**MUTHUR**

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

**Requirements Document**

**Edition 0.1**

**16-Feb-11**

|  |  |  |
| --- | --- | --- |
| **DOCUMENT CHANGE HISTORY** | | |
| **Edition** | **Date** | **Description of Change** |
| 0.1 | 16-Feb-11 | First draft |
|  |  |  |

**Contents**

1 Introduction 6

1.1 Purpose of this Document 6

1.2 Related Documents 6

1.3 Terms 7

1.4 Font and Diagram Conventions 7

1.5 Definition of MUTHUR 7

2 Requirements, Goals and Approaches 8

2.1 Architectural Qualities 8

3 MUTHUR System Design 9

3.1 High Level Architecture 9

3.2 MUTHUR Client 10

3.2.1 Registration 10

3.2.2 List of FEMs 12

3.2.3 Federation Execution State Controller 13

3.3 MUTHUR 14

3.3.1 Startup/Initialization 14

4 Parameter and Configuration Files 15

5 Preliminary Database Table Schemas 15

**Tables**

No table of figures entries found. **In your document, select the words to include in the table of contents, and then in the Formatting Palette under Styles, click a heading style. Repeat for each heading that you want to include, and then insert the table of contents in your document. You can also create a table of contents by clicking the Create with Manual Formatting option and then type the entries manually.**

**Figures**

**No table of figures entries found.**  
In your document, select the words to include in the table of contents, and then in the Formatting Palette under Styles, click a heading style. Repeat for each heading that you want to include, and then insert the table of contents in your document. You can also create a table of contents by clicking the Create with Manual Formatting option and then type the entries manually.

# Introduction

## Purpose of this Document

This document describes the design of the enhancements to the NexSim product and the implementation of the middle ware system referred to as MUTHUR.

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 design documentation make heavy use of this configuration especially in terms of the Federation Object Model (FOM), which describes the data model that is associated with any federation execution. The implementation supports the use of any FOM that meets the needs of the federation designers and the participating federates and is not tied in any way to the FOM used by the federation of the Frasca FTD and the Flight Operations lab.

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 is intended to be a high-level design document. Although in some areas design details are suggested, the primary goals of this document are to describe the functions of MUTHUR, identify the major modules and classes, and define the significant features of the class interfaces.

## Related Documents

This document is derived from the following documents, which are available in the MUTHUR document repository:

* NexSim Frasca Integration Proposal
* Draft SubK FFP Frasca - CSC 1-7-11
* MUTHUR - Requirements

## Terms

This section provides definitions for terms that will be used in this document.

OSGi – Open Service Gateway Initiative

Bundle –

## Font and Diagram Conventions

The following font convention is used in this document:

* **ExecutableProgram** (50% gray underline)
* **MUTHUR**
* **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.

# MUTHUR System Design

## High Level Architecture

There are two main components to MUTHUR – the core functionality that resides in the what is commonly referred to as middle ware and the client side API that allows clients or federates to interface with the middle ware and participate in federation executions.

The core or what we will refer to as MUTHUR in this document is the middle ware component and is responsible for

* Object model definition
* Federate registration
* Federation initialization
* Federation synchronization
* Object instantiation
* Object ownership
* Time management

MUTHUR runs as a service in the FUSE OSGi, container, which is an open source product, maintained by Progress software. FUSE’s OSGi foundation provides a number of technical capabilities that will help us meet our short and long term product requirements. It provides

* Modularity – an architecture that enforces a component based development approach
* Versioning – different versions of the same components can be deployed to the same container facilitating product enhancement and maintenance
* Deployment – components can be deployed into the container at run-time allowing services and capabilities to be added without re-compilation or even restarting the container or even an executing federation

In addition to its technical features it’s also the integration platform for the FAA (Federal Aviation Administration). It is clear that integration of existing systems rather than creating more monolithic, “stove pipe” applications is a major initiative within the FAA technical management. Whether its systems, simulator or tools the FAA is encouraging vendors to work with existing systems to integrate rather than duplicating efforts and FUSE has been chosen as the integration platform.

MUTHUR is composed of a set of bundles that are deployed to the FUSE container. All the services provided by MUTHUR are contained within this set of bundles and interact with the client side.

The client side or simply the client is a set of APIs (application programming interface) that shields federates from the complexity of interfacing with MUTHUR and allows federates to seamlessly participate in a federation execution. There are clients for Java and C++. The C++ is either a Windows DLL or shared library and the Java client is a jar file. For either the federate simply links the DLL or loads the jar into their simulator, application or tool. To participate within a federation the client uses the API and follows a well-documented protocol that is detailed later in this document.

