

1. Programmers Guide 6.0	3
1.1 Application Development	7
1.1.1 TENA Naming and Versioning Conventions	8
1.1.2 Obtaining TENA Software	12
1.1.2.1 TENA Installer Overview	13
1.1.3 Application Code Layout Considerations	14
1.1.3.1 TENA Software Directory Structure	15
1.1.4 Application Considerations	17
1.1.4.1 Application Resignation	18
1.1.4.2 Attribute Timestamps	20
1.1.4.3 Middleware Provided Boost and ACE-TAO Support	21
1.1.4.4 Multiple Version Support	23
1.1.4.5 Runtime Exception Handling	26
1.1.4.6 Smart Pointers	30
1.1.5 Example Applications	33
1.1.5.1 Arbitrary Values	35
1.1.5.2 Environment Variables	37
1.1.5.3 Example Application Configuration Options	38
1.1.6 Building Example Applications	39
1.1.7 Running Example Applications	40
1.1.8 Log Files	42
1.1.9 Distributing TENA Applications	43
1.1.10 Application Object Model Linking	46
1.1.11 TENA in MFC Applications	48
1.1.12 API Naming Conventions	50
1.1.13 Using TENA in a Library	51
1.2 Event Management	56
1.2.1 Alerts	57
1.2.2 Startup Services	59
1.2.3 TENA Console	60
1.3 Execution Manager	61
1.3.1 EM Configuration Parameters	64
1.3.2 EM Fault Tolerance	72
1.3.3 EM Recovery	75
1.3.4 Multicast Support	77
1.3.5 OM Consistency Checking	79
1.3.5.1 Type Registry	81
1.3.6 Running EM as a Service	83
1.4 Glossary	86
1.5 Local Class	92
1.5.1 Accessing Application Data from a Local Class	96
1.5.2 Local Class Constructor	99
1.5.3 Local Class Implementation Registration	103
1.5.4 Local Class Inheritance and Polymorphism	109
1.5.5 Using Local Method with SDO Pointer Argument	111
1.6 Message	114
1.6.1 Accessing Application Data from a Message	117
1.6.2 Adding Message Support to Existing Application	120
1.6.3 Message Constructor	126
1.6.4 Message ID	130
1.6.5 Message Implementation Registration	131
1.6.6 Message Inheritance and Polymorphism	137
1.6.7 Message Sender	140
1.6.7.1 Message Communication Properties	142
1.6.8 Message Subscription	143
1.6.9 Message Unsubscribe	150
1.7 Middleware Introduction	152
1.8 Middleware Services	155
1.8.1 Advanced Filtering	156
1.8.1.1 Filter	160
1.8.1.2 Tag	162
1.8.1.2.1 Computing Tag based on State Update	165
1.8.2 Configuration Mechanism	167
1.8.2.1 Middleware Configuration Parameters	181
1.8.2.1.1 Recommended Parameters for Working LAN Environment	187
1.8.2.2 Network Address	189
1.8.3 Execution Management Services	191
1.8.4 Middleware IDs	197
1.8.5 Middleware Metadata	201
1.8.5.1 Application-Specific Clock	208
1.8.6 Middleware Threading Model	212
1.8.7 Publication Services	213
1.8.7.1 Best Effort Transport	215
1.8.7.1.1 Multicast Address Assignment	217
1.8.7.2 Reliable Transport	219
1.8.8 Subscription Services	221
1.8.8.1 Best Match	225
1.8.8.2 Callback Framework	230
1.8.8.3 Object Model Subsetting	233

1.8.8.4 Observer Mechanism	234
1.8.8.4.1 Reusable Observers	238
1.8.8.5 Subscription Slicing	244
1.9 Object Models	245
1.9.1 Meta-Model	247
1.9.1.1 Fundamental Meta-Model Types	249
1.9.1.2 Class Meta-Model Type	250
1.9.1.3 Local Class Meta-Model Type	252
1.9.1.4 Message Meta-Model Type	254
1.9.1.5 Attribute Meta-Model Construct	256
1.9.1.5.1 Const Attribute Qualifier	257
1.9.1.5.2 Optional Attribute Qualifier	259
1.9.1.5.3 Readonly Attribute Qualifier	261
1.9.1.5.4 Private Attribute Qualifier	262
1.9.1.6 Operation Meta-Model Construct	263
1.9.1.6.1 Const Operation Qualifier	264
1.9.1.6.2 Oneway Operation Qualifier	265
1.9.1.7 Inheritance Meta-Model Support	266
1.9.1.8 Class Pointer Meta-Model Construct	267
1.9.1.9 Remote Method Exception Meta-Model Construct	268
1.9.1.10 Enumeration Meta-Model Construct	269
1.9.1.11 Vector Meta-Model Construct	270
1.9.1.12 Package Meta-Model Construct	271
1.9.1.13 Import Meta-Model Construct	272
1.9.1.14 Package Scoped Constant Meta-Model Construct	273
1.9.2 TENA Definition Language	274
1.9.2.1 UML Support	277
1.9.3 Object Model Compiler Overview	278
1.9.3.1 Uploading Object Models	279
1.9.3.2 Installing Object Models	282
1.9.4 Object Model Definition	283
1.9.5 Object Model Implementation	284
1.9.5.1 Sharing Object Model Implementations	285
1.9.6 Example Object Models	288
1.9.7 TENA Standard Object Models	291
1.9.7.1 Orientation Conventions	293
1.9.8 TENA Candidate OM Process	295
1.10 Programmers Guide 6.0.1 Addendum	299
1.11 Programmers Guide 6.0.2 Addendum	300
1.12 Programmers Guide 6.0.3 Addendum	301
1.13 Programmers Guide 6.0.4 Addendum	303
1.14 Programmers Guide 6.0.5 Addendum	305
1.15 Programmers Guide 6.0.6 Addendum	310
1.16 Programmers Guide 6.0.7 Addendum	311
1.17 Programmers Guide 6.0.8 Addendum	319
1.18 Stateful Distributed Object	321
1.18.1 Adding SDO Support to Existing Application	325
1.18.2 SDO Creation	332
1.18.2.1 SDO Communication Properties	336
1.18.2.2 SDO Concurrency Properties	338
1.18.2.3 SDO Initializer	339
1.18.2.4 SDO Servant	341
1.18.3 SDO Destruction	345
1.18.4 SDO ID	347
1.18.5 SDO Inheritance and Polymorphism	348
1.18.6 SDO Pointer	351
1.18.7 SDO Reactivation	357
1.18.8 SDO Remote Method	360
1.18.8.1 Accessing Application Data from a Remote Method	365
1.18.9 SDO Subscription	367
1.18.9.1 SDO Proxy	375
1.18.9.1.1 Discovery Order	378
1.18.9.1.2 Received State Processing	379
1.18.10 SDO Unsubscribe	381
1.18.11 SDO Update	384

Programmers Guide 6.0

Static Documentation Warning

This document was statically generated based on modification date of: 2022-09-12. The latest version of this document can be found here: <https://www.trmc.osd.mil/wiki/display/MW/Programmers+Guide+6.0>

TENA Middleware Programmer's Guide 6.0

Preface

The Purpose of this Programmer's Guide

This TENA Middleware Programmer's Guide describes Release 6 of the TENA Middleware, a fundamental component of the Test and Training Enabling Architecture (TENA). This guide examines the software interface and operation in considerable detail. Experienced developers should find the TENA Middleware Programmer's Guide, along with the companion TENA Hands-On Training course, sufficient to begin using TENA. System architects and designers making existing range software operate with TENA or planning new TENA-compliant range software should find this guide helpful for identifying and assessing important issues.

Readers' Presumptions and Expectations

The intended readers of this guide are the software engineers, systems engineers, and programmers who will use the TENA Middleware and integrate it with range applications. Readers are presumed to have experience with the general software concepts associated with TENA. Readers are also encouraged to review project background material, such as those documents listed below.

- [FI 2010 Joint Overarching Requirements Document \(JORD\)](#)
- [FI 2010 Technical Capabilities Requirements Document \(TCRD\)](#)
- [TENA Architecture Reference Document](#)
- [TENA Training Briefings](#)
- [TENA Briefings and Papers](#)
- [TENA Architecture Management Team \(AMT\) Proceedings](#)

General information regarding TENA and the TENA Software Development Activity (TENA SDA) activity can be sent to the feedback email address (feedback@tena-sda.org).

Document Versions and Refinements

Version 6.0 of the TENA Middleware Programmer's Guide is applicable to all 6.0.x versions of the TENA Middleware. Where necessary, middleware version specific comments are included in the overall guide. Additionally, addendum pages for the different middleware releases are provided to explain any relevant changes and new features unique to a particular version of the middleware.

Although this document has been reviewed for correctness and clarity, it is anticipated that future refinements and/or corrections will be necessary. The document associated with the latest release of the middleware will be updated or expanded as necessary. Once a new version of the middleware is released, the corresponding documentation of the older releases will be archived. All of the middleware documentation versions can be accessed from [Middleware Documentation Page](#). A complete PDF of the Programmer's Guide can be downloaded from this page: [Complete Programmers Guide](#).

Any comments, corrections, or questions concerning the Programmer's Guide can be directed to the TENA web-based helpdesk system, <http://www.tena-sda.org/helpdesk/browse/MW>. User feedback will help ensure that the documentation continues to address the needs of the community.

Related Documents

There are a number of related documents associated with the TENA Middleware that are listed below.

- [TENA Middleware Release Notes](#)
- [TENA Middleware Installation Guide](#)
- [TENA Middleware API Guides](#)
- [TENA Training Material](#)

TENA Middleware C++ API Guide

The TENA Middleware API (application programmer interface) guide is intended to support application developers with web-based access to the files associated with the middleware programming interface. These files are primarily programming header files which are automatically processed (using the [doxygen](#) product) to produce linked web pages.

- [Release 6.0.9 Middleware C++ API Guide](#)

How to Create a Local Copy of the Middleware API Guide

1. Check that `doxygen` is installed
 - From the shell type: `doxygen -v`
2. Check that `graphviz` is installed
 - From the shell type: `dot -V`
3. source the TENA environment script, `$(TENA_HOME)/$(TENA_VERSION)/scripts/tenaenv-$(TENA_PLATFORM)-v$(TENA_VERSION).sh`
 - E.g., source `~/Applications/TENA/6.0.9/scripts/tenaenv-macos11-clang12-v6.0.9.sh`
4. Run make in `$(TENA_HOME)/$(TENA_VERSION)/doc/doxygen`
 - `make -C $(TENA_HOME)/$(TENA_VERSION)/doc/doxygen all`

Programmers Guide Pages

Search

- Application Development
 - [TENA Naming and Versioning Conventions](#)
 - [Obtaining TENA Software](#)
 - [TENA Installer Overview](#)
 - [Application Code Layout Considerations](#)
 - [TENA Software Directory Structure](#)
 - [Application Considerations](#)
 - [Application Resignation](#)
 - [Attribute Timestamps](#)
 - [Middleware Provided Boost and ACE-TAO Support](#)
 - [Multiple Version Support](#)
 - [Runtime Exception Handling](#)
 - [Smart Pointers](#)
 - [Example Applications](#)
 - [Arbitrary Values](#)
 - [Environment Variables](#)
 - [Example Application Configuration Options](#)
 - [Building Example Applications](#)
 - [Running Example Applications](#)
 - [Log Files](#)
 - [Distributing TENA Applications](#)
 - [Application Object Model Linking](#)
 - [TENA in MFC Applications](#)
 - [API Naming Conventions](#)
 - [Using TENA in a Library](#)
- Event Management
 - [Alerts](#)
 - [Startup Services](#)
 - [TENA Console](#)
- Execution Manager
 - [EM Configuration Parameters](#)
 - [EM Fault Tolerance](#)
 - [EM Recovery](#)
 - [Multicast Support](#)
 - [OM Consistency Checking](#)
 - [Type Registry](#)
 - [Running EM as a Service](#)
- Glossary
- Local Class
 - [Accessing Application Data from a Local Class](#)
 - [Local Class Constructor](#)
 - [Local Class Implementation Registration](#)
 - [Local Class Inheritance and Polymorphism](#)
 - [Using Local Method with SDO Pointer Argument](#)
- Message
 - [Accessing Application Data from a Message](#)
 - [Adding Message Support to Existing Application](#)
 - [Message Constructor](#)
 - [Message ID](#)
 - [Message Implementation Registration](#)
 - [Message Inheritance and Polymorphism](#)
 - [Message Sender](#)
 - [Message Communication Properties](#)
 - [Message Subscription](#)

- Message Unsubscribe
- Middleware Introduction
- Middleware Services
 - Advanced Filtering
 - Filter
 - Tag
 - Computing Tag based on State Update
 - Configuration Mechanism
 - Middleware Configuration Parameters
 - Recommended Parameters for Working LAN Environment
 - Network Address
 - Execution Management Services
 - Middleware IDs
 - Middleware Metadata
 - Application-Specific Clock
 - Middleware Threading Model
 - Publication Services
 - Best Effort Transport
 - Multicast Address Assignment
 - Reliable Transport
 - Subscription Services
 - Best Match
 - Callback Framework
 - Object Model Subsetting
 - Observer Mechanism
 - Reusable Observers
 - UniqueIDMapper
 - Subscription Slicing
- Object Models
 - Meta-Model
 - Fundamental Meta-Model Types
 - Class Meta-Model Type
 - Local Class Meta-Model Type
 - Message Meta-Model Type
 - Attribute Meta-Model Construct
 - Const Attribute Qualifier
 - Optional Attribute Qualifier
 - Readonly Attribute Qualifier
 - Private Attribute Qualifier
 - Operation Meta-Model Construct
 - Const Operation Qualifier
 - Oneway Operation Qualifier
 - Inheritance Meta-Model Support
 - Class Pointer Meta-Model Construct
 - Remote Method Exception Meta-Model Construct
 - Enumeration Meta-Model Construct
 - Vector Meta-Model Construct
 - Package Meta-Model Construct
 - Import Meta-Model Construct
 - Package Scoped Constant Meta-Model Construct
 - TENA Definition Language
 - UML Support
 - Object Model Compiler Overview
 - Uploading Object Models
 - Installing Object Models
 - Object Model Definition
 - Object Model Implementation
 - Sharing Object Model Implementations
 - Example Object Models
 - TENA Standard Object Models
 - Orientation Conventions
 - TENA Candidate OM Process
- Programmers Guide 6.0.1 Addendum
- Programmers Guide 6.0.2 Addendum
- Programmers Guide 6.0.3 Addendum
- Programmers Guide 6.0.4 Addendum
- Programmers Guide 6.0.5 Addendum
- Programmers Guide 6.0.6 Addendum
- Programmers Guide 6.0.7 Addendum
- Programmers Guide 6.0.8 Addendum
- Stateful Distributed Object
 - Adding SDO Support to Existing Application
 - SDO Creation
 - SDO Communication Properties
 - SDO Concurrency Properties
 - SDO Initializer
 - SDO Servant
 - SDO Destruction
 - SDO ID

- [SDO Inheritance and Polymorphism](#)
- [SDO Pointer](#)
- [SDO Reactivation](#)
- [SDO Remote Method](#)
 - [Accessing Application Data from a Remote Method](#)
- [SDO Subscription](#)
 - [SDO Proxy](#)
 - [Discovery Order](#)
 - [Received State Processing](#)
- [SDO Unsubscribe](#)
- [SDO Update](#)

Previous Versions of the Middleware API Guide

- [Release 6.0.8 Middleware C++ API Guide](#)
- [Release 6.0.7 Middleware C++ API Guide](#)
- [Release 6.0.6 Middleware C++ API Guide](#)
- [Release 6.0.5.1 Middleware C++ API Guide](#)
- [Release 6.0.5 Middleware C++ API Guide](#)
- [Release 6.0.4 Middleware C++ API Guide](#)
- [Release 6.0.3 Middleware C++ API Guide](#)
- [Release 6.0.2 Middleware C++ API Guide](#)
- [Release 6.0.1 Middleware C++ API Guide](#)

Application Development

Application Development

Collection of topics related to the development of TENA applications.

Application Development Topics

- [TENA Naming and Versioning Conventions](#) — Description of the conventions used for naming and versioning TENA software.
- [Obtaining TENA Software](#) — Explains how to obtain the TENA Middleware.
- [Application Code Layout Considerations](#) — Provides guidance on how application developers can arrange their software files with respect to the TENA software structure.
- [Application Considerations](#) — Collection of application consideration topic pages.
- [Example Applications](#) — Example applications are automatically generated using the particular object model of interest to illustrate the operation of the middleware in conjunction with the object model constructs.
- [Building Example Applications](#) — Overview on building auto-generated example applications.
- [Running Example Applications](#) — Overview on running auto-generated example applications.
- [Log Files](#) — TENA applications create a log file (unless disabled) whenever the application encounters an exception (i.e., error, warning).
- [Distributing TENA Applications](#) — Provides guidance for deploying TENA applications to be used on other computer systems.
- [Application Object Model Linking](#) — Information regarding application linking with object models.
- [TENA in MFC Applications](#) — Guidance related to the use of TENA in MFC applications.
- [API Naming Conventions](#) — Description of the conventions used for naming software elements that are part of the middleware API.
- [Using TENA in a Library](#) — Guidance related to the use of TENA in a separate library that is loaded by the main program.

TENA Naming and Versioning Conventions

TENA Naming and Versioning Conventions

The TENA Middleware and related software products follow a naming and versioning convention to properly communicate information about the product that is needed by the user to ensure proper operation. This information includes a descriptor for the computer platform associated with the software product and the version number. The versioning of the middleware and other products is used to indicate interoperability relationships of the different versions.

Description

Computer Platform Descriptors

A compact text string is used to represent the particular computer operating system and compiler used for the particular software product distribution. The version of the operating system and compiler is included in the platform descriptor. All of the computer platform and compiler combinations supported by the TENA software are shown in the corresponding [Installation Guide](#) for the particular middleware release. Examples of these platform descriptors are `rh-e15-gcc41-64` and `w7-vs2010`. Note that for each platform descriptor there may be a similar descriptor with a `-64` for 64-bit architectures and a `-d` for the version built with debugging information.

 — Users are recommended to use 64-bit versions of the middleware (as opposed to 32-bit versions) where available due to the increased performance. Additionally, the debugged versions should only be used when actively debugging application code because the debugged code can be several orders of magnitude slower, especially with the Windows operating systems.

Middleware Naming and Versioning

The middleware version follows the three number convention `I.J.K`. The first number, `I`, represents major releases of the middleware and typically involves API (Application Programming Interface) changes. The second number, `J`, represents runtime compatibility – meaning that middleware versions with the same first two numbers (`I.J`) will interoperate in a common execution. The third number, `K`, represents a minor middleware change.

Upgrading existing applications to a new middleware version with the same first number, but different second and/or third numbers will only require the application software to be re-compiled and re-linked with the new middleware version.

The TENA Middleware distribution names follow the convention shown in the panel below. Platform descriptors are unique to the particular middleware version, as platforms are added and removed based on user community needs.

The UNIX distributions primarily use the suffix `.bin`, Apple Mac distributions use the suffix `.command`, and the Windows distributions use the suffix `.exe`. These distributions use the [TENA Installer](#) to create a self-extracting installation with graphical user interface support.

Middleware Versions

Middleware Version = I.J.K, where

- I : Major release, ensures API interoperability,
- J : Ensures runtime interoperability,
- K : Minor release

Middleware Distribution Naming

The notation for middleware distribution file names is:

```
TENA-MiddlewareSDK-v<middlewareVersion>.<packageIdentifier>@Product@<platformDescriptor>.{bin,exe,command},
```

where,

<middlewareVersion>	The three number version identifier of the middleware.
<packageIdentifier>	Capital letter to differentiate different package contents because the middleware package contains more than just the middleware (e.g., TENA Console, standard object models).
<platformDescriptor>	Text string representing computer platform.

Examples:

- TENA-MiddlewareSDK-v6.0.4.A@Product@rhel5-gcc41-64-d.bin
- TENA-MiddlewareSDK-v6.1.2.G@Product@w7-vs2012.exe

Note that this naming convention has been changed with release 6.0.4 of the middleware. The previous middleware distributions used a date notation instead of the packageIdentifier, but this caused confusion because of incorrect implied meaning of the date.

Object Model Naming and Versioning

Object model names are organized according to the owning User Group to avoid name collisions of the object model names or model components. User Groups represent organizations, projects, or activities. Additional information on User Groups can be found on the [User Groups page](#).

The name of the object model must begin with the User Group name. This applies to the TENA Definition Language (TDL) file name and all of the object model software distribution file names. Since a User Group may have multiple object models, the name of the particular object model will follow the User Group name, with a hyphen ('-') used as a separator. For example, consider the User Group named "Acme" with an object model named "Rocket", which would combine for an object model name "Acme-Rocket".

There are no TENA constraints on the length limitations of object model names, but since these names are used in the auto-generated software, length limitations may be encountered with operating systems, compilers, and other tools — most notably Microsoft products. Therefore, users are encouraged to use short, but descriptive names. The TENA project utilizes a "toggle-case" convention for naming in which the first character of separate words is capitalized, and acronyms use a common case (either all upper case or all lower case), e.g., "GPSguidance".

Object model definitions may evolve over time, so keeping track of different versions is important to ensure interoperability between applications using the same object model. The object model naming conventions for versioning is to append the object model name with a "-v" separator followed by the version identifier. The version identifier is typically a numerical counter (i.e., -v1, -v2), although users may want to use multiple numbers to define additional meaning (e.g., -v2.3.1, -v2.4).

There are two components associated with object models: definition and implementation. An object model definition contains the necessary code for an application to use the object model, allowing an application to publish and subscribe to the types defined by the object model. An object model implementation contains common code associated with constructors and methods from Local Classes or Messages defined in the object model. For example, the `TENA::TSPI` object model contains a number of Local Classes that perform coordinate conversions, and an object model implementation is required for applications using this object model.

Since multiple object model implementations may exist for the same object model definition, the object model implementations require unique implementation names. Furthermore, the object model implementations may evolve over time and versioning is required. In some cases an object model will not have any implementation code and only a definition will exist.

The naming and versioning convention for the object model related products are summarized below.

Object Model Naming

Object Model definition naming convention is:

```
<UserGroup>-<objectModelName>-v<defVersion>,
```

where

<UserGroup>	Name of the owning User Group.
<objectModelName>	Name of particular object model.
<defVersion>	User defined version identifier for the object model definition.

Example:

- Acme-Rocket-v2.3.tdl.

Object Model implementation naming convention is:

<UserGroup>-<objectModelName>-v<defVersion>-<implName>-<implVersion>,

where

<UserGroup>	Name of the owning User Group.
<objectModelName>	Name of particular object model.
<defVersion>	User defined version identifier for the object model definition.
<implName>	Name of particular object model implementation.
<implVersion>	User defined version identifier for the object model implementation.

Example:

- libAcme-Rocket-v2.3-GPSguidance-v3.2.1-osx107-gcc42-v6.0.3.1.dylib.

Object Model Distribution Naming

The software distributions associated with object models are built for a particular computer platform, and the distribution file names include the platform descriptor. Additionally, the object model distributions are dependent on the particular version of the TENA Middleware used for building the software. These distributions will include the object model implementation if one exists.

There are object model distributions available for different programming languages. The default distribution is for the C++ programming language, although there is support for the Java programming language and the Microsoft .NET programming languages. Each programming language distribution will include a different distribution identifier in the file name, as shown below.

C++ Distribution:

<UserGroup>-<objectModelName>-Distribution-v<defVersion>@Product@<platformDescriptor>-v<middlewareVersion>. {bin,exe,command}

Java Distribution:

<UserGroup>-<objectModelName>-JavaDistribution-v<defVersion>@Product@<platformDescriptor>-v<JavaBindingVersion>.MW<middlewareVersion>. {bin,exe,command}

.NET Distribution:

<UserGroup>-<objectModelName>-DotNetDistribution-v<defVersion>@Product@<platformDescriptor>-v<DotNetBindingVersion>.MW<middlewareVersion>. {bin,exe,command}

where

<UserGroup>	Name of the owning User Group.
<objectModelName>	Name of particular object model.
<defVersion>	User defined version identifier for the object model definition.
<platformDescriptor>	Text string representing computer platform.
<middlewareVersion>	The three number version identifier of the middleware.
<JavaBindingVersion>	Version identifier for the Java language binding.
<DotNetBindingVersion>	Version identifier for the .NET language binding.

Example:

- Acme-Rocket-Distribution-v2.3@Product@xp-vc80-v6.0.4.exe
- Acme-Rocket-JavaDistribution-v2.3@Product@xp-vc80-v1.0.3.MW6.0.4.exe

Note that there is no separate version identifier for the C++ language binding as it is currently coupled with the middleware release.

⚠ Only visible to MW-team because needs to be updated and it needs to discuss candidate OM modifiers. The above needs to be reviewed and updated based on what actually gets implemented with 6.0.4.

Version Modifiers

When using a beta version of the middleware, it is useful to not have to increment the version identifier of any object models that are being upgraded simultaneously until the beta testing has been completed. In this situation a version modifier can be used for the object model file names, so that after testing is completed the object model version number does not have to be unnecessarily incremented.

The version modifier is a character string, corresponding the beta version of the middleware, that is inserted between the "-v" and the object model version identifier. For example, the object model file `Acme-Rocket-v3.tdl` would become `Acme-Rocket-v*RC.1-*3.tdl` for "RC.1" (Release Candidate 1) of the middleware. Once the beta testing is completed, the object model file can be renamed `Acme-Rocket-v3.tdl`.

ℹ — Note that only the object model TDL file name needs to include the version modifier. Any import statements involving object models with version identifiers do **not** need to be changed to include the version modifier.

Obtaining TENA Software

Obtaining TENA Software

Registered users of the TENA website can download TENA Middleware distributions from the [TENA Repository](#).

Description

Only registered users of the TENA website can download and use the TENA Middleware. Users can submit account requests from the main web page: <http://www.tena-sda.org>. The approval process requires a manual review of the account request and may take several days to process. An email will be delivered to the requester when the account is approved.

Once logged into the TENA website with an approved account, users can navigate to the TENA Repository (<https://www.tena-sda.org/repository>) and select the "View Middleware" button. Under the middleware page, the menu allows users to select the particular middleware version of interest. The resulting page will display all of the available middleware distribution files with download buttons to obtain the particular distribution.

The middleware distribution files are packaged with the [TENA Installer](#) and include multiple components. In addition to the middleware, the distribution package contains all of the [TENA Standard Object Models](#), [Example Object Models](#), and [TENA Console](#). Executing the distribution file will (with default settings) start a GUI window that allows the user to specify the installation location and deselect any components to be installed. Under Windows operating systems, it is recommended to install the TENA software at the root directory, "C:\", due to path limitation issues that can occur with Microsoft products.

Users are requested to follow the appropriate installation guide for details regarding the proper installation and testing of the TENA Middleware (see [Middleware Installation Guides](#)).

TENA Installer Overview

TENA Installer Overview

The TENA Installer is used to package software components into installable images. For the TENA project, these software components include the TENA Middleware, Object Model definitions and implementations, and other utilities such as the TENA Console. Users are able to use the TENA Installer for distributing their application software, along with TENA components if appropriate. The TENA Installer provides a graphical user interface to support the installation of software components across the computer platforms supported by the TENA project. For details on the TENA Installer, see the documentation at the [TENA Installer space](#).

Description

The TENA Installer is a utility that can be used for bundling zip files into a self-extracting executable that will install the contents of the zip files and optionally run post-install scripts. Each software component, represented by its zip file, can specify a default installation location for the contents of the zip file. This allows a use to bundle TENA Middleware components, such as object model distributions, along with application code where the TENA software components are installed into the default `TENA_HOME` location and the application code can be installed in a separate location.

Each software component can define a post-install script (written in the Python programming language) that is executed after the software has been installed. Typical post-install script behavior includes configuring user environment variables or creating shortcuts/links.

A graphical user interface assists the user in changing the default installation location for the software components and selecting what software components should be installed. The TENA Installer can create self-extracting installation images across all of the computer platforms support by the TENA project.

Additional information and usage details on the TENA Installer can be found at the [TENA Installer space](#).

Application Code Layout Considerations

Application Code Layout Considerations

TENA application developers are able to arrange their software files according to their particular needs. The auto-generated example applications, based on a particular object model (OM), are installed in the TENA directory structure under `<TENA HOME>/<version>/src/<OM name>`. Users can create sub-directories under the `src` directory following the example application code layout for their applications if convenient, or a completely alternative location and application software structure can be used.

Description

The TENA software is installed onto a computer file system according to a well defined directory structure, see [TENA directory structure page](#) for a description of this structure. A component of the TENA software is the auto-generated example applications that are based on a particular object model. The software files for these example applications are installed under the `<TENA HOME>/<version>/src/<OM name>` directory and follow a specific structure for the various files associated with the example applications.

TENA application developers are able to work directly in the `src` directory structure, including modifying the auto-generated example application files to accommodate the particular user application requirements. The [TENA Installer](#) will automatically detect modified files in the `src` directory to prevent a re-installation from overwriting files that a user has modified.

Users are also able to develop their TENA applications in other file system locations that are independent of the TENA installation location. When building TENA applications, developers are strongly encouraged to build and link against the TENA middleware and object model files in the TENA installation directory structure, versus attempting to copy the TENA files into an alternative directory structure or following different naming conventions. Information about building the auto-generated example applications can be found on the [Building Example Applications page](#).

Usage Considerations

The directory/file layout and naming was done to support the simultaneous installation of different versions and different platforms (i.e., computer operating system, operating system version, compiler, and compiler version) into a common file system. Note that the use of the automatically generated [Example Application](#) build files (e.g., Visual Studio files on Windows, Makefiles on UNIX) require the user to change environment variable values to switch between different versions or platforms (see the [Build File Environment Variables documentation page](#) for additional information).

Current (and future) TENA products are expected to operate more effectively using the generated directory/file layout and naming conventions. Users considering to alter this directory structure will need to evaluate the requirements for such changes and the consequences. Contact the [TENA helpdesk](#) (<https://www.tena-sda.org/helpdesk/>) for additional information and/or guidance.

TENA Software Directory Structure

TENA Software Directory Structure

When the TENA Middleware and related products are installed on a computer's file system, the various software files are installed according to a directory structure designed to support multiple versions and various computer platforms (operating system, compiler). The files are organized to support activities associated with building and running TENA products and user developed TENA applications. Users can build and run applications using the different middleware versions and platform types contained within this directory structure. TENA utilities rely on this directory structure to manage software dependencies and users are encouraged to preserve this TENA directory structure.

Description

The TENA Middleware and related products are installed in a common directory structure used to support effective building and running of TENA applications. The "root" of the directory structure (referred to as "`TENA_HOME`") can be located anywhere on the filesystem, although for the Microsoft operating systems it is recommended to install the TENA software at the top-level directory (e.g., C:) due to path length limitations that are common with Microsoft products.

Top level directories under the `TENA_HOME` location are defined in the following table. The variable `<PLATFORM>` is used to indicate the descriptor for the computer platform, e.g., "xp-vc80", "fc6-gcc41". `<MW_VERSION>` is used to define the particular middleware version, e.g., "6.0.0", "5.2.2".

TENA Installation Directory Structure

Directory Name	Description
<code>all/bin /<PLATFORM></code>	Contains middleware version independent files, currently limited to binary files, such as the TENA Installer.
<code>components</code>	Collection of component descriptor files that describe the various TENA software components that have been installed within this directory structure. Various installation and build tools utilize these files and should not be modified.
<code>lib</code>	Common directory for software libraries used for various TENA products and versions. A common directory is used to simplify the setting of <code>LD_LIBRARY_PATH</code> on UNIX machines to locate shared object libraries. This common directory is also used for Windows object libraries, .lib files, that are needed for linking applications and any Java jar files used by the TENA products.
<code><MW_VERSION></code>	Middleware version specific directory that contains various files associated with the middleware.
<code><MW_VERSION>/bin</code>	Includes <code><PLATFORM></code> specific sub-directories for all of the middleware binary executable files. For Microsoft operating systems, the .dll files used for runtime linking are installed in this directory.
<code><MW_VERSION>/build</code>	Contains files to support build operations for the auto-generated example applications.
<code><MW_VERSION>/doc</code>	Contains documentation for the middleware.
<code><MW_VERSION>/include</code>	Contains the header files needed for building TENA applications.
<code><MW_VERSION>/log</code>	Used as default location for diagnostic log files generated by the middleware and related products.
<code><MW_VERSION>/scripts</code>	Includes script files that are used to set up environment variables used for build and running the auto-generated example applications.
<code><MW_VERSION>/src</code>	Contains the files associated with the auto-generated example applications. See table below for additional details on the example application files.
<code><MW_VERSION>/tdl</code>	Contains the object model TDL (TENA Definition Language) files that define the object models that have been installed.

