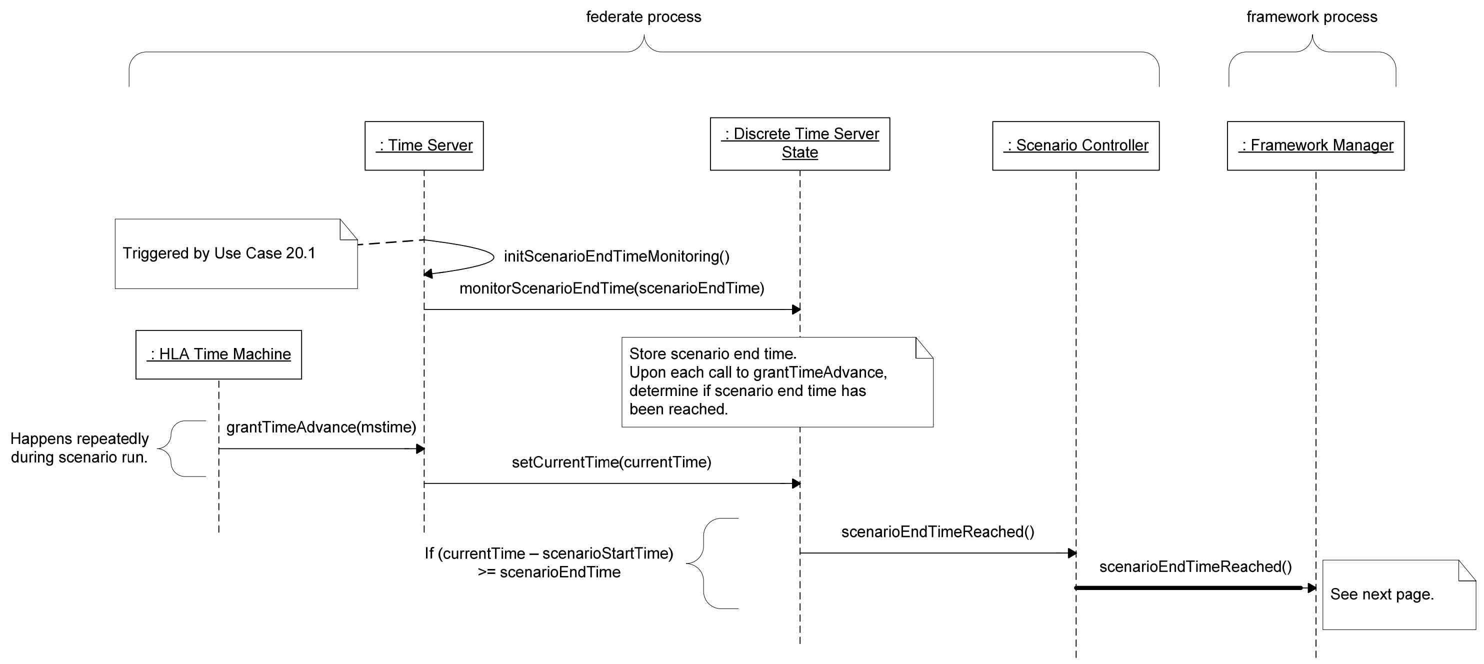
   currentScenarioTime – scenarioStartTime >= scenarioRunTime

   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.