The client and MUTHUR communicate with each using an asynchronous messaging protocol based on the JMS (Java Messaging Service) interface. This allows MUTHUR to effectively operate in an environment where it has no direct control over a federation’s participants and network availability and performance are unpredictable. It also provides a challenge in handling the asynchronous nature of messaging. One can never assume that a call will be completed and must take into account time outs. Results are returned after the caller has initially returned via callbacks or interface classes. Clients are shielded from much of this behavior but must still take the asynchronous nature of the messaging protocol into account when using the client API.

## MUTHUR Client

### Registration

The first thing that a client should complete is the registration with MUTHUR. The user supplies the federate name and a RegistrationCallback pointer. One should try to use unique federate names to help identifying processes and filter diagnostic and logging information. A GUID (Global Unique IDentifier) containing a timestamp is appended to the name to form a unique identifier. The timestamp can be used to sort or filter logging, database and other reporting data.

Like the vast majority of calls to MUTHUR this is asynchronous. The client provides a pointer a RegistrationCallback object, which implements the Callback interface. The call returns immediately with the results of the call, either success or error returned later via the callback. If the call succeeds the onSuccess() method is called and the onError() method is called if an error occurs. A timeout value is always provided in the callback, which is used to ensure that a response will ultimately be received. This is a common pattern repeated throughout the API.

Registration assumes that MUTHUR is running and has successfully completed the start up or initialization phase.

**Register** (FederateName, RegistrationCallback)

* Connect to JMS server
* Create session
* Create GUID (name + GUID) – include timestamp in GUID for sorting
* Get new Request\_ID from RequestIDFactory.create()
* Create the reply queue MUTHUR.GUID.RESPONSE.QUEUE for this federate
* Connect to MUTHUR.GUID.RESPONSE.QUEUE
* Create consumer on MUTHUR.GUID.RESPONSE.QUEUE
* Connect to MUTHUR.REGISTRATION.QUEUE
* Register consumer on MUTHUR.REGISTRATION.QUEUE
* Create single instantiation of RequestToCallbackMap
* Create RegistrationMessage object:
  + Name
  + GUID
  + CreateTimestamp (time that the message was created)
  + Request\_ID (from factory)
  + MUTHUR.GUID.RESPONSE.QUEUE (to be used by MUTHUR for responses)
  + Time out value (default value should always be assigned)
* Add Request\_ID, RegistrationCallback entry to RequestToCallbackMap
* Serialize RegistrationMessage to XML
* Put XML on MUTHUR.REGISTRATION.QUEUE

A JMS consumer should be created to consume messages on the response queue that are sent from MUTHUR to this federate. The queue name should be in the form MUTHUR.GUID.RESPONSE.QUEUE with GUID being formed from the federate name + GUID + current timestamp. The registration request should include the response queue name in the message, which MUTHUR will map to the GUID. All responses from MUTHUR to requests by this federate will be received by this federate in the consumer for this queue. The key method on any consumer is the onMessage(Message m) : void, which is called whenever a message is put on a queue where a consumer is attached.

The following is the pseudo code for the onMessage() method for the consumer attached to the MUTHUR.GUID.RESPONSE.QUEUE.

* **onMessage**(Message m)
  + Extract XML from Message
  + Create RequestProcessResult from XML
  + Get request ID from RequestProcessResult object
  + Search RequestToCallbackMap for request ID
  + If found
    - Get Callback pointer entry (removes) using request ID key
    - Create success message m
    - Call Callback.onSuccess(m)

Note that each Callback object implements the Callback interface as in the RegistrationCallback concrete class. The RegistrationCallback class implements both the onSuccess() and onError() methods with each receiving a pointer to a Message interface class. The Message object can be serialized in the form of a well-formed XML doc that describes the success or failure of the operation. The Callback and Message classes attempt to provide a generic approach to handling Callbacks.

The following is the pseudo code for the onSuccess () and onError () methods for the RegistrationCallback object passed to the register () method.

* **onSuccess**(Message m)
  + Signals that registration was successful and the join process can begin
* **onError**(Message m)
  + Error occurred during registration
  + Log and respond based on error type

RequestTimeoutWatchdog Thread

This is a background thread that is created at library load time and continually iterates over the list of requests every n number of seconds and removes any that have exceeded their allowed time out value. When a response is received at a federate from MUTHUR the request id is used from the response to remove the matching request id from the RequestToCallbackMap. If a matching request cannot be found then it’s assumed that the request has timed out. The following steps describe what this thread will do every configurable number of seconds.