Users attempting to run TENA executables (e.g., Execution Manager, TENA Console) will need to explicitly reference the `TENA_HOME /<TENA_VERSION>/bin/<PLATFORM>` location, or have this location defined in their `PATH` environment variable. When building TENA applications, the `TENA_HOME /<MW_VERSION>/include` directory is needed for accessing the various middleware header files. The `TENA_HOME/lib` directory is needed for linking and running TENA applications under UNIX operating systems, or this location can be added to the user's `LD_LIBRARY_PATH` environment variable. For Microsoft operating systems, the `TENA_HOME/lib` directory is needed for linking TENA applications and the `TENA_HOME /<MW_VERSION>/bin /<PLATFORM>` directory is needed for running these applications, or this can be included in the user's `PATH` environment variable.

Each time that a user installs a new object model distribution, header files will be installed in the `TENA_HOME/<MW_VERSION>/include` directory, libraries will be installed in the `TENA_HOME/lib` and `TENA_HOME/<MW_VERSION>/bin/<PLATFORM>` directories, and the TDL file will be installed in the `TENA_HOME/<MW_VERSION>/tdl` directory. Additionally, any auto-generated example applications associated with the object model will be installed in the `TENA_HOME/<MW_VERSION>/src` directory.

The directory layout for the auto-generated example application code under the `TENA_HOME/<MW_VERSION>/src` begins with a directory that matches the name of the object model followed by multiple application directories for publishing and subscribing applications for each type of SDO or Message defined in the object model. The individual publisher and subscriber application directories are named with the outer package name of the object model, followed by the name of the SDO or Message, then the keyword of either "Publisher" or "Subscriber", and then the version identifier for the object model. There is also a directory for an application that is capable (through configuration parameters) of publishing and subscribing to all SDOs and Messages defined in the object model. This application is named for the object model name (as opposed to the package and type names). Some example application directory names are "Example-Vehicle-Subscriber-v1" and "Example-Vehicle-v1-AllPublishSubscribe-v1".

As mentioned, by default the example applications generated for a particular object model through the TENA website will contain a publisher and subscriber for every SDO and Message type in the object model, as well as an application that is capable of publishing and subscribing to every object model type.

The directory layout for every auto-generated example application follows the same structure. The directories and key files for this structure are identified in the following table.

TENA Example Application Directory Structure

Directory or File Name	Description
<code><APP_NAME></code>	Top level application directory that contains main build files.
<code><APP_NAME></code> <code>/localCla</code> <code>ssImpls.h</code>	Special file used to support Microsoft compilers to force linkage of object model implementation libraries.
<code><APP_NAME></code> <code>/myBuildF</code> <code>iles</code>	Contains files to assist in building the example application.
<code><APP_NAME></code> <code>/myHelpers</code>	Contains several helper files that are used to support application configuration and provide functions for printing attribute values associated with the object model SDOs and Messages.
<code><APP_NAME></code> <code>/myImpl</code>	Contains a directory layout based on the object model SDO and Message types, in which each type has a sub-directory that contains several implementation classes. For SDOs, there is a class that creates SDO servants and a class that is used to update SDO servants. For Messages, there is a class uses to construct and send messages. These classes initialize all of the attribute values, and in the case of SDOs will update these values incrementally. There are also several "observer" classes that are auto-generated for each SDO or Message in these directories. The observer classes enable the application developer to implement specific behavior that happens in response to an event associated with the SDO or Message.

The auto-generated applications provide completely working example TENA programs based on a particular object model. These applications provide a foundation for developers to use for testing or to tailor for their particular needs. Developers are encouraged to based their TENA applications based on the structure and behavior of the auto-generated example applications.

Application Considerations

Application Considerations

Considerations associated with developing TENA applications are captured within this section of the User Guide.

Specific topics related to application considerations are captured in children pages that are linked below:

- [Application Resignation](#) — Proper application resignation is required to ensure other execution members destruct objects from resigned application and cease communication.
- [Attribute Timestamps](#) — SDO (stateful distributed object) and Message attributes may need to be associated with a timestamp to represent the time of validity for the attribute values.
- [Middleware Provided Boost and ACE-TAO Support](#) — Middleware versions prior to 6.0.5 included support for third-party Boost and ACE-TAO libraries that may now necessitate application changes.
- [Multiple Version Support](#) — Some applications need support to use multiple versions of the middleware, object models, or third-party components in the same application.
- [Runtime Exception Handling](#) — Runtime exceptions thrown from the TENA Middleware that TENA Applications should handle
- [Smart Pointers](#) — TENA Middleware uses smart pointers for many API components to deal with memory management of API components shared between middleware and application.

Application Resignation

Application Resignation

As a publish-subscribe system, applications may establish network connections with other applications to exchange objects and messages. An application needs to resign from the execution to notify those applications that have active network connections that need to be closed or have discovered objects that need to be destructed. If an application is abnormally disconnected from an execution, then the [TENA Console](#) should be used to manually remove the application from the execution.

Description

In a TENA execution, applications establish network communication connections with other applications in order to exchange information and services. Some applications will discover SDOs (Stateful Distributed Objects, referred to as objects) from other applications. These discovered objects will receive state updates from the publishing application and may support remote methods that the subscribing application can invoke on the object contained within the publishing application. When an application resigns from the execution, by destroying the `TENA::Middleware::Execution` object, the middleware will perform the necessary communication with the Execution Manager and other applications to ensure that network connections are properly closed and applications are notified of discovered objects being destroyed.

When an application abnormally disconnects from a TENA execution (e.g., application crash, network failure), any objects created by that application will be left in an unusable state. These objects can't be updated and any attempt to invoke a remote method on these objects will result in an runtime exception being encountered by the invoking application. Additionally, when an application attempts to resign from the execution and still has network connections to an abnormally terminated application, the resign process will attempt to communicate with the abnormally disconnected application and may result in communication timeout delays.

Generally speaking, the TENA Middleware is unable to automatically clean up the objects that were created by an abnormally terminated application because it is not always able to detect a failed application. For example, an unresponsive application may be stopped in a debugging tool or heavily burdened from a processing or memory perspective, so having the middleware take automatic action may introduce other problems. Therefore, the middleware requires operator intervention to clean up objects from abnormally terminated applications. This clean up process is referred to as "Application Removal".

The application removal capability allows an operator to assert that a particular application has abnormally terminated and any objects that were created by that application are no longer usable. The [TENA Console](#) supports a "removal" operation, which will have the [Execution Manager](#) attempt to check that the targeted application is non-responsive and then inform the other applications that the targeted application has been removed from the execution. If other applications are holding objects belonging to the removed application, a destruction event will be delivered for each object. Additionally, internal to the TENA Middleware system any communication connections with the removed application will be closed.

Usage Considerations

Participants in a TENA event should attempt to avoid abnormal application termination as much as possible by not exiting applications without having the application programmatically resign from the execution. The normal resign process will ensure that the TENA Middleware is able to perform the necessary processing to notify applications holding objects belonging to the resigning application and shut down any established communication connections in an efficient manner.

When abnormal application termination is unavoidable, an event operator will be required to issue the "remove application" command through the TENA Console. When the remove application command is invoked, the console requests that the Execution Manager performs the operation. The Execution Manager will attempt to communicate with the application and upon unsuccessful communication it will cause a destruction callback to be generated for all applications that are holding an SDO Proxy belonging to the removed application.

If the Execution Manager is able to successfully communicate with the application that is attempting to be removed, the `remove` command will fail and indicate that the application cannot be removed from the execution.

In certain circumstances (e.g., temporary network outage) an application may be removed, but after some period of time the removed application may attempt to resume normal operations. In this situation, the removed application will have an inconsistent view of the execution and may cause inconsistencies for other applications. Therefore, it is important for the event operators to ensure that applications that are removed through the TENA Console do not resume processing.

Execution Resignation Guidance

The procedures used to resign from an execution depends on the particular needs of the application. For typical applications that are only joined to a single execution and have completed all TENA related processing, it is most effective to invoke the `reset()` method on the `TENA::Middleware::RuntimePtr` object. This will destruct all of the middleware API objects and release memory and other resources associated with the middleware. If the application is joined to multiple executions, or may subsequently join an execution, then the Runtime object should persist and the `reset()` method should be invoked on the appropriate `TENA::Middleware::ExecutionPtr` object. For other applications, it may be appropriate to only destruct an individual `TENA::Middleware::SessionPtr` object.

In some circumstances, the application may want to resign from the execution, but it is important for the application to process any pending middleware events that may exist in the callback queue. There may be pending middleware events (e.g., object updates, messages) if the application has been unable to process all of the callbacks in the time provided to the [Callback Framework](#) operations. When it is necessary to process any pending events during shutdown, the application should first issue unsubscribe operations for all of type subscriptions. The unsubscribe operation will prevent future discovery and update events for that particular type. The application can then process all of the pending callback events before destructing the appropriate API object (i.e., Runtime, Execution, or Session).

i — When resigning, the application should destruct the API object (using the `reset()` method) with the largest possible scope, where scope is defined in descending order as `TENA::Middleware::RuntimePtr`, `TENA::Middleware::ExecutionPtr`, and then `TENA::Middleware::SessionPtr`.

C++ API Reference and Code Examples

Applications join an execution by invoking the `joinExecution` method on the `TENA::Middleware::Runtime` object. Maintaining at least one reference to the `ExecutionPtr` returned from this method will ensure that the application remains joined to the execution. When the `ExecutionPtr` reference count goes to zero due to the `ExecutionPtr` objects going out of scope, or the application explicitly invoking the `reset` method, the `Execution` object will be destroyed and the application will resign from the execution.

When the application is shutting down, the largest scoped middleware object should be destroyed based on the needs of the application (see Execution Resignation Guidance discussion above). This is typically done using the `reset()` method on these objects (actually the smart pointers that hold these objects). Note that if the Runtime is destroyed, the associated Executions and Sessions will also be destroyed.

```
// Destroying the Runtime will also destroy the associated Execution and Session objects.  
TENA::Middleware::RuntimePtr->reset();
```

Application Exit Detector

The middleware provides a utility that can be used with applications to register a signal handler to intercept "control-c" actions. The "control-c" action will normally terminate an application, but the "Exit Detector" utility will ensure that the application properly resigns before the application is terminated. All of the auto-generated example applications illustrate the use of this Exit Detector (as shown by the code fragment below).

```
#include <TENA/Middleware/Utils/ExitDetector.h>  
...  
// Create an ExitDetector object for graceful termination of the main loop  
TENA::Middleware::Utils::ExitDetector exitDetector;  
...  
// Here's a simple way to handle any exit, logout or shutdown call;  
// however, signals can be handled individually (see  
// TENA/Middleware/Utils/ExitDetector.h).  
if ( exitDetector.exitDetected() )  
{  
    std::cout << "\n\tWARNING: Exit requested -- gracefully exiting!\n"  
          << std::endl;  
    break;  
}
```

Attribute Timestamps

Attribute Timestamps

When a publishing application updates a Stateful Distributed Object (SDO) or send a Message, the attribute values may correspond to a particular time in which the attribute value is valid. For example, if a radar system is publishing a measurement, it may be important for the subscribing systems to know the timestamp of that measurement. Many Object Models include TENA::Time as an attribute that is used to provide a timestamp for the attribute values. In some cases, it may be necessary to provide different timestamps for different attributes that are contained within a single SDO update or Message.

Description

When designing an object model, it is necessary to determine if there is a need to add an attribute that represents the time of validity for the attribute values. Often, the object model will use the TENA::Time Local Class to represent this timestamp, either directly or contained within the TENA::TSPI object model. Unless otherwise noted in the object model documentation, the TENA::Time attribute is used to indicate the time of validity for all of the attribute values that need to be associated with a timestamp.

In some circumstances, it may be necessary for different attribute values to correspond to different timestamps. When this situation arises, it is best to design the object model with these different timestamps, including defining the additional TENA::Time attributes as "optional" in case they may not always be used with a particular update. Subscribing systems will need to check if the optional timestamps are set, and then either use the default timestamp or the optional attribute unique timestamp for their processing.

Middleware Provided Boost and ACE-TAO Support

Middleware Provided Boost and ACE-TAO Support

Background

Versions of the TENA Middleware prior to 6.0.5 provided header files and partial library support for boost and ACE/TAO. In addition, the example applications used boost and ACE in some relatively minor ways. The 6.0.5 release removes the use of boost and ACE in the public Middleware API and also from the example applications. This was done to allow use of multiple versions of the Middleware in the same application and so that users could use their own version of boost and/or ACE with the Middleware. Users who developed applications following the earlier versions of the example applications and used boost and/or ACE will find that those applications will not build with 6.0.5. Depending on the extent of use of boost and ACE, there are basically two courses of action: (1) remove the use of boost and ACE by providing alternative implementations or (2) install a distribution of boost and/or ACE. Users who wish to install boost or ACE/TAO can take advantage of already-built distributions in the TENA repository.

Typical Compile Errors

Applications that made use of boost and ACE following the previous example applications will see the following types of compilation errors. Applications that made use of additional boost or ACE functionality will, of course, find additional problems due to missing header files.

Typical errors with Visual Studio

- error C2653: 'boost' : is not a class or namespace name
- fatal error C1083: Cannot open include file: 'boost/lexical_cast.hpp'
- error C2504: 'noncopyable' : base class undefined
- fatal error C1083: Cannot open include file: 'boost/noncopyable.hpp'
- error C2065: 'ACE_SYNCH_MUTEX' : undeclared identifier
- error C2923: 'TENA::Middleware::Utils::MW605::AtomicOp' : 'ACE_SYNCH_MUTEX' is not a valid template type argument for parameter 'ACE_LOCK'

Typical errors with gcc

- error: 'boost' has not been declared
- fatal error: boost/lexical_cast.hpp: No such file or directory
- error: invalid type in declaration before ';' token boost::noncopyable
- error: 'ACE_SYNCH_MUTEX' was not declared in this scope
- error: template argument 1 is invalid
TENA::Middleware::Utils::MW605::AtomicOp<ACE_SYNCH_MUTEX, TENA::int32> &
^

Typical errors osx1011-clang73

- error: use of undeclared identifier 'boost'
- fatal error: 'boost/lexical_cast.hpp' file not found
- fatal error: 'boost/noncopyable.hpp' file not found
- error: use of undeclared identifier 'ACE_SYNCH_MUTEX'
- error: unknown type name 'AtomicOp'

Alternative Implementations

The following alternative implementations that remove boost and ACE are suggested based on the new versions of the example applications.

Alternative to boost::lexical_cast

boost::lexical_cast has a lot of capability, but it is frequently overkill for simple cases. An alternative approach is provided by std::to_string(). Compilers that support c++11 provide std::to_string(). For those compilers which do not support this part of the c++11 standard, TENA provides implementations in TENA/types.h.

Alternative to boost::noncopyable

TENA provides a simple alternative to boost::noncopyable with TENA::Middleware::Utils::noncopyable defined in TENA/Middleware/Utils/noncopyable.h.

Alternative to TENA::Middleware::Utils::AtomicOp

For compilers that support the c++11 standard, atomic functionality is provided by std::atomic. gcc44 does not fully support c++11, but it does provide std::atomic by including the header file stdatomic.h. Visual Studio 2010 does not support std::atomic. When using VS 2010 the recommended approach is to use Windows Interlocked API described at [https://msdn.microsoft.com/en-us/library/windows/desktop/ms684122\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/ms684122(v=vs.85).aspx).

Install boost and/or ACE/TAO

Users who want to continue to use boost and ACE/TAO should be able to install (pretty much) whatever version they would like to use. For convenience the TENA Repository provides already-built versions that are consistent with those used internally by the Middleware. The boost version is 1.60.0.1 which has some minor modifications to the base boost version 1.60.0 to eliminate some compiler warnings. Similarly, the version of ACE/TAO is TAO 2.3.4.1. Users will need to set paths for compiling and linking as they would for any other external library. We suggest using BOOST_ALL_DYN_LINK and referring to the boost documentation.

Multiple Version Support

Multiple Version Support

i Certain applications, such as gateways, may need to link against multiple versions of the TENA Object Models or TENA Middleware. Additionally, the TENA Middleware makes use of third-party components such as ACE, TAO, and Boost software that user application code may want to use, but may have a requirement to use a different version of these third-party components than was used by the TENA Middleware. TENA Middleware release 6.0.4 has provided a **partial** capability with multiple version support, in which multiple versions of a particular TENA object model can exist in the same compilation unit (i.e., separate object code library), but the middleware and third-party products (ACE, TAO, Boost) only supports multiple versions that are maintained in separate compilation units. The next release of the TENA Middleware is expected to fully support using multiple versions of the middleware and the third-party products in the same compilation unit.

! Release 6.0.4 Multiple Version Support Limitations

During 6.0.4 development, it was determined that we were unable to fully complete the planned multiple version support capability and in order to support the needs of the user community it was necessary to only provide a partial capability of the multiple version support mechanism with the middleware 6.0.4 release. Unless there is a time critical need for multiple version support, users are encouraged to wait for the next release of the middleware which is not expected to have the current limitations associated with this capability.

The release 6.0.4 multiple version limitation is that any code files associated with a particular version (whether object model, middleware, or third-party library) can not be mixed with code files with a different version in the same "compilation unit". A compilation unit is the collection of code files that are used by a single invocation of the compiler to produce a particular object file. In other words, Release 6.0.4 provides link-time but not compile-time multi-version support. This limitation requires that applications that need to combine different versions need to ensure that particular invocations of the compiler do not use code files belonging to different versions. Unfortunately, this can be tricky because the compiler may not always detect when there are multiple definitions and the particular version used will often depend on the include path order. Additionally, in order to transfer data from code using one version to code using a different version will require the application developer to define a new data structure that can be shared between the separate compilation units.

For example, assume that an application needs to bridge one execution using the Bar-v1 object model and another execution using the Bar-v2 object model. The application code can be separated so that some code only using the Bar-v1 definition code and other application code will only use the Bar-v2 definition code. In order to transfer Bar data between the code, a new data structure will need to be defined that holds the attribute values for the objects and messages that need to be transferred. On the other hand, if data does not need to be transferred between the different code versions then no additional work is needed.

An additional limitation associated with multiple object model versions is that different local method implementations are not supported. For example, the TENA-Time object model has a local method implementation to support conversions between different time representations. This local method implementation needs to be updated when leap seconds are introduced by the governing body of UTC (coordinated universal time). An application that attempted to bridge multiple versions of the TENA-Time object model that used different local method implementation would not be supported by middleware release 6.0.4.

Users interested in using the limited multiple version support capability in release 6.0.4 of the middleware are encouraged to open a MW helpdesk case to explain their use case so that the limitation can be properly reviewed within this use case and assistance can be provided if necessary.

Description

Prior to release 6.0.4 of the TENA Middleware, it was not possible to link multiple versions of the object models, middleware, or third-party components used by the middleware in the same application. This caused certain gateway applications to have to be architected to use some alternative inter-process communication mechanism with separate processes. Additionally, application code that used any of the third-party software products used by the TENA Middleware were required to use the same code version used by the middleware.

! – Note that multiple version support only applies to release 6.0.4 and later of the TENA Middleware.

Multiple Version Support for Object Models

Some applications may be required to support multiple versions of the same object model (e.g., TENA-Radar-v3 and TENA-Radar-v3.1). A versioned namespace mechanism was developed for the TENA object model code to wrap all of the object model code in a namespace that is unique to the version of the object model (as well as being unique to the middleware version). The application code files need to include the appropriate versions of the object model header files, and if both versions need to co-exist in the same application code file, the object model API elements will need to be fully qualified. The namespace qualification contains the version of the object model and the version of the middleware, e.g., `TENA::Radar::TENA_Radar_v3_1_MW604`.

In order for an application to support multiple versions of the same object model, it is required to join two separate TENA Executions. This is because a single TENA Execution is not permitted to support two different versions of the same object model, since doing so would prevent interoperability of the applications written according to two different object model interfaces. When a typical application joins an execution, the middleware is capable of determining all of the object models linked by that application and automatically providing those object model definitions to the Execution Manager during the join process. The Execution Manager records all of these object model definitions and is able to detect if applications attempt to join the execution with conflicting object model definitions. This is referred to as [OM Consistency Checking](#). In the case of an application joining two separate executions, it is necessary for the application to explicitly inform the middleware of the object model types that the application plans to use for that execution (during the `join()` operation). This allows an application to use one version of an object model with one execution and a different version of the same object model with a different execution.

In the case of an application joining multiple executions, there is an alternative `joinExecution` API method on the `TENA::Middleware::Runtime` API class. This alternative method includes an argument of type `TENA::Middleware::ExecutionJoinOptions` that enables the application developer to specify the particular object model types to be used for the particular execution being joined. The `ExecutionJoinOptions::setSpecifiedObjectModels()` method takes a `std::vector<std::string>` that contains a string representation of the object model, including the version identifier. A string representation of the object model name and version can be obtained through the `getObjectName()` function associated with every object model definition.

The code fragment below illustrates how an application would join two executions using different versions of the TENA-Radar object model.

```
#include <TENA-Radar-v3/TENA/Radar/RadarSystem.h>
#include <TENA-Radar-v3.1/TENA/Radar/RadarSystem.h>
...
// It is only necessary to obtain the object model name from one of the types in the object model file
std::vector< std::string > omNames1;
omNames1.push_back( TENA::Radar::TENA_Radar_v3_MW604::RadarSystem::getObjectName() );

std::vector< std::string > omNames2;
omNames2.push_back( TENA::Radar::TENA_Radar_v3_1_MW604::RadarSystem::getObjectName() );

TENA::Middleware::ExecutionJoinOptions joinOptions1;
joinOptions1.setSpecifiedObjectModels( omNames1 );

TENA::Middleware::ExecutionJoinOptions joinOptions2;
joinOptions2.setSpecifiedObjectModels( omNames2 );

// Join first execution using TENA-Radar-v3 object model
TENA::Middleware::ExecutionPtr pExecution1(
    pTENAmiddlewareRuntime->joinExecution( endpointVector1, joinOptions1 ) );

// Join second execution using TENA-Radar-v3.1 object model
TENA::Middleware::ExecutionPtr pExecution2(
    pTENAmiddlewareRuntime->joinExecution( endpointVector2, joinOptions2 ) );
```

If the application attempts to use an object model type that was not specified in the object models for that execution using the `joinOptions` argument, a `TENA::Middleware::ObjectTypeInconsistencyError` is thrown. This exception can occur with creating a `ServantFactory`, creating a `MessageSender`, invoking `subscribe()`, or invoking `changeSubscription()`.

Multiple Version Support for Middleware

Although not typical for the needs of the user community, some gateway applications may need to utilize different versions of the TENA Middleware in the same process space. In order to support this requirement, all of the middleware code is wrapped within a versioned C++ namespace, e.g., `MW604`, to enable the use of multiple versions. Through the use of appropriate "using" statements, typical usage of the middleware will not require specification of the versioned namespace. In situations where multiple middleware versions need to be supported within the same process space, the application code will typically be segmented such that the individual application code files will only need to access a single version of the middleware. These code files will need to include the appropriate middleware header files for the particular middleware version used by those files, and other application code files using a different version of the middleware will include the header files for that version. Release 6.0.4 of the middleware is only link compatible with multiple versions of the middleware and it is necessary to keep the different versions in separate compilation units. For example, an application can use one library that uses one version of the middleware and have another library that uses a different version of the middleware, linking those libraries into a single executable. This restriction is expected to be addressed with the next release of the middleware.

Multiple Version Support for Third-Party Components

The third-party products used with the current TENA Middleware implementation are:

- ACE (<http://www.cs.wustl.edu/~schmidt/ACE.html>) – "The ADAPTIVE Communication Environment (ACE) is a freely available, open-source object-oriented (OO) framework that implements many core patterns for concurrent communication software. ACE provides a rich set of reusable C++ wrapper facades and framework components that perform common communication software tasks across a range of OS platforms. The communication software tasks provided by ACE include event demultiplexing and event handler dispatching, signal handling, service initialization, interprocess communication, shared memory management, message routing, dynamic (re)configuration of distributed services, concurrent execution and synchronization."
- TAO (<http://www.dre.vanderbilt.edu/~schmidt/TAO.html>) – "standards-based, CORBA middleware framework that allows clients to invoke operations on distributed objects without concern for object location, programming language, OS platform, communication protocols and interconnects, and hardware. TAO is designed using the best software practices and patterns that we have discovered in our work on ACE in order to automate the delivery of high-performance and real-time QoS to distributed applications."
- Boost (www.boost.org) – "Boost provides free peer-reviewed portable C++ source libraries. We emphasize libraries that work well with the C++ Standard Library. Boost libraries are intended to be widely useful, and usable across a broad spectrum of applications."

Release 6.0.4 of the middleware is only link compatible with multiple versions of these third-party products and it is necessary to separate the different versions in separate compilation units (similar to the limitation with multiple versions of the middleware described above).

Although compiling a single code unit with the TENA Middleware and a different version of one of these third-party components is not supported, there are situations in which it can be made to work. There are two basic issues. The first issue is collisions of symbols within a namespace. This problem can be resolved by defining one or more of the preprocessor symbols TENA_BOOST_HAS_MULTIPLE VERSIONS, TENA_ACE_HAS_MULTIPLE VERSIONS, and TENA_TAO_HAS_MULTIPLE VERSIONS as appropriate. This prevents importing symbols from the Middleware-versioned namespace into the unversioned namespace. For example, defining TENA_BOOST_HAS_MULTIPLE VERSIONS eliminates the statement "namespace boost = boost_MW604". References to boost symbols found in the Middleware would then require scoping with boost_MW604. Boost symbols from the user's version of boost would be scoped by boost::.

The second and more difficult issue with mixing in another third-party library in the same compilation unit is the problem of include paths. The header files for the TENA Middleware as well as the third-party libraries have been modified to use only relative paths for other included header files. In addition, the various #define macros in the third-party libraries have been modified to be version specific. These changes only provide part of the solution, however. The fundamental problem of dealing with include path issues remains. While it is possible to sort out include path issues in some cases, this use case is not supported.

Runtime Exception Handling

Runtime Exception Handling

The TENA Middleware uses C++ exceptions to communicate exceptional situations requiring special handling back to the application. Applications need to "handle" these exceptions (using try/catch blocks), or the exception will cause the application to be terminated. This page documents all the parts of the Middleware API that throw exceptions, what specific exceptions are thrown, and under what circumstances the exceptions are thrown.

Description

This page attempts to provide, in a single location, documentation of exceptions thrown by the TENA Middleware. This page is an aggregation of the exception documentation contained in the TENA Middleware API doxygen documentation. The TENA Middleware API does not document exceptions using C++ exception specifications on function declarations. This is in keeping with industry-recognized best practices in the field of C++ software development. For a detailed explanation of the shortcomings of C++ exception specifications, see [Herb Sutter's article](#) on exception specifications. For more background on exceptions in general, see the [cplusplus.com tutorial on exceptions](#).

Properly handing exceptions is important for creating robust and high quality applications. Uncaught exceptions will cause the application to terminate abnormally even though the application may be written to handle the exception and continue processing. Application developers are encouraged to review all TENA Middleware API invocations and determine if there are potential exceptions thrown by the invocation and ensure that the invocation is wrapped in a try/catch block to handle in a manner that will not cause the application to terminate abnormally.

The table below lists TENA C++ Middleware API functions that throw exceptions, what exceptions they throw, and why. The .NET and Java language bindings define exceptions equivalent to the C++ TENA exceptions; C++ STL exceptions are mapped to the most appropriate .NET or Java exception.

Function	Exception	Thrown when...
TENA::Middleware::ApplicationConfiguration's constructor and parse()	TENA::Middleware::ConfigurationError	An option is given an invalid value or is missing a value, or a configuration file could not be read.
TENA::Middleware::ApplicationConfiguration's constructor and parse()	TENA::Middleware::UsageRequested	The "help" flag is set in the configuration.
TENA::Middleware::Configuration's constructor	TENA::Middleware::ConfigurationError	An option is given an invalid value or is missing a value.
TENA::Middleware::Utils::Setting's constructor	std::logic_error	An illegal setting name is specified.
TENA::Middleware::Utils::BasicConfiguration::setSettingValue()	std::invalid_argument	The value given for a setting is not compatible with the setting's type.
TENA::Middleware::Session::evokeCallback() and evokeMultipleCallbacks()	user-code exception	If any exception is thrown by the user-supplied implementation of the virtual "execute()" method of TENA::Middleware::Callback, which includes the various event handlers of AbstractObservers for each SDO and Message type.
TENA::Middleware::init()	TENA::Middleware::RuntimeAlreadyExists	A Runtime already exists. In Release 6.0.3 and later, use TENA::Middleware::getExistingRuntime() to get the already-created Runtime object.
TENA::Middleware::init()	std::logic_error	Internal ACE static object initialization failed.
TENA::Middleware::init()	std::invalid_argument	Validation failed on the given tenaConfiguration. A required option may be missing.
TENA::Middleware::dynamicCast()	std::bad_cast	Cast failed <ul style="list-style-type: none">when casting a NonNullSmartPtr (An exception must be thrown, since a null pointer cannot be returned in this case),when casting an SDOpointer.

TENA::Middleware::Endpoint::parseEndpointString() and TENA::Middleware::Utils::Endpoint::parseEndpointString()	std::invalid_argument	Any of the endpoint profiles in the input string are incorrect.
TENA::Middleware::LocalClassMethodsFactoryRegistry::registerFactory()	std::invalid_argument	Factory is null or factory's description is blank.
TENA::Middleware::LocalClassMethodsFactoryRegistry::registerFactory()	std::logic_error	<ul style="list-style-type: none"> • Factory has already been registered, or • LocalClassMethodsFactoryRegistry::lock has been called for the registry.
TENA::Middleware::LocalClassMethodsFactoryRegistry::getFactory()	std::bad_cast	MethodsFactoryType does not match the type of the LocalClassMethodsFactory corresponding to the given key.
TENA::Middleware::LocalClassMethodsFactoryRegistry::getFactory() and TENA::Middleware::Runtime::joinExecution()	TENA::Middleware::NoLocalClassMethodsFactory	No LocalClassMethodsFactory has been registered for the given MethodsFactoryType. Note that joinExecution() calls an internal TypeRegistry validate() method, which checks that a local methods factory has been registered for every local class in the application's linked-in OM definitions.
TENA::Middleware::Runtime::joinExecution()	TENA::Middleware::ExecutionManagerNotFound	No executionManager found listening at specified endpoint(s).
TENA::Middleware::Runtime::joinExecution()	TENA::Middleware::ArbitraryValueNotAllowed	The executionManager was configured not to allow applications with arbitrary values, but this application was compiled with arbitrary values.
TENA::Middleware::Runtime::joinExecution()	TENA::Middleware::ExecutionAlreadyJoined	The application is already joined to the given execution manager. Search through Runtime::getExecutions(), find the desired Execution object (by looking at Endpoints in the Execution Metadata), and use it instead. In TENA versions > 6.0.3, the ExecutionAlreadyJoined exception has a getExistingExecution() method which can be used to get the ExecutionPtr returned by the earlier successful joinExecution() call.
TENA::Middleware::Runtime::joinExecution()	TENA::Middleware::IncompatibleMiddlewareVersion	The joining application's TENA version does not match that of the execution manager via standard match rules (or exactly, if exact match is required by the execution manager).
TENA::Middleware::Runtime::joinExecution()	TENA::Middleware::NoLocalClassMethodsFactory	No LocalClassMethodsFactory has been registered for the given MethodsFactoryType. This is usually caused by failing to link a required ObjectModel implementation library.
TENA::Middleware::Runtime::joinExecution()	TENA::Middleware::ConfigurationFileNotFound	The configuration file specified in the joinExecution method was not found.
TENA::Middleware::Runtime::joinExecution()	TENA::Middleware::PostRegisterJoinExecutionException	An error occurs while setting up the Execution object after joining. This could be a problem in getting the Execution resources, starting the heartbeat thread, setting up the reliable and best-effort data distribution machinery, creating Execution metadata, or setting up machinery to monitor the fault tolerance replica status.
TENA::Middleware::Runtime::joinExecution()	TENA::Middleware::ObjectModelTypeInconsistencyError	An attempt to join the execution using ExecutionJoinOptions with OMs specified and there is an inconsistency among the OM names, OM types and/or OM versions that are specified.
TENA::Middleware::Utils::SmartPtr dereferences (This applies to RuntimePtr, SubscriptionPtr, etc.)	TENA::Middleware::Utils::InvalidAccess	Dereferencing a null pointer.

