**MUTHUR**

\_\_\_\_\_\_\_\_\_\_\_\_\_

**Developers Guide**

**Edition 0.1**

**04-Apr-11**

|  |  |  |
| --- | --- | --- |
| **DOCUMENT CHANGE HISTORY** | | |
| **Edition** | **Date** | **Description of Change** |
| 0.1 | 04-Apr-11 | First draft |
|  |  |  |

**Contents**

1 Introduction 5

1.1 Purpose of this Document 5

1.2 Font and Diagram Conventions 5

1.3 Definition of MUTHUR 6

2 Requirements, Goals and Approaches 7

2.1 Architectural Qualities 7

3 Components 7

3.1.1 Federate 7

3.1.2 Controlling Federate 8

3.1.3 Federation 8

3.1.4 MUTHUR 8

4 Data Structures 8

4.1 Federate Registration Handle (FRH) 8

4.2 Federation Execution Handle (FEH) 9

4.3 Federation Execution Model (FEM) 9

4.4 Events 9

4.4.1 Requests and Responses 9

4.4.2 Data Objects 11

4.4.3 System 14

5 Ambassador 14

5.1 Overview 14

5.2 Callbacks and Asynchronous Messaging 15

5.3 Protocol and Flow 15

5.4 Initialization 17

5.5 Registration 17

5.6 List Federation Execution Models 17

5.7 Joining a Federation 18

5.8 Data Subscription 19

5.9 Ready To Run 19

5.10 Publishing Data 20

5.11 Federation Termination 20

**Figures**

Figure 1 - Callback example 15

Figure 2 - Ambassador to MUTHUR protocol 16

Figure 3 - Ambassador initialization 17

Figure 4 - Federate Registration 17

Figure 5 - List Federation Execution Models 18

Figure 6 - Join federation 18

Figure 7 - Data subscription request 19

Figure 8 - Ready to run request 20

Figure 9 - Publishing data 20

Figure 10 - Federation termination event 20

# Introduction

## Purpose of this Document

MUTHUR provides synchronization, data sharing and ownership management, common model representation, time management and other related functions required to allow two or more simulations and/or applications to seamlessly interoperate and form a “federation” of systems that appear as a single, cooperating system. While it is intended and targeted at simulation or synthetic environments there is nothing in the design that limits participation in a federation to simulators.

The initial goal of MUTHUR is to form a federation that includes at least the flight operations lab located at MTSU and a flight training device or flight simulator from Frasca International.

The long-term vision of MUTHUR is to provide for the construction and execution of federations encompassing dozens or more synthetic environments, applications or tools distributed across a vast network. This long-term goal will be accomplished in a phased approach delivering those functions required for the initial deployment and building on those with each additional phase.

This document is intended to be a developer’s guide to integrating simulators, applications and tools with MUTHUR via its client API that is largely encompassed within the Ambassador class. The Ambassador class encapsulates all the complexity of interfacing with MUTHUR allowing the developer to deal with simple data structures and method calls. It provides initialization, a request/response protocol, data subscription and publication and system event notifications.

The document will provide a brief description of the overall architecture of MUTHUR along with the major components and concepts that a developer will need to know. It describes the data structures and methods that a developer will need to know.

## Font and Diagram Conventions

The following font convention is used in this document:

* **ExecutableProgram** (50% gray underline)
* **Class**
* **Method()** or **Class::Method()**
* ***database\_table***
* *database\_table\_column*

The following symbols are used in several activity diagrams.



## Definition of MUTHUR

MUTHUR is middleware that allows two or more simulations, applications or tools to seamlessly interoperate and form a “federation” of systems that appears to the outside world as a single, cooperating system.

MUTHUR provides the following functions:

* Synchronized operation – coordinates the start of the federation execution, intentions for federates for publication and subscription, starts the federation execution, signals end of the federation and synchronizes the cleanup and termination of the federation execution.
* Common object model – an agreed upon set of classes and attributes that can be published from or subscribed to by members of a federation execution.
* Data sharing – federates can publish and subscribe to data (classes and attributes) that is described in the FOM. MUTHUR allows federates to express interest in data and coordinates the distribution of data to these federates to satisfy these subscriptions.
* Time management – federates can utilize time management facilities provided by MUTHUR to ensure that the federate proceeds through the federation execution in a synchronized manner. Federates can be time regulating where they actively contribute to or help regulate the federation execution. Federates can also be cooperative and be driven by the federation. Federates can also assume both roles – regulating and cooperative.
* Object ownership – as objects are created they are owned by at least one federate which is maintained by MUTHUR. Ownership of objects can also be transferred between federates.