Sequence Diagram for Use Case 33.2

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

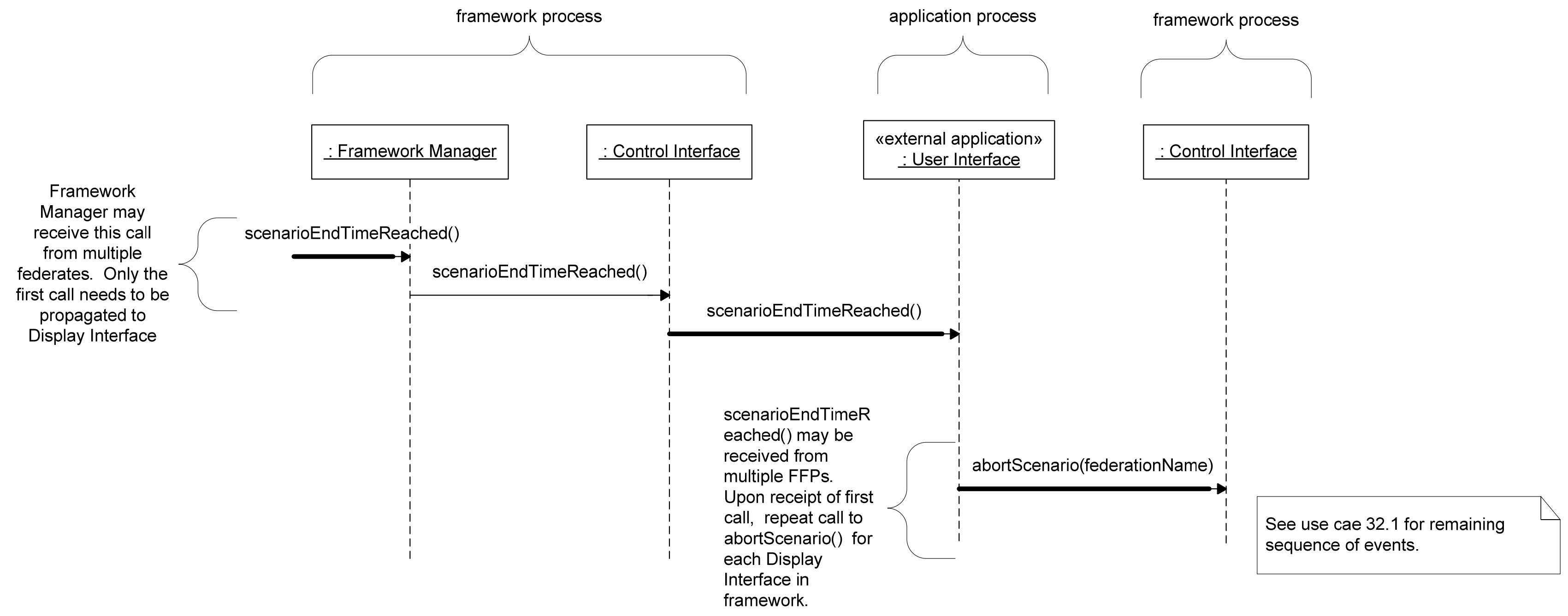
Sub-case a: Scenario is executing in discrete time.
Diagram 1 of 2


federate process
framework process
: Time Server
: Discrete Time Server State
: Scenario Controller
: Framework Manager
Triggered by Use Case 20.1
initScenarioEndTimeMonitoring()
monitorScenarioEndTime(scenarioEndTime)
: HLA Time Machine
Store scenario end time.
Upon each call to grantTimeAdvance, determine if scenario end time has been reached.
Happens repeatedly during scenario run.
grantTimeAdvance(mstime)
setCurrentTime(currentTime)
scenarioEndTimeReached()
If (currentTime – scenarioStartTime) >= scenarioEndTime
scenarioEndTimeReached()
See next page.

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.


framework process
application process
framework process
: Framework Manager
: Control Interface
«external application» : User Interface
: Control Interface
Framework Manager may receive this call from multiple federates. Only the first call needs to be propagated to Display Interface
scenarioEndTimeReached()
scenarioEndTimeReached()
scenarioEndTimeReached()
scenarioEndTimeReached() may be received from multiple FFPs. Upon receipt of first call, repeat call to abortScenario() for each Display Interface in framework.
abortScenario(federationName)
See use cae 32.1 for remaining sequence of events.

Use Case 33.2

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 scenarioEndTime / 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:

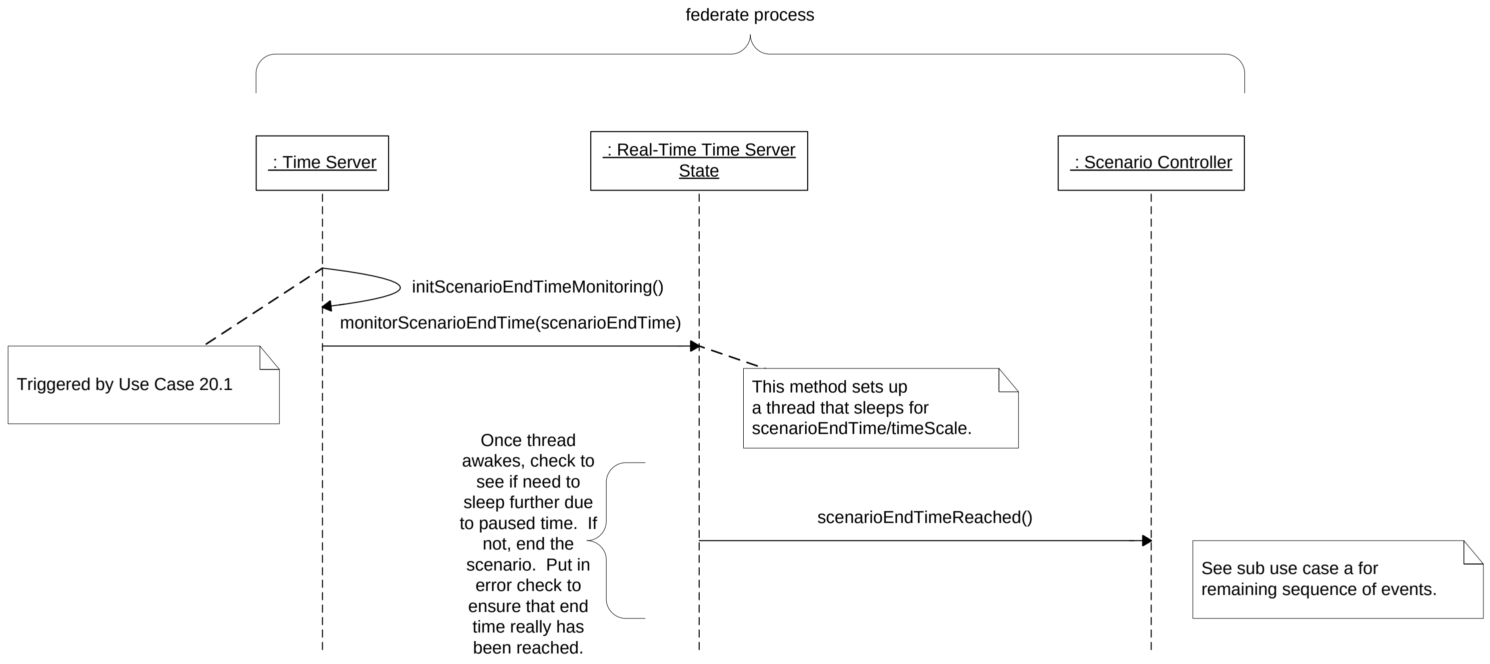
   currentWallTime – scenarioStartTime - totalPausedTime >= scenarioRunTime

   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 b: Scenario is executing in real time.



federate process
: Time Server
: Real-Time Time Server State
: Scenario Controller
initScenarioEndTimeMonitoring()
monitorScenarioEndTime(scenarioEndTime)
Triggered by Use Case 20.1
This method sets up a thread that sleeps for scenarioEndTime/timeScale.
Once thread awakes, check to see if need to sleep further due to paused time. If not, end the scenario. Put in error check to ensure that end time really has been reached.
scenarioEndTimeReached()
See sub use case a for remaining sequence of events.

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.

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.

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.