TENA::Middleware::Utils::ManagedPtr dereferences (This applies to ExecutionPtr, SessionPtr, ServantPtr, ServantFactoryPtr, MessageSenderPtr, ProxyPtr, and SDOpointerPtr.)	TENA::Middleware::Utils::InvalidAccess	Dereferencing a null or invalidated pointer.
dereference of TENA::Middleware::ExecutionPtr	TENA::Middleware::Utils::InvalidAccess	An attempt to use this pointer is made after the Runtime object that created it has been destroyed.
ServantFactory MessageSender	TENA::Middleware::ObjectModel::TypeInconsistencyError	An attempt is made to create a ServantFactory or MessageSender for an inconsistent OM type (as determined by the list of OM names that were specified when the app joined the execution).
ServantFactory::createServant() ServantFactory::reactivateServant() MessageSender::create()	TENA::Middleware::ExecutionNotConfiguredForBestEffort	An attempt is made to create a best-effort publisher when the executionManager was not started with best-effort delivery settings.
ServantFactory::createServant() ServantFactory::reactivateServant() MessageSender::create() MessageSender::send() ServantPublicationStateView::commitUpdater() Servant::commitUpdater()	TENA::Middleware::Timeout	An invocation on a remote subscriber times out.
ServantFactory::createServant() ServantFactory::reactivateServant() MessageSender::create() Runtime::joinExecution() subscribe() ~Execution()	TENA::Middleware::CannotContactExecutionManager	An attempt is made to send an event to an ExecutionManager, but the send has not succeeded within the time limit given by the "cannotContactEMretryTimeLimitInSeconds" configuration parameter. An executionManager fault has been reported. Fail-over will occur if executionManager replicas exist.
ServantFactory::createServant() ServantFactory::reactivateServant() MessageSender::send() ServantPublicationStateView::commitUpdater() Servant::commitUpdater() Proxy::[anyRemoteMethod]()	std::invalid_argument	An attempt is made to transmit an object (SDO publication state, Local Class, or Message) from a publisher to a subscriber (or vice versa, in the case of an RMI) and the object is missing a required attribute.
ServantPublicationStateView::createUpdater() and Servant::createUpdater()	std::runtime_error	There is an active PublicationStateUpdater for the Servant and the TENA::Middleware::BlockingConcurrencyProperties was set for this Servant.
Constructing a TENA enum from an integer	std::invalid_argument	The given value is outside of the range of permissible values for the enum.
subscribe() and changeSubscription()	TENA::Middleware::ObjectModel::TypeInconsistencyError	An attempt is made to subscribe to an inconsistent OM type (as determined by the list of OM names that were specified when the app joined the execution).
subscribe() and changeSubscription()	std::invalid_argument	Subscription is invalid (i.e, 0), or subscription->getMessageCallbackFactory() is not valid.
subscribe(), changeSubscription(), and unsubscribe()	TENA::Middleware::Timeout	An invocation on a remote publisher or executionManager times out.
subscribe(), changeSubscription(), and unsubscribe()	TENA::Middleware::NetworkError	An unrecoverable network error occurs.
get_[anyOptionalAttribute]() on a Proxy publication state, Message, or Local Class	std::logic_error	An attempt is made to read an unset attribute.

Proxy::[anyRemoteMethod]()	TENA::Middleware::Timeout	An RMI on the remote servant times out.
Proxy::[anyRemoteMethod]()	std::exception	An exception not specified in the formal TDL exception specification clause of the remote method is thrown by the remote method. An error alert is also generated automatically.
Proxy::[anyRemoteMethod]()	TENA::Middleware::RemoteMethodImplError	A RemoteMethodImplError is thrown by a Remote Method Implementation. A RemoteMethodImplError exception is never initiated by the Middleware. It is thrown by a remote method implementation if the implementer wants the Middleware not to generate an error alert when passing the exception from the Servant to the Proxy. It is, therefore, useful for gateways that relay remote method invocations from one execution to another.
SDOpointer::subscribe()	std::invalid_argument	Subscription is invalid.
SDOpointer::pullPublicationState() SDOpointer::subscribe()	TENA::Middleware::NetworkError	An unrecoverable network error occurs.
SDOpointer::pullPublicationState() SDOpointer::subscribe()	TENA::Middleware::Timeout	An invocation on the remote publisher times out.
SDOpointer::pullPublicationState() SDOpointer::subscribe()	std::invalid_argument	SDOpointer's Execution is different from the given Session's Execution.
TENA::Middleware::Filter constructor	std::invalid_argument	An attempt is made to construct a specific-tag filter, but no tag is given.
TENA::Middleware::Filter::matchesOnlyWhichTag()	std::logic_error	An attempt is made to get the tag of a Filter that has no tag.
TENA::Middleware::Tag::get_tag()	std::logic_error	An attempt is made to get the value of a Tag that has no value.

Exceptions thrown by various functions throughout the Middleware:

Exception	Description
std::bad_alloc	Can be thrown by almost any TENA Middleware function. Indicates an out-of-memory condition.
TENA::Middleware::NetworkError	A generic base class, extended by specific exceptions. Can be thrown by any TENA Middleware function that uses the network. Indicates a low-level CORBA or other network-related error.

Smart Pointers

Smart Pointers

The Middleware employs a few different types of smart pointers to manage the lifetimes of objects.

Description

When using dynamic allocation in nontrivial C++ applications, one typically needs some strategy to track the use of dynamically allocated resources so that they can be freed when they are no longer needed. A smart pointer encapsulates such a strategy in a class that overloads `operator->`. In general, smart pointers are class templates that are parameterized on the pointed-to type. The middleware makes use of these smart pointers to handle the memory management strategies with the C++ objects that are shared between the middleware and the application code.

Usage Considerations

There are four different varieties of smart pointers used in the Middleware API:

- `std::auto_ptr`
- ! — The "auto_ptr" was deprecated in c+11 and replaced by "unique_ptr", and the "auto_ptr" has been completely removed from c+17. As a result, the "auto_ptr" has been completely removed from the TENA Middleware 6.0.7 API (and all subsequent Middleware versions) and replaced by "unique_ptr"; however, earlier versions of the Middleware use the "auto_ptr".
- `unique_ptr`
- `TENA::Middleware::Utils::SmartPtr`
- `TENA::Middleware::Utils::NonNullSmartPtr`
- `TENA::Middleware::Utils::ManagedPtr`

Common Smart Pointer Features

The various types of smart pointers apply different usage semantics and memory management strategies. However, they have several common features described below.

Instance creation

When creating an object that will be managed by a smart pointer using `new`, the result of `new` should be passed directly to the smart pointer constructor:

```
SmartPtr< MyClass > objPtr( new MyClass );
```

However, when using the Middleware API, one typically gets a smart pointer to a newly allocated object as the return value to a factory function (i.e., a function responsible for creating instances of a type).

Accessing the Pointed-To Object

Nearly all access to the pointed to object should be through `operator->`. This works as it would with a raw pointer, allowing access to the members of the pointed-to instance.

One can dereference the smart pointer with `operator*`; however, special care should be taken when doing so. Code using a reference to the pointed-to object is not participating in the smart pointer's management of the object; and consequently, the object may be destroyed without the code using the reference becoming aware of that fact.



Due to the nature of `ManagedPtr`'s semantics, it is not possible to provide any sort of lifetime guarantee to code using a reference to the pointed-to instance. `operator*` exists on `ManagedPtr` as an implementation artifact; however, it cannot be used safely in code that uses the Middleware.

Check for Validity

`SmartPtr` and `ManagedPtr` have a member function `isValid` that can be used to check whether the pointer is valid; i.e., not null. The member function `get` can be used to establish the validity of a `std::unique_ptr`. Note that a smart pointer to a discovered SDO Proxy may currently be valid, even though the corresponding SDO Servant has been destroyed and the destruction processing has not been completed by the subscribing application. Additional exception handling is required in some circumstances when using smart pointers.

Significantly, `NonNullSmartPtr` provides no such facility. Because the semantics of `NonNullSmartPtr` guarantees that it cannot be null, a `NonNullSmartPtr` can always be assumed to be valid.

Releasing the Pointed-To Object

The most common way of relinquishing ownership of the pointed-to object is to allow the smart pointer to go out of scope and be destroyed. When the last smart pointer pointing to an object is destroyed, the object will be destroyed as well.

Occasionally it is inconvenient to rely on scope to disassociate a smart pointer from the resource it owns. Smart pointers typically include a `reset` member that takes, optionally, a new raw pointer to own instead of the current one. Calling `reset` with no arguments is equivalent to passing 0 (null) and is consequently not supported by `NonNullSmartPtr`.

Thread Safety

Smart pointers are typically thread safe to the extent that any locking to enforce or enable the semantics of the smart pointer will be performed in the smart pointer implementation. For instance, there is no problem having two `SmartPtrs` to the same instance in different threads where the `SmartPtr`s themselves have independent lifetimes.

Variant-Specific Considerations

`std::unique_ptr`

`unique_ptr` is a smart pointer that owns and manages another object through a pointer and disposes of that object when the `unique_ptr` goes out of scope. `unique_ptr` disallows the copy operation that was the weakness of `auto_ptr`. With `unique_ptr`, transferring ownership of the pointer being managed is done with move semantics using `std::move()`.

```
std::unique_ptr< Example::Tank::RemoteMethodsInterface > pRemoteMethods( // Note: auto_ptr replaced by unique_ptr
    new MyApplication::Example_Tank::RemoteMethodsImpl );
.....
.....
Example::Tank::ServantPtr pServant(
    pServantFactory->createServant(
        std::move( pRemoteMethods ), // Note: std::unique_ptr<> will be NULL after this call. Note: std::move *initializer,
        communicationProperties ) );
```

Since the `unique_ptr` can be moved and not copied, `unique_ptr` is used to signify ownership transfer in an API.

Functions that return a `unique_ptr` are transferring ownership of the pointed-to instance to the caller. The caller can either let the returned `unique_ptr` go out of scope (and destroy the pointed-to instance) or `release` the pointed-to instance to prolong its lifetime. In the latter case, it is typical to use a different type of smart pointer to manage the lifetime of the pointed-to instance. The constructors for some smart pointer types can take a moved `unique_ptr` directly and allow skipping an explicit call to `release`.

Functions that take a moved `unique_ptr` are assuming ownership of the pointed-to instance. The caller should not attempt to use the pointed-to instance subsequent to calling such a function.

`TENA::Middleware::Utils::SmartPtr`

`SmartPtr` is a reference-counted smart pointer. This is probably the most frequently used type of smart pointer in C++ code. Consequently, it has the greatest potential for reuse in client code independent of the rest of the Middleware API.

As the name implies, a reference-counted smart pointer maintains a reference count that is incremented when copies of the smart pointer are added and decremented when those copies are destroyed. The pointed-to object is destroyed when the count drops to zero. Most of the time, a reference counted smart pointer can be used without thinking much about the mechanics of the count. However, there are a couple of situations one must take care to avoid.

Duplicate Reference Counts

While multiple `SmartPtrs` can point to an object, they must all share the same reference count. This happens automatically when assigning `SmartPtr` to another or creating a copy using the copy constructor. But you will wind up with duplicate counts if raw pointers to the same object are used to set the value of more than one `SmartPtr` instance:

```
MyClass * rawPtr = new MyClass;
SmartPtr< MyClass > smartPtr1( rawPtr );
SmartPtr< MyClass > smartPtr2( rawPtr );
```

Because `smartPtr1` and `smartPtr2` are unaware of each other's existence, they have separate reference counts. And for each instance, the `MyClass` instance will be destroyed when the count drops to zero. This is referred to as "double deletion"; and the most likely result is a crash in your program.

Reference Cycles and `SmartWeakPtr`

A more subtle problem to avoid is the creation of reference cycles. In the simplest case, object A has a `SmartPtr` to object B; and object B has a `SmartPtr` to object A. Even if all other references to object A and object B drop to zero, the objects never get destroyed because they continue to "own" each other. The memory for these objects is consequently leaked. Reference cycles can be arbitrarily large and complex, involving many more players; e.g., A owns B which owns C which in turn owns A. They can be hard to identify even once leaked memory has been detected. Reference counted smart pointers are an aid in memory management; not a substitute for thinking about it. It is important to consider the points of ownership for objects in your application or library.

Sometimes, avoiding a cycle is not so easy. There are instances in which one wants to be able to transmit ownership to client code while incrementing the reference count would be redundant (because the lifetime of the instance is guaranteed by some part of the code upon which the current code depends). `SmartWeakPtr` is intended for exactly those scenarios.

A `SmartWeakPtr` doesn't actually participate in ownership; it does not increment the reference count. But clients can obtain a `SmartPtr` from from a `SmartWeakPtr` that shares a reference count with whatever `SmartPtr` was used to construct the `SmartWeakPtr`:

```
SmartPtr< MyClass > smartPtr( new MyClass );

SmartWeakPtr< MyClass > weakPtr( smartPtr );

// Elsewhere in code that can see weakPtr, but not smartPtr...
SmartPtr< MyClass > acquiredSmartPtr = weakPtr.lock()
```

Acquiring a `SmartPtr` from a `WeakPtr` is said to "lock" the owned resource because the resource is kept from being destroyed for the lifetime of the returned `SmartPtr` (and any copies of it).

TENA::Middleware::Utils::NonNullSmartPtr

`NonNullSmartPtr` is very similar to `SmartPtr`, with the significant difference indicated in its name: it is not allowed to be null. Attempting to initialize or reset a `NonNullSmartPtr` to null will result in a `std::invalid_argument` exception. Because it cannot be null, `NonNullSmartPtr` has no `isValid` member function. Any `NonNullSmartPtr` can be assumed to be valid without performing a check. In the Middleware API, local class and message pointers are `NonNullSmartPtrs`.

TENA::Middleware::Utils::ManagedPtr

SDO pointers, `TENA::Middleware::ExecutionPtr`, and `TENA::Middleware::SessionPtr` are `ManagedPtrs`. The `ManagedPtr` is reference counted like the `SmartPtr`; however, it can be invalidated by a manager. This means that while you aren't using it, a `ManagedPtr` can be made invalid. Checking a `ManagedPtr` for validity before using it might seem like a good idea; however, the `ManagedPtr` could be made invalid immediately **after** your validity check (and before your subsequent code executes). So, instead, one should be prepared for the possibility of a `TENA::Middleware::Utils::InvalidAccess` exception whenever a `ManagedPtr` is dereferenced.

Unlike the other smart pointers presented here, `ManagedPtr` satisfies needs rather particular to the Middleware. Using `ManagedPtr` correctly with types outside the Middleware API can be error-prone and is not, in general, recommended.

Example Applications

Example Applications

When an object model is processed through the Object Model Compiler, fully working example applications that illustrate the typical operations of the middleware in conjunction with the object model types are automatically created. The source code of these example applications are provided to assist users with understand middleware operations or using as the basis for a real TENA system.

Example Applications

Included with object model distributions is the source code associated with example applications that illustrate the operation of the middleware in conjunction with the particular object model constructs. These fully working example applications enable programmers to learn the basic operation of the middleware from either a publisher or subscriber perspective. In addition to illustrating how to perform basic middleware operations, the example applications can be used for testing other applications by either publishing certain objects/messages to be received by some subscribing application under development, or used to print out the object/message attribute values produced by some publishing application under development.

These example applications are designed to be simplistic, but can be modified to incorporate the requirements for more sophisticated TENA applications. It is important to note that these auto-generated example applications publish "dummy" attribute values (see [Arbitrary Values documentation page](#)) and there is no implementation code provided with any remote methods that may be defined in the object model. Users are required to carefully review the example application code before attempting to use for operational applications.

Layout of the Example Application Source

The example application source code is installed in the "src" directory under the particular middleware version followed with directories based on the object mode, e.g., \$TENA_HOME/6.0.3/src/Example/Example-Vehicle-v1. Example applications are created for each SDO and Message type defined within the object model, with a separate Publisher and Subscriber application. There is also an example application named "AllPubSub" that supports publishing and subscribing to all SDO and Message types defined in the particular object model.

Within each example application directory, there is the application source code and the necessary build files. All of the build files are included in the installation, even though some of those build files (i.e., Makefiles for UNIX operating systems and Visual Studio files for Microsoft operating systems) will not apply to the target operating system and compiler. Users are required to select the appropriate build file for their environment. Detailed instructions related to building and running the example applications is provided in the [Installation Guide documentation](#).

The items found in an example application installation is shown below, with a brief description of the contents. The particular directory location shown below is for \$TENA_HOME/6.0.3/src/Example/Example-Vehicle-v1/Example-Vehicle-v1-AllPublishSubscribe-v7.1.1. The version identifier "7.1.1" applies to the Object Model Compiler plugin used to generate the C++ example application code. The example application code generated for other programming languages, such as Java, would have a different version identifier. If there was the need to update the example application plugin for a particular middleware release, there could be multiple instances of the example application installed and in this situation users should use the later versioned example application code.

```
myBuildFiles  
myHelpers  
myImple  
Example-Vehicle-v1-AllPublishSubscribe-v7.1.1.config  
Example-Vehicle-v1-AllPublishSubscribe-v7.1.1.cpp  
Example-Vehicle-v1-AllPublishSubscribe-v7.1.1-2005.sln  
Example-Vehicle-v1-AllPublishSubscribe-v7.1.1-2008.sln  
Example-Vehicle-v1-AllPublishSubscribe-v7.1.1-2010.sln  
localClassImpls.h  
Makefile  
Makefile-win.mk
```

Build Files

The .sln project files in the top-level example application directory correspond to Visual Studio build files. The Makefile and Makefile-win.mk are the build scripts for GNU Make on UNIX and Windows systems respectively. More information on build files can be found in the [Build Files -v6](#) documentation.

Example configuration

The Example-Vehicle-v1-AllPublishSubscribe-v7.1.1.config file provides fine-grained configuration options for the Example-Vehicle example application. By modifying this file you can control the appropriate level of subscription and publishing detail as well as the Execution Manager connection information. There is more detailed information within the file itself concerning your options.

Source Code

The Example-Vehicle-v1-AllPublishSubscribe-v7.1.1.cpp is the entry point for the Example-Vehicle example application. This file contains the main function and any auxilliary functions and inclusions needed to get started quickly. This file is heavily documented and is worth understanding on its own.

Helper Source Files

The myHelpers directory contains auxilliary source files used to control the way that configuration is loaded and information printed. These files can be modified to your own needs, although the default behaviors are likely sufficient in many cases.

Implementation Source Files

The myImpl directory contains the base functional creation and update servants as well as a basic subscriber.

Related Example Application Topics

- [Arbitrary Values](#) — The auto-generated example application code uses an ArbitraryValue mechanism to generate default values to support test and evaluation programs.
- [Environment Variables](#) — The auto-generated build files associated with the example applications utilize environment variables to support the building and running of these example applications.
- [Example Application Configuration Options](#) — Defines the configuration options for the auto-generated example applications.

Arbitrary Values

Arbitrary Values

The TENA Object Model Compiler can automatically generate all of the software files necessary for creating a TENA application that publishes and/or subscribes to the various object and messaged defined in the object model. Publishing objects and messages requires the application to select type proper values for the various attributes, as well as supplying type proper values for the remote method arguments and return values. The ArbitraryValue mechanism is used by these example applications to ensure that arbitrary values are of the proper type. And to ensure that event operators are able to detect applications that may be mistakenly be using these arbitrary values, the use of the ArbitraryValue mechanism can be disabled by configuration of the Execution Manager.

Description

The [TENA Object Model Compiler](#) supports the automated generation of working example software applications that utilize the user specified object model. These applications publish and subscribe to the SDOs (Stateful Distributed Objects) (i.e., objects) and Messages defined within the specified object model. The automatically generated applications need to provide properly typed values for the published object and message required attributes. These automatically generated values are supported through the use of a C++ class template named `ArbitraryValue`.

The `ArbitraryValue` class template is designed to provide a default value for all of the fundamental TENA data types. For example, the arbitrary value for an integer is set to zero, a floating point value is set to 0.0, and a string is set to be empty. In some cases, the arbitrary value is set to a specific value to improve the illustrative nature of the automatically generated programs (e.g., increment the value by one on each update).

The benefit of having the automatically generated example applications provide arbitrary values is so that users can quickly obtain completely working example programs for their particular object model. The auto-generated software can then be used as the foundation for user developed applications that are attempting to meet specific requirements associated with an operational application/system.

A problem that can occur is when an application developer has forgotten to replace the auto-generated `ArbitraryValue` code with real attribute values.

- **⚠ AVAILABLE only in Release 6.0.4 and lower**, the EM will allow applications using the `ArbitraryValue` to join the execution by default. To prevent applications using `ArbitraryValues` from joining the execution, operators must start the EM with the configuration parameter, `disallowArbitraryValue`.
- **⚠ AVAILABLE only in Release 6.0.5 and higher**, the EM will prevent applications using the `ArbitraryValue` from joining the execution by default. To allow applications using `ArbitraryValues` to join the execution, operators must start the EM with the configuration parameter, `allowArbitraryValue`.

When applications are needed for operational use, `ArbitraryValues` should not be used. To enable this, applications must be compiled without the compiler directive, `TENA_MIDDLEWARE_ALLOW_ARBITRARY_VALUE`, to ensure proper detection of the use of `ArbitraryValues`. By default, the auto-generated build files, (i.e., Makefiles for UNIX, and Visual Studio project files for Windows), define the compiler directive `TENA_MIDDLEWARE_ALLOW_ARBITRARY_VALUE`, and this directive must be manually removed to enable detection of the use of `ArbitraryValues`. In addition, all `ArbitraryValues` should be removed from the application code.

- **⚠ AVAILABLE only in Release 6.0.4 and lower**, only applications built WITHOUT the `TENA_MIDDLEWARE_ALLOW_ARBITRARY_VALUE` compiler directive will be permitted to join the execution when the `disallowArbitraryValue` configuration parameter is set for the EM. By default, `ArbitraryValues` ARE allowed.
- **⚠ AVAILABLE only in Release 6.0.5 and higher**, applications built WITH the `TENA_MIDDLEWARE_ALLOW_ARBITRARY_VALUE` compiler directive will be permitted to join the execution only when the `allowArbitraryValue` configuration parameter is set for the EM. By default, `ArbitraryValues` ARE NOT allowed.

Usage Considerations

Developers using the auto-generated example applications can utilize the software for investigation and testing using the `ArbitraryValue` type. When TENA application developers are attempting to develop an operational application, all occurrences of `ArbitraryValue` should be replaced with actual values that are appropriate for the application. The use of `ArbitraryValue` can be detected by removing the compiler directive `TENA_MIDDLEWARE_ALLOW_ARBITRARY_VALUE` found in either the corresponding Makefile under UNIX operating systems, or in the Visual Studio project files under Windows operating systems. Under Visual Studio, the compiler directive is found in the properties window for Configuration Properties:C/C++:Preprocessor: Preprocessor Definitions.

- **⚠ AVAILABLE only in Release 6.0.4 and lower**, TENA event operators may want to detect applications that are using `ArbitraryValues`, because operational applications should be providing actual attribute values and not relying on auto-generated arbitrary values. In this situation, the EM must be started with the configuration parameter, `disallowArbitraryValue`, to prevent applications that are using `ArbitraryValues` from joining the execution.
- **⚠ AVAILABLE only in Release 6.0.5 and higher**, TENA event operators can detect applications that are using `ArbitraryValues`, because the EM disallows the use of `ArbitraryValues` in joining applications by default. If applications using `ArbitraryValues` are to be allowed in the execution, the EM must be started with the configuration parameter, `allowArbitraryValue`.

C++ API Reference and Code Examples

TENA application developers are not expected to make use of the `ArbitraryValue` class since it is meant to be used for automatic, code-generation purposes. An example of the use of `ArbitraryValue` is shown in the code fragment below. In this example, the Person-Publisher application is using `ArbitraryValues` for the "x" and "y" attributes for the `Example::Location` local class.

```
// create a LocalClass instance for attribute location
// \todo In a real application, instances of \c ArbitraryValue<T>
//       should be replaced with values that are sensible for the
//       particular application.
Example::Location::LocalClassPtr p_PersonUpdater_location_LocalClass(
    Example::Location::create(
        /* xInMeters */ ArbitraryValue< TENA::float64 >(),
        /* yInMeters */ ArbitraryValue< TENA::float64 >() ) );
```

Environment Variables

Environment Variables

The auto-generated build files associated with the example applications utilize environment variables to support the building and running of these example applications. These environment variables are not required for general TENA applications, only for the example applications and other custom TENA applications that want to follow the same build file pattern.

Environment Variables

When running the `tenaenv` script (see the section [Building Example Applications](#) for more information) the build environment amenable to building example applications is established. The goal of these environment variables is to isolate changes in versions of the TENA middleware from specific user build infrastructure.

Having said that; when the `tenaenv` script is run the following variables are created:

Variable Name	Description
<code>TENA_HOME</code>	Set to the root directory of the TENA software installation location.
<code>TENA_PLATFORM</code>	Set to the platform descriptor that defines the OS, compiler, and versions.
<code>TENA_VERSION</code>	Set to the version number of TENA Middleware.

In addition, depending on your operating system, the following environment variables will be appended in the following:

PATH

In a UNIX environment, the `PATH` variable will have the following appended:

```
PATH = ${TENA_HOME}/${TENA_VERSION}/bin/${TENA_PLATFORM}: ${TENA_HOME}/all/bin/${TENA_PLATFORM}: ${TENA_HOME}/all/bin: \
${TENA_HOME}/${TENA_VERSION}/scripts: ${TENA_HOME}/all/scripts: ${PATH}
```

From a Windows-esque perspective the append will be:

```
PATH = %TENA_HOME%\%TENA_VERSION%\bin\%TENA_PLATFORM%;%TENA_HOME%\all\bin\%TENA_PLATFORM%;%TENA_HOME%\%TENA_VERSION%\scripts;%PATH%
```

Library Variables

On a UNIX environment (or in Cygwin on Windows), both the `LD_LIBRARY_PATH` and `DYLD_LIBRARY_PATH` variables will be appended in the following ways:

```
LD_LIBRARY_PATH = ${TENA_HOME}/lib
DYLD_LIBRARY_PATH = ${TENA_HOME}/lib
```

In the case of this example application tutorial simply working against the defaults established by the `tenaenv` script should be sufficient. However, when you choose to setup your own production development environment then finer-grained control might be required. In that case, please refer to the comprehensive [Middleware Installation Guide](#) for more information regarding the modification of the TENA-related environment variables.

Example Application Configuration Options

Example Application Configuration Options

The auto-generated example applications support a common set of configuration parameters that can be used to control the behavior of the application when running. These applications can be used for testing other TENA applications or testing network connectivity.

Application Configuration Options

Aside from the required arguments (as we will see in the section [Running Example Applications](#)), example applications take the following optional arguments:

Argument	Parameter(s)	Default	Description
<code>help</code>	<code>none</code>	<code>none</code>	Displays a full listing of configuration parameters for the TENA Middleware and the example application programs
<code>publish</code>	<code>all, none, or CSV</code>	<code>all</code>	Specify which elements to publish (only with <code>AllPublishSubscribe</code> applications)
<code>subscribe</code>	<code>all, none, or CSV</code>	<code>all</code>	Specify which elements to publish (only with <code>AllPublishSubscribe</code> applications)
<code>bestEffort</code>	<code>none</code>	<code>none</code>	Specify the use of BestEffort (UDP/IP multicast) for SDO state change updates as well as for messages
<code>numberOfIterations</code>	<code><number></code>	<code>30</code>	The number of iterations (updates and/or callback intervals) the application will run
<code>verbosity</code>	<code><number></code>	<code>4</code>	A verbosity level argument, ranging from 0 (least output) to 4 (most output)
<code>noOptional</code>	<code>none</code>	<code>none</code>	Disables updating optional attributes in messages and SDOs. ⚠ AVAILABLE only in Release 6.0.8 and higher

CSV refers to a comma-separated list of values such as `foo,bar,baz`

Building Example Applications

Building Example Applications

In this section we will discuss how and where to get and install an object model distribution and build the auto-generated example application using the Example-Vehicle object model. This process applies to other object models as well.

Getting Your Model Distribution

Before we start into building the auto-generated example application we have to take a few minutes to browse for and retrieve it. For this exercise, we will be using one of the TENA example object model `Example-Vehicle`. This section will walk you through the process of navigating the TENA Repository in order to obtain all of the parts necessary for working with the example applications.

The TENA Repository

The TENA Repository is a website used to store, search, view, and download various software components, including those associated with object models. To access the TENA Repository, open a web-browser and enter the following URL into its address bar: <https://www.trmc.osd.mil/repository>. From this opening page you will want to click on the browse object models link (or menu item). From the search bar, type "Example-Vehicle" and select the Example-Vehicle object model link to access the details page for this particular object model.



Just a moment

Have you downloaded the current version of the TENA middleware? The middleware can be downloaded from the repository by clicking the Products:Middleware menu item.

The middleware distribution includes the object model distributions for the Example and TENA standard object models, so it is not necessary to download the Example-Vehicle object from the repository. Generally, users will want to work with example applications for their particular object model though, so going to the Repository to download and install the object model distribution will be necessary.

Building the Example Application

Navigate to the installation directory for the example application of interest. Refer to the [Example Applications](#) documentation page for information related to the directory location and contents.

Before building the example application it is necessary to set the environment variables for your operating system. Scripts are provided in a directory under the middleware version being used (e.g., `$TENA_HOME/6.0.3/scripts`) that can be run to set the environment variables used by the example application build files. Utilize the script associated with your operating system and shell preference (e.g., `tenaenv-w7-vs2010-64-v6.0.3.bat`, `tenaenv-rhel6-gcc44-64-v6.0.3.sh`).

Under the Windows operating system, it is necessary to re-start any programs after environment variables have been changed. It may also be necessary to take additional steps for environment variables to be properly acknowledged. Refer to the [Installation Guide](#) for details related to setting and checking that environment variables have been properly established.

To finally build the source you can simply navigate to the path under the `src` directory for the example application that you desire to build. In the `src` directory you will see a number of sub-directories for various object models – one of which will be `Example-Vehicle-v1`. Within this directory you will see a number of directories corresponding to the different example application sources.

- `Example-Notification-Publisher-v7.1.1`
- `Example-Notification-Subscriber-v7.1.1`
- `Example-Vehicle-Publisher-v7.1.1`
- `Example-Vehicle-Subscriber-v7.1.1`
- `Example-Vehicle-v1-AllPublishSubscribe-v7.1.1`

For this tutorial we will build the `Example-Vehicle-v1-AllPublishSubscribe-v7.1.1` example application.

Navigate to the `Example-Vehicle-v1-AllPublishSubscribe-v7.1.1` directory. There will be a number of build files ranging from Visual Studio solution files to Makefiles. On UNIX systems, simply run `make` to build the example application. On Windows, it is necessary to start the Visual Studio program (either manually or by double clicking the appropriate `sln` file). When using Visual Studio, it is necessary to check at the top of the tool that the appropriate Debug/Release and Win32/x64 setting is defined, and then "Build Solution" can be selected from the "Build" menu.

If the middleware and object model code has been properly installed and the environment variables have been properly set up, the build operation will produce an executable binary named `Example-Vehicle-v1-AllPublishSubscribe-v7.1.1` (or `Example-Vehicle-v1-AllPublishSubscribe-v7.1.1.exe` on Windows) which will be used when we [run our example application](#).

Running Example Applications

Running Example Applications

In this section we will talk about the steps involved with running and modifying the example application source code. In addition we will talk about some of the details of the source and what the parts mean.

Prerequisites for Running Example Applications

The following instructions assume that the middleware has been properly installed and the example application has been built. The [Building Example Applications](#) documentation page discusses the procedures for building these applications. This documentation will continue to utilize the Example-Vehicle-v1-AllPublishSubscribe-v7.1.1 example application, but the procedures apply to any auto-generated TENA example application.

Running the TENA Execution Manager

After setting up the environment variables used by the example applications using the `tenaenv` script (see [Building Example Applications](#)), the [Execution Manager](#) can be started in a Windows command or shell window with the following command:

```
executionManager -listenEndpoints <emHostname>:<emPort>
```

Note that is generally recommended to start the Execution Manager using the [TENA Console](#), since this event management tool provides additional insight into the distributed event, but for these procedures manually starting the Execution Manager process is sufficient.

The parameters `emHostname` and `emPort` should be filled in with your local machine's name or IP address that is associated with the computer's network interface that will be running the `executionManager` executable and port respectively. You will see a number of informational messages printed to your console, one of which will be useful in a moment:

```
To join this execution, use one of the following endpoints:  
{ {[192.168.12.82:53013]} }  
e.g.  
-emEndpoints iiop://192.168.12.82:53013
```

Running the Example Vehicle Publisher/Subscriber

Open another command or shell window and navigate again to the place where the TENA middleware was installed. Under UNIX operating systems, it is necessary to set up the environment variables for each shell using the `tenaenv` script. Continuing with navigation to `src/Example-Vehicle-v1/ExampleVehicle-v1-AllPublishSubscribe-v7.1.1` directory to locate the previously built example application executable. The example application can then be started with the following command.

```
./Example-Vehicle-v1-AllPublishSubscribe-v1 -emEndpoints <emHostname:emPort> -listenEndpoints <hostname>
```

The `emHostname` and `emPort` comprises the network address used when the `executionManager` process was started. The `listenEndpoints` for the example application is used to indicate the name (or IP address) associated with the computer's network interface that is running the example application. The `listen` port for the example application is optional, as the middleware will select an available port if not specified. Defining the particular port for TENA applications may be necessary when network devices are involved in the distributed event and may only permit certain ports to be used for TCP communication. If firewall configurations restrict the available port range that applications can use, the `portspan` option can be used such as "192.168.1.1:55100/portspan=10".