# Requirements, Goals and Approaches

## Architectural Qualities

MUTHUR’s system architecture and design attempts to realize the following attributes:

* *Flexibility* is essential for MUTHUR to be able to adapt to the many changes that take place in its environment.
* *Generality* is necessary for MUTHUR to efficiently accommodate the wide variety of data types, users, and requirements that characterize its environment.
* *Scalability* is the means by which MUTHUR can support federations that range in size of several federates to hundreds of federates sharing massive amounts of data and executing for long periods of time.
* *Interoperability* is fundamental to ensure that federates of all types, simulations, applications and tools, are able to participate in federation executions.
* *Reliability* in synchronization, execution and data sharing are critical to ensure that the synthetic environment maintains a level of predictable behavior and reliability in order to support modeling, training and research and development efforts.
* *Maintainability* and *evolvability* are key qualities for MUTHUR design and development activities in light of MUTHUR’s long-term mission and the change that characterizes its environment.
* *Openness* and *conformance to community, national, and international standards* are essential for MUTHUR to fulfill its interoperability objectives, and to provide a level of usability in both systems and data sufficient to meet users’ needs.
* *Modularity* and *layering* are necessary to help address MUTHUR’s needs for flexibility and maintainability.

# Components

### Federate

A participant in a simulation coordinated and controlled by **MUTHUR** is referred to as a federate. A federate is most commonly a simulator but can also be an application, management tool or any system that can utilize and abide by the protocol for participating in a federation or collection of federates.

### Controlling Federate

A federation is made up of a collection of federates. While it is not required many federations will be designed to have one federate be the controlling federate. While the full range of the controlling federate’s responsibility depends on the federation design, generally they are responsible for controlling the flow of the federation execution. At the very least this entails determining how long the federation executes as well as when termination conditions have been met during the execution.

### Federation

A federation is a collection of federates that has agreed to join together and cooperates to form a single simulation system. A federation is a logical formation that is created once the required federates have all joined, expressed interest in data via subscriptions, notified the other participants that they are prepared to start and then received the notification that all parties can now begin execution.

The federation execution continues until either the system or one of the federates raises an end execution request. Once all federates including the controlling federate accept the event then the system will send all participating federates a federation termination event. Most commonly the controlling federate is responsible for issuing the federation termination condition when it recognizes or determines the end of execution has been reached.

### MUTHUR

All federations are maintained, coordinated and synchronized by the middleware referred to as MUTHUR. MUTHUR registers federates, maintains all available Federation Execution Models, synchronizes the creation of a federation, maintains all subscriptions registered by the federates, distributes published data to the appropriate interested federates and manages the time slices and steps each federate through the execution of a federation.

Before any federate can register or federations execution can take place MUTHUR must be started and running on the network. MUTHUR is location agnostic and can reside anywhere on the network as long as it accessible to all federates including a remote node across a WAN. Federates connect to and interact with MUTHUR via the Ambassador component API which will be explain in a later section.

# Data Structures

This section describes the data, structures and objects that a developer will encounter when interfacing with MUTHUR via the Ambassador API.

## Federate Registration Handle (FRH)

A federation registration handle (FRH) uniquely identifies the federate to **MUTHUR** and to others in any federations in which the federate participates. A FRH is returned from a successful call to the Ambassador::register () method and is used in the majority of subsequent calls to the **Ambassador**.

## Federation Execution Handle (FEH)

The federation execution handle (FEH) is returned from a call to **Ambassador::joinFederation** (). The FEH is then used in any calls to the **Ambassador** in order to associate the federate to the federation. The FEH is used to uniquely identify an instance of a federation execution and is valid only for the duration of the federation. Once notice of the termination of the federation is received by the federate the handle is invalid and should no longer be used.

## Federation Execution Model (FEM)

Federation execution models are data structures that describe essential characteristics of a federation. FEMs are stored in **MUTHUR** and are retrieved by clients via a call to **Ambassador::listFederationExecutionModels** () passing a FRH and **AmbassaborCallback**. A list of FEMs are returned in the **ListFedExecModelsResponse** via the **AmbassadorCallback::onSuccess** () method.

FEMs contain the name, description, FEM id, logical start time of the federation execution, expected duration of the federation execution and the list of federate names required in order for the federation execution to begin. This list of required federates is used during the join step of a federation execution. When a join is requested by a federate a FEM is sent as a parameter. The list of required federates must also join the federation before the process can proceed to the next step. The logical start time and duration is used by the controlling federate in the federation to determine when the end of federation execution condition has been reached.