* Iterate over the RequestToCallbackMap
* Get the next entry
* Check if timeout duration has been exceeded (now – time submitted)
* If timeout duration exceeded removed item from map
* Construct error Message m with timeout error code associated text
* Call Callback.onError(m)

### List of FEMs

Before joining a federation, a federate very likely would want to get a list of available FEMs (Federation Execution Models). Recall that FEMs describe attributes of a federation such as:

* Name of federate
* Logical Start Time
* Phase time outs
  + Join
  + Publish intentions
  + Subscription intentions
  + Housekeeping
* Required federates
  + Name of federate
  + Time management role (regulating, non-regulating or both)
* Federated Object Model

**GetFederationExecutionModels**(Callback\*)

Returns the list of FEMs that are currently available via this instance of MUTHUR in the onSuccess() method of the Callback implementation. In the onSuccess(Message m) method of the Callback the **ObjectFactory::create(Message)** is called to create the correct object and populated with the attributes from the Message. A FEM from this list will be used in the **FederationExecutionStateController::Join**() method that initiates the federation execution process towards the execution phase and finally synchronizes shutdown and clean-up.

### Federation Execution State Controller

It is expected that clients follow a set of steps in order to participate as a federate in the execution of a federation. While it’s possible to do these as single calls to MUTHUR through the client the Federation Execution State Controller (FESC) provides a single class that encapsulates the calls and routes the responses to the next step in the process. If an error occurs than the error handler is called and the client handles it in the appropriate manner.

While it’s not required, it’s anticipated that there will be some level of coordination between the different federates as to when the federation execution will occur or be started. The FESC allows this coordinate to be as loose as necessary as federates can join when they are ready and MUTHUR will ensure that each federate has joined and communicate that to the FESC which will then proceed to the next step.

The steps that the FESC takes the user through are:

1. Join – federate is attempting to join a federation defined by a FEM.
2. Publication Intentions – given the FOM associated with the FEM a list of classes that this federate.
3. Subscription Intentions – the federate publishes all the classes and/or attributes that it is interested in getting notifications when objects of these classes are created, modified or deleted.
4. Ready To Execute – federate is now able to begin its execution and participate in the federation execution.
5. End Condition Signaled – the ending condition for the federation was signaled by a federate. Each federate has the opportunity to perform housekeeping and house cleaning.
6. Termination – the federation execution has terminated.

The process starts with the call to the FESC::Join() method passing in a Federation Execution Model returned from a call to GetFederationExecutionModels(). All of the required federates listed in the FEM must complete the join process before the FESC will return from the join call and calls the next step in the process – publication intentions. PublicationIntentions(ListofClasses) takes a list of classes that will be published by this federate from the FOM. Once all federates publication intentions have been processed then MUTHUR returns a response to all federates and the SubscriptionIntentions(ListofClasses) is called by the FESC. Once the all federate subscription intentions have been received and registered then the ReadyToExecute() method is called and the federate can begin participating in the federation execution. At some time during the execution a federate will signal or raise the ending condition and MUTHUR will call the EndConditionSignaled() method. Federates can then begin shutdown and housekeeping chores. Once those are complete Terminate() method is called which signals that the federate has terminated and the federation will be cleaned up by MUTHUR.

## MUTHUR

### Startup/Initialization

One of the requirements of MUTHUR’s start up sequence is for the creation of the set of common JMS destinations used for receiving requests including registration. These steps are coordinated with the JMS service provided by the ActiveMQ service resident in the FUSE server, which must be deployed and configured. This configuration will be part of the standard deployment for MUTHUR.

Startup/Initialization

* Create RegistrationManager instance
* Start JMS server
* Connect to server
* Create session
* Create MUTHUR.REGISTRATION.QUEUE
* Connect MUTHUR.REGISTRATION.QUEUE
* Create consumer on MUTHUR.REGISTRATION.QUEUE
* Create MUTHUR.REQUEST.QUEUE
* Create consumer on MUTHUR.REQUEST.QUEUE

Consumer implementation for MUTHUR.REGISTRATION.QUEUE

* onMessage(Message m)
  + Create RegisterUser object from message
  + Add to RegistrationManager
  + Extract name of user response queue name created by user (<federate name>.<guid>.RESPONSE.QUEUE)
  + Create producer for user response queue
  + Create REGISTRATION\_RESPONSE message
  + Serialize REGISTRATION\_RESPONSE message to XML
  + Put XML on user response queue

Consumer implementation for MUTHUR.REQUEST.QUEUE

* onMessage(Message m)
  + Call MessageFactory.create(m) to create the appropriate MessageObject
  + Call RequestProcessor.process(MessageObject) returning RequestProcessResult
  + Serialize RequestProcessResult to XML
  + Put XML on (<federate name>.<guid>.RESPONSE.QUEUE

# Parameter and Configuration Files

[TBD]

# Preliminary Database Table Schemas