After running the command above you will see some diagnostics reported to standard output for the process related to the objects and messages being published. The default behavior of the example applications is to cycle through 30 updates for the objects and message being published. An example output is shown below.

```

There are 0 discovered Example::Vehicle SDOs on the list.
There have been 0 Example::Vehicle Discovery events.
There have been 0 Example::Vehicle State Change events.
There have been 0 Example::Vehicle Destruction events.

There are 0 queued Example::Notification messages.

Updating the Example::Vehicle::Servant's state #30: ... Done.
Example::Vehicle.name Update # 30
Example::Vehicle.team Example::Team_Red
Example::Vehicle.location
    location.xInMeters 0
    location.yInMeters 0

Example::Vehicle::Metadata
    ID='DIME::ID( 192.168.12.82, 6950, 1266505276, 7 )'
    StateVersion=31
    ReactivationCount=0
    getPublisherEndpointsString='[{192.168.12.82:58085}]'
    PublisherApplicationID=1
    PublisherSessionID=0
    FilteringContext={ Tag=<No Tag> }
    CommunicationProperties=TENA::Middleware::Reliable
    MulticastAddressString='N/A'
    isTerminated=false
    TypeIDpath={ Example::Vehicle from Example-Vehicle-v1 OM (0x00000000ee61bb07) }
    TimeOfCreation =2010-02-18T15:01:16.105915-0500
    TimeOfDiscovery=<uninitialized>
    TimeOfCommit =2010-02-18T15:01:46.125957-0500
    TimeOfReceipt =<uninitialized>
Sending Example::Notification::Message # 30: ...
Example::Notification.text

```

The precise details printed by the example application will vary depending on your specific computer's configuration, but the above should generally match.

And that's all there is to building and running example applications.



For a more comprehensive guide to running example applications see the [Middleware Installation Guide](#).

Log Files

Log Files

TENA applications create a log file (unless disabled) whenever the application encounters an exception (i.e., error, warning). Normally, the exception message will also be written to the application's "standard output" stream and a distributed Alert message will be sent to all active TENA Consoles.

Description

TENA applications (by default) create a log file that contains any alert messages encountered by the application. The alerts consist of errors, warnings, and notes that represent unusual conditions that should be reviewed by the application operator. The Execution Manager will also create a log file and record any operator commands in the file.

The log files, by default, are written to the directory \$TENA_HOME/\$TENA_VERSION/log using the following filename template <toolname>-<date>-<time>-<pid>-msgs.txt, e.g., tenaConsole-20100115-134131-27192-msgs.txt.

These log files will include any [Alerts](#) generated by the application and are a valuable diagnostic tool when a TENA application has experienced any anomalies when running. Users are encouraged to review these log files under those circumstances, as well as periodically inspect the log files during application development.

 TENA applications will produce log files by default when any application is run and experiences Alerts. These files will not be deleted, so users are responsible for removing these log files when no longer needed.

Configuration Parameters used to Control Log Files

Parameter Name /Syntax	Description
logDir <ARG>	The directory for diagnostic and error log files. Default location is \$TENA_HOME/\$TENA_VERSION/log.
noErrorLog	Disable error log file (messages will only go to std::cout). Default setting is to produce an error log file.
applicationName <ARG>	This configuration option is used to specify the application name as it appears in log file names. Default name is based on process name obtained through operating system.

Example Log File

```
Log file opened by '/TENA/6.0.0/bin/rhel5-gcc41-64-d/executionManager' at 2010-01-15T13:40:46.764599NOTE: EM ID 0: Registering an execution manager replica with ID number 1 listening on endpoint set {[192.168.12.37:51001]} [2010-01-15T13:41:10.305617]

NOTE: EM ID 0: Attaching a console application with ID number 2 listening on endpoint set {[192.168.12.37:51018]} [2010-01-15T13:41:31.390251]
NOTE: EM ID 0: Registering an application with ID number 3 listening on endpoint set {[192.168.12.37:34900]} [2010-01-15T15:07:05.540031]
NOTE: EM ID 0: Attaching a console application with ID number 4 listening on endpoint set {[192.168.12.75:2451]} [2010-01-15T15:07:57.617342]
WARNING: TENA console 2 attempted to terminate objects for unknown application with ID 3423. [2010-01-15T15:08:28.318580,TENA/Middleware/ExecutionManager/ExecutionManagerImpl.cpp:4287:terminateObjectsBelongingTo,TID=1132456256]
NOTE: EM ID 0: Unregistering an application with ID number 3 listening on endpoint set {[192.168.12.37:34900]} [2010-01-15T15:23:49.599026]
```

Distributing TENA Applications

Distributing TENA Applications

Like many executable programs, the applications built using the TENA Middleware have dependencies on a certain run-time environment to be able to run properly. This includes any dependent shared libraries, as well as environment settings that vary for each of the supported operating systems for the TENA middleware. Although many different approaches to packaging applications are possible, the TENA Project has developed an approach used by the components of the middleware itself. Following this approach for TENA applications may provide benefits, since it leverages the directory structure and environment already established for the TENA Middleware.

The general process of distributing one or more applications is simply to package up everything each application needs to run, and collect the files into some kind of archive. The steps are outlined below.

- Identify the application parts — executable programs, configuration files, data files, and documentation.
- Identify dependent libraries.
- Decide upon a partitioning and directory structure.
- Create a suitable archive and distribute it to target machines.

Identifying the Application Parts

In the simplest case, this will be a single executable program. For the example source code distributed with the TENA Middleware distribution, each separate directory generates a single executable. For example, building the project in \$TENA_HOME/6.0.3/src/Example-Vehicle-v1/Example-Vehicle-Publisher-v7.1.1 using the build files included will create a single executable in that directory, Example-Vehicle-Publisher-v7.1.1.exe (or Example-Vehicle-Publisher-v7.1.1 on non-windows platforms). Generally speaking, the application parts will include the executable binary, libraries, configuration files, runtime data files, and documentation.

Identifying Dependent Libraries

An application program for the TENA middleware will have dependencies on one or more shared libraries. These are .dll files on windows, .so files on unix platforms, or .dylib files on OSX. These dependencies must be deployed along with the application file. The developer will usually already understand the dependencies of the application, but there are OS specific tools to help identify them. On windows, the tool is dumpbin. On the various unix flavors, the tool is ldd. On OSX, it is otool. Consult the specific documentation for details. Below is an example using dumpbin on Windows.

```
C:\Install\TENA\6.0.3\src\Example-Vehicle-v1\Example-Vehicle-Publisher-v7.1.1> dumpbin /dependents Example-Tank-Publisher-v7.1.1.exe

Dump of file Example-Vehicle-Publisher-v7.1.1.exe
File Type: EXECUTABLE IMAGE
Image has the following dependencies:
    libTENA_Middleware-w7-vs2010-64-v6.0.3.1.dll
    libExample-Vehicle-v1-w7-vs2010-64-v6.0.3.1.dll
    MSVCP100.dll
    MSVCR100.dll
    KERNEL32.dll
```

Running dumpbin requires setting up a Visual Studio Command Prompt from the Start Menu. The libraries listed here are typical for a TENA application. The first library listed in the TENA Middleware library and libExample-Vehicle-v1-w7-vs2010-64-v6.0.3.1.dll library is for the object model definition. If the application used additional object models, those libraries would be listed as well. Object models may also have implementations that would be listed as a dependency. There are several operating system specific files identified with this executable: MSVCP100.dll, MSVCR100.dll, KERNEL32.dll. As a rule, the compiler libraries (e.g., MSVCR100.dll) will be available on all machines with Visual Studio installed, or machines that have installed the Microsoft Visual C++ Redistributable package.

A developer needs to understand how each library is used and any requirements for third party products used in their application. Where appropriate, these third party products should be bundled with the application distribution to simplify the installation requirements for the end-user. However, there are several considerations that need to be evaluated before bundling third-party libraries:

- Does the third-party software license prevent the re-distribution of the software or have other restrictions? For example, some freely available products require that the license file is distributed with any product using the third-party library. In this situation, the application should include the license files with the software package contents (for example, the TENA Middleware SDK product places all of the license files in the .../TENA/doc/licenses directory).
- Does the vendor of third-party software provide periodic updates to address security vulnerabilities? If so, then it may be appropriate for the end-user to be responsible for downloading the third-party product and ensuring that it is properly updated, as installing a pre-bundled library could expose the user to a security vulnerability if it is out of date.
- Does the third-party product need to be installed using operating specific installation tools? If so, then it may not be possible to include the third-party software in the distribution. For example, bundling the Visual Studio MSVC libraries directly will not work – they must be installed in the fashion defined by Microsoft using a file download from their site.

Microsoft Visual C++ Redistributable Packages

As mentioned above, the Microsoft Visual Studio compiler libraries can't be bundled with the application product. Users need to download the Redistributable from Microsoft to properly install the files for the runtime needs of the application. Although it is possible that the Redistributable Package download file could be included with the application product, this is not recommended since the download file could be out of date by the time the user performs the installation and an older version with vulnerabilities could be installed. It is recommended that the application product provides installation instructions that informs users of the need to download the Microsoft Visual C++ Redistributable Package when necessary.

Links to the Microsoft Visual C++ Redistributable Packages for the currently supported compilers are listed below:

- [Visual C++ 2010 Redistributable Package -- https://www.microsoft.com/en-us/download/details.aspx?id=5555](https://www.microsoft.com/en-us/download/details.aspx?id=5555)
- [Visual C++ 2012 \(update 4\) Redistributable Package -- https://www.microsoft.com/en-us/download/details.aspx?id=30679](https://www.microsoft.com/en-us/download/details.aspx?id=30679)
- [Visual C++ 2013 Redistributable Package -- https://www.microsoft.com/en-us/download/details.aspx?id=40784](https://www.microsoft.com/en-us/download/details.aspx?id=40784)

Note that the Visual C++ 2015, 2017 and 2019 all share the same redistributable files. The latest supported Visual C++ Redistributable Packages for Visual Studio 2015, 2017 and 2019 are located at:

- [Visual C++ 2015, 2017 and 2019 Redistributable Package -- https://support.microsoft.com/en-us/topic/the-latest-supported-visual-c-downloads](https://support.microsoft.com/en-us/topic/the-latest-supported-visual-c-downloads)

If usability is a concern regarding computers without Visual Studio or the Redistributable Package installed, the TENA Installer supports the use of a post-install script that could attempt to detect if the dependent libraries are installed on the user's computer and provide instructions to download the appropriate redistributable package.

Decide upon a Partitioning and Directory Structure

Partitioning

Partitioning a software distribution into more than one part may have advantages for configuration management. The object models and low level utility libraries used in an exercise should be stable before all application development is complete. So providing one software distribution with all the shared object models (actually OM definition libraries and OM implementation libraries) and utilities, and a second software distribution with just application binaries, might make sense. Then updates to applications would only require redistributing the application distribution.

Decisions about how to partition software distribution will be different and must be made depending on the detailed requirements for a particular distributed execution, and how the versions and dependencies of applications and libraries are managed. From a TENA perspective, the only important idea is that all applications use compatible versions of all object model definitions and (typically) implementations. Because of this, it is generally recommended that a distributable archive containing all object models along with their implementations be made, so that all applications in an exercise use a consistent set of libraries for this shared communication.

Directory Structure

The TENA Middleware uses a defined directory structure and establishes an environment to allow TENA Middleware applications that adhere to that structure to be run, simply. The directory structure is described in more detail in [TENA Software Directory Structure](#). But the basic idea is simple.

Windows Platforms

All shared libraries (.dll files) and all executables are installed under a directory `TENA_VERSION/bin/TENA_PLATFORM`. `TENA_VERSION` is simply the version of the current middleware, such as "6.0.3". `TENA_PLATFORM` is the platform descriptor the middleware uses to differentiate between library versions, such as `w7-vs2010` or `osx-gcc42-64-d`. When the middleware is installed, the `PATH` environment variable is set to include this directory. Extracting an archive with this structure under the `%TENA_HOME%` directory will allow the executables and shared libraries to be resolved and run properly.

Non-windows Platforms

On non-windows platforms, the shared libraries (.so files or .dylib files) are installed under the `lib/` directory. The actual libraries generated by TENA are all named explicitly to include information about the platform and version of the TENA middleware used (for example, `libExample-Vehicle-v1-f12-gcc44-v6.0.1.dll`) so putting them in a common `lib/` directory does not cause any ambiguity. The environment needed to dynamic load these variables can be set by the `tenaenv-TENA_PLATFORM-vTENAVERSION.sh` scripts in the `scripts` directory. Executables are still stored in the directory `TENA_VERSION/bin/TENA_PLATFORM`. `TENA_VERSION` is simply the version of the current middleware, like "6.0.3". `TENA_PLATFORM` is the platform descriptor the middleware uses to differentiate between library versions, like `-w7-vs2010-64` or `osx-gcc42-64-d`. The `PATH` environment is also set by the `tenaenv` scripts.

Following this structure on a non-windows platform will allow the distributions to take advantage of the TENA Middleware environment scripts. Once the environment is set using a `tenaenv` script, and applications distributed in this way and extracted under `$TENA_HOME` would be able to be run.

The different directory structures used for Windows and non-Windows platforms is driven by the differences in the environment needed for application development. This structure is used by the middleware and is a compromise between development time and run time, and the different platforms.

Create a suitable archive and distribute it to the target machines

Once a directory structure and contents are identified, simply making an archive will allow the applications to be distributed to other computers, by extracting them. If the directory structure above was followed, extracting them under the `TENA_HOME` directory will be correct for the run time environment established for TENA Middleware installations (the various settings of `PATH`, `LD_LIBRARY_PATH`, `DYLD_LIBRARY_PATH`).

Internally we have standardized on the use of .zip archives on all platforms. Tools for creating zip archives are easily available on all platforms. In addition, an internally developed tool, the [TENA Installer](#) allows creating self extracting installation images on all the TENA supported platforms (Windows, Linux, OSX, and Solaris). This tool allows packaging a single or multiple .zip into an installer. The generated installation image can allow certain features to be optional, can run post installation scripts, and can even be used to run platform specific installation of third party libraries. See the documentation on the TENA Installer for more details.

Application Object Model Linking

Application Object Model Linking

TENA applications will link against one or more individual object models. Only object models that are used by the application for publication or subscription should be included when building the application. Linking unnecessary object models will create dependencies on those object models and object model versions that may necessitate re-building the application if there is an object model version mismatch with other applications joined to the TENA Execution.

Description

In addition to compiling and linking against the middleware include files and libraries, TENA applications need to include header files and libraries for object models that are used by the application. Object models are used if the application is publishing an item from the object model (e.g., Stateful Distributed Object, Message), or if the application is subscribing to an item from the particular object model. Additionally, the application may be using an object model indirectly through composition or inheritance.

After an application determines the necessary object models (and their versions) that are needed, the application source code will need to include the necessary header files for the object model definitions used by the application. For example, if the application is subscribing to `Example::Vehicle` objects, then the `Vehicle.h` header file needs to be included in the application source code. This include statement (i.e., `#include <Example/Vehicle.h>`) is typically placed in the main C++ application file. Note that this include statement does not specify the particular version of the object model associated the `Example::Vehicle` definition being used. The object model version identification is included in the build files (i.e., Makefile, Microsoft Visual Studio project file) to simplify the process when an application needs to migrate to a different object model version.

The build files will need to define the include directory to be used for the object model definition, and the include directory name will include the object model version (e.g., `-I$(TENA_HOME)/$(TENA_VERSION)/include/OMs/Example-Vehicle-v1`). Note that the object model include files are placed in a middleware versioned directory by default as shown by the `$(TENA_VERSION)` variable in the example above. Although the object model include files may not be different between middleware versions, using this source code layout helps to manage the build complexity. Also note that the variables shown in the above example are used with the automatically generated example source code, but are not strictly required for TENA application directory.

In addition to ensuring that appropriate header files are included in the application source code and the build file include statements specify the correct object models, the application build files also need to link against the appropriate object model definition libraries. The object model definition library name includes the object model version identifier, the computer platform designator, and the middleware versions (e.g., `-lExample-Vehicle-v1-$(TENA_PLATFORM)-v$(TENA_VERSION)`).

Object models may contain [Local Class](#) implementations that are needed to use a particular object model within an application. An object model implementation is needed when a non-default local class constructor or local class methods are defined.

When object model implementation code is necessary, the TENA application is required to also link against the appropriate object model implementation library. Generally, there may be multiple object model implementations for the same object model definition and the middleware was designed so that only the build file library link specification needs to be changed in order to use a different object model implementation.

Due to potential linker optimization deficiencies, TENA applications are required to include an object model implementation header file that forces linkage with the object model implementation library. The header file include statements only reference the name of the local class so that there are no application source code changes to switch from one object model implementation to a different implementation. An example of the include statement in the source code is, `#include <TENA/GPSTime/Impl.h>` which is for the `GPSTime` local class. In order to isolate the application code from the object model implementation name and version, the build file will need to specify the appropriate include file directory and library, e.g., `-I$(TENA_HOME)/$(TENA_VERSION)/include/OMs/TENA-Time-v2, -lTENA-Time-v2-$(TENA_PLATFORM)-v$(TENA_VERSION)`.

In summary, the requirements for building and linking applications against object models are shown in the table below.

Application Requirement	Example
Include object model header file, typically in main C++ file.	<code>#include <Example/Vehicle.h></code>
Add object model header file directory to build file.	<code>-I\$(TENA_HOME)/\$(TENA_VERSION)/include/OMs/Example-Vehicle-v1 (for UNIX Makefiles)</code> <code>Properties:C/C++:Additional Include Directories add \$(TENA_HOME)\\$(TENA_VERSION)\include\OMs\Example-Vehicle-v1 (for Windows Visual Studio)</code>
Add object model definition library to build file.	<code>-lExample-Vehicle-v1-\$(TENA_PLATFORM)-v\$(TENA_VERSION) (for UNIX Makefiles)</code> <code>Properties:Linker:Input:Additional Dependencies add libExample-Vehicle-v1-\$(TENA_PLATFORM)-v\$(TENA_VERSION).lib (for Windows Visual Studio)</code>
Include object model implementation header file, if implementation required.	<code>#include <TENA/GPSTime/Impl.h></code>
Add object model implementation header file directory to build file, if implementation required.	<code>-I\$(TENA_HOME)/\$(TENA_VERSION)/include/OMs/TENA-Time-v2 (for UNIX Makefiles)</code> <code>Properties:C/C++:Additional Include Directories add \$(TENA_HOME)\\$(TENA_VERSION)\include\OMs\TENA-Time-v2 (for Windows Visual Studio)</code>

Add object model implementation library to build file, if implementation required.

-lTENA-Time-v2-fullConversion-v2-\$(TENA_PLATFORM)-v\$(TENA_VERSION) (for UNIX Makefiles)
Properties:Linker:Input:Additional Dependencies add libTENA-Time-v2-fullConversion-v2-\$(TENA_PLATFORM)-v\$(TENA_VERSION).lib (for Windows Visual Studio)

Building and Linking against Unnecessary Object Models

Applications may incorrectly build and link against unnecessary object models. Although there is no application problem (from a software perspective) in building and linking against unnecessary object models, it will cause the application to be dependent on that particular object model version when joining a TENA Execution. When an application joins an execution, the Execution Manager will ensure that all applications joining the execution are using compatible object model versions. If an application is unnecessarily dependent on a particular object model version, it can cause a problem when a newer version of the object model is being used by other applications in the execution. Therefore, it is prudent for an application to only build and link against object models that are actually needed by the application.

Although a future capability may be developed within the TENA Middleware to automatically notify developers when the application is built and linked against unnecessary object models, currently it is the responsibility for the developer to only use needed object models. The most straight forward technique to determine whether an object model is necessary is to remove (test by commenting out the #include statements) all of the object model definition (and implementation, if any exist) header files associated with a particular object model. The header files associated with a particular object model can be determined by examining the directory structure under \$(TENA_HOME)/\$(TENA_VERSION)/include/OMs for the object model being evaluated.

If all of the object model definition (and implementation) header files for a particular object model are commented out and the application continues to successfully build, then that object model is not needed and all references to the object model should be removed. In the source code, the include statements that were commented out can be deleted. The build files should then be modified to remove references to the object model as well.

TENA in MFC Applications

TENA in MFC Applications

This documentation provides guidance related to the use of TENA in Microsoft Foundation Class (MFC) applications.

Introduction

Microsoft Foundation Classes (MFC) is a C++ library that provides access to the Microsoft Windows API. MFC is available to applications developed with Microsoft Visual Studio if the "Use of MFC" setting in the project's configuration properties is set to "Use MFC". Applications that use MFC are often called "MFC applications".

Differences From Non-MFC Applications

No main Function

MFC applications do not allow user code to be placed in a `main` function, as is normally done in C++ applications. Instead, the MFC application developer typically writes classes that derive from `CWinApp` and `CFrameWnd`, and implements methods on those classes.

TENA uses [ACE](#), which typically uses the standard C++ `main` function. Therefore, MFC applications must take a special step to inform ACE that the application is an MFC application. This is done by defining `ACE_HAS_MFC=1` in the MFC application's Visual Studio project: Project -> Properties -> Configuration Properties -> C/C++ -> Preprocessor -> Preprocessor Definitions -> Add "ACE_HAS_MFC=1" to the list of definitions.

If you forget to do this, the compiler will produce the following error message when compiling the TENA Windows configuration header files:

```
fatal error C1189: #error : "When building MFC applications, you must define ACE_HAS_MFC=1"
```

No Direct Inclusion of Windows.h

MFC applications should not `#include Windows.h` directly.

In order to get declarations for standard windows functions and classes, MFC applications should `#include afxwin.h` (which internally `#includes Windows.h`). Any attempt to `#include Windows.h` before `#including afxwin.h` will result in a compile error like this:

```
fatal error C1189: #error: WINDOWS.H already included. MFC apps must not #include <windows.h>
```

It is safe to `#include Windows.h` after `#including afxwin.h`, since the compiler sees that `Windows.h` was already `#included` (via `afxwin.h`) and so skips the second `#inclusion`.

TENA MFC applications generally need not be concerned about this issue. TENA applications should generally `#include "TENA/Middleware/config.h"` before any other `#includes`. If the application being compiled is an MFC application, "TENA/Middleware/config.h" will detect this and will `#include "afxwin.h"`.

⚠ – TENA Release 6.0 Beta 4 and earlier had a shortcoming: "TENA/Middleware/config.h" `#included` `Windows.h`, even for MFC applications. To work around this, TENA MFC applications needed to `#include "afxwin.h"` (or "afxext.h") **prior** to `#including "TENA/Middleware/config.h"`.

WINVER Must Be #Defined

In MFC applications, the `WINVER` C PreProcessor symbol must be defined before including any MFC header files (such as "afxwin.h"). Failure to do so will produce a compile-time warning like the following:

```
WINVER not defined. Defaulting to 0x0502 (Windows Server 2003)
```

TENA MFC applications generally need not be concerned about this issue. "TENA/Middleware/config.h" (which should be `#included` first) detects whether the application being compiled is an MFC application, and if so, it will `#define WINVER` if it wasn't already defined.

Related Topics and Links

- [Microsoft Foundation Classes](#)
- [ACE](#)
- ⚠ MW-5367 - Compiler errors RESOLVED

API Naming Conventions

API Naming Conventions

The TENA Middleware API attempts to follow certain naming conventions for software elements (e.g., classes, methods, attributes) that are part of the API (Application Programming Interface). Consistent conventions help with understanding and readability of the software.

Description

CamelCase Naming Convention

All of the TENA Middleware API elements follow the "CamelCase" naming convention, summarized as:

- Whether the first character of the name is capitalized depends on type of software element (see type conventions below).
- Subsequent words within a name are capitalized, unless following an upper case acronym, in which case the word used lower case (e.g., GPSpowerLevel).
- Acronyms use the same case (whether upper or lower case).

Namespaces

- The TENA Middleware uses Namespaces to organize the API based on a functional decomposition. These namespaces start with a capital letter and may be nested within another namespace, e.g., TENA::Middleware::SDO. Multi-word namespaces follow the CamelCase convention.

Classes and Typedefs

- The various classes and Typedefs that make up the middleware API begin with a capital letter and follow the CamelCase convention.

Attributes

- Public attributes start with a lower case letter and follow the CamelCase convention.
- Private attributes start with an underbar ('_'), then a lower case letter and follow the CamelCase convention.

Methods

- Methods start with a lower case letter and follow the CamelCase convention.
- If the method is associated with setting or getting an Object Model class or message attribute, the method is named `set_attributeName()`, `get_attributeName()`, and `is_attributeName_set()`, where `attributeName` is the actual name of the attribute defined in the object model.
- If the method has a return value with certain engineering units, the method name will end with "`In Units`", where `Units` is the particular units, e.g., `get_distanceInMeters()`.

Enumerations

- Enumerations start with an upper case letter and follow the CamelCase convention.
- To avoid scoping problems, object model enumerations will prefix each value with the name of the enumeration, e.g., `Color { Color_Red, Color_Green, Color_Blue}`, with each value after the prefix starting with a capital letter and following the CamelCase convention.

Exceptions

- Exceptions start with an upper case letter and follow the CamelCase convention.

Using TENA in a Library

Using TENA in a Library

This documentation provides guidance related to the use of TENA in a library that is loaded by the main program.

Introduction

Although it is unusual, in some circumstances it may be necessary to separate the application code that is making use of the TENA Middleware into a library. The main program would then dynamically load this library to utilize the TENA capabilities. Under Windows, this type of library has the extension .dll, and under UNIX the library can have the extension .so or .dylib (for the Mac OS). The mechanisms used to dynamically load libraries is different between Windows and UNIX operating systems, and are discussed in the following sections.

In the following code examples, there is a main application that will attempt to dynamically load the library containing the code that uses the TENA Middleware. The main application does not need to include any header files associated with the TENA Middleware. The separate library will contain code that performs the usual configuration processing, join a TENA execution, and perform TENA operations. The code below is greatly simplified in comparison to what a real application may need to do with respect to exchanging data and control between the main program and the dynamically loading library, but should be sufficient to provide the essential elements needed to support this type of application structure.

Windows TENA Library Example

The main application only needs to define the necessary entry point in the library that will be used after the library is loaded. For this example, the command line argc, argv parameters are the only arguments passed into the library function. Pseudocode for the library function is shown below.

```

#include <TENA/Middleware/config.h>
#include <TENA/Middleware/Runtime.h>
#include <TENA/Middleware/RuntimePtr.h>
#include <TENA/Middleware/ApplicationConfiguration.h>
#include <TENA/Middleware/Utils/ARGV.h>

#include <string>
#include <iostream>

#define TENA_export_int extern "C" __declspec(dllexport) int __cdecl

TENA_export_int startTENA( int argc, char* argv[] )
{
    int status = 0;
    try
    {
        TENA::Middleware::Utils::ARGV args( argc, argv );
        ApplicationConfiguration appConfig( args(argc), args(argv() ) );

        std::vector< TENA::Middleware::Endpoint > endpointVector(
            appConfig["emEndpoints"].getValue<
                std::vector< TENA::Middleware::Endpoint > >());
    }

    TENA::Middleware::RuntimePtr pRuntime(
        TENA::Middleware::init( appConfig.tenaConfiguration() ) );

    TENA::Middleware::ExecutionPtr pExecution(
        pRuntime->joinExecution( endpointVector ) );

    TENA::Middleware::SessionPtr pSession(
        pExecution->createSession( "TestSession" ) );

    // Do other TENA stuff
}

catch( std::exception const & e )
{
    TENA_ERROR << "Exception: " << e.what() << std::endl;
    status = -1;
}
catch( ... )
{
    TENA_ERROR << "Unknown exception" << std::endl;
    status = -1;
}
return status;
}

```

In the main application, a typedef is defined for the library function named "StartTENA" as shown in the following code fragment. The code shown attempts to dynamically load the library and invoke the function startTENA. Since the main application does not contain a definition for the function called from the library, it is necessary to cast the function pointer to match the expected signature defined by the typedef.

```

#include <windows.h>
typedef int (__cdecl *StartTENA)(int,char** const);

// other application header stuff

int
main( int argc, char* argv[] )
{
    // Do application stuff

    // Attempt to load the library that uses the TENA Middleware
    HMODULE hinstLib = LoadLibrary( (LPCSTR)TEXT("libTENAstuff.dll") );
    if ( hinstLib == NULL )

    {
        std::cerr << "Could not load library libTENAstuff.dll" << std::endl;
        return -1;
    }

    // Since the main application does not include the library function definition, it is necessary to perform a
    cast
    StartTENA startTENA = reinterpret_cast<StartTENA>( GetProcAddress( hinstLib, "startTENA" ) );

    if ( startTENA != NULL )
    {
        (startTENA)( argc, argv );
    }
    else
    {
        std::cerr << "ERROR: could not dynamically load function startTENA: "
        "GetProcAddress error " << GetLastError() << std::endl;
        return -1;
    }

    // Do other application stuff

    FreeLibrary( hinstLib );

    return 0;
}

```

UNIX TENA Library Example

The main application only needs to define the necessary entry point in the library that will be used after the library is loaded. For this example, the command line argc, argv parameters are the only arguments passed into the library function. Pseudocode for the library function is shown below.

```

#include <TENA/Middleware/config.h>
#include <TENA/Middleware/Runtime.h>
#include <TENA/Middleware/RuntimePtr.h>
#include <TENA/Middleware/ApplicationConfiguration.h>
#include <TENA/Middleware/Utils/ARGV.h>

#include <string>
#include <iostream>

#define TENA_export_int extern "C" __attribute__((visibility("default"))) int

TENA_export_int startTENA( int argc, char* argv[] )
{
    int status = 0;
    try
    {
        TENA::Middleware::Utils::ARGV args( argc, argv );
        ApplicationConfiguration appConfig( args(argc), args(argv() ) );

        std::vector< TENA::Middleware::Endpoint > endpointVector(
            appConfig["emEndpoints"].getValue<
                std::vector< TENA::Middleware::Endpoint > >());
    
```

TENA::Middleware::RuntimePtr pRuntime(
 TENA::Middleware::init(appConfig.tenaConfiguration()));

```

        TENA::Middleware::ExecutionPtr pExecution(
            pRuntime->joinExecution( endpointVector ) );
    
```

TENA::Middleware::SessionPtr pSession(
 pExecution->createSession("TestSession"));

```

        // Do other TENA stuff
    }
    catch( std::exception const & e )
    {
        TENA_ERROR << "Exception: " << e.what() << std::endl;
        status = -1;
    }
    catch( ... )
    {
        TENA_ERROR << "Unknown exception" << std::endl;
        status = -1;
    }
    return status;
}

```

In the main application, a typedef is defined for the library function named "StartTENA" as shown in the following code fragment. The code shown attempts to dynamically load the library and invoke the function startTENA. Since the main application does not contain a definition for the function called from the library, it is necessary to cast the function pointer to match the expected signature defined by the typedef.

```

#include <dlfcn.h>
typedef int (*StartTENA)(int,char** const);

// other application header stuff

int
main( int argc, char* argv[] )
{
    // Do application stuff

    // Attempt to load the library that uses the TENA Middleware

#if defined(__APPLE__)
    std::string libraryName( "libTENAstuff.dylib" );
#else
    std::string libraryName( "libTENAstuff.so" );
#endif

void * libHandle = dlopen( libraryName.c_str(), RTLD_LOCAL|RTLD_LAZY );
if ( !libHandle )

{
    std::cerr << "Could not load library " << libraryName << ":" << dlerror() << std::endl;
    return -1;
}

// Since the main application does not include the library function definition, it is necessary to perform a
cast
StartTENA startTENA = reinterpret_cast<StartTENA>( dlsym( libHandle, "startTENA" ) );

if ( startTENA != NULL )
{
    (startTENA)( argc, argv );
}
else
{
    std::cerr << "ERROR: could not dynamically load function startTENA: " << dlerror() << std::endl;
    return -1;
}

// Do other application stuff

dlclose( libHandle );

return 0;
}

```

Event Management

Event Management

TENA provides event management support with distributed alert messages, embedded diagnostics, and a management console.

Event Management Support

- [Alerts](#)
- [Startup Services](#)
- [TENA Console](#)

Alerts

Alerts

When the TENA Middleware, which is associated with a TENA application, encounters an unusual situation an "Alert" will be generated. By default this Alert will be written to the standard output stream associated with the application, as well as sent as a distributed message that is displayed for any TENA Console operators.

Description

Alerts are generated whenever the middleware encounters a condition that warrants a `Warning` or `Error` message. By default, these messages are sent to the standard output stream (i.e., `std::cout`), written to the TENA application's log file created in the "log" directory of the TENA installation, and sent to any active [TENA Consoles](#).