## Events

The communication between federates and between federates and **MUTHUR** are based on a message system. The primary payload of these messages is the event, which is modeled by the IEvent class from which all events are derived.

### Requests and Responses

**MUTHUR** supports a request/response mode of communication where federates can send requests and receive responses to those requests asynchronously via callbacks. This section describes each of these requests and it’s corresponding response.

#### FederateRegistration

Before a federate can proceed with interacting with **MUTHUR** and participating in a federation it must first register with **MUTHUR**. The registration requires that the federate provide a federate name, which can be any character string that will be unique among federates. This federate name is provided in a previous call to **Ambassador::getInstance** (), which is explained in a later section. The user calls **Ambassador::register** () passing an **AmbassadorCallback** object.

The response is returned in a **FederationRegistrationResponse** in the **AmbassadorCallback::onSuccess** () method. The federate registration handle (FRH) can be retrieved via the **FederationRegistrationResponse::getRegistrationID** () and is used in subsequent calls to **Ambassador**.

Note that an **AmbassadorCallback** is included in all calls to the **Ambassador** and is assumed in subsequent explanations. If a request is successful the corresponding response is returned in the **Ambassador::onSuccess** () method. If an error occurs then an exception will be returned in **Ambassador::onError** () with an explanation of the error contained in the **Exception** parameter. The following explanations of each method call assume that each request is successfully executed.

#### ListFedExecModels

Before joining a federation a user provide a FEM that will define the federation. One way to get this FEM is to obtain one from the full list of available FEMs that are maintained by **MUTHUR**. To obtain this list, a user calls **Ambassador::listFederationExecutionModels** () passing the FRH from the registration.

The response is returned in a **ListFedExecModelsResponse** in the **AmbassadorCallback::onSuccess** () method. The list of FEMs can be retrieved by a call to **Ambassador::getFEMList** () and iterated over to obtain the desired FEM.

#### JoinFederation

Once the desired FEM is retrieved it is used with the FRH to join the associated federation with a call to **Ambassador::joinFederation** () passing it the FRH and FEM.

The response is received in a **JoinFederationResponse** in the **AmbassadorCallback::onSuccess** () method. The **JoinFederationResponse** contains the FEM handle, which uniquely identifies a federation execution and is retrieved with a call to **JoinFederationResponse::getFederationExecutionHandle** ().

#### DataSubscription

To receive instantiations, updates and removals of data from other participants in a federation a client must issue a **DataSubscriptionRequest** request passing the FRH, FEH, a list of class names and a **DataEventCallback** callback to **Ambassador::addDataSubscriptions** (). The names are those classes for which the client will receive all instantiations or creations of, modifications to and deletions of a particular class. The **DataEventCallback** will receive any publications of these classes (instantiations, modifications and deletions) made by other federates during the federation execution.

The response is received in the **Ambassador::onSuccess** () and is simply an acknowledgement that the subscriptions succeeded.

#### ReadyToRun

Once the client has submitted any data subscriptions and is ready to participate in the federation execution, it calls **Ambassador::readyToRun** () with a FRH and FEH.

The response is received in the **Ambassador::onSuccess** () and is simply an acknowledgement that the request was acknowledged.

### Data Objects

With the Ambassador::addDataSubscriptions() call a client registers interest in the creation, modification and deletion of classes and their objects. These data objects are currently limited to the following classes, which should provide adequate support for the anticipated participants in the expected air traffic management, and ATC focused federation executions. All publications are received in the DataEventCallback that was registered in the call to Ambassador::addDataSubscriptions() in the DataEventCallback::onEvent() method.

All data object are derived from the IDataObject interface. In addition they contain a common set of attributes referred to as the object id, which can be used as a unique key for the object. These attributes are:

* createTimestampMSecs - time in milliseconds when the object was created
* objectUUID – unique identifier in itself for this particular object
* tailNumber – tail number for the particular air craft that uniquely identifies that air craft
* acid – air craft call sign

Any of the data objects can be created with a no-arg constructor and still provide a unique object instance.