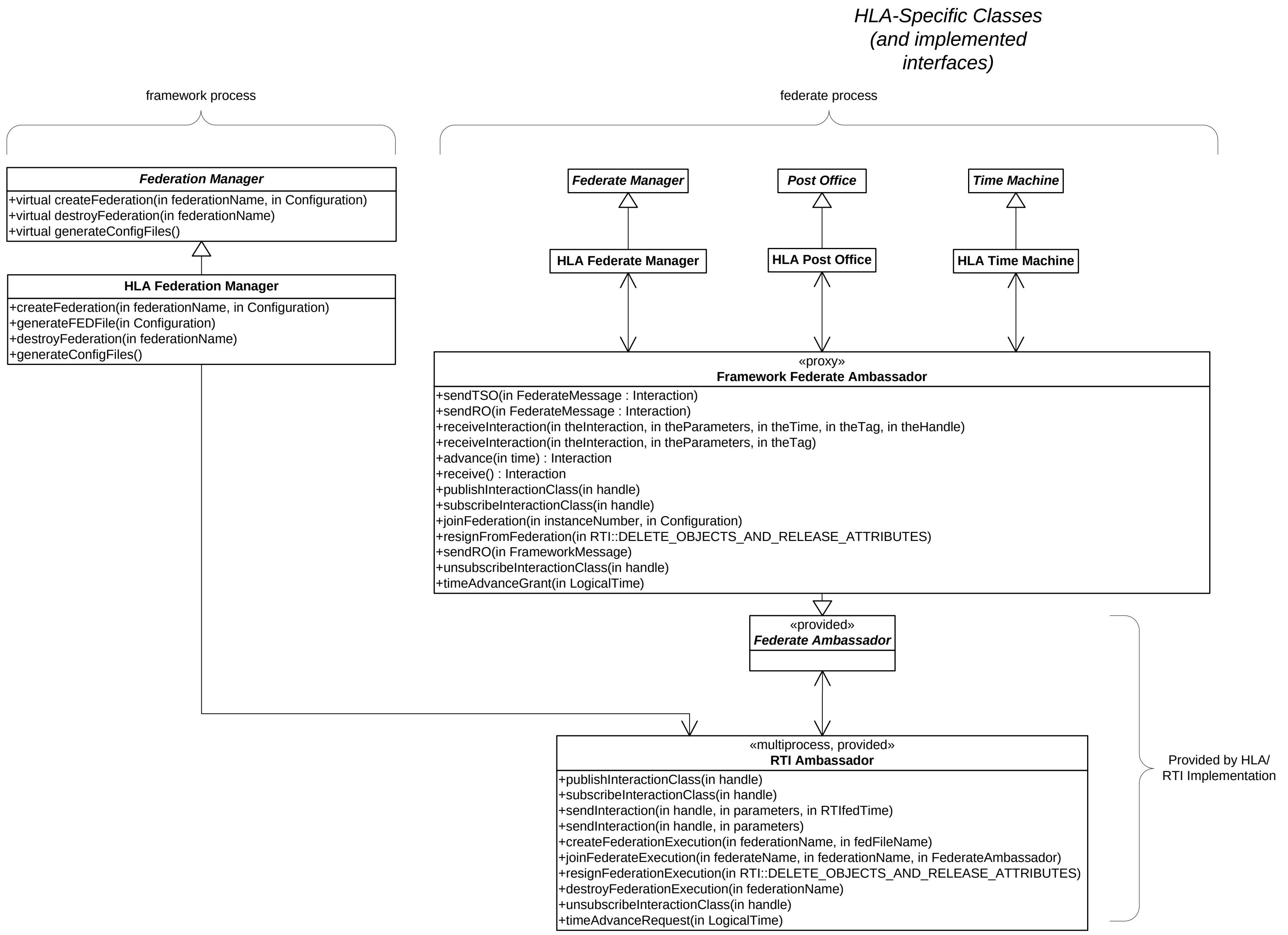
Other Design Concepts

Error Handling

- As hinted at in some of the sequence diagrams, C++ exceptions are going to be used to indicate error conditions, as opposed to return codes.
- This has not been fleshed out in detail, but a likely strategy is to define a base class such as “Framework Exception” and subclass the different types of errors off of that.   When an an error or exceptional condition is encountered, the appropriate exception object is created and thrown.
- Decisions about how to handle exceptions may need to be determined on a case by case basis. The most severe response would be to not run or shut down a federation, federate, or scenario. The least severe would be to log a warning and move on.

HLA As Middleware

- Choice of RTI Implementation
  - Surveys indicated freeware DMSO version not optimal
  - Most reputable vendor products seem to be MÄK and Pitch RTIs
  - MÄK has some advantages over Pitch
    - MÄK has support for Solaris 10
    - Implemented in C++ (Pitch is in Java)
    - Other AMDD projects (ANTARES and NSSAT) use MÄK
  - We are inclined to go with MÄK.
  - Items to investigate
    - Runtime performance of MÄK
    - Compiler incompatibilities. MAK requires Sun C++ compiler. C++ code compiled with different compilers cannot be linked. This may require a C wrapper around calls to RTI.