If an alert is generated by an application, it indicates that the application has encountered an abnormal condition that should be investigated. Users should check for alerts in the log files after the running of a TENA application that may have encountered abnormal behavior.

The [configuration parameters](#) associated with the TENA Middleware alert mechanism is shown in the table below.

Configuration Parameter	Description
<code>noErrorLog</code>	Disable error log file (messages will only go to <code>std::cout</code>).
<code>logDir <location></code>	The directory for diagnostic and error log files. : Default value = "TENA_HOME/TENA_VERSION/log".
<code>disableAlertOnError</code>	Disables sending of alert error messages to consoles.
<code>disableAlertOnWarning</code>	Disables sending of alert warning messages to consoles.
<code>disableAlertOnNote</code>	Disables sending of alert note messages to consoles.

In addition to the middleware using the Alert mechanism, applications can use the alert mechanism to generate application specific alerts. In addition to the `Error` and `Warning` alerts, the Alert mechanism supports a `Note` and an `Out` construct that operate in a similar manner (except that the `Out` mechanism only works locally and does not send alert messages to TENA Consoles). The `Note` is used to report useful information, but not necessarily something that is unusual or indicates a problem. The `Error`, `Warning`, and `Note` alerts are automatically decorated with information such as the date and time the alert was issued, and the file name and line number where the alert was generated. The `Out` construct does not provide any decoration.

Middleware Alert Information

Middleware Alerts are reported to TENA Consoles, log file, and/or the standard output for the application. The description of these alerts will sometimes contain detailed information from the underlying communication software (CORBA standard) used by the middleware. The following notes are intended to help understand these alerts, but users are encouraged to submit a MW helpdesk case if assistance in needed in understanding the middleware alerts.

- Transient Error
 - A transient communication error indicates that the middleware attempted to communicate with a distributed node, but there was no positive or negative acknowledgment for that communication. This can happen when there are certain types of communication faults.
 - The middleware will attempt to retry when a transient communication error occurs (middleware configuration parameter default value is three retries). If the communication retries succeeds, the middleware will issue an Alert of type `Note` to indicate that a transient error occurred. If the retries do not succeed, the middleware will issue an Alert of type `Error` to indicate that there was a communication failure and the middleware was unable to communicate with another distributed node.
 - If the middleware encounters a communication error or continues to encounter transient errors that succeed with retries, the network engineers should be notified to investigate. When there are communication errors, the execution may be in an inconsistent state and it may use the Repair Execution mechanism after the communication problems have been resolved.
- Timeout Error
 - When the middleware is waiting for a response to a communication request to a distributed node, a timer (middleware configuration parameter) will be used to define how long the middleware will wait for a response. If this timer expires, the middleware will issue a timeout error indicating that the particular communication attempt failed.
 - A timeout error will typically occur when the remote computer is completely unresponsive or unreachable. This type of error can also happen with firewalls.
- No Usable Profile
 - This error indicates that the middleware was unable to communicate to the other endpoint, and this error message typically accompanies other communication errors, such as transient errors.
- Object Not Exist
 - An object not exist error indicates that the middleware was able to communicate with a distributed node (e.g., another application), but the middleware associated with that node was unable to locate the object associated with the communication request.
 - An object not exist error can occur if an application abnormally terminates and is then restarted using the same communication endpoint.

Usage Considerations

Disabling Alerts going to Standard Output

If it is necessary for an application to disable alerts being sent to standard output (i.e., `std::cout`), the following code can be added after the application has called the `TENA::Middleware::init` function. This code removes `std::cout` from the alert streams.

```
TENA::Middleware::Utils::Debug::OstreamWrapper coutWrapper( std::cout );
TENA::Middleware::Utils::Debug::Logger::instance().removeStream( coutWrapper );
```

Some aspects of the middleware Alert mechanism are accessible to application developers in order to support more advanced features, such as filtering certain Alerts based on their content. Users attempting to perform this type of advanced processing with Alerts should submit a [helpdesk](#) ticket that explains the processing requirements.

Release 6.0.5 Update Middleware version 6.0.5 and beyond, execution scoped Alert macros were created (e.g., `TENA_EX_ERROR`, `TENA_EX_NOTE`) that allows the application developer to scope the Alert to a particular Execution if the application uses multiple Executions. The "EX" version of the macros requires `ExecutionID` as a parameter.

C++ API Reference and Code Examples

The TENA Alert constructs are defined in the header file `TENA_HOME/TENA_VERSION/include/TENA/Middleware/Utils/Debug/TENAostream.h`. There are four Alerts that can be used by application developers that correspond to the Error, Warning, Note, and Out alert constructs discussed above:

- `TENA_ERROR`
- `TENA_WARNING`
- `TENA_NOTE`
- `TENA_OUT`

These alerts are used as an `ostream` in which additional text strings can be added to the alert message. The code example below illustrates the use of these alerts.

```
catch ( std::exception const & ex )
{
    TENA_ERROR << "Unexpected Standard Exception: " << ex.what() << std::endl;
}
```

As mentioned above, these alert macros (except for `TENA_OUT`) will automatically insert the file name and line number from where the macro was used. Additionally, the timestamp (as determined by the clock value provided by the operating system) will be included in the alert message.

 — Users should be sensitive in the number of alert messages that may be generated through the use of these macros. It is sometimes useful to provide a rate-limiting mechanism to only use the macro the first several times it occurs, or only the first time every 15 minutes.

Startup Services

Startup Services

In some environments, it may be useful to automatically startup TENA utilities (e.g., Execution Manager) and tools (e.g., Data Collector) when the operating system is started. This page contains notes related to setting up services for various TENA utilities/tools on different operating systems.

Description

TENA Utility/Tool Prerequisites

In some situations it may be useful to automatically startup certain TENA utilities or tools when the computer operating system is started. The utilities and tools developed by the TENA SDA will attempt to support these startup service operations by designing the utilities/tools so that there is not a requirement for manual input or a need to support an output device (e.g., GUI, standard output). This support will typically be accomplished through configuration parameters, such as:

- noGUI – Disable the use of a GUI (graphical user interface), if a default GUI exists.
- noninteractive – Disable the need for standard input, if a default command-line interface exists.
- quiet – Disable writing to standard output or standard error.

The [Execution Manager](#), [TENA Data Collection System](#), and [TENA Canary](#) products have all been designed to enable operation as a startup service using configuration support to disable input and output devices. But providing the ability to disable input/output are only prerequisites to being able to have these utilities/tools be automatically launched when a computer operating system starts. In order to automatically run a TENA utility/tool at operating system startup it is necessary to use an appropriate mechanism and procedures for the particular operating system.

Operating System Startup Mechanisms

There is no common mechanism for starting services across the different operating systems. As expected, the various Windows operating systems operate quite differently from the UNIX operating systems with respect to startup services. And the different UNIX operating system flavors can operate quite differently in terms of the fundamental mechanism (e.g., BSD /etc/rc scripts, System V runlevel mechanism) and the procedural details.

Usage Considerations

At this time, the TENA SDA project is attempting to share information related to the use of these startup mechanisms, but is not attempting to build a generalized, cross-platform, startup service. This wiki page is intended to capture notes from the user community related to running TENA utilities and tools as operating system startup services. Users should submit a Middleware (MW) helpdesk case (<https://www.tena-sda.org/helpdesk/MW>) if they are able to share experiences/techniques associated with these startup services.

Windows Startup Mechanisms

- The preferred approach for developing a startup service for Windows is to use Visual Studio to create a Windows Services Application. The following MSDN page provides documentation for this development: <http://msdn.microsoft.com/en-us/library/y817hyb6.aspx>.
- Another approach that has been used in the past, that did not require Visual Studio development, was using Windows Resource Kit tools INTRSV.R, EXE and SRVANY.EXE. These tools have not been updated by Microsoft since releasing for Windows Server 2003, although some users continue to use them on later versions of Windows operating systems. The microsoft support article, <http://support.microsoft.com/kb/137890>, provides instructions for running an executable or batch file as a service using these tools. A simple batch file can be created for running the Execution Manager as shown in the code below.

```
@echo off  
start c:\TENA\6.0.3.1\bin\w7-vs2010-64\executionManager.exe -listenEndpoints myhost:55100 -nonInteractive -quiet
```

- The Cygwin project which provides an emulation of UNIX services for Windows computers provides a utility for running cygwin-based scripts and executables as a Windows service: <http://cygwin.com/cygwin-ug-net/using-cygserver.html>.
- There are likely freely available and commercial tools available to assist with running executables and batch files as Windows services, such as [NSSM](#) (<http://nssm.cc/usage>) and [AlwaysUp](#) (<http://www.coretechnologies.com/products/AlwaysUp/>).

⚠ – Note that the techniques and products listed above are provided without actual practical experience. Users attempting to use TENA utilities/tools as a Windows service are encouraged to research these and other options. Users are encouraged to share their experiences with the community and perhaps a common implementation can be supported in the future.

TENA Console

TENA Console

The TENA Console is an event management GUI (graphical user interface) utility that can be used for monitoring applications joined to an execution. Through the TENA Console, operators can examine the configuration parameters used by an application and the runtime operational parameters associated with the application. Network monitoring commands are also supported by the TENA Console to detect both Reliable and Best Effort communication problems. Multiple instances of a TENA Console can be simultaneously connected to an execution at various locations on the network supporting the execution.

Description

The TENA Console is an event management tool that assists operators in understanding the status of their TENA execution. Through the console, an operator can obtain information about the particular applications connected to an execution, including configuration settings and runtime diagnostics. The console can also be used to actively monitor communication between the different systems associated with the execution.

In some situations, it may be useful to utilize the TENA Console network monitoring capabilities with computer systems using a known TENA application. The [TENA Canary](#) was created for this purpose. The canary has a GUI for a simple TENA application that can be used in conjunction with the TENA Console capabilities.

The TENA Console and Canary User Guides are found through the following links:

- [Console User Guide](#)
- [Canary User Guide](#)

Execution Manager

Execution Manager

Every TENA execution utilizes an Execution Manager process to manage group membership. When TENA applications attempt to join or resign from the execution, they are required to perform these requests with the Execution Manager. The Execution Manager is used to introduce publishing and subscribing applications to enable them to make direct connections when appropriate. Execution wide configuration parameters, such as multicast properties and object model definitions, are maintained by the Execution Manager. Multiple Execution Managers can be utilized to provide fault tolerance support.

Description

A TENA Logical Range Execution requires an Execution Manager process operating on the common network. An **Execution Manager (EM)** software process exists for each particular execution running on the network. The EM is used to ensure consistent object model definitions, maintain execution-specific information and settings, assist in exchanging initialization information when an application joins an execution, and provide other execution management services.

When applications attempt to join an execution, the EM will perform several "check-out" operations and assign the application with an execution unique `ApplicationID` (see [Middleware IDs](#)). The application check-out operations include the following items:

- **Compatible Middleware Version Checking** — When an application attempts to join an execution, the EM will check the middleware version of the joining application and compare with the middleware version of the EM. If the joining application is using an incompatible middleware version, it will be prevented from joining the execution. Middleware version identifiers use three integers, `I.J.K`, where a common `I` indicates API-compatibility and a common `J` indicates runtime-compatibility. Therefore the middleware compatibility checking, by default, ensures that the `I.J` is the same for the EM and the joining application. If appropriate, the event coordinator can use the EM configuration parameter, `requireVersionMatch`, to force the joining application to match the middleware version exactly. See the [TENA Versioning Conventions](#) page for additional information.
- **Object Model Consistency Checking** — When an application attempts to join an execution, every object model construct (e.g., class, message, constant) used by that application is registered with the EM, along with a checksum of each OM type definition (and value in the case of a constant). If an application attempts to register an object model construct that has a different checksum, an object model definition inconsistency is detected, and the application's request to join is denied. Additionally, for each joining application, the EM will record the name for each registered object model implementation (associated with local classes and messages), and a mismatch in the object model implementation name provided by a joining application is compared to any object model implementations that are already registered for currently-joined applications will cause the join request to fail. To permit different object model implementation versions to be used in the execution, specify the `allowImplementationMismatch` configuration option for the EM. See the [Object Model Consistency Checking](#) page for additional information.
- **ArbitraryValue Checking** — The TENA Object Model Compiler will automatically generate complete working applications based for object models submitted by users. For these applications to build and run "out of the box" without modification, the object model compiler must provide arbitrary values for the attribute values of the various objects and messages, as well as return values for methods. A special C++ class, `ArbitraryValue`, was created to support the use of arbitrary values for various data types (e.g., value of 1 for an integer, value "one" for a string). The use of the `ArbitraryValue` type simplifies the testing of automatically-generated applications, but these arbitrary values should not be used for "real" events.
 - **⚠ Release 6.0.5 and higher behavior** – By default, the EM will NOT allow applications using arbitrary values to join the execution. The EM configuration parameter, `allowArbitraryValue`, can be specified to allow applications that are using the `ArbitraryValue` type to participate in the execution.
 - **⚠ Prior to Release 6.0.5 behavior** – The EM configuration parameter `disallowArbitraryValue` can be used by an event operator to ensure that all applications attempting to join the execution were compiled without the use of `ArbitraryValue`, and, therefore, ensure that proper user code has been added for setting attribute values and method return values. See the [Arbitrary Values](#) page for additional information on this mechanism.
- **Communication Checking** — When an application attempts to join an execution, the EM will attempt to establish a network connection using TCP/IP with the requesting application. Although the application can communicate to the EM, a test of the reverse communication is used to ensure that there are no firewall restrictions that prevent the EM and other applications from establishing connections with the application attempting to be join. If the connection attempt fails, the EM will prevent the application from joining the execution by throwing a runtime exception to the application indicating that there was a communication problem, potentially caused by a firewall.

In addition to managing the group membership (i.e., application joins and resigns) for the execution, the EM supports the necessary broadcast communication mechanism to announce application subscription interests. When an application first subscribes to a particular object model type, it must broadcast that interest to all applications. When a publishing application receives notification of these subscription interests, the middleware checks if the application is publishing the object model type (and Advanced Filtering criteria, if used) that matches the subscription interest. If there is a match, the publishing application will attempt to establish a **direct** connection with the subscribing application to send object discoveries, updates, destructions, or messages. Note that Best Effort multicast traffic does not require a direct TCP/IP connection for sending updates or messages, but a TCP/IP connection is still needed for object discoveries and destructions.

The Execution Manager is responsible for application heartbeat monitoring, unless this feature is disabled using the `disableHeartbeat` configuration parameter. The heartbeat interval is defined for all applications with the `heartbeatIntervalInSeconds` parameter. Application heartbeat processing instructs each application to periodically send a communication ping (using TCP/IP) to the EM. If the EM does not receive a heartbeat ping within a time interval defined by twice the `heartbeatIntervalInSeconds`, then the EM will issue an Alert message to notify all replica EMs and TENA Console operators of a potential problem.

Similar to applications, [TENA Consoles](#) connect to the Execution Manager to obtain information related to the execution. The EM notifies each connected Console whenever there are membership and/or status changes for the execution by sending alerts. See the [Alerts](#) page for additional information regarding execution Alerts.

Using a centralized process, such as the Execution Manager, significantly simplifies certain operations associated with the distributed event. For example, consider the additional complexity and performance cost in detecting object model inconsistencies in a dynamic membership environment without a centralized process. The downside of using a centralized process for these algorithms is the potential of a fault in the operation of the EM, due to a communication problem, a computer problem, or a software problem. In order to mitigate the risks and side-effects associated with a single point of failure, the EM supports an optional fault-tolerance mechanism. This mechanism allows the event operators to start multiple Execution Managers on different computers at different locations on the network. These multiple EMs work together in a primary-secondary arrangement, whereby a secondary (or replica) EM will automatically take over for a primary EM in the case of a fault. Additional information related to this mechanism can be found on the [EM Fault Tolerance](#) page.

⚠ AVAILABLE only in Release 6.0.5 and higher, the Execution Manager provides an automated save/recovery capability to improve the availability of the execution management capabilities by maintaining a recovery file (actually a SQLite file) that contains the current state of the EM. If the EM is abnormally terminated, the recovery file allows the EM to be re-started with the state that was previously saved. This feature is enabled when the `recoveryDir` configuration parameter is specified. Please refer to the [EM Recovery](#) page for additional information.

In summary, the Execution Manager is a logically centralized process (logically centralized, because the fault tolerance mechanism permits multiple, redundant processes to be established) that coordinates membership, group communication, and other execution-wide activities. Each execution is required to have an Execution Manager process. Although Execution Managers are used to broker connections between the applications joined to the execution, specific object updates and messages do not flow through the Execution Manager.

Additional topics related to the Execution Manager can be found in the following pages:

- [EM Configuration Parameters](#) — Description of configuration parameters associated with the Execution Manager.
- [Multicast Support](#) — TENA applications can utilize the UDP multicast protocol for sending SDO updates and messages.
- [OM Consistency Checking](#) — Execution Manager mechanism to detect incompatible object model definitions and implementations.
- [EM Fault Tolerance](#) — Mechanism to start multiple Execution Managers to tolerate faults caused by software, computer, or network problems.
- [EM Recovery](#) — Execution Manager mechanism to improve fault-tolerance and promote precise execution-management capabilities.
- [Running EM as a Service](#) — Mechanism to install and run the Execution Manager as a service on a specified host.

Usage Considerations

The first step in creating a TENA execution is to start the EM process. This can be done manually or using the [TENA Console](#) tool. In both cases, the user must provide the network endpoint(s) (i.e., `listenEndpoints`) that the EM will use to communicate with other TENA applications and utilities. The endpoint provided to the EM is a combination of the host name (or IP address) of the network interface and the port number. The port number is an integer between 1024 and 65535. (Well-known ports in the range 0-1023 are not allowed — web servers, for instance, already use port 80). Users must select a port number between 1024 and 65535 that is not already in use. If the requested port is already in use, the EM will terminate and display a diagnostic message stating that the requested port is already in use on that machine and network interface. Note that multiple processes on a single machine may not bind to the same port.

TENA Middleware SDK Product

An example manual invocation of the EM (as installed by the TENA Middleware SDK Product installer) is shown below using the network interface named `homer`.

```
executionManager -listenEndpoints homer:55100
```

Execution Manager Product

The EM can also be installed as a separate TENA Product. An example manual invocation of the EM (as installed by the EM Product installer) is shown below using the network interface named `homer`. Note that the EM start script is located in the `$TENA_HOME/executionManager-<version>` subdirectory.

```
start.sh -listenEndpoints homer:55100
```

After the EM has been started, TENA applications can then attempt to join the execution. Each application is required to provide the network endpoint of the EM (i.e., `emEndpoints`), as well as the listening endpoint(s) for the application itself. An example invocation of an application on computer `marge` is shown below.

```
Example-Vehicle-Publisher-v1 -emEndpoints homer:55100 -listenEndpoints marge
```

Note that the port number associated with the `listenEndpoints` for the application is optional and the operating system will select an available port number automatically. On networks where there are firewalls that block incoming TCP connections, it may be necessary to specify the `listenEndpoints` port number and have the firewall configured to support incoming connection requests on that port.

i — Unless required because of firewall restrictions, it is recommended that applications don't specify the port number for their `listenEndpoints`. This is due to the possibility of a remote application attempting to use a bad cached connection after the application is re-started, during which there was an abnormal application termination or communication fault of the re-started application. In this situation, the application holding the bad cached connection may encounter additional communication delay in re-establishing a connection with the re-started application. By allowing the operating system to select a random port, any bad cached connection will not attempt to be used in communicating with the re-started application.

The behavior of the Execution Manager can be controlled by a number of configuration parameters, including `listenEndpoints`. These parameters are managed with the [TENA Configuration Mechanism](#). The key parameters are shown in the table below; refer to the [EM Configuration Parameters](#) page for a complete listing of parameters.

Parameter Name	Description
<code>listenEndpoints</code>	The endpoint(s) on which this process should listen for requests, e.g., use <code>-listenEndpoints <hostname>:50000</code> to listen on port 50000 on the host <code><hostname></code> .
<code>multicastProperties</code>	Enable use of multicast and specify the address, port and range. The format for the argument is: <code><baseMulticastAddress>:<portNumber>:<numberOfAddresses></code> .

As indicated by the parameter above, the Execution Manager is responsible for defining multicast parameters for the execution. UDP Multicast communication can be used to disseminate information between publishing and subscribing applications. Details on the multicast specification can be found on the [Multicast Support](#) page.

While running manually, the Execution Manager supports several simple input commands. These services are invoked by typing text commands into the window running the EM. The supported commands are listed in the table below.

Execution Manager Terminal Services

Command	Description
<code>show version</code>	Display the middleware version of the EM.
<code>show config</code>	Display the configuration parameters for the EM.
<code>show endpoints</code>	Display the endpoints of the EM.
<code>show allowables</code>	Display the supplied configuration of allowable participants.
<code>show om impl rules</code>	Display the supplied configuration of OM Implementation rules.
<code>set passphrase</code>	Define a passphrase that is required for any TENA Console attempting to attach to the execution.
<code>force rollover</code>	Forcibly rollover the primary EM to a replica EM, which becomes the new primary EM. ⚠ AVAILABLE only in Release 6.0.5 and higher.
<code>forcequit or forceexit</code>	Forcibly shutdown the EM. If the EM is using a recovery directory and applications are joined in the execution, the DB file will be retained. ⚠ AVAILABLE only in the EM version 6.0.7.2 and higher.
<code>quit or exit</code>	Instructs the EM to shutdown.

EM Configuration Parameters

Execution Manager Configuration Parameters

The Execution Manager is a centralized process used to assist in the management of a TENA execution. Event operators can configure the behavior of the Execution Manager through the setting of configuration parameters.

Description

The configuration parameters available for the [Execution Manager](#) are defined in the following table. Refer to the [configuration mechanism](#) documentation for how these parameters values can be set.

i — The default Execution Manager parameter values are usually appropriate for typical environments in which TENA applications are distributed across a wide-area network. If, however, all TENA applications will be connected on a single local area network (LAN) and the network is configured correctly, then parameters can be adjusted (typically by shortening timeouts and reducing numbers of retry attempts) for optimal performance. See documentation for [recommended settings for a stable, working, LAN environment](#).

Configuring the Execution Manager using Environment Variables

The Execution Manager configuration parameter values may be specified using environment variable. The configuration parameters shown below must be prefixed with "**TENA_EM_**". Any [TENA Middleware](#) configuration parameter value that is set as an environment variable for use by the Execution Manager must be prefixed with "**TENA_MW_**".

i – visible to MW-team only.

! The Description text should match what is in the code. The attributes are also listed in the order that they show up in the code.

There are middleware parameters used by the EM that should be listed above, but there does not seem to be an easy way to determine which middleware parameters are used and what are not used.

Execution Manager Configuration Parameters

Configuration Parameter	Description	Default Value	Data Type
listenEndpoints	The endpoint(s) on which this process should listen for requests, e.g., use -listenEndpoints <hostname>:55100 to listen on port 55100 on the host <hostname>. The port value is optional and it is recommended to allow the middleware to select the port value. It may be helpful to specify the allowable port range using the portspan option, such as "192.168.1.1:55100/portspan=10". Note: IANA (www.iana.org) recommends using ports in the range 49152 - 65535 for private uses.	User required.	std::vector<TENA::Middleware::Utils::Endpoint>
emEndpoints	The endpoint(s) on which the primary execution manager is listening for requests e.g., -emEndpoints <hostname>:55100 to listen on port 55100 on the host <hostname>. Used when this executionManager is to act as a secondary EM. Multiple EM endpoints need to be separated by semicolons and enclosed in double quotes.	No secondary EMs used.	std::vector<TENA::Middleware::Utils::Endpoint>
multicastProperties	Enable use of multicast and specify the address, port and range. The format for the argument is: <baseMulticastAddress>:<portNumber>:<numberOfAddresses>. IANA (www.iana.org) advises that multicast addresses range between 239.192.0.0 and 239.255.255.255 and that port numbers range between 49152 and 65535.	No multicast is used.	Utils::MulticastProperty
requireVersionMatch	Require joining agents to have the exact same TENA Middleware version as the ExecutionManager. The default behavior is to only ensure that the first two version numbers of the middleware version match. For example, version 5.2.1 and 5.2.2 are permitted without this configuration parameter set.	not set	Parameter is a "flag" type, in which there is not an explicit data type used; the parameter is either set or not set.

minimumVersion	Specify a minimum middleware version, allowing applications that have a later middleware version to join. Applications that have an earlier middleware version are prevented from joining the execution. The format of the option is "-minimumVersion <version>", where version.major is required (e.g. "-minimumVersion 6"). The other components of the version number (minor, patch, and extra) are optional. Optional values which are not specified assume a default value of '0' (equivalent to "-minimumVersion 6.0.0.0"). The configuration parameters "-requireVersionMatch" and "-minimumVersion" are mutually exclusive. If both are specified, an error message is displayed and the EM fails to start. If the version of the EM (e.g. 6.0.5) is less than the specified minimum version (e.g. 6.1), a TENA_WARNING will be displayed and the EM will start normally. ⚠ — Available in executionManager version 6.0.5 and beyond.	not set	std::string
announcementThreads	Number of threads to use to process announcement events coming from user applications. Set to 0 to disable the feature. ⚠ — Available in executionManager version 6.0.9 and beyond.	13	std::uint32
allowCompatibleBetaVersions	Allow applications joining a non-Beta execution to use a compatible TENA Middleware Beta release. The default behavior is to allow Beta releases to interact only with the same or one version earlier Beta release. The major and minor version numbers are required to match. In addition, the newer Beta release must be marked as compatible with earlier versions. This option does not affect replica EMs.	not set	Parameter is a "flag" type, in which there is not an explicit data type used;the parameter is either set or not set.
allowCompatibleUntaggedVersions	Allow applications joining an execution to use a compatible TENA Middleware untagged version. The default behavior is to allow untagged versions to interact only with the same untagged version. The major and minor version numbers are required to match. This option does not affect replica EMs.	not set	Parameter is a "flag" type, in which there is not an explicit data type used;the parameter is either set or not set.
nonInteractive	Disable the interactive console command interface, although output may still be written to the console window (depending on the "quiet" option).	not set	Parameter is a "flag" type, in which there is not an explicit data type used;the parameter is either set or not set.
verbose	Use the verbose setting for writing output information.	not set	Parameter is a "flag" type, in which there is not an explicit data type used;the parameter is either set or not set.
quiet	Suppress standard output.	not set	Parameter is a "flag" type, in which there is not an explicit data type used;the parameter is either set or not set.
passphrase	The passphrase used to restrict console access. Using a passphrase requires console operators to provide that passphrase to connect to the execution. The passphrase cannot contain single or double quotes.	not set	std::string

registrationAlwaysFails	For testing purposes, always raise failedToRegisterExecutionAgent execution when application attempts to join.	not set	Parameter is a "flag" type, in which there is not an explicit data type used;the parameter is either set or not set.
allowImplementationMismatch	Permit an application to join this execution, if it provides an object model implementation description that is different from that of a currently-joined application. Note that this option and the <code>omImplementationRulesFile</code> option are mutually exclusive.	not set	Parameter is a "flag" type, in which there is not an explicit data type used;the parameter is either set or not set.
allowArbitraryValue	Allow an application to join this execution if it was compiled with <code>TENA_MIDDLEWARE_ALLOW_ARBITRARY_VALUE</code> defined. By default, applications using arbitrary values are prevented from joining the execution. Automatically-generated applications use arbitrary values to aid in providing working applications, but these arbitrary values should be removed for operational events. ⚠ — Available in executionManager version 6.0.5 and beyond.	not set	Parameter is a "flag" type, in which there is not an explicit data type used;the parameter is either set or not set.
disallowArbitraryValue	Prevent an application from joining this execution if it was compiled with <code>TENA_MIDDLEWARE_ALLOW_ARBITRARY_VALUE</code> defined. ⚠ — Deprecated in executionManager version 6.0.5 and beyond, as this is now the default behavior.	not set	Parameter is a "flag" type, in which there is not an explicit data type used;the parameter is either set or not set.
dontSendArbitraryValueNote	Allow an application to join this execution if it was compiled with <code>TENA_MIDDLEWARE_ALLOW_ARBITRARY_VALUE</code> defined, but don't send a Note to the TENA Console saying the Application was allowed to join.	not set	Parameter is a "flag" type, in which there is not an explicit data type used;the parameter is either set or not set.
writeEndpointToFile	Used to write the execution manager endpoint to a file.	not set	<code>std::string</code>
rmiTimeoutInMillisconds	Timeout duration (in milliseconds) used by the execution manager when invoking a twoway method. ⚠ — Obsolete in executionManager version 6.0.3 and beyond. Use the "twowayTimeoutInMilliseconds" Middleware Configuration option instead.	20000	<code>TENA::uint32</code>
heartbeatIntervalInSeconds	Sets the heartbeat interval used by applications, execution managers and consoles. Heartbeat processing is used to detect communication or application faults. The minimum value is 15 seconds.	60 (seconds)	<code>TENA::uint32</code>

disableHeartbeat	Disables the use of heartbeat processing for all participants.	not set	Parameter is a "flag" type, in which there is not an explicit data type used; the parameter is either set or not set.
allowableParticipantsFile	The name of the local file that contains the list of hostnames/IP addresses from which TENA applications and Consoles will be allowed to join/observe this execution. The list of allowable participants is static and cannot be changed during runtime. If a TENA application (or TENA Console) attempts to join this execution from an IP address that is not in the list of allowable participants, a NetworkError is generated. This parameter is only valid for the primary EM. Any attempt to use this parameter for a replica EM will cause the replica EM to fail. The list of allowable participants is communicated to all replica EMs. See usage notes below for additional information regarding this file. ! — Available in executionManager version 6.0.3 and beyond.	not set	std::string
omImplementationRulesFile	By default, a TENA Execution will require that all object model local class and message implementations are consistent and prevent applications from joining the execution with a different implementation. An implementation rules file can be specified to control this object model implementation consistency checking. The details associated with this file are described below. Note that this option is mutually exclusive with the allowImplementationMismatch option. ! — Available in executionManager version 6.0.4 and beyond.	not set ! — Available in executionManager version 6.0.6.1 and beyond. A default OM Implementation Rules file is provided in the executionManager's config directory (for compatibility of 6.0.4.2, 6.0.5.x and 6.0.6 TENA standard OM implementations).	std::string
recoveryDir	The directory for the database containing EM state. When this option is set, the EM will both save, and where relevant, recover, its state using the database in the specified recoveryDir. If the specified directory does not exist, it will be created. In addition, the recovery directory must be writeable. If not, an exception is thrown and the EM terminates. ! — Available in executionManager version 6.0.5 and beyond. ! — Prior to executionManager version 6.0.9, if this option was not set, then both save and recovery capabilities are disabled for the EM.	On Windows, the default recovery directory is: <ul style="list-style-type: none">%APPDATA%\TENA\recoveryDir On all other platforms, the default recovery directory is: <ul style="list-style-type: none">\$HOME/TENA/recoveryDir ! — Prior to executionManager version 6.0.9, no default value for the recovery directory was set, although the config file shipped with the executionManager set the value to: <ul style="list-style-type: none">TENA_HOME/TENA_VERSION/save	std::string
disableRecovery	If this option is set, then both save and recovery capabilities are disabled for the EM. ! — Available in executionManager version 6.0.9 and beyond.	Not set	Parameter is a "flag" type, in which there is not an explicit data type used; the parameter is either set or not set.
twowayTimeoutInMilliseconds	Timeout duration (in milliseconds) used by the execution manager when invoking a twoway method. ! — Available in executionManager version 6.0.3 and beyond. For versions prior to 6.0.3, use "rmiTimeoutInMilliseconds.	20000	TENA::uint32

Hidden Parameters

This section is only visible to members of the TENA SDA "team" (i.e., `TENA-team`).

The following parameters are defined as "hidden" and are not visible to users when configuration parameters are listed. The justification for hiding the visibility of these parameters may be to avoid user confusion for parameters that should not typically be modified, or the parameters may negatively affect the behavior of the application. Please contact the TENA Development Team for additional information or consideration in using these parameters.

Hidden Configuration Parameter	Description	Default Value
<code>allowAnyVersion</code>	Allow applications joining an execution to use any TENA Middleware version. No checking is made for compatibility between versions. This option does not affect replica EMs.	not set

Usage Considerations

`listenEndpoints` Argument

The Execution Manager is required to define the value for the `listenEndpoints` argument. This parameter requires the hostname or IP address associated with the network interface that should be used to communicate with other TENA applications. The port number for the `listenEndpoints` is not required to be specified, but may need to be used if there are firewall devices that have specific port numbers opened for communication with remote sites. Refer to the [Network Address page](#) for additional information.