#### SpawnAircraft

The SpawnAircraft class represents a new aircraft that can be added to the simulation. This class contains initial position data, flight plan data, taxi out time and expected departure time. The attributes are:

* Flight Position
  + deptArptCode
  + arrArptCode
  + latitudeDegrees
  + longitudeDegrees
  + altitudeFt
  + groundspeedKts
  + headingDegrees
  + airspeedKts
  + rollDegrees
  + yawnDegrees
  + sector
  + center
* Flight Plan
  + source
  + aircraftType
  + deptTime
  + arriveTime
  + cruiseSpeedKts
  + route
  + deptCenter
  + arriveCenter
  + deptFix
  + arriveFix
  + physicalClass
  + weightClass
  + userClass
  + numAircraft
* Taxi data
  + taxiOutTime
* Departure Data
  + actualDeptTime

#### FlightPosition

FlightPosition objects provide an update on the flight position of a previously spawned aircraft whether in the air or on the ground. In order to ensure that the updates can be applied to the current aircraft the appropriate key attributes such as acid and tailNumber need to match the corresponding current active aircraft. This matching of key attributes applies to all updates against current aircraft. The attributes are:

* deptArptCode
* arrArptCode
* latitudeDegrees
* longitudeDegrees
* altitudeFt
* groundspeedKts
* headingDegrees
* airspeedKts
* rollDegrees
* yawnDegrees
* sector
* center

#### FlightPlanFiled

FlightPlanFiled objects are used to update the flight plans or routes of a current aircraft that match the key attributes. The attributes are:

* source
* aircraftType
* deptTime
* arriveTime
* cruiseSpeedKts
* route
* deptCenter
* arriveCenter
* deptFix
* arriveFix
* physicalClass
* weightClass
* userClass
* numAircraft

#### AircraftTaxiOut

AircraftTaxiOut events are generated when an aircraft leaves the gate to begin taxiing out for departure. The attribute is:

* taxiOutTime

#### AircraftDepartureData

#### AircraftDepartureData events should be generated when the aircraft departs from the airport. The actual definition for departure is left up to the initiator of the event but is generally held to be when the wheels leave the runway and the plane moves into local control. The attribute is:

* actualDeptTime

#### AircraftArrivalData

AircraftArrivalData events should be generated when an aircraft arrives at its destination. The actual definition of arrival is left up to the initiator of the event but is generally held to be when the aircraft has touched its wheels on the runway. The attribute is:

* arriveTime

#### KillAircraft

KillAircraft objects are created when a user deletes or removes an aircraft. The attributes are either the unique ID taken from the particular object or a match on the aircraft tail number. The attributes are:

* tailNumber
* acid

### System

System events are issued by **MUTHUR** to participating federates during federation execution. The user will receive system events to notify them of particular states in the execution. System events will be received in the SystemCallback registered by the user during the call to Ambassador::readyToExecute(). Events will be received in the SystemCallback::onEvent() method and can be received at any time during the federation execution.

#### BeginExecution

TODO

#### EndExecution

TODO

#### FederationTermination

TODO

# Ambassador

## Overview

The Ambassador serves as the interface between a federate and MUTHUR. It’s through this interface that a federate registers, joins federations, participates in federation executions, shares data with federates and in the case of a controlling federate controls federation executions.

The previous sections gave an idea of the data structures, concepts and to some extent how to interact with MUTHUR through the Ambassador. This section will go into more detail about these concepts along with examples of how to actually use the Ambassador API.

NOTE: All examples are currently provided in Java with C++ examples to follow shortly.

## Callbacks and Asynchronous Messaging

It is assumed that MUTHUR and any potential federates will be different system processes and more than likely will not even reside on the same machine and will need to communication across a network. Given this separation of processes and unpredictability in location, MUTHUR utilizes a messaging system to provide communication with it’s federates wherever they may reside on the network. With the unpredictability of networks in terms of availability and performance one cannot assume that calls will either return in time or return at all. For these reasons all calls in the Ambassador are asynchronous in that they return almost immediately and return their responses in callbacks.

The use of callbacks requires a slightly different approach then the standard procedure call. With callbacks, calls are asynchronous and return immediately to the caller with the response returned sometime later via the callback. The following is an example of a call to the Ambassador::register method. Note that the method provides an instance of the AmbassadorCallback interface class. The method returns immediately and proceeds. When the response is returned the results are either provided in onSuccess() or onError() depending on the outcome.

ambassador.register(new AmbassadorCallback() {

public void onSuccess(IEvent event) {

if (event != null) {

if (event instanceof FederateRegistrationResponse) {

rr = (FederateRegistrationResponse) event;

String regID = rr.getRegistrationID();

}

}

}

public void onError(Exception e) {

System.out.println(e.getLocalizedMessage());

}

});

Figure - Callback example

If the call does not return in time or times out, then AmbassadorCallback::onError() is called with the appropriate message that the call exceeded the time out. The Ambassador has a default time out value of 30 seconds which can be overridden with a call to Ambassador::setTimeOutSecs(). All calls are subject to timing out except if the timeout value is set to -1 which denotes an indefinite time to live for all calls.