HLA-Specific Classes (and implemented interfaces)
framework process
federate process
Federation Manager
+virtual createFederation(in federationName, in Configuration)
+virtual destroyFederation(in federationName)
+virtual generateConfigFiles()
HLA Federation Manager
+createFederation(in federationName, in Configuration)
+generateFEDFile(in Configuration)
+destroyFederation(in federationName)
+generateConfigFiles()
Federate Manager
Post Office
Time Machine
HLA Federate Manager
HLA Post Office
HLA Time Machine
«proxy»
Framework Federate Ambassador
+sendTSO(in FederateMessage : Interaction)
+sendRO(in FederateMessage : Interaction)
+receiveInteraction(in theInteraction, in theParameters, in theTime, in theTag, in theHandle)
+receiveInteraction(in theInteraction, in theParameters, in theTag)
+advance(in time) : Interaction
+receive() : Interaction
+publishInteractionClass(in handle)
+subscribeInteractionClass(in handle)
+joinFederation(in instanceNumber, in Configuration)
+resignFromFederation(in RTI::DELETE_OBJECTS_AND_RELEASE_ATTRIBUTES)
+sendRO(in FrameworkMessage)
+unsubscribeInteractionClass(in handle)
+timeAdvanceGrant(in LogicalTime)
«provided»
Federate Ambassador
«multiprocess, provided»
RTI Ambassador
+publishInteractionClass(in handle)
+subscribeInteractionClass(in handle)
+sendInteraction(in handle, in parameters, in RTIfedTime)
+sendInteraction(in handle, in parameters)
+createFederationExecution(in federationName, in fedFileName)
+joinFederateExecution(in federateName, in federationName, in FederateAmbassador)
+resignFederationExecution(in RTI::DELETE_OBJECTS_AND_RELEASE_ATTRIBUTES)
+destroyFederationExecution(in federationName)
+unsubscribeInteractionClass(in handle)
+timeAdvanceRequest(in LogicalTime)
Provided by HLA/ RTI Implementation

Application Capture (1 of 2)

- In order to incorporate applications into the framework, need to gather information about them, e.g. I/O function calls, time function calls, message structure definitions, etc. This should be automated as much as possible.
  - Serialization code can be auto-generated given a C-style struct used for I/O. This can be done by modifying existing scripts.
  - Given a list of I/O function call prototypes, function prototypes for the Federate I/O Handler can be partially automatically generated. This was prototyped using a simple test application. Time-related calls are also a possibility for auto-replacement, although there is more variety in how they need to behave.
  - Scripts could be written to automatically search for certain types of function calls, or programming practices that may cause difficulty when incorporating an application.

Application Capture (2 of 2)

- For applications that cannot be brought into framework without source code modifications, it would be ideal to automate those changes so they can be recreated or reversed.

- In summary, advantages to automatic application capture are:
  - Reducing the amount of tedious manual coding, thus reducing chances for error.
  - Bookkeeping of exactly what was need to bring applications into framework, providing guidance for future applications.
  - User-friendly GUIs could be created to guide developers through the process of incorporating a new application into the framework.

Risk Areas

- Use of interposition libraries to intercept calls made by VxWorks middleware still under investigation (no show-stoppers yet).
- RTI performance is unknown. There are several good indications about MÄK RTI performance, but we don’t have solid numbers on latest versions yet.
- Time needed to incorporate and test federates may be greater than anticipated.
- May need to work around compiler incompatibilities
  - MAK RTI requires Sun C++ compiler
  - Different versions of g++ compiler may be used between federates and framework.

Potential Work Beyond FY08

- Use of OASIS as middleware instead of HLA
- Support for additional programming languages, e.g. Java or MATLAB
- Additional federates (IABM? TSCEI?)
- Additional federate behavior (tactical data links, more engagement support, additional threats, combat identification)
- Advanced features described in the requirements

Related Resources

- IR&D Sharepoint Website:
  - http://aplteam/jhuapl/ADSD/projects/CSSF/
- IEEE 1516 HLA Specifications
  - Downloadable from http://ieeexplore.ieee.org/xpl/standards.jsp
- UML Overview
  - http://en.wikipedia.org/wiki/Unified_Modeling_Language