`allowableParticipantsFile` Argument

The `allowableParticipantsFile` argument is optional and is **AVAILABLE only in Release 6.0.3 and higher**. If the `allowableParticipantsFile` argument is specified for the primary ExecutionManager, the contents of this file must be a simple list of allowed hostnames and/or IP addresses — one per line, e.g.:

```
yourHost.yourDomain.org  
192.168.12.24  
10.11.12.14
```

This is a list of not only IP addresses of machines running TENA Applications that can join the execution, but also the IP addresses of machines that can observe the execution (e.g., run a TENA Console or `adminConsole`). If a machine is not on the list, it can neither join nor observe the execution. Comments are permitted in the file and can be included as single entries or at the end of lines containing valid hostname/IP address entries. If the specified `allowableParticipantsFile` is empty, missing or contains an invalid entry, an exception is thrown and the Execution Manager will fail.

The `allowableParticipantsFile` argument is only valid for the primary ExecutionManager. Any attempt to use this parameter for a replica ExecutionManager will cause the replica ExecutionManager to fail. The list of allowable participants is communicated to all replica ExecutionManagers.

`omImplementationRulesFile` Argument

⚠ — Available in EM Product 6.0.6.1 and beyond. A default OM Implementation Rules file is provided in the EM Product's config directory (for compatibility of 6.0.4.2, 6.0.5.x and 6.0.6 TENA standard OM implementations).

The Execution Manager (EM) will perform object model local class and message implementation consistency checking when applications join the execution. By default, the EM will prevent applications from joining an execution when they are attempting to use a different object model implementation. For example, if one application joined the execution using the `TENA-Time-v2-fullConversions-v3` implementation and another application attempted to join using the `TENA-Time-v2-fullConversions-v2` implementation, the EM would prevent that application from joining because of the implementation difference. In this particular situation preventing the application from joining is the correct behavior because the `-v3` implementation was updated to account for a leap second that was added to the time conversion code.

In some circumstances, there may be multiple local class and message implementations that are compatible and should be permitted to be used in the execution. The `omImplementationRulesFile` configuration argument can be used to support this use case. In this situation, the people responsible for managing the TENA execution would create a file that contains the object model implementation consistency rules that would be used by the EM. Note that the `allowImplementationMismatch` and `omImplementationRulesFile` options are mutually exclusive, and passing both arguments to the EM will generate an exception.

The object model implementation rules file supports the following syntax and conventions.

1. File comments can be used when preceded by a '#' character.
2. Individual rules are defined in a single line within the file as:

```
<OM_Name>[1].<Type_Name>[0-1]:<Implementation_Name>[1+].
```

where:

OM_Name is the name of the object model along with the version (e.g., TENA-TSPI-v5). Each rule line must define the object model name, which is indicated by the cardinality notation "[1]".

Type_Name is the fully qualified name of the particular local class or message type to which the rule applies (e.g., TENA::UTCTime). Since an object model may define multiple local class and message types, the rule may only apply to a particular type. If the Type_Name is not specified, then the rule is assumed to apply to all local class and message types that have an implementation (indicated by the cardinality notation "[0-1]"). Note that a rule that specifies the OM_Name.Type_Name takes precedence over a rule that only specifies the OM_Name.

Implementation_Name list the names of all valid implementations that can be used for the specified OM Name and, optionally, specific Type Name. At least one Implementation Name must be provided and multiple names can be listed using whitespace as the separator (indicated by the cardinality notation "[1+]").

MW 6.0.4.2 and MW 6.0.5 and MW 6.0.6 Interoperability

```
# For TENA MW 6.0.4.2 and MW 6.0.5 and MW 6.0.6 interoperability
TENA-TSPI-v5:fullConversion-v2 StdImpl,1.0.0 StdImpl,1.0.1
TENA-PlatformType-v2:csvReader-v2 StdImpl,1.0.0 StdImpl,1.0.1
TENA-Time-v2:StdImpl,1.0.0 StdImpl,1.0.1 StdImpl,1.1.0 StdImpl,1.1.1 StdImpl,1.1.2
TENA-UniqueID-v3:userDefined-v2 StdImpl,1.0.0 StdImpl,1.0.1
TENA-PlatformDetails-v4:checkValues-v2 StdImpl,1.0.0 StdImpl,1.0.1
TENA-Radar-v3.1:checkValues-v1 StdImpl,1.0.0 StdImpl,1.0.1
TENA-Engagement-v4:checkValues-v2 StdImpl,1.0.0 StdImpl,1.0.1
TENA-Exercise-v1:constructors-v2 StdImpl,v1.0.0 StdImpl,1.0.1
```

Some examples for the implementation rules are shown below.

OM Implementation Rules Examples

Only allow StdImpl,1.0.0 and StdImpl,1.0.1 for the TENA-TSPI-v5 object model.

TENA-TSPI-v5:StdImpl,1.0.0 StdImpl,1.0.1

Only allow fullConversion-v3 and fullConversion-v4 for the TENA-TSPI-v5 object model.

TENA-TSPI-v5:fullConversion-v3 fullConversion-v4

Only allow fullConversion-v2 for the TENA::UTC implementation in the TENA-TSPI-v5 object model.

TENA-TSPI-v5.TENA::UTCtime:fullConversion-v2

All applications using TENA-TSPI-v5 must use either fullConversion-v2 or all applications must use fullConversion-v3, as the two rules defined below operate in an exclusionary manner. Either implementation is acceptable for the execution and the application that joins first will define the accepted implementation.

TENA-TSPI-v5:fullConversion-v2
TENA-TSPI-v5:fullConversion-v3

All applications using TENA-TSPI-v5 must use either StdImpl,1.0.0 or all applications must use StdImpl,1.0.1, as the two rules defined below operate in an exclusionary manner. Either implementation is acceptable for the execution and the application that joins first will define the accepted implementation.

TENA-TSPI-v5:StdImpl,1.0.0
TENA-TSPI-v5:StdImpl,1.0.1

All applications using TENA-TSPI-v5 must use either fullConversion-v2 or fullConversion-v3, OR all applications must use either fullConversion-v3 or fullConversion-v4, as the following two rules operate in an exclusionary manner dependent on the order in which applications using these implementations join. If the first two different implementations are fullConversion-v2 and fullConversion-v3, then subsequent applications using fullConversion-v4 will not be allowed to join.

TENA-TSPI-v5:fullConversion-v2 fullConversion-v3
TENA-TSPI-v5:fullConversion-v3 fullConversion-v4

All applications using TENA-TSPI-v5 must use either StdImpl,1.0.0 or StdImpl,1.0.1, OR all applications must use either StdImpl,1.0.1 or StdImpl,1.0.2, as the following two rules operate in an exclusionary manner dependent on the order in which applications using these implementations join. If the first two different implementations are fullConversion-v2 and fullConversion-v3, then subsequent applications using fullConversion-v4 will not be allowed to join.

TENA-TSPI-v5:StdImpl,1.0.0 StdImpl,1.0.1
TENA-TSPI-v5:StdImpl,1.0.1 StdImpl,1.0.2

3. Note that implied OM Implementation rules (i.e., those not explicitly specified in a rules file) require that joining applications adhere to the OM implementations that are already in use at the time that they join the execution. In this situation, the first application that joins the execution with an object model local class or message implementation without a defined rule will specify the implementation name/version that must be used by all subsequent applications that join the execution using the local class or message. This is how object model implementation consistency checking works in TENA v6.0.3 and earlier.

4. In general the OM Implementation rules file is provided to enforce specific requirements regarding the use of OM implementations for execution participants (denote which OM implementations and versions are allowed). Though the rules for all OM implementations that are in use for an execution need not be specified in a rules file, the use of this file as a complete specification of all supported OM implementations for the execution is not precluded.

Related Topics and Links

- [Configuration Mechanism](#)
- [Middleware Configuration Parameters](#)

EM Fault Tolerance

Execution Manager Fault Tolerance

The Execution Manager (EM) is a centralized process used to support execution-wide activities and information related to a TENA execution. Since this process is susceptible to problems with the network, the computer running the process, or the software process itself, event coordinators can utilize the Execution Manager Fault Tolerance mechanism to eliminate or minimize the effects of these potential problems. This fault tolerance mechanism permits the configuration of multiple Execution Manager processes running on different computers across the network in a manner that allows a replica Execution Manager to quickly assume responsibilities for a failed primary EM.

Description

A number of "distributed algorithms" used internally by the TENA Middleware rely on the Execution Manager (EM) process. The Execution Manager adheres to a software design pattern referred to as the Singleton pattern, because logically there is a single instance of this process that can be used to synchronize and arbitrate activities. An example of a distributed algorithm implemented by the Execution Manager is the generation of an execution-wide unique application identifier (i.e., ID). Without the Execution Manager, each application would need to attempt to generate their own application identifier and then use some communication mechanism to check with every other application to ensure uniqueness.

Whenever there is a single software process required for the proper operation of a distributed event, there is a vulnerability that the process may encounter an abnormal termination, network failure, or computer failure. The Execution Manager Fault Tolerance capability enables event coordinators to configure multiple Execution Managers to tolerate these faults. The configuration involves identifying the location of the **primary** Execution Manager and the location of each replica Execution Manager.

An Execution Manager's location is specified by its [network address](#), which includes the host name or IP address of the network interface and the port number that the Execution Manager listens to for communication purposes. Event coordinators will prepare a list of Execution Manager network addresses that are planned to be used for a particular event. At the start of the TENA execution, the primary Execution Manager will be started on the appropriate machine, as shown in the command line example shown below (where `emPrimaryHostname` is the host name of the machine running the primary Execution Manager, and the 55100 is an example port number).

```
executionManager -listenEndpoints emPrimaryHostname:55100
```

The replica Execution Managers should then be started on their designated hosts. Replica Execution Managers can be run on the same machine as the primary, as well as other computers that are connected to the primary on a properly functioning computer network. The syntax used to start a replica Execution Manager includes a configuration option for the `emEndpoints` that defines the network address for the primary Execution Manager.

Additionally, the replica Execution Manager can provide multiple network addresses for the `emEndpoints` that lists other potential Execution Manager addresses, in the event that the primary Execution Manager can't be contacted. When the replica Execution Manager starts, it will process the list of `emEndpoints` attempting to communicate with each address to locate the primary Execution Manager for the event. If the replica Execution Manager is unable to locate a primary Execution Manager using the `emEndpoints` configuration list, it will report an error message and terminate. An example invocation for a replica Execution Manager is shown below with the replica hostname, `emReplicaHostname-1`, and additional replica hostnames, `emReplicaHostname-2` and `emReplicaHostname-3`.

```
executionManager -listenEndpoints emReplicaHostname-1:50101 \
  -emEndpoints "emPrimaryHostname:50100;emReplicaHostname-1:50101;emReplicaHostname-2:50102;emReplicaHostname-3:50103"
```

As each replica EM is started, it will attempt to communicate with the primary Execution Manager. Upon successful startup, the primary Execution Manager will add each replica Execution Manager to the "succession order" list, and ensure that the list is provided to every other replica Execution Manager. Once a replica Execution Manager is successfully registered with the primary Execution Manager, all related state information maintained by the primary Execution Manager will be relayed to the replica Execution Managers in the event that the primary Execution Manager has a fault.

When a primary Execution Manager fault is detected, the first available replica Execution Manager will be promoted to become the new, primary Execution Manager. The Execution Manager state replication and primary succession activities will occur transparently to the individual TENA applications, although there may be some additional communication and processing, when a replica is promoted to a primary or a new replica is added to the succession list.

Each TENA application that is participating in an event using the Execution Manager Fault Tolerance capability should know the list of Execution Managers (initial primary and replica EMs). When an application attempts to join the execution, the middleware will attempt to locate the primary Execution Manager according to the configuration option `emEndpoints` list. If the application exhausts the addresses specified in the `emEndpoints` list without locating a primary Execution Manager, an exception (`ExecutionManagerNotFound`) will be thrown. Note that applications are not required to list all of the Execution Managers, but providing the full list will improve the applications ability to join the execution in the event that the initial primary Execution Manager(s) have encountered a fault.

TENA applications specify the `emEndpoints` in a similar manner to the replica Execution Managers as shown in the example application invocation below.

```
Example-Vehicle-Publisher-v1 -listenEndpoints appHostname:60100 \
  -emEndpoints "emPrimaryHostname:50100;emReplicaHostname-1:50101;emReplicaHostname-2:50102;emReplicaHostname-3:50103"
```

During normal operations, TENA applications will only attempt to communicate with the Execution Manager during join, resign, subscription and heartbeat operations. At any time, if the application encounters an error while attempting to communicate with the Execution Manager, it will attempt to communicate with the next Execution Manager as defined by the order specified in the EM succession list. Internal to the TENA Middleware, an application obtains the EM succession list as part of the execution join operation, and this list may be updated as replica EMs are added or removed from the list. If the application is unable to locate a primary Execution Manager, the exception (`ExecutionManagerNotFound`) will be thrown.

Usage Considerations

Event coordinators are responsible for designing the Execution Manager Fault Tolerance configuration for an event, i.e., defining the network addresses for the primary and replica Execution Managers. This list of Execution Manager addresses should be communicated to all of the TENA application operators, so that each application is started with the correct `emEndpoints` list. Note that applications are not required to define multiple EM addresses, since access to only a single operational Execution Manager is necessary. But, providing multiple addresses is helpful in the event that there is an EM fault.

When defining the location of the primary and replica Execution Managers, the event coordinators should consider how many execution manager faults need to be tolerated, recognizing that an excessive number of replica Execution Managers may slow down the application join and resign operations due to the need for distributed state replication (although that delay should be minimal).

⚠ — Prior to Release 6.0.5, the fault tolerance mechanism required all Execution Managers to be started prior to any applications joining the execution or TENA Consoles connecting to the execution. **For Release 6.0.5 and higher, a replica Execution Manager can be started and registered in the execution at any time during the event.** Please refer to helpdesk case [MW-3292](#) for additional information.

Multiple Execution Manager instances can run on a single machine in order to handle a software fault, but it will not protect the execution from a fault caused by that computer or the network connected to that computer. Current guidance for event coordinators needing Execution Manager Fault Tolerance is to run a primary and replica Execution Manager on a single machine, and then start additional replica EMs on one or more machines on the same Local Area Network (LAN) as the primary machine. The rationale for keeping the primary and replica Execution Managers on the same LAN is to avoid network partitioning issues that occur in many Wide Area Network (WAN) events.

EM Fault Tolerance does not repair network partitions.

Network partitioning occurs when a portion of the network becomes disconnected from other portions of the network. When this partitioning occurs between Execution Manager instances, there can be a primary EM established on both sides of the partition, and the overall event will be in an inconsistent state. If the network partition is repaired, the existence of two primary Execution Managers will continue to propagate this inconsistent execution state (in fact, there will appear to be two separate executions, although there can still be connections between individual applications). When an operator detects that network partitioning has occurred, the subsequent behavior of the entire event with respect to those partitioned applications should be considered suspect and the event should be re-started, if necessary.

Force Rollover Command

⚠ This feature is AVAILABLE only in Release 6.0.5 and higher.

When a primary EM is started in a console window, the `"force rollover"` command can be used to forcibly rollover the primary EM to a replica EM, which becomes the new primary EM. This option is also supported in the TENA Console in the Execution Manager's tab. If the forced rollover command is issued and there are no replica EMs in the execution, an exception is displayed for the primary EM. After the forced rollover completes, one of the replica EMs will be the new primary EM. In addition, the primary EM that is being rolled over can either be terminated (shutdown normally) or remain in the execution (as a replica EM).

1. The result of the `"force rollover"` command for the old primary EM is shown in the output below, where the old primary EM remains in the execution as a replica EM.

```

> force rollover

Do you want to terminate this EM after the rollover is complete? Enter 'Y' or 'N'!
> N

NOTE: EM ID 0: Forced rollover of replica EM ID 0 - IN PROGRESS. [2016-06-07T22:52:09.155009Z,TENA,
executionManager,trunk-6.0.5,Utility,source,all/ExecutionManagerImpl.cpp:6822:forceExecutionManagerRollover,
TID=0x7fff75217300]
>
NOTE: Signaling replica fault for replica EM ID 0 [2016-06-07T22:52:09.156347Z,TENA,executionManager,trunk-
6.0.5,Utility,source,all/ExecutionManagerImpl.cpp:6872:forceExecutionManagerRollover,TID=0x7fff75217300]
>
NOTE: EM ID 0: Sent a replicaFault event for EM ID 0 [2016-06-07T22:52:09.157523Z]
>
NOTE: EM ID 0: Handling a replicaFault event for EM ID 0 [2016-06-07T22:52:09.157731Z]
>
NOTE: EM ID 0: Replica EM ID 1 is considered Primary. [2016-06-07T22:52:09.157855Z]
>
NOTE: EM ID 0: Forced rollover of replica EM ID 0 - COMPLETED. [2016-06-07T22:52:09.158152Z,TENA,
executionManager,trunk-6.0.5,Utility,source,all/ExecutionManagerImpl.cpp:6895:forceExecutionManagerRollover,
TID=0x7fff75217300]
> >
NOTE: EM ID 0: Handling a newPrimaryReplica event for EM ID 1 [2016-06-07T22:52:09.158581Z]
```

2. The result of the "force rollover" command is shown in the output below, where the old primary EM is shutdown.

```

> force rollover

Do you want to terminate this EM after the rollover is complete? Enter 'Y' or 'N'!
> Y

NOTE: EM ID 0: Forced rollover of replica EM ID 0 - IN PROGRESS. [2016-06-07T22:56:56.298762Z,TENA,
executionManager,trunk-6.0.5,Utility,source,all/ExecutionManagerImpl.cpp:6822:forceExecutionManagerRollover,
TID=0x7fff75217300]
>
NOTE: Signaling replica fault for replica EM ID 0 [2016-06-07T22:56:56.300092Z,TENA,executionManager,trunk-
6.0.5,Utility,source,all/ExecutionManagerImpl.cpp:6872:forceExecutionManagerRollover,TID=0x7fff75217300]
>
NOTE: EM ID 0: Sent a replicaFault event for EM ID 0 [2016-06-07T22:56:56.301240Z]
>
NOTE: EM ID 0: Handling a replicaFault event for EM ID 0 [2016-06-07T22:56:56.301443Z]
>
NOTE: EM ID 0: Forced rollover of replica EM ID 0 - COMPLETED. [2016-06-07T22:56:56.301747Z,TENA,
executionManager,trunk-6.0.5,Utility,source,all/ExecutionManagerImpl.cpp:6895:forceExecutionManagerRollover,
TID=0x7fff75217300]
> >
NOTE: EM ID 0: Handling a newPrimaryReplica event for EM ID 1 [2016-06-07T22:56:56.302352Z]
> NOTE: Shutdown complete. [2016-06-07T22:56:58.330574Z]
```

EM Recovery

Execution Manager Recovery

The Execution Manager (EM) is a centralized process used to support execution-wide activities and information management related to a TENA execution. Since this process is susceptible to problems with (1) the network, (2) the computer running the process, or (3) the software process itself, event coordinators can utilize the Execution Manager Recovery mechanism to eliminate or minimize the effects of these potential problems. This EM recovery mechanism relies on a database persistence mechanism to store relevant EM state that can be used to recover the primary EM in the event of the abnormal termination or failure of the EM process.

 **This feature is AVAILABLE only in Release 6.0.5 and higher.**

Description

The Execution Manager is a process that synchronizes and arbitrates activities within an execution. For example, the Execution Manager can generate an execution-wide unique application identifier (i.e., ID). Without the Execution Manager, each application would need to attempt to generate their own application identifier and then use some communication mechanism to check with every other application to ensure uniqueness.

Whenever there is a critical software process required for the proper operation of a distributed event, there is a vulnerability that the process may encounter an abnormal termination, network failure, or computer failure. The EM recovery mechanism enables a failed EM to be recovered back to its most recent state prior to the failure. This allows execution-wide activities, such as joining or resigning, to continue after the EM is recovered. The use of the EM Recovery feature, along with the EM Fault Tolerance mechanism, vastly improves the availability of the execution management infrastructure.

The EM Recovery feature is implemented by having an EM save its internal state to an external archive. Currently, a SQLite database file is saved to a local disk and employed as the persistence mechanism. If the primary EM abnormally terminates for any reason, the primary EM can be recovered (at the same listenEndpoint that was in use previously) using the "recoveryDir" configuration option. If there were replica EMs in the execution when the primary EM abnormally terminated, and a new primary EM has NOT been designated prior to the old primary EM's recovery, then the restarted primary EM will recover as a primary EM. If a replica fault event occurred after the primary EM abnormally terminated, and a new primary EM was designated, then the recovered EM will register in the execution as a new replica EM.

Recovery of an EM

When an EM is started (from the command-line or using the TENA Console), that EM will save/recover its internal state when the `recoveryDir` option is specified. Note that only a primary EM can be recovered using the saved data. When a replica EM fails and is recovered, it will communicate with the primary EM to retrieve the latest state of the execution.

1. The EM save/recovery behavior is enabled when the `recoveryDir` configuration parameter is specified.
2. The location of the save/recovery directory is set using the `recoveryDir` configuration parameter.
3. If an EM terminates abnormally, the recovery capability is enabled on restart when the `recoveryDir` configuration parameter is specified. The recovery directory should be set to the same location each time the EM is started (before and after any abnormal termination).
4. Each EM has its own database file and is uniquely identified by its `listenEndpoints` value. The EM's state is saved in a SQLite database file in the designated recovery directory location, where the file name is `TENA_ExecutionManager_DB_<listenEndpoint>.sqlite`. The SQLite database is removed when the EM terminates normally.
5. The `listenEndpoints` value for a recovered EM must be the same as when the EM originally started.

 – Note that when a replica EM terminates abnormally and is recovered, it will recover as a new replica EM (ignoring the contents of the database and removing the old database).

 – Note that when a primary EM terminates abnormally and one or more replica EMs are registered in the execution, the execution may designate a new primary EM. In this case, when the old primary EM is recovered, it will determine that the execution has a new primary EM, and it will start as a new replica EM.

Usage Considerations

When an EM is started with the specified `listenEndpoints` value, the EM will check whether there is an existing SQLite database file, which contains data for an EM at the specified `listenEndpoint`. If a matching file exists, the EM will be recovered.

If the `recoveryDir` configuration parameter was specified when the EM was started initially (prior to abnormal termination), then this option must be specified with the same value for that EM to be recovered. When the EM process terminates abnormally (due to some fault), the EM process can be recovered without interruption to the execution participants. An example command for recovery of the EM using the network interface named `homer` is shown below.

```
executionManager -listenEndpoints homer:55100 -recoveryDir /TENA/6.0.5/save
```

In unusual circumstances, one may want to restart the EM (without recovery) at the same `listenEndpoints` (as was set for the EM prior to its termination). Doing so starts a new EM instance (denoting a new execution). Therefore, the applications that are joined to the execution that was started prior to the EM's termination will NOT be informed of the new EM (since it is not being recovered). Consequently, these applications will not be able to communicate with this new EM instance. If this is the intended behavior, it can be accomplished in the following manner:

1. By omitting the `recoveryDir` option, or
2. By specifying a different value for the `recoveryDir` option, or
3. By removing the SQLite database file.

Multicast Support

Multicast Support

TENA applications collaborate by exchanging SDO updates and messages using either "Reliable" or "Best Effort" communication protocols. Best effort communication utilizes the UDP multicast protocol in which the application performs a single communication write and the network devices replicate the message to the necessary destinations. Multicast communication can improve performance and scalability, but the protocol is unreliable and applications need to be designed to tolerate lost network packets.

Description

Within a TENA execution, applications exchange information in terms of SDO state updates and messages. An application can transmit this information using either *Reliable* or *Best Effort* communication, where Reliable communication uses the TCP/IP transport protocol and Best Effort communication uses the UDP Multicast transport protocol.

Reliable (TCP/IP) communication uses a point-to-point connection between the publisher and the subscriber, so if there are multiple subscribers interested in the publisher's information there will need to be a connection from the publisher to each subscriber. The publisher will then perform a network write operation for each subscriber every time there is an SDO state update or message to be delivered. The TCP/IP communication mechanism is a reliable protocol such that the publishing application will be notified if a network error (or similar problem) occurred in delivering the data to the subscribing system.

Best Effort (UDP Multicast) communication only requires the publishing application to perform a single network write operation, and then relies on the network devices to deliver the data to all subscribing systems. Multiple multicast addresses are supported. A publishing application will send its data on a particular multicast address and a subscribing application must "join" that multicast address to receive the data. Since the publishing application only needs to perform a single network write operation, independent of the number of subscribers, there is less overhead than with Reliable, TCP/IP, communication. Unfortunately, this anonymous publish-subscribe protocol is not reliable and the publisher has no guarantee that data was delivered to the subscribers.

Even though Best Effort communication is unreliable, there is a performance gain over Reliable communication and using Best Effort communication may be acceptable in circumstances where potential loss of data can be tolerated. For example, if the published data is updated frequently and subscribers are not required to process every update, then Best Effort communication may provide increased scalability for the event.

When an execution needs to use Best Effort communication, the Exercise Manager (EM) must be configured with the appropriate multicast address range to be used for the event. If multiple executions using Best Effort are going to run simultaneously over the same network, then the multicast address range should be designed to not overlap to prevent multicast cross-talk. The TENA Middleware will reject multicast traffic if it doesn't belong to the current execution, but there can be a performance cost if a computer is receiving unwanted multicast traffic.

The EM configuration parameter named `multicastProperties` uses the form `<baseMulticastAddress>:<portNumber>:<numberOfAddresses>`, such as `225.25.2.1:55200:254`. IANA (www.iana.org) recommends that multicast addresses should be in the range between 239.192.0.0 and 239.255.255.255, with port number in the private range between 49152 and 65535. When an application joins a particular execution, the EM will indicate if Best Effort communication is supported and the particular multicast address range that is to be used. The middleware uses a complicated, but deterministic algorithm that maps a particular object model type (and Advanced Filtering criteria) to a particular multicast address (see the [Multicast Address Assignment](#) page for details on the mapping algorithm).

Publishing applications dictate whether their SDO updates or messages are sent using Reliable or Best Effort transport. The communication properties can be sent on a per SDO or per message basis. Subscribing applications need to be prepared to receive SDO updates and messages using either communication mechanism (Reliable or Best Effort).

If an application attempts to perform updates or send messages using Best Effort transport when the Execution Manager has not been configured for multicast operation, a runtime exception will occur. The particular exception is `TENA::Middleware::ExecutionNotConfiguredForBestEffort` and applications can develop their code to catch this exception and update/send using Reliable transport if they need to write their application conditionally with attempting Best Effort transport first.

Usage Considerations

As mentioned, Best Effort (UDP multicast) communication is an unreliable protocol and network packets sent using this protocol can be discarded without notification to the sender. Application developers should only choose to use Best Effort communication when the performance benefit is required (based on engineering analysis and not speculation) and the overall event can tolerate the undetected loss of network packets sent using this protocol.

TENA Messages do not typically make good candidates for Best Effort communication because the message is sent once and there is no recovery if the network does not deliver the packet — unlike SDO updates in which a subsequent update would allow a subscriber to recover from a lost packet (or packets). SDO updates can be sent using Best Effort communication if the updates occur relatively frequently and subscribers are not required to process every single update performed by the publishing application.

In some cases, subscribing applications may design code to periodically check the timestamp of the last received update and if an update has not been received within the expected period of time the application can issue a warning to the application operator. Subscribing applications are permitted to "pull" the publication state from the SDO servant (using `pullPublicationState()`), but this mechanism is accomplished with a blocking distributed invocation with indeterminate response time which may introduce undesirable application behavior.

When using Best Effort communication, the event operators need to decide on the range of multicast addresses to be used for the event. It is prudent to verify through testing that the computers and network devices supporting the TENA execution are properly configured for UDP multicast when Best Effort transport is required. This testing should use the entire planned multicast address range. Please contact the local system administrators/network engineers for this testing or contact the TENA Helpdesk if additional assistance is required.

Additional multicast addresses will enable better data segmentation (i.e., minimize the use of the same multicast address for different data), but computer operating systems, network interface cards, and network devices may have limits to the number of multicast addresses that can be effectively supported. Although several hundred multicast addresses is commonly used, consult the documentation for these devices for limitations or guidance on the number of multicast addresses to be used for an event. Testing of these devices with the full range of multicast addresses active should also be performed prior to the event.

When using Best Effort (UDP multicast) communication for a TENA execution, the developers of the applications publishing information that should be sent best effort need to ensure that the correct `CommunicationProperties::Best_Effort` is defined when creating SDOs or sending messages. It is recommended that applications are developed to permit a configuration option to switch between Reliable and Best Effort communication. Subscribing applications do not need to specify the communication property.

The **Execution Manager** associated with the execution using Best Effort communication needs to be started with the appropriate `multicastProperties` configuration parameter value. Applications using Best Effort communication will have the configuration parameter `multicastInterface` default to be the same interface as defined by the `listenEndpoints` parameter value, so it is not necessary to define the `multicastInterface` parameter value unless (for some bizarre reason) that Reliable traffic is sent on a different interface from Best Effort traffic.

By default, an application's `multicastTTL` parameter value will be "1". This parameter refers to the "Time To Live" for a multicast packet and indicates whether network devices will propagate the multicast packet outside of the current local area network. A value of "1" indicates that the packet will not be propagated outside of the local area network. Network devices use the following values to define the forwarding behavior of multicast packets:

TTL Value Meanings

TTL Value	Description
0	restricted to the same host
1	restricted to the same subnet
32	restricted to the same site
64	restricted to the same region
128	restricted to the same continent
255	unrestricted

Related Topics and Links

- [Execution Manager Configuration Parameters](#)

OM Consistency Checking

Object Model Consistency Checking

The various applications participating in a TENA execution are typically developed by different organizations and at different times. The object models used by these applications may contain definitions or implementations that are incompatible due to version differences, and this may introduce interoperability problems. The TENA Object Model Compiler automatically generates a series of typecode checksum values for the definition of the various object model constructs so that the Execution Manager can detect object model differences when an application attempts to join an execution. Additionally, the Execution Manager can be configured to detect and reject applications that use different object model implementations (i.e., software code used to implement object model constructors and methods). An application that attempts to join an execution with incompatible object model definitions (or object model implementations, if configured) will receive a runtime exception at join time indicating the object model inconsistency.

Description

A TENA Object Model is used by an application to formally define the information that is exchanged with other TENA applications. This information is modeled as SDOs (Stateful Distributed Objects) and messages. TENA applications can *publish* and/or *subscribe* to the different types of SDOs and messages defined in the object model(s) that are used to build the application.

Due to the evolution of object models and applications, it is possible that different applications will attempt to join a TENA execution using inconsistent object model types. For example, one application may subscribe to a type containing a Location that was defined to have attributes double x and double y, while the publishing application may have used an object model where Location was defined to have attributes double x, double y, and double z. The TENA Middleware needs to detect these kinds of object model inconsistencies when an application attempts to join the execution.

The OM Compiler is used to parse an object model and automatically generate the object model definition software that is linked with applications using that particular object model. Included with the object model definition software is a "typecode definition key" for every type in the object model. The typecode definition key is a software generated checksum that is used to ensure that every application joining the execution uses a consistent set of types.

When an application attempts to join an execution, it provides the Execution Manager with the information (including the type name and the typecode checksum) for each object model type against which that application is linked. The Execution Manager maintains the set of all type information in use for the particular execution. If an application attempts to join the execution with type information which is inconsistent with types registered by currently joined applications, the Execution Manager will refuse to allow the application to join the execution, instead throwing an exception such as:

NOTE: ... Exception raised: Type consistency failure: Application listening at endpoint {[xx.xxx.xxx.xx:ppppp]} tried to register type 'tmp:consistencyCheck_SDO' with typeid 2873499957 from object model tmp-consistencyCheck-v2 but type name was previously registered with typeid 1770087170 in object model tmp-consistencyCheck-v1.

Object Model Implementation Consistency Checking

In addition to the **object model type** consistency checking that is unconditionally provided by the Execution Manager, it may be useful for TENA executions to enforce the use of common **object model implementations** in the execution. Each local class and message methods factory implementation must define a unique description, since multiple implementations may exist for a particular type. The implementation description is used by the Execution Manager to perform implementation consistency checking. Similar to how joining applications will provide object model type information to the Execution Manager when joining an execution, the application will also provide object model implementation description strings for each methods factory implementation. By default, the Execution Manager is configured to refuse to allow an application to join an execution, if its object model implementation descriptions do not match those of currently joined applications. This behavior can be overridden by using the "allowImplementationMismatch" configuration parameter for the Execution Manager.

As a relevant real-world example, a commonly used TENA object model, `TENA::Time`, provides support for the conversion of a time value into different representations (e.g., Coordinated Universal Time, GPS Time, etc.). When a new leap second was added by the International Earth Rotation and Reference System Service (IERS), a new version of the `TENA::Time` object model implementation code was required. During the transition to the new implementation version of `TENA::Time`, it was prudent for the TENA event coordinators to ensure that all applications were using the updated version of the `TENA::Time` implementation.