## Protocol and Flow

One of the main functions of MUTHUR is to synchronize and coordinate the execution of several simulators and/or applications in order to form a single simulated environment referred to as a federation. In order to do so each of the participants in the federation must follow a set protocol in order to establish the federation, share data among the participants, coordinate the start and execution of the federation and finally determine when the end condition for the federation execution has been reached.

The following steps are performed by each federate in concert with MUTHUR during its lifetime. The steps in blue are performed when a federate initially starts up. The steps in red are performed each time a federate wants to join or participate in a federation execution. All calls are through the Ambassador API.

|  |  |  |  |
| --- | --- | --- | --- |
| Step | Federate | MUTHUR | Result |
| 1 | ::getInstance () | Returns a connection to the Ambassador | Initializes the Ambassador and establishes connects to MUTHUR |
| 2 | ::register () | Returns registration handle | Establishes federate inside MUTHUR |
| 3 | ::listFederationExecutionsModels () | Returns a list of registered FEMs | Provides federate with FEMs to establish a federation |
| 4 | ::joinFederation () | Returns a federation execution handle | Establishes a federation execution context |
| 5 | ::addDataSubscriptions () | Acknowledges subscriptions | Records subscriptions for the federate for a federation execution |
| 6 | ::readyToRun () | Returns ready to run messages to all participants | Federate execution is started |
| 7 | ::publishData () |  | Distributes data to subscribers in the federation execution |
| 8 | ::terminateFederation () | Returns message that termination complete | Terminates the federation execution |

Figure - Ambassador to MUTHUR protocol

Step 1 and 2 initializes the Ambassador and establishes the federate as a potential participant in the future federations and must be completed prior to step 4. Steps 4 – 8 encompass the life cycle of a federation. Steps 4 – 6 must each be completed by ALL participants before any of the others may move onto the next steps. Step 7 of course can be executed multiple times and it’s expected that a federate could potentially execute it thousands of times during a federation execution depending on its role, the makeup of the federation and the length of the execution. Step 8 is generally called by the controlling federate which is determined by the designers of a federation execution and ends the federation execution.

It’s expected that steps 4 -8 will be repeated multiple times by an application after completing the initialization in steps 1 and 2.

## Initialization

Before using the Ambassador it must be initialized which is made very simple for the user. This is done simply be calling the Ambassador::getInstance() passing the federate name as the only parameter. This returns an Ambassador reference which the user can use throughout it’s interaction with Ambassador. Below is an example of the getInstance() call.

private static Ambassador ambassador;

…

ambassador = Ambassador.getInstance("MyFederateName");

Figure - Ambassador initialization

## Registration

Federates must register with MUTHUR which returns a registration handle. The registration handle uniquely identifies the federate with MUTHUR and is used in many of the calls to MUTHUR. The following is an example of a call to Ambassador::register() passing in an AmbassadorCallback to receive the response.

private static FederateRegistrationResponse rr;

…

ambassador.register(new AmbassadorCallback() {

public void onSuccess(IEvent event) {

if (event != null) {

if (event instanceof FederateRegistrationResponse) {

rr = (FederateRegistrationResponse) event;

String regID = rr.getRegistrationID();

}

}

}

public void onError(Exception e) {

System.out.println(e.getLocalizedMessage());

}

});

Figure - Federate Registration

Note that the response is received in the AmbassadorCallback::onSuccess() or AmbassadorCallback::onError method depending on the outcome.