When the EM detects an inconsistency in the object model implementation(s) that are being registered by a joining application, the EM will emit an error similar to that shown below, and the application will be prevented from joining the execution (unless `allowImplementationMismatch` is set for the EM).

NOTE: ... Exception raised: OM Impl Compatibility Mismatch: for type TENA::Tests::OMconsistency::msg: Application listening at endpoint {[xx.xxx.xxx.xx:ppppp]} tried to join with implementation description 'TENA,Tests-OMconsistency-v1-TestImpl,trunk-1.0.0, ObjectModelImpl,source,all' but previous application has joined with 'TENA,Tests-OMconsistency-v1-StdImpl,trunk-1.0.0, ObjectModelImpl,source,all'. The OM Impl versions are incompatible. This is not allowed. The OM Impl name and the major and minor numbers of the version must match for the same OM name and OM type. For example, 'TENA,TSPI-v5-StdImpl,1.0.0, ObjectModelImpl,source,all' and 'TENA,TSPI-v5-StdImpl,1.0.1, ObjectModelImpl' are compatible by default. To specify alternate OM impl support for your execution, use the 'omlImplementationRulesFile' configuration option.

Usage Considerations

Object model type consistency checking is automatically enforced by the Execution Manager and there are no development or operational requirements.

The optional object model implementation consistency checking is enabled by default and requires all applications participating in the execution to use object model implementations that have the same implementation description string (ensuring the use of the same OM implementation type and version). Implementation consistency checking can be disabled by starting the Execution Manager with the "allowImplementationMismatch" configuration parameter.

When developing a new version of an existing object model implementation, care should be used to refine the description string returned by the `getDescriptor()` method for the corresponding `MethodsFactoryImpl` class. It is recommended that the description string is based on the object model implementation name, including the version identifier.

C++ API Reference and Code Examples

Object model type consistency checking is automatically done by the TENA Middleware and the OM Compiler's generated object model definition code. Some of the underlying typecode information used by the consistency checking mechanisms is available to application developers. Each object model SDO, Message, or Local Class type has a corresponding `Type` class that can be used for type identification purposes. See the [Type Registry page](#) for more information about these `Type` classes and the `TypeRegistry`.

Object model implementation consistency checking is dependent on the object model implementation developer's proper use of the implementation description string, as discussed above. An example of the object model implementation description string that is returned by the `getDescriptor()` method is shown below. Note that in this example, the description string matches the object model implementation name, including the version identifier. Users are encouraged to follow a similar convention to ensure that the description string is correctly specified for each version of an object model implementation.

```
std::string const
TENA::TSPI::PositionConversion::MethodsFactoryImpl::getDescriptor() const
{
    return "TENA-TSPI-v5-positionConversion-v1";
}
```

Related Topics and Links

- [TENA Object Models \(OMs\)](#)
- [TENA Object Model Compiler \(OM Compiler\)](#)

Type Registry

Type Registry

The TENA Middleware utilizes a type registry to detect incompatible object model definitions when an application attempts to join an execution. The Object Model Compiler generates a `TypeID` (i.e., checksum) and other type information for the definition of each object model construct. Application developers can access the `TypeID` and other type information through a `TypeRegistry` mechanism, as well as using variables and functions associated with the generated object model code.

Description

The Middleware type registry is a process-wide singleton which contains type information about the object model types against which the application is linked. Application developers can use this type registry for reporting or identifying particular object model types. The type information available is described in the following table.

Object Model Type Information

Information Item	Description
Type Name	Name (string representation) of the particular object model type (e.g., <code>Example::Vehicle</code>).
Type ID	Numerical identifier (64-bit integer) used to uniquely identify the object model type.
Type Path	Vector of <code>TypeIDs</code> corresponding to the type inheritance hierarchy. Note that the order of the <code>TypeIDs</code> is from most-derived to least-derived, so that <code>TypeIDpath().front()</code> will always be that type's <code>TypeID</code> .
Object Model Name	Name (string representation) of the object model from which the type came from (e.g., "Example-Vehicle-v1").

This type information is available for all object model types defined in the object model(s) linked into the application. Application developers can develop software that invokes the static functions or utilizes the static variables defined in the generated code for the object model definitions.

Additionally, the middleware provides access to a `TypeRegistry` class that can be used to provide this type information at runtime by performing a lookup based on a particular `TypeID`. Application developers can obtain the `TypeID` associated with an SDO, Message, or Local Class from the metadata associated with the object model element.

Usage Considerations

The actual numerical `TypeID` associated with a particular object model type is a checksum (associated with the [OM Consistency Checking](#) mechanism) that is generated based on the fully qualified name of the object model element and the complete definition of that element. So, as long as the name of the object model element does not change and the definition of that element does not change, the `TypeID` will be the same from one application to another, or one execution to another.

Application developers may want to utilize the `TypeID` for a compact representation to uniquely identify the type of a particular object model element, such as in a database. In this situation, a separate table should be saved that maps the `TypeID` to the string representation of the type.

When using the `TypeIDpath` on SDO Proxies, Local Classes, and Messages received by a subscriber it will describe the complete inheritance hierarchy regardless of what the subscriber is linked against. For example, suppose a Vehicle subscriber (not linked against the inherited Humvee object model definition) discovers a Humvee SDO. In this situation, `TypeIDpath` in the metadata will return a vector with two `TypeIDs`: [<HumveeTypeID>, <VehicleTypeID>]. If the subscriber calls `TypeRegistry::lookupTypeInfo(<HumveeTypeID>)`, the `TypeRegistry` will throw an exception because the subscriber is not linked against the Humvee object model, and therefore <HumveeTypeID> is unknown.

C++ API Reference and Code Examples

The automatically generated code for each object model definition includes a header file for each object model element (e.g., SDO, message, local class). These header files are installed under the `include/OMs/<objectModelName>/<packageName>` directory and named for the name of the object model element (e.g., `Vehicle.h`). In these object model element header files are several static methods and variables associated with the object model type information. Application developers can utilize these methods and variables within their application as needed. The definition of these methods and variables are shown below.

```

static
::TENA::Middleware::TypeIDpath const &
toTypeIDpath();

static
::std::string const &
toTypeName();

static
::std::string const &
getObjectName();

static ::TENA::Middleware::TypeID const typeID;

```

As mentioned in the description above, applications can utilize a `TypeRegistry` object to obtain this object model type information during runtime. The `TypeRegistry` object is a "singleton" that is obtained through the static method `TypeRegistry::getInstance()`, which returns a reference to the `TypeRegistry` created by the middleware for this application.

Object model type information maintained by the `TypeRegistry` is stored in a `TypeInfo` data structure that is defined as shown below, along with the methods that lookup the appropriate `TypeInfo`.

```

struct TypeInfo
{
    std::string typeName;
    std::string omName;
    ExecutionManagement::IDL::MetatypeEnum metatype;
};

enum MetatypeEnum
{
    METATYPE_ENUM,
    METATYPE_LOCALCLASS,
    METATYPE_CONSTANT,
    METATYPE_MESSAGE,
    METATYPE_SDO
};

virtual
TypeInfo const&
lookupTypeInfo( TypeID const & typeID )
    const = 0;

virtual
TypeInfo const&
lookupTypeInfo( TypeIDpath const & typeVector )
    const = 0;

```

As shown, the `TypeInfo::metatype` is an enumeration to indicate whether the object model type is an enumeration, local class, constant, message, or SDO.

Application code can obtain the `TypeID` or `TypeIDpath` from the metadata associated with the object model element. Refer to the [Metadata](#) documentation for information on how to access and use the metadata provided by the middleware.

Related Topics and Links

- [Object Model](#)
- [Object Model Consistency Checking](#)
- [Middleware Metadata](#)

Running EM as a Service

Running the Execution Manager as a Background Service

Introduction

The Execution Manager may be started in any of the following ways:

- As a stand-alone process in a command shell, using either a `start.sh` script or the `executionManager` command.
- Using the TENA Console.
- As a Background Service.



Starting the Execution Manager as a Background Service is available in the EM Product version 6.0.7 and beyond.

Installing the Execution Manager Product

To run the Execution Manager as service, the Execution Manager **must be installed using the EM Product distribution**.



This service feature is not provided by the Execution Manager that is installed as part of the TENA MiddlewareSDK distribution.

When the Execution Manager Product is installed, its directory structure is as follows:

- <install-dir>/executionManager-v6.0.7
 - bin (executable and runtime libraries for Windows)
 - lib (runtime libraries)
 - config (editable configuration files)
 - log (default log file location)
 - save (default recovery file location)
 - license (third party license files)
 - start script (Windows batch script or bash script)

Managing the Execution Manager Service

Before starting the Execution Manager as a service, ensure that configuration file has been properly edited to meet the requirements of the execution. See [EM Configuration Parameters](#) for general Execution Manager configuration information.

A. Starting the Execution Manager in a Command Shell

The default installed configuration file, `executionManager.config`, is editable, and the file should be modified to meet the requirements of the the execution.

The following example demonstrates how to run the Execution Manager in a command shell using the `start.sh` script and the configuration file. Performing this action prior to starting the Execution Manager as a service helps to ensure that the configuration file is correct.

Once the Execution Manager process is started, type 'exit' to shut it down normally.

```

$ <install-dir>/executionManager-v6.0.7/start.sh (use start.bat on Windows)
executionManager Version: 6.0.7
  Middleware Version: 6.0.7
    TENA Platform: w81-vs2013-64-d

To join this execution, use one of the following endpoints:

{[10.0.0.146:55100]}

For example: -emEndpoints 10.0.0.146:55100

Enter 'help' to display this list of commands.
Enter 'show version' to display the TENA Middleware version and the version of this EM.
Enter 'show config' to display the supplied command line and full configuration options.
Enter 'show endpoints' to display the endpoints of the execution manager.
Enter 'show allowables' to display the supplied configuration of allowable participants.
Enter 'show om impl rules' to display the supplied configuration of OM Implementation rules.
Enter 'set passphrase [<phrase>]' to set, or without a phrase to clear.
Enter 'force rollover' to forcibly rollover to a new primary execution manager.
Enter 'quit' or 'exit' to shutdown the execution manager.

Accepting requests ...

> exit
NOTE: Shutting down.....
NOTE: Shutdown complete.

```

B. Installing the Execution Manager Service

To install the Execution Manager as service, ensure that the user that has adequate permissions, (e.g. *root* or *Administrator* permissions).

The following example demonstrates how to install the Execution Manager as a service using the *start.sh* script. The *start.sh* script should also be used to remove the Execution Manager service.

The supplied *userName*, if given, will be assigned as the owner of the Execution Manager service. The service can be optionally started at boot time by specifying the *startAtBoot* parameter.

```

(Showing CentOS 7)
$ <install-dir>/executionManager-v6.0.7/start.sh -installService [-userName rknights] [-startAtBoot]
NOTE: TENA ExecutionManager service installed

```

C. Starting the Execution Manager Service

The following example demonstrates how to start/stop the Execution Manager as a service using the *start.sh* script.

Note that when the Execution Manager is started as a service, the *quiet* and *nonInteractive* configuration parameters are set.

```

(Showing CentOS 7)
$ <install-dir>/executionManager-v6.0.7/start.sh -startService
Starting TENAExecutionManager (via systemctl): [ OK ]
NOTE: TENA ExecutionManager service started

$ <install-dir>/executionManager-v6.0.7/start.sh -stopService
Stopping TENAExecutionManager (via systemctl): [ OK ]
Terminated

Note. If requesting start or stop from a non-admin user, the admin password will be requested.
Sudo access would be a better way to start and stop the Execution Manager service, see System Administrator for details.

```

D. Removing the Execution Manager Service

The following example demonstrates how to remove the Execution Manager service using the *start.sh* script.

```
(Showing CentOS 7)
$ <install-dir>/executionManager-v6.0.7/start.sh -removeService
NOTE: TENA ExecutionManager service removed
```

Glossary

Glossary

The following glossary attempts to define common terms and acronyms associated with the TENA (Test and Training Enabling Architecture) project.

Item	Definition
Abstract Class	A class with no instances that is created only for the purpose of defining an interface or set of interfaces that then can be implemented in other classes that inherit from the abstract class.
ACE	See Adaptive Communication Environment.
ACE ORB	See TAO.
Adaptability	A system's ability to function effectively in a complex and rapidly changing environment. Adaptability includes scalability, portability, and configurability as well as the ability to respond intelligently to real-time events and surprising threats.
Adaptive Communication Environment (ACE)	A cost-free, open-source, object-oriented software environment that implements design patterns for many aspects of distributed computing.
Aggregation	The ability of instantiated objects to have dynamic relationships (other than 'is-a' and 'has-a') with object instances.
AMT	See Architecture Management Team.
API	See Application Programming Interface.
Application	A unit of software deployment that can stand alone and that performs a single broadly defined function (e.g., word-processors, spreadsheets, drawing packages, etc.). Applications can be composed of smaller software components, and can also be combined into larger software systems, but as the name implies an application generally has a single overall function. Obviously, the distinctions between components, applications, and systems are somewhat vague. Generally, components cannot stand on their own and tend to run only in a single process space. Applications can run on their own can either run on a single processor or be distributed. Systems include both software and hardware.
Application Programmer's Interface (API)	The programming interface visible to the developers. It is defined partly by generated code based on the TENA object model, while other parts of the interface are more static in nature.
Architecture	As defined by the Institute of Electrical and Electronic Engineers (IEEE), an architecture is "the fundamental organization of a system embodied in its components, their relationships to each other and to the environment, and the principles guiding its design and evolution." DoD has specified an architectural framework for describing information system architectures for C4ISR systems (called the C4ISR Architecture Framework) that describes three fundamental architectures (or architectural views): the Operational Architecture, the Technical Architecture, and the System Architecture.
Architecture Management Team (AMT)	The project team created to oversee and approve the design and construction of the TENA.
Auto Pointer	An auto pointer is a 'smart pointer' responsible for deleting the contained pointer when the container goes out of scope (or is destroyed).
C4ISR	Command, Control, Communications, Computers, Intelligence, Surveillance, and Reconnaissance.
Callback Services	Those services in the TENA Middleware that provide a flexible control mechanism for making invocations across the execution. A user application can provide a thread (or threads) of control to the middleware for executing callback objects (waiting in the queue) when it is appropriate for the application.
Candidate TENA Objects	Those object definitions (and potentially implementations) that have been used in multiple test or training events, are proven to be effective, and are widely applicable. They also are approved by the AMT as ready for submission to the RCC for standardization, and also may evolve more quickly, given the need, until actual submission to the RCC for standardization.

Class	A template for creating a given object, specifying what state and methods the object has but not giving them values. An object is an instantiation of a class.
Client	An application that wishes to read or use a TENA object created by another TENA application is called a 'client' of that object. Clients do not interact directly with servant objects, but instead do so through intermediary objects called 'proxies.'
Clock Services	Manage time issues for the facility that perform synchronization and time setting services as well as maintain a global clock for exercises.
Common Object Request Broker Architecture (CORBA)	A standard generated by the Object Management Group (OMG). Software that conforms to the CORBA standard supports distributed computing in heterogeneous distributed environments. A CORBA-conformant Object Request Broker (ORB) supports interoperability among distributed software components in a location-transparent manner. The CORBA specification is available online at http://www.omg.org/corba/
Component (Software)	A unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties. Components are independently created and deployed packages of software that can be combined to create applications. Components are not objects, but they can contain many objects. In general, applications are created by combining a few tens of components together using 'glue' code (either other software, scripts, or visual scripts).
Composition	The feature whereby TENA objects ("root" objects) may contain other TENA objects (contained objects). Only root objects have independent existence. Contained objects depend on their container object for their identity.
Data Streams	A communication mode allowing data streams to be sent between applications based on the applications' needs. Support for data streams is intended to provide TENA applications with native capability to send and receive audio, video, and telemetry effectively without incurring undue encapsulation and transmission overhead.
Design Patterns	Design patterns encompass "tried and true" solutions tailored for specific object-oriented software design problems and objectives. Design patterns not only make it easier to reuse successful designs and architectures, but also make it easier to communicate such designs and architectures to other developers.
Developers (Application)	The developers and model builders at DTCs and throughout the range community who build new applications and integrate legacy systems that reside at ranges. Primarily, application developers are concerned with the algorithmic correctness and functionality of their applications and not with the details of the distributed federation. However, they are concerned with interoperability between their applications and the rest of the execution. Application developers use the TENA API.
Developer (Infrastructure)	Developers who design and build TENA implementations. "Build the frameworks."
Developer (Execution)	Developers who build TENA object model definitions and define overall execution/event characteristics.
Distributed Interest-based Message Exchange (DIME)	DIME is an extensible framework for Publish/Subscribe message passing systems. DIME enables the TENA middleware to exchange information efficiently and to minimize unwanted data on the network as well as on individual machines. DIME also implements the change notification capability defined in the TENA functional capabilities document.
Distribution Services	Manage the distribution of data between facility assets and infrastructure components.
DoD	Department of Defense.
Encapsulation	The property of a software construct that separates interface from implementation. In programming language implementations, encapsulation means a defined object interface is the only external access to the object's functions and data. Component systems-- such as those based on CORBA and COM specifications that have interfaces specified in interface definition languages-- provide encapsulation that is even more powerful. The encapsulating interface is language independent.
Endpoint	An endpoint is made up of the host name or IP address of a computer's network interface and the port number. The network naming service uses endpoints to listen for requests from the TENA Middleware.
Error Handling	The overall process a software program uses to deal with errors generated during its operation.
Event Channel Messaging	A communication mode allowing single messages to be sent between applications based on the needs of those application.

Exception Handling	The more specific means (<i>i.e.</i> , try-throw-catch code) a program employs to catch and deal with errors.
Execution Management	Those TENA Middleware services that support joining with and resigning from executions.
Execution Manager	The Execution Manager is a process initiated manually by the developer before running an execution. This process assists in defining execution-unique names, assists in the efficient dissemination of publication state between applications, and provides other execution management services.
Foundation Initiative 2010 (FI2010)	A joint interoperability initiative from the Director of Operational Test and Evaluation whose goal is to provide the core products necessary to (1) enable interoperability among ranges, facilities, and simulations in a quick, cost-efficient manner and (2) foster reuse for range asset utilization and for future developments.
Framework	An object-oriented application framework is a reusable 'semi-complete' application that can be specialized to produce software applications. Object-oriented application frameworks are reusable designs of all or part of a system represented by a set of abstract classes and the way instances of these classes interact.
High Level Architecture	A Department of Defense (DoD) Architecture intended for use by the DoD Modeling and Simulation community. A middleware system that implements this architecture is called an "RTI" for "Run-time Infrastructure."
HLA	See High Level Architecture.
Implementation	The actual code/software that implements the functionality behind the interface.
Inheritance	This is the ability of a class to specialize another class, while preserving 'substitutability.'
Interface Definition Language (IDL)	Used to specify object interfaces, that is, services an object agrees to provide to client objects. Microsoft's Interface Definition Language (MIDL) defines COM object interfaces. Object Management Group (OMG) IDL defines CORBA object interfaces in a language-neutral way, for use with CORBA-compliant Object Request Brokers. The OMG's CORBA Specification includes a syntax, semantics, and implementation language bindings for OMG IDL. When this document mentions IDL with no additional qualifier, the reference is to OMG IDL.
Interoperability	The ability of independently developed and deployed systems (or system components) to seamlessly work together and exchange information to solve a user's problem. Interoperability is required between range resources; across phases of the acquisition, test, and training processes; between range resources and C4ISR resources; and between range resources and simulations.
Latency	See Subscription Latency Time.
Local Methods Invocation	Invoking local methods means invoking those methods that can only be called by the object's owner application.
Local TENA Middleware	The specific part of the TENA Middleware that is linked in to a user's application or TENA resource during execution.
Logical Range Object Model (LROM)	The specific subset or variant of the TENA Object Model that is used in a logical range execution, and defines those objects used. The LROM may contain some RCC Standard TENA Objects, some candidate TENA objects, and their objects unique to a given range event.
LROM	See Logical Range Object Model.
M&S	Modeling and Simulation.
Message Services	Provide a form of communications based on a discrete, packetized data oriented service for any purposes applicable within facility operations.
Name Binding	A name-to-object association that is always defined relative to a naming context.
Namespaces	Namespaces define a scope where global identifiers and global variables are placed.

Network Naming Service (NNS)	The NNS is a process initiated manually by the developer before running an execution. The NNS ensures that each LRE has a unique name.
Network Reference	The network reference is similar to an endpoint in that it defines the IP address and port number. The network reference differs from an endpoint in that it represents a distributed object reference used to contact the Network Naming Service.
NNS	See Network Naming Service.
Notification	Implements the ability for the application to attach observers to SDOs to be notified when the proxy's publication state is modified.
Object	State, behavior, and identity; the instantiation of a class. An object is a basic unit of functionality in an object-oriented software system, combining state, behavior and identity into one software construct.
Object Framework	This framework provides the base classes necessary to implement the TENA object services. This framework provides relationship services, introspection, method operations and other object services defined in the TENA functional capabilities document.
Object Management Group (OMG)	An international trade organization that promotes portability and interoperability among software components by specifying an architecture for object-oriented distributed systems, the Object Management Architecture (OMA) reference model, and populating this framework with detailed specifications. OMG membership now exceeds 800 according to a recent press release.
Object Model	See TENA Object Model.
Object Request Broker (ORB)	A software construct that brokers method invocations between components in a distributed environment, providing the facility of location transparency for objects in a system. The ORB's capabilities are specified in the OMG's Common Object Request Broker Architecture (CORBA) specification.
Object Services	Responsible for creation, deletion, and introspection of objects, along with supporting services for manipulation of attributes, methods, and relationships.
Observer	Code that creates a dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
Operational Architecture	See Architecture.
Polymorphism	Class definitions may inherit from other classes but must conform to the base class's interface. Polymorphism is when these classes may then override the specific behavior invoked using a given interface.
Process	In software, a thread of control in the execution of a computer program that manages its own unique set of resources. A computer process is distinguished from a thread in that a process has its own resources, while a thread shares the resources of its parent process with other threads in that process.
Proxies	Representative objects that reside in the client's process space and stand in for the servant object. Client applications do not interact directly with one another, but rather use proxies to obtain an object's publication state.
Publication Services	Those services within the TENA Middleware that give applications the ability to create stateful distributed objects (SDOs), messages, or data streams that will be accessible to other applications within the execution.
Publication State	Those attributes that an object is capable of publishing to other applications, such as 'Get Value' methods. See also, publish/subscribe.
Publish	Each application publishes certain types of information (the publication state) to which any other application can subscribe. This is similar in effect to HLA, DIS, and other PDU-based communication systems. (See also Subscribe.)
QoS	Quality of Service
Resource	See TENA Resources.
RCC	Range Commanders Council.
Remote Method Invocation	The process of providing distributed access to remote, user-defined services (e.g., CORBA, Java RMI). Those methods that can be called by applications acting as remote clients.
Reuse	The ability to use a system (or system components) at sites or in test or training events other than those in which the software was originally designed to operate. Reuse implies componentization and modularity; standard, documented interfaces; and encapsulation of legacy capabilities. Reuse enables significant savings in long-term development and maintenance costs and is an effective and efficient path to sharing and interoperability.

Servers /Servants	An application that wishes to create and modify a TENA object is called a 'server' of that object. The served object is called a 'servant.'
Session Object	Session objects are the primary interface mechanism between an application and the TENA Middleware.
SDO	See Stateful Distributed Object.
Singleton	The term given to an object instance that, by definition, is the only one allowed to exist within an application's process.
Stateful Distributed Object (SDO)	An instance of a TENA class that generally combines an interface and publication state. The interface is accessible across the distributed system. The publication state may be updated and reflected to subscribers. SDOs are the unification of two different types of distributed computing paradigms: a message-passing publish and subscribe mechanism combined with a distributed object/remote method invocation mechanism.
Subscription and Discovery	Enables the application to subscribe to certain types of data and then discover remote SDOs.
Subscription Latency Time	The amount of "lag" time that passes after an application subscribes, but before a callback is returned with appropriate information.
Subscription Services	When developers wish to join an application to an execution, subscription services allow for applications to indicate the types of TENA objects in which they are interested to discover.
TAO (ACE ORB)	A free, open-source software built on top of the Adaptive Communication Environment (ACE). TAO is a high-performance implementation in wide-spread use in many existing systems throughout industry, government, and university research.
TDL	TENA Definition Language.
Technica l Architecture	See Architecture.
TENA	See Test and Training Enabling Architecture.
TENA Compliant	An application is compliant with the TENA architecture if it can communicate meaningfully about some subset of the objects in the TENA Object Model with other applications using the TENA middleware, and therefore participate fully in a TENA execution.
TENA Definition Language (TDL)	TDL is based on the Object Management Group's CORBA Interface Definition Language (IDL). IDL consists of a language-neutral way of defining interfaces. TDL adds several features to IDL for TENA in a language-neutral format.
TENA Execution (LRE)	A collection of collaborating applications that share a common object model for the purposes of exchanging data and making remote requests.
TENA Middlew are	The software that links range resources together into an execution. The "TENA Middleware" is the component that is in between (<i>i.e.</i> , in the middle of) range resources.
TENA Objects	An instantiation of a TENA class. There are three categories of TENA objects, representing the different stages in the standardization process. Logical Range Object Model (LROM) objects are created and used by a given execution for their specific needs, but which have not undergone any rigorous standardization or testing. Candidate TENA objects are those objects submitted to the AMT for "de-conflicting" with other, similar objects from other ranges. Standard TENA objects have been tested at multiple ranges, approved by the Range Commander's Council, and inserted into the TENA Object Model for use by the entire range community.
TENA Object Model	Those object definitions (and potentially implementations) that have been approved and standardized by the RCC and are held to be common among all the ranges. The TENA Object Model consists of standardized classes that define what types of objects can be instantiated as well as auxiliary information such as basic types, structures, interfaces, constants, exceptions, arrays, and strings that can be used within class definitions.
TENA Resources	Resources such as (1) range applications compiled to use TENA Middleware services for interaction; (2) gateway applications that bridge TENA systems with legacy or other protocols or architectures; and (3) TENA tools and databases that are configured for a particular event.

Test and Training Enabling Architecture (TENA)	The purpose of TENA is to enhance and enable interoperability and reuse of software resources among the Range Community. The most important components of TENA are the specification of a common infrastructure (the TENA Middleware) and the specification of a common object model (the TENA Object Model). Using the C4ISR architecture framework, TENA is a Technical Architecture in that it specifies rules and standards for achieving interoperability and reuse. However, the specification of the TENA Object Model makes TENA more than a Technical Architecture-- it is a Technical Architecture combined with portions of a Domain-Specific Software Architecture.
Type	An abstract interface. A given object can implement one or more types.
Unsubscribe	The opposite of subscription; each un-subscription negates a previous subscription. An application can't unsubscribe to things to which it hasn't subscribed in the first place.

Local Class

Local Class

A TENA Local Class is similar to a C++ class: it provides an abstraction that composes data and provides methods to operate on that data. Like a C++ class, it supports polymorphism. It is composed of fundamental type attributes, TENA Messages, and other TENA Local Classes. A TENA Local Class instance exists entirely in, and is visible only to, a single C++ process. (This is what makes it "local".) TENA Local Classes can be used as members of, and as parameters or return values of methods on, TENA SDOs, TENA Messages, and other TENA Local Classes.

Typical reasons for using Local Classes include:

- They provide a way to ensure that proper attribute values are set by publishers.
- They provide a way for subscribers to define common attribute transformation operations.

Description

A TENA **Local Class** is similar to a TENA Class (i.e., an SDO, Stateful Distributed Object) in that each is composed of both methods and attributes. However, the methods and attributes of a Local Class are always local with respect to the application holding the Local Class instance. Invoking a method on an SDO, in general, involves an RMI (remote method invocation) to a remote machine where the method's computation is actually performed. Invoking a method on a Local Class always results in the method's computation being performed locally in the invoking application's process space.

The attributes of a Local Class are always local to the application holding the instance of the Local Class. This is in sharp contrast to the attributes (i.e., state) of an SDO. Even though SDO proxy attribute values are accessed as if they were local to the application, in fact, their values may have been set in a remote application and cached in the local application after having been disseminated. The parameters or return types of methods on a Local Class follow the same rules as those for an SDO, i.e., everything in the TENA Meta-Model is allowed except an SDO (in which case a `Class Pointer` can be used). Since Local Classes are always passed by value, they are legal parameters and return types.

Similarly, attributes of a Local Class may be any of the fundamental data types or complex constructs from the TENA Meta-Model, with the exception of an SDO (as mentioned, a `Class Pointer` can be used). Since a Local Class is inherently local, composing it from constructs that are generally remote (i.e., SDOs) is impossible.

The Local Class concept greatly improves the power and expressiveness of the TENA Meta-Model. While the SDO concept is very adept at modeling many real-world problems, often situations arise where it is impractical to use RMIs or remotely modifiable attributes.

For example, consider a simple object model that has a method to calculate the local time offset from a time value attribute contained within an SDO. Performing this time offset calculation as an RMI, while possible, would be grossly inefficient in most real distributed systems.

The Local Class concept allows for object model definitions that support client-side methods on Local Classes that are contained in SDOs or Messages. For example, an object model can define a Local Class that represents an entity's position defined in a particular coordinate system. The Local Class can provide methods to be executed by the receiving (i.e., client-side) application that can convert the position to another coordinate system. These coordinate conversion operations would be performed locally (with respect to the client application). The `TENA::TSPI` object model is designed with Local Classes in this manner.

This capability helps to improve interoperability by ensuring consistent implementation of low-level support functions, as well as by promoting reusability of these Local Class implementations.

It is recommended that even simple data structures that are needed within an execution should be defined as Local Classes when the grouping of attributes should be treated as a single construct and may require future local behavior (i.e., constructors and/or methods). Applications using Local Classes need to use attribute accessor/mutator mechanisms to get/set the Local Class attributes.

Consider the following TDL (TENA Definition Language) example:

```

package Example
{
    local class Location
    {
        float64 xInMeters;
        float64 yInMeters;
    };

    local class AreaOfEffect
    {
        AreaOfEffect( Location centerOfArea, float64 radius );

        boolean isLocationAffected( in Location here );

        readonly Location centerOfArea;
        readonly float64 radius;
    };
}

```

A simple data structure `Location` was defined in the example object model as a Local Class even though there is no local behavior needed. Using a Local Class helps to emphasize the coupling of these attributes from a modeling perspective, and may simplify evolution if local behavior or derived Local Classes were ever needed in the future.

The `AreaOfEffect` Local Class has behavior, specifically the method named `isLocationAffected` taking `Location` and `radius` arguments, and returning a boolean value. Note well that invoking the `isLocationAffected` method never results in a remote method invocation, and the `AreaOfEffect` instance actually lives in the local address space. In other words, the `isLocationAffected` method is always implemented locally within the process that contains the particular `AreaOfEffect` instance in question. This is in sharp contrast to an SDO, in which case a method invocation may result in a remote method invocation across the network (if the SDO instance in question is running on another machine connected to the network) or via shared memory.

Local Classes can be contained in an SDO or Message, and therefore can be disseminated as part of the SDO's publication state or as part of a sent Message, e.g.,

```

message Explosion
{
    ...
    AreaOfEffect affectedArea;
}

```

Applications using the above object model can subscribe to `Explosion` messages. The received message will contain an `AreaOfEffect` instance as a message attribute. Invocations of the `isLocationAffected` method on the `AreaOfEffect` Local Class will always execute in the same process as the received message.

Local Class Creation

Application developers use a static `create` method whenever a Local Class needs to be created. Although usually only publishing applications need to create Local Classes, subscribing applications may also need to create Local Classes. Subscriber applications typically obtain a Local Class from the SDO state or Message without having to use the `create` mechanism.

One use case for subscribers to need to create Local Classes is when there is a need to copy a received Local Class and modify its state (perhaps to resend or use a non-const local method). Another use case for subscribing applications creating a Local Class is when multiple Local Class representations exist for a common item and users are required to create a specific Local Class from a generic Local Class holder — this technique is used with the `TENA::TSPI` object model.

A simple illustration of creating a Local Class is shown in the code fragment below. The `Location` Local Class is created by using the static `create` method. In the case of a Local Class without any constructor defined in the object model definition, such as `Location`, the `create` method requires arguments for each non-optional attribute (see [optional attribute documentation](#)). The code example below uses the values "3.0" and "4.0" for example values. The `create` method for the `AreaOfEffect` Local Class, which declares a constructor in the object model, uses arguments based on that constructor declaration (i.e., `Location` and `float64`).

```

Example::Location::LocalClassPtr location( Example::Location::create( 3.0, 4.0 ) );
Example::AreaOfEffect::LocalClassPtr area( Example::AreaOfEffect::create( location, 5.0 ) );

```

 — In summary, when a Local Class does not define a constructor in the object model, the `OMC` auto-generated associated `create` method will contain arguments for all of the non-optional attributes, and when one or more constructors are defined, the associated `create` method(s) will have arguments that match the object model constructor declaration(s).

Note that Local Classes are always accessed through a pointer mechanism named `LocalClassPtr` in the appropriate namespace of the type. This pointer mechanism is designed to prevent the use of a "null" pointer, so users can always be guaranteed that the `LocalClassPtr` refers to an actual Local Class and there is no need to test if the pointer is non-null or valid.

Additional information related to creating Local Classes can be found in the [Local Class constructor page](#).

Local Class Cloning

An application receiving a Local Class through a subscription or remote method argument will be provided an immutable version of the Local Class. An immutable Local Class prevents the user from being able to modify the state of the Local Class. In some cases it is necessary to obtain a copy of the received Local Class that allows the state to be modified. In this situation, the application can invoke the "clone" method on the original Local Class to obtain a deep copy of the Local Class in which the `set` and other local methods that can modify the state are available.

A simple code example is shown in the fragment below.

```
Example::Location::ImmutableLocalClassPtr location( pVehicle->get_location() );  
  
Example::Location::LocalClassPtr locationCopy( location->clone() );
```

Local Class Implementation

When using Local Classes without any constructor or method behavior, the auto-generated example application source code does not require any modification by the application developer. The static `create` method for the Local Class will require values for all non-optional attributes.

When the Local Class defines constructor or method behavior, the application developer is responsible for providing the implementation code needed by the Local Classes, in much the same way that the programmer must provide the implementation code needed for any methods contained in an SDO. For an SDO, the programmer provides an implementation for the `RemoteMethodsImpl` class when the particular SDO has methods. Similarly, for a Local Class, the programmer provides an implementation of `LocalMethodsFactoryImpl` and `LocalMethodsImpl` classes when the particular Local Class has any methods.

The OMC auto-generated code for the `LocalMethodsFactoryImpl` and `LocalMethodsImpl` classes is included with the example applications. The specific installation location for these classes is illustrated below with the `Example-Tank` object model and `AreaOfEffect` Local Class type.

```
TENA_HOME/TENA_VERSION/src/Example-Tank-v1/Example-Tank-v1-SharedLocalImpl-v1/myImpl/Example/AreaOfEffect  
/SharedLocalImpl
```

Note that all of the Local Class implementation code associated with a particular object model is installed in a separate directory for each object model. The `SharedLocalImpl` library that results from building the implementation code is shared among all of the OMC auto-generated example applications for the object model. This directory structure is used because a Local Class implementation is typically developed once and shared among multiple applications.

See the [sharing object model implementation documentation page](#) for additional information on sharing Local Class implementation code.

The default behavior of the OMC auto-generated `LocalMethodsFactoryImpl` class is sufficient for most use cases. However, if Local Class constructor(s) or method(s) need application data, the application developer may need to customize the `LocalMethodsFactoryImpl` class. (See the below section on [Accessing Application Data from a Local Class](#) for more information.)

The OMC auto-generated `LocalMethodsImpl` class needs to be modified to provide appropriate behavior for the constructor(s) and method(s). However, the OMC auto-generated code uses the [Arbitrary Value mechanism](#), which provides default behavior suitable for application testing and evaluation. An example of the auto-generated code for the `AreaOfEffect::isLocationAffected` method is shown below.

```
bool  
Example::AreaOfEffect::SharedLocalImpl::  
LocalMethodsImpl::  
isLocationAffected( Example::Location::ImmutableLocalClassPtr const & here )  
{  
    // This is just an auto-generated "stub" example implementation. It is  
    // provided merely as a convenient starting point. This "stub" implementation  
    // should be replaced with the real implementation.  
  
    return TENA::Middleware::ArbitraryValue< bool >();  
}
```

Normally, users will replace the `ArbitraryValue` statements (as shown above) with specific Local Class implementation code.

Although it is common for all applications in an Execution to use the same Local Class implementation code, it is possible that multiple implementations could be used by different applications. Applications select the appropriate Local Class implementation code by compiling against the appropriate Local Class header files and by linking against the appropriate Local Class implementation code library. Using the correct header files and libraries is controlled by the build files (i.e., Makefiles for UNIX and Visual Studio project files for Windows).

The middleware uses a technique that involves static registration of the Local Class implementation code which allows the middleware to construct a Local Class (with appropriate implementation) when an SDO or Message containing the Local Class is received from another application. Due to the compiler /linker behavior for some computer platforms (specifically Windows and MAC OS X), it is necessary for an application to reference a "dummy" symbol in the Local Class implementation code library to force proper linkage. This registration is accomplished with the application including the OMC auto-generated `Impl.h` file for the appropriate Local Class, such as the example header file referenced below.

```
TENA_HOME/TENA_VERSION/src/Example-Tank-v1/Example-Tank-v1-SharedLocalImpl-v1/include/Example/AreaOfEffect/Impl.h
```

Additional details related to this registration process can be found in the [Local Class implementation registration documentation page](#).

Local Class Polymorphism

The TENA meta-model supports single inheritance of Local Classes, such as having a `Derived` Local Class inherit from a `Base` Local Class. In this situation, the `Derived` class can add attributes and methods to the existing `Base` definition (as long as the attribute method names are unique).

Local Classes are always accessed through a pointer mechanism named (within the appropriate namespace) `LocalClassPtr`. When an application is provided a derived Local Class that is accessed through a `LocalClassPtr` corresponding to an inherited type, invoking a method on the Local Class will behave in a polymorphic manner (i.e., the client will invoke the derived Local Class method).

Note that the polymorphic behavior requires the application to be compiled against the derived Local Class definition. In the case of [object model subsetting](#), a subscribing application may only be compiled against the base class type, such as the `Base` type from the example above. If the publishing application provides a `Derived` type, the subscribing application will "slice" the provided Local Class so that only the `Base` attributes and methods will be available.

Additional details related to Local Class polymorphism and slicing can be found in the [Local Class inheritance and polymorphism documentation page](#).

Accessing Application Data from a Local Class

In some circumstances, the user defined Local Class implementation code needs access to particular application data. For example, the Local Class implementation code may need to access a database that is connected to the application. In these circumstances, the implementation developer needs to customize the Local Class factory to ensure that the needed application data is provided to each Local Class instance created by the middleware.

Additional details related to accessing application data from a Local Class implementation can be found in the [Local Class application data access documentation page](#).

Accessing Application Data from a Local Class

Accessing Application Data from a Local Class

Sometimes, a Local Class method or constructor implementation requires access to some data other than the local class state and the method arguments. To meet this requirement, the local class methods factory (`LocalMethodsFactory`) implementation class's constructor can take any user-defined parameters. The factory can then pass any data along to the `LocalMethods` objects that it constructs and returns from its `create()` method.

⚠ When a Local Class implementation is required, it must be implemented in C++. The underlying requirement of the TENA Middleware is that a single shared implementation be used by all applications in an execution. Only a C++ implementation library is easily usable from C++, Java and .Net applications.

Description

Local Class objects may be constructed by the TENA Middleware, as well as by the user application. The Middleware may construct a Local Class instance in any of several situations, such as when a Local Class is received over the network or when the user application invokes the `create` function in the Local Class's namespace. Since the user application does not always construct the Local Class instance, if any of the Local Class methods need access to any of the user application's data, there needs to be some way for that data to get passed to the Local Class methods implementation. This is accomplished by having the user application supply a Local Class methods factory to the Middleware. The factory is constructed by the user application, and can therefore have a constructor that takes any application-specific data. The Middleware then calls upon this factory to generate a Local Class methods instance whenever the Middleware needs one.

The only requirements that the Middleware places on this factory are that it must have one or more `create` methods taking Local Class state and Local Class constructor parameters, and that the `create` methods must return a local methods object. As long as it does these things, the factory can do anything, such as taking application data in its constructor and passing that application data to the Local Class methods objects returned by its `create` methods. Experienced software engineers might recognize this as the "Factory Pattern".

An example may help to illustrate the concept better. Consider the `Example::HomeStation` Local Class whose TDL looks like this:

```
local class HomeStation
{
    // Ensure name is not the empty string
    HomeStation( string name, Location location );

    void set_name( in string name ); // Ensure name is not the empty string
    float64 distanceFrom( in Location location ) const;

    readonly string name;
    Location location;
};
```

Note that the code fragments referenced are for particular versions of the Example object models, specifically `Example-Tank-v3`. This object model may be updated/refined as necessary.

Local Methods Customizations for accessing Application Data

Suppose that your application has a logging system and needs to log information whenever the `set_name` method is invoked. Suppose that any code wishing to use this logging system needs access to a "Logger" object. Your local methods implementation object, which supplies the implementation for the `set_name` method, will need access to the `Logger` object. The `set_name` method itself cannot take the `Logger` object as a parameter because the `set_name` method's parameters are predetermined by the object model. Instead, you will need to pass the `Logger` object to your `Example::HomeStation::LocalMethodsFactory` implementation object, which will then pass it to the local methods implementation object.

Your local methods factory implementation looks (in relevant part) like the following (with the class declaration and definition combined). Note that there may be multiple `create` methods using different parameters that would all need to be changed to add the `Logger` argument.

```

class LocalMethodsFactoryImpl
: public Example::HomeStation::LocalMethodsFactory
{
public:
    LocalMethodsFactoryImpl( Logger & rLogger )
    : rLogger_( rLogger );

    virtual
    Example::HomeStation::LocalMethodsPtr const
    create( Example::HomeStation::State & rState,
            std::string name,
            Example::Location::ImmutableLocalClassPtr const & location )
    {
        return Example::HomeStation::LocalMethodsPtr(
            new LocalMethodsImpl( rLogger_, rState, name, location ) );
    }
    ...
private:
    Logger & rLogger_;
};

```

Your local methods implementation looks (in relevant part) like this (with the class declaration and definition combined):

```

class LocalMethodsImpl
: TENA::Middleware::Utils::noncopyable,
  public virtual Example::HomeStation::LocalMethods
{
public:
    LocalMethodsImpl( Logger & rLogger,
                      Example::HomeStation::State & rState,
                      std::string const & name,
                      Example::Location::LocalClassPtr const & location )
    : rLogger_( rLogger )
    {
        ...
    }
    ...
private:
    Logger & rLogger_;
};

```

Release 6.0.5 Update ⚠ – Prior to middleware version 6.0.5, the type `boost::noncopyable` was used instead of the `TENA::Middleware::Utils::noncopyable` type.

Use of Arbitrary Value

The automatically generated example application code, including the user defined constructors and methods for Local Classes, will utilize the [ArbitraryValue](#) mechanism to automatically generate arbitrary values (e.g., value of "1" for an integer, value of "1.0" for a floating point number). This mechanism is used so that the automatically generated example application code will compile and run without modification.

When an implementation developer is customizing the behavior of a local methods implementation, all occurrences of `ArbitraryValue` should be replaced with proper values. Removing the default compiler directive "`TENA_MIDDLEWARE_ALLOW_ARBITRARY_VALUE`" from the implementation build file (Makefile under UNIX and Project file under Visual Studio) will ensure that the `ArbitraryValue` is not being used.

i — Remove compiler directive "`TENA_MIDDLEWARE_ALLOW_ARBITRARY_VALUE`" from implementation build file.

Implementation `getDescription`

All local method implementations are required to implement the method `getDescription` that provides a description name (*yes, `getName` would be a better name for this method*) for the particular implementation. Developers of a local method implementation need to modify all `getDescription` methods within their implementation library (if more than one exist) to provide this name intended to be unique from all other implementations.

The string returned from `getDescription` is used by the middleware to detect whether all applications joining an execution are using the same implementation for a particular Local Class type. Although enforcing that all applications use the same implementation is not a requirement, it is typical the appropriate behavior (and the default behavior for the middleware).

The descriptive implementation name should also include a version identifier, e.g., "fullConversions-v1", in order to differentiate between different implementation versions of the same software. Different versions are typically necessary to correct defects or improve functionality.

 — Provide descriptive name in all `LocalFactory::getDescription` methods.

Local Methods Factory Registration

Now that the local methods factory and local methods implementation has been written to use the application data (such as a `Logger` object), the appropriate application data needs to be provided by the application and the local methods factory needs to be registered with the middleware. The default (and automatically generated) local method factory registration process is done through a global initialization technique. In the default `localMethodsFactoryImpl.cpp` file (e.g., `myImpl/Example/HomeStation/LocalMethodsFactoryImpl.cpp`), a `RegistryFactory` class is defined and is created in an anonymous namespace as shown in the code below for the `Example::HomeStation` type.

```
/// An anonymous namespace to keep its contents "hidden" inside this file.
namespace
{
    class RegisterFactory
    {
    public:
        RegisterFactory()
        {
            boost::shared_ptr< Example::HomeStation::LocalMethodsFactory > factory(
                new Example::HomeStation::SharedLocalImpl::LocalMethodsFactoryImpl );

            TENA::Middleware::LocalClassMethodsFactoryRegistry::
                getInstance().registerFactory( factory );
        }
    }; // End of class RegisterFactory

    // Create an object, thereby invoking the RegisterFactory constructor
    RegisterFactory registerFactory;
} // End of namespace
```

The motivation of the static registration of the local methods implementation factory is that application developers only need to link the library containing the implementation code and the factory will be automatically registered with the middleware. Additional information related can be found on the [Local Class Implementation Registration documentation page](#).

Since the custom local methods implementation has added an argument (or arguments) for application data, the default `RegisterFactory` code will not compile. The above `RegisterFactory` code should be deleted because applications using the custom local methods implementation will need to manually register the local methods factory. Once the `RegisterFactory` code is removed from the implementation, application developers attempting to use this custom implementation will need to make the following invocation prior to attempting to join an execution.

```
boost::shared_ptr< Example::HomeStation::LocalMethodsFactory > factory(
    new Example::HomeStation::SharedLocalImpl::LocalMethodsFactoryImpl( logger );

TENA::Middleware::LocalClassMethodsFactoryRegistry::
    getInstance().registerFactory( factory );
```

Local methods implementation developers can also chose to define static methods within their `LocalMethodsFactoryImpl` class to perform this registration process and simplify the interface for users of the implementations. Although the `LocalMethodsFactoryImpl` class could define a static method to set application data, e.g., `setLogger`, this approach is not recommended because if the application forgets to invoke this method the application will encounter a runtime exception the first time a `LocalMethodsFactory` is needed, versus detecting the problem at join execution time.

Local method implementation developers are encouraged to review the [Sharing Object Model Implementations documentation page](#) for guidance on developing and sharing object model implementation code.

Usage Considerations

Thread Safety

Developers of Local Class implementations that utilize application data need to be aware of thread safety issues with the local method implementations. Since multiple middleware programming threads, as well as application threads, may be simultaneously invoking local methods, any use of application data needs to be done in a thread safe manner.

For example, consider a local methods implementation that utilized an `ostream` to log information about the invocation of a Local Class method. If every Local Class created is provided that same `ostream` object, there can be concurrent access to that stream resulting in undefined behavior. In this situation, the method implementation should also be provide access to a common thread locking primitive (e.g., `ACE::Mutex`) and this lock should be acquired before the `ostream` object is used and released when completed.

 — Thread safety is required for application data used within the local methods implementations.

Local Class Constructor

Local Class Constructor

The TENA Meta-Model supports definition of user specific Local Class constructors, which along with default constructors are used to create Local Classes. User specific Local Class constructors can be used to implement range checking of attribute values or perform other initialization operations when a Local Class is created. If user specific Local Class constructors are not defined in the object model, then default constructors will be defined; one constructor with no attribute value arguments and another constructor for all non-optional attributes.

⚠ When a Local Class implementation is required, it must be implemented in C++. The underlying requirement of the TENA Middleware is that a single shared implementation be used by all applications in an execution. Only a C++ implementation library is easily usable from C++, Java and .Net applications, so this restriction is imposed. Avoiding the use of constructors and local class methods in an object model ensures that implementation will not be required.

Description

Release 6 of the TENA Middleware provides the ability to explicitly define constructors for both [Local Classes](#) and [Messages](#). Constructors are used to initialize attributes, either directly from parameters or by performing some computation, when an object is instantiated.

Within the TENA Meta-Model, Local Class constructors are optional. User defined Local Class constructors can be used when it is necessary to provide specific implementation behavior to check the Local Class attribute values or perform specific implementation behavior at construction time.

For example, if it is necessary to ensure that a Local Class attribute value is non-zero, a constructor can be defined that includes the attribute value (as well as other attributes, as appropriate) to ensure that the particular attribute value is "range checked" at construction time. This attribute can be defined as `readonly` in the object model definition to prevent the generation of a `set` method that would allow the attribute value to be set after construction.

Another use case for user-defined constructors would be when there are dependencies between multiple attribute values that can only be effectively checked together – using a constructor with these dependent attribute values to perform this checking and marking these attributes as `readonly` would satisfy this need.

User defined Local Class constructors can also be necessary to perform implementation specific initialization operations. For example, a Local Class implementation that was required to connect to an external database may need to establish that database connection at construction time. Note that in this particular example, the `LocalClassFactory` could be used with static methods for the database operations if that provides a better design for the needs of the implementation.

As mentioned, Local Class constructors are optional and if no constructors are specified in the object model, a default constructor with no arguments and a constructor with an argument for each non-optional attribute are automatically generated. Otherwise, only the specified constructors are available.

For the Local Class constructors, whether user-defined or the default generated, the API (application programming interface) uses static `create` functions that match the argument syntax defined by the constructors. These `create` functions are defined in the namespace defined by the object model `Package` structure.

A simple auto-generated `create` function for the `Example::Location` Local Class that does not define any user specified constructors is shown in the code fragment below. This `Example::Location` Local Class has two `float64` (64 bit floating point numbers) numbers for X and Y coordinate values and an instance can be created through the `create` method.

```
// Location TDL syntax
local class Location
{
    float64 xInMeters;
    float64 yInMeters;
};

// Location Local Class creation
TENA::float64 xLocation( 2.0 );
TENA::float64 yLocation( 3.0 );
Example::Location::LocalClassPtr pLocation(
    Example::Location::create( xLocation, yLocation ) );
```

Local Class instances are held within a reference counted pointer (so called "smart" pointer), as indicated by the type suffix `ptr`. This is because multiple references to the same Local Class instance may exist within an application's process space (including middleware references) and only when all references are released will the underlying Local Class instance be destroyed and memory released.

An example of a Local Class that has a user defined constructor is illustrated in the code fragment below. The `AreaOfEffect` constructor defined requires the `centerOfArea` and `radius` to be specified, which is used in the `create` invocation.

```

// AreaOfEffect TDL syntax
local class AreaOfEffect
{
    AreaOfEffect( Location centerOfArea, float64 radius );

    boolean isLocationAffected( in Location here );

    readonly Location centerOfArea;
    readonly float64 radius;
};

// AreaOfEffect Local Classe creation
TENA::float64 xCenter( 3.0 );
TENA::float64 yCenter( 4.0 );
Example::Location::LocalClassPtr pCenter(
    Example::Location::create( xCenter, yCenter ) );
TENA::float64 radius( 5.0 );
Example::AreaOfEffect::LocalClassPtr pArea(
    Example::AreaOfEffect::create( pCenter, radius ) );

```

Range Checking

It is often appropriate for constructors to range-check attributes when the Local Class is initialized. In order to properly range-check an attribute at construction, the local class needs to be modified as follows:

1. Mark each attribute to be range-checked as `readonly`. This causes the OMC (Object Model Compiler) to not generate a `set_AttributeName` method for setting the attribute directly.
2. Add an explicit constructor that assigns the values for all of the attributes that need to be range-checked.

Additionally, if an attribute requiring range-checking needs to be able to be changed after construction, you should add a `set_AttributeName` method to the local class or message, and implement it with the appropriate range checking logic.

Constructor Implementation Development

When designing an object model with Local Classes that declare constructors, an implementation for these constructors must be developed to be used applications using the object model. The TENA Object Model Compiler (OMC) will automatically generate the source code for a "stub" implementation of any user defined Local Class constructors (as well as any Local Class methods). This generated implementation code is associated with a shared implementation library that contains all of the Message and Local Class implementation code for the particular object model.

The generated shared implementation library is installed in the `TENA_HOME/TENA_VERSION/src/OM_NAME/OM_NAME-SharedLocalImpl-v1/myImpl` directory. Users can modify the "SharedLocalImpl" name and the "1" version if desired. For each Message or Local Class that contains a constructor and /or local method, a sub-directory will exist based on the Message or Local Class name. Under this directory will be a `SharedLocalImpl/LocalMethodsImpl.cpp` file that contains the "stub" code for the Local Class constructor implementation.

Object model implementation developers can begin with the automatically generated shared implementation library code and modify to support the specific implementation needs. Details on the generated Local Class constructor "stub" code is described in the API section below.

Once the object model implementation code has been built and tested, the implementation developer can package up the necessary files to share with other users of that object model. The [sharing object model implementations documentation page](#) provides additional information on sharing the developed implementation code.

Additionally, applications that use object models that require implementations need to register the appropriate implementation (since multiple object model implementations are possible). The [Local Class implementation registration documentation page](#) provides information on the registration process.

C++ API Reference and Code Examples

When an application is just **using** a Local Class with defined constructors, the necessary API of concern is the default `create` function(s) discussed above. This function allows an application to create a particular `LocalClass` instance.

If it is necessary to **develop** the Local Class constructor implementation, then the developer needs to modify the auto-generated shared implementation library code associated with the particular object model.

As an example, consider the `Example-Tank` object model `AreaOfEffect` Local Class type (object model TDL shown above). The generated code that needs to be modified for the constructor implementation is defined in the source code file `src/Example-Tank-v1/Example-Tank-v1-SharedLocalImpl-v1/myImpl/Example/AreaOfEffect/SharedLocalImpl/LocalMethodsImpl.cpp`. Note that the version number of the `Example-Tank` object model is expected to change, so the "-v1" in this directory location may be different.

This auto-generated `LocalMethodsImpl.cpp` "stub" implementation code is shown below. The code will build and run without modification, but the behavior uses arbitrary values and default behavior that needs to be replaced with the necessary behavior needed by the implementation developer.

```

Example::AreaOfEffect::SharedLocalImpl::
LocalMethodsImpl::
LocalMethodsImpl( Example::AreaOfEffect::State & rState,
    Example::Location::ImmutableLocalClassPtr const & /* centerOfArea */,
    TENA::float64 /* radius */ )
:
rState_( rState )
{
    // This constructor is invoked by the
    // Example::AreaOfEffect::SharedLocalImpl::LocalMethodsFactoryImpl
    // as a result of a call to
    // Example::AreaOfEffect::create(
    //     centerOfArea, radius )
    //
    // This is a "stub" example implementation. It is provided as a starting
    // point. This "stub" implementation should be replaced with the real
    // implementation.

    // Initialize all the non-optional attributes of
    // Example::AreaOfEffect
    /// \todo In a real application, instances of \c ArbitraryValue<T>
    /// should be replaced with values that are sensible for the
    /// particular application.
    Example::Location::LocalClassPtr centerOfArea(
        Example::Location::create(
            /* xInMeters */ TENA::Middleware::ArbitraryValue< TENA::float64 >(),
            /* yInMeters */ TENA::Middleware::ArbitraryValue< TENA::float64 >() ) );
    this->rState_.set_centerOfArea( centerOfArea );

    this->rState_.set_radius( TENA::Middleware::ArbitraryValue< TENA::float64 >() );
}

```

Specifically, the implementation developer should remove all occurrences of `ArbitraryValue` and set the appropriate Local Class attribute values using the constructor arguments. Note that the name of the constructor arguments, `centerOfArea` and `radius` in this example, are commented out to avoid compiler warnings associated with unused arguments. Implementation developers should remove these comment markers and use the constructor arguments to set the attribute values, constrained in the `State` object, appropriately.

As an example, the "stub" implementation can be changed as illustrated below. The `radius` attribute value is checked to ensure it is non-negative before setting the value contained in the `State` object. If the range value received is negative, then the constructor throws a `std::invalid_argument` exception. Note that the `float64` value of the range is converted to a string using `boost::lexical_cast` so it can be included in the message portion of the exception.

```

#include <boost/lexical_cast.hpp>

Example::AreaOfEffect::SharedLocalImpl::
LocalMethodsImpl::
LocalMethodsImpl( Example::AreaOfEffect::State & rState,
    Example::Location::ImmutableLocalClassPtr const & centerOfArea,
    TENA::float64 radius )
:
rState_( rState )
{
    // Check if radius is non-negative

    if (radius >= 0.0)
    {
        this->rState_.set_centerOfArea( centerOfArea );
        this->rState_.set_radius( radius );
    }
    else
    {
        throw std::invalid_argument( std::string(" radius (" )
            + boost::lexical_cast< std::string >( radius )
            + ") out of range [0.0, +infinity]"));
    }
}

```

Note that within the automatically generated `LocalMethodsImpl` classes that an additional constructor is defined to be used when the middleware needs to create a Local Class from information sent across the network (i.e., middleware receives a message instance from a publishing application). A destructor is also automatically generated for the `LocalMethodsImpl` classes. Neither the network constructor or destructor will typically need to be specialized by the implementation developer and the automatically generated code will suffice.

Java API Reference and Code Examples

When an application is just **using** a Local Class with defined constructors, the necessary API of concern is the default `create` function(s) discussed earlier. This function allows an application to create a particular `LocalClass` instance.

If it is necessary to **develop** the Local Class constructor implementation, then the developer needs to modify the auto-generated shared implementation library code associated with the particular object model. **⚠ For the current TENA Middleware, all object model library implementation code must be implemented in C++.** Because a single Local Class implementation library must be shared among all applications, developed in any supported language, that publish or subscribe to a given Local Class type, the library must be implemented in a language that is usable for any language supported by the TENA Middleware. There is no efficient way to access a Java implementation or .Net implementation from a C++ program. In contrast, most of the access for the Java and .Net bindings is already accomplished by wrapping existing C++ libraries for use by Java and .Net.

In order to develop an implementation for Local Classes in an object model, a developer would have to download a C++ Object Model distribution, develop the implementation code using as described above. Then simply use `java.lang.System.loadLibrary` to load the resulting library, before creating an `Execution` in the application.

Local Class Implementation Registration

Local Class Implementation Registration

Local Classes can be created by an application or by the middleware in response to receiving an SDO update or a message from another application. When a Local Class has a user defined constructor or method, a `LocalMethodsFactory` is needed so that whenever the Local Class is created the appropriate implementation code is utilized. The factory must be registered with the middleware, which is automatically done with the automatically generated example applications by simply including the appropriate implementation header file and linking with the associated library. Application developers are allowed the freedom to customize this registration mechanism for explicit control if necessary.

 – All Local Class implementations must be implemented in C++. This allows the use of the resulting libraries from any of the MW supported languages.

Description

Local Class Implementation Development

When designing an object model with Local Classes that declare constructors or methods, an implementation for these constructors must be developed to be used applications using this Local Class type. The TENA Object Model Compiler (OMC) will automatically generate the source code for a "stub" implementation of any user defined Local Class constructors and methods. This generated implementation code is associated with a shared implementation library that contains all of the Message and Local Class implementation code for the particular object model.

For each Message within the object model that contains a constructor and/or local method, a `SharedLocalImpl` sub-directory will exist followed by a directory name based on the Message or Local Class name (e.g., `SharedLocalImpl/Example_HomeStation`). Under this directory will be a `LocalMethodsImpl.cpp` file that contains the "stub" code for the Local Class constructor implementation. This stub code can be used to develop a proper implementation for the Local Class or Message.

Object model implementation developers can begin with the automatically generated shared implementation library code and modify to support the specific implementation needs. In the case of TENA standard object models, and other object models registered in the TENA Repository, the implementations have already been developed and are included with the object model packages.

The automatically generated example shared implementation library is installed in the `TENA_HOME/TENA_VERSION/src/OM_NAME/OM_NAME-SharedLocalImpl-v1/myImpl` directory. Users should modify the "SharedLocalImpl" name if a custom implementation is being developed. Changing the name of the implementation requires finding all occurrences of `SharedLocalImpl` in the generated source example code, and replacing it with the desired name. This includes renaming any directories named `SharedLocalImpl`. Once done, the resulting source code will generate an implementation with a different name.

Local Class Implementation Registration

A Local Class has two parts: State and `LocalMethods`. If a Local Class type specifies (in the object model definition) neither constructors nor methods, a `LocalMethods` implementation (and therefore `LocalMethodsFactory`) is not necessary. If a constructor or method is declared, an application must supply a `LocalMethods` and `LocalMethodsFactory` implementation that satisfies the interface specified in the definition header file.

 — Note that the Object Model Compiler will create default constructors for Local Classes in which no user defined constructors are defined. In this situation, the implementation code for the default constructors is included with the object model definition code and a `LocalMethods` and `LocalMethodsFactory` is not needed. Only user defined constructors require a separate implementation.

The `LocalMethodsFactory` is called by the Middleware to create the `LocalMethods` implementation objects for any Local Classes that are created. Local Classes can be created explicitly by the application, or they can be created by the middleware in response to receiving a local class from some other publishing application. Therefore, a `LocalMethodsFactory` implementation must be registered for each Local Class type (to be used by the application) that has a user defined constructor or method before an application can join the execution.

When does the application pass the `LocalMethodsFactoryImpl` object to the Middleware? It needs to be done early on, before any Local Class can be created. Therefore, the TENA Middleware has this rule: If an application (potentially) uses a local class with constructors or methods defined, a `LocalMethodsFactory` must be registered in the application before that application calls `TENA::Middleware::Runtime::joinExecution`.

How does the application pass the `LocalMethodsFactoryImpl` object to the Middleware (i.e., How does the application register the local class methods factory implementation with the Middleware)? The technical answer: The application calls the `registerFactory()` method on the `TENA::Middleware::LocalClassMethodsFactoryRegistry` as follows:

```
boost::shared_ptr< Example::HomeStation::LocalMethodsFactory > factory(
    new SharedLocalImpl::Example_HomeStation::LocalMethodsFactoryImpl );

TENA::Middleware::LocalClassMethodsFactoryRegistry::
    getInstance().registerFactory( factory );
```

But, it would be somewhat onerous for the application developer to have to write this code manually for every single Local Class. To assist with this, the automatically generated example application provides the `RegisterFactory` code in the `LocalMethodsFactoryImpl.cpp` source file. Not only does this file implement the `LocalMethodsFactory`, but it also declares an instance of `RegisterFactory` that performs the automatic registration of the implementation **during global initialization**. Through this mechanism, registration occurs by including a single header file (e.g., `Example/HomeStation/Impl.h`) and linking the associated implementation library (e.g., `libExample-Tank-v4-SharedLocalImpl-v7.2.0-w8-vs2012-v6.0.4.lib`).

Working Around Defective Linkers

Some compilers and linkers, such as Microsoft Visual C++, have a design flaw by which they fail to link code that has no references to it. (This is incorrect behavior because that code, despite having no references to it, may be invoked at static initialization time, and may do important things. If the linker, in a well-intended attempt to be efficient, omits that code, the application can fail to do things that it is supposed to do.) Because of this defect, the static initialization code described above that performs local methods factory registration fails to get called with certain linkers unless extra steps are taken to deceive the defective linker into performing correctly.

To work around this linker design flaw, the automatically generated example application code defines a dummy "forceLinkage" variable at the bottom of the `LocalMethodsFactoryImpl` definition file, alongside the code that performs the registration via static initialization. The generated code also contains an `Impl.h` header file (one for each Local Class or Message) that references the dummy "forceLinkage" variable. That reference, as the name implies, deceives the defective linker into linking the local methods factory registration code into the executable so that it gets invoked upon static initialization. To simplify even further, the example application code uses a `localClassImpls.h` header file that #includes all of the `Impl.h` header files for all the Local Classes and Messages in the object model. The application has only to #include that one `localClassImpls.h` header file to cause all of the local class methods factories to be registered automatically with the TENA Middleware.

What if more fine-grained control is needed?

The `exampleApplication` plugin used by the Object Model Compiler generates example applications that assume that all the local methods in a given object model are implemented in a single library. If, for some reason, the local methods implementation for individual Local Classes and/or Messages needs to be separated into multiple libraries, that is very possible. In fact, as described above, the `localClassImpls.h` header file is simply an aggregation of header files for each Local Class with a method or constructor defined. For example, the `Example-Tank-v4-SharedLocalImpl` `localClassImpls.h` header file aggregates these "fine-grained" header files:

- `Example/AreaOfEffect/Impl.h`
- `Example/ComplexAreaOfEffect/Impl.h`
- `Example/Explosion/Impl.h`
- `Example/HomeStation/Impl.h`

If multiple implementation libraries exist, the application build files just need to be adjusted to include the appropriate fine-grained `Impl.h` header files and link with the necessary implementation libraries.

Explicit LocalMethodsFactory Registration

The `RegisterFactory` class is intended to be used to support automatic registration during static initialization time. If it is necessary to support applications being able to explicitly register the appropriate