## List Federation Execution Models

A federate can request a list of registered federation execution models (FEM) from MUTHUR. While a federate could construct a FEM manually it is recommended that you use those from MUTHUR to ensure consistency between federates when referring to a particular FEM. The following is an example of a call to retrieve the list of registered FEMs from MUTHUR.

private FederationExecutionModel fem;

…

ambassador.listFederationExecutionModels(getRegistrationHandle(),

new AmbassadorCallback() {

public void onSuccess(IEvent event) {

if (event != null) {

if (event instanceof ListFedExecModelsResponse) {

ListFedExecModelsResponse listOfFEMResponse = (ListFedExecModelsResponse) event;

Iterator<FederationExecutionModel> iter =

listOfFEMResponse.getFemList().iterator();

while (iter.hasNext()) {

FederationExecutionModel fem = iter.next();

if(fem.getName().equalsIgnoreCase("MyFEMName")){

System.out.println("Found my FEM.");

break;

}

}

}

}

}

public void onError(Exception e) {

System.out.println("Request for list of FEMs failed");

}

});

Figure - List Federation Execution Models

The example returns a ListFedExecModelsResponse object which contains a list of FEMs which is retrieved via a call to ::getFemList(). The example iterates over the list of FEMs and simply extracts the single FEM whose name matches and breaks out of the iteration.

## Joining a Federation

Once you have a FRH (federation registration handle) and a FEM (federation execution model) you can then join a federation. MUTHUR will return a FEH (federation execution handle) to be used in subsequent Ambassador calls requiring a FEH. The following show the FRH and FEM that was previously retrieved being used as arguments in the Ambassador::joinFederation() call. The FEH is retrieved from the JoinFederationResponse object in the AmbassadorCallback::onSuccess() by calling JoinFederationResponse::getFederationExecutionHandle().

private String federationExecutionHandle;

…

ambassador.joinFederation(federationRegistrationHandle, fem,

new AmbassadorCallback() {

public void onSuccess(IEvent event) {

if (event != null) {

if (event instanceof JoinFederationResponse) {

JoinFederationResponse jfResponse = (JoinFederationResponse) event;

federationExecutionHandle =

jfResponse.getFederationExecutionHandle();

}

}

}

public void onError(Exception e) {

System.out.println("Join failed");

}

});

Figure - Join federation

## Data Subscription

After joining the federation a federate should register interest in any data. Data that is published as instantiations, modifications and deletions of classes. The following example shows a federate registering for three classes – SpawnAircraft, FlightPositionUpdate and KillAircraft. Each instantiation, modification or deletions to these classes will be sent to this federate for appropriate processing.

List<String> classNames = new Vector<String>();

classNames.add("SpawnAircraft");

classNames.add("FlightPositionUpdate");

classNames.add("KillAircraft");

ambassador.addDataSubscriptions(regID,

federationExecutionHandle, classNames, new AmbassadorCallback() {

public void onSuccess(IEvent event) {

System.out.println("Subscriptions added.");

}

public void onError(Exception e) {

System.out.println("Error adding subscriptions.");

}

}, new DataEventReceiver());

/\*\*

\*

\*/

private class DataEventReceiver implements DataEventCallback {

public void onEvent(IDataObject dataObject) {

System.out.println("DataEventCallback.onEvent() called");

}

}

Figure - Data subscription request

Note that a new callback, DataEventCallback which is implemented in the class DataEventReceiver which will receive all subscriptions.

## Ready To Run

After the subscription are registered the federate should notify MUTHUR that it is ready to participate in the federation by calling Ambassador::readyToRun(). The following example shows this call which, if successful results in a call to the AmbassadorCallback::onSuccess(). If the call fails then the federation will not enter into the execution stage and is terminated.

ambassador.readyToRun(regID, federationExecutionHandle, new AmbassadorCallback() {

public void onSuccess(IEvent event) {

System.out.println("Ready to run request acknowledged.");

}

public void onError(Exception e) {

System.out.println("Ready to run request was not acknowledged");

}

}, new MySystemEventCallback());

/\*\*

\*

\*/

private class MySystemEventCallback implements SystemEventCallback {

public void onEvent(IEvent systemEvent) {

System.out.println("SystemEventCallback.onEvent() called");

}

}

Figure - Ready to run request

Also note that another callback is used as the SystemEventCallback as the fourth parameter, which will receive all the system notification sent from MUTHUR concerning this federation execution.

## Publishing Data

Federates can publish data by calling Ambassador::publishData() with an object derived from IBaseDataObject like SpawnAircraftData. The example below creates the data and publishes it to the federation by using the FEH. All federates subscribing to this class will receive this data update other than the publishing federate.

SpawnAircraftData dataObject = new SpawnAircraftData();

ambassador.publishData(regID,

federationExecutionHandle, dataObject);

Figure - Publishing data

## Federation Termination

To terminate the federation execution simply call the Ambassador::terminateFederation() with the FEH. This will initiate system messages to all participating federates that the federation execution is terminating and to proceed with clean up and shutdown.

ambassador.terminateFederation(federationExecutionHandle);

Figure - Federation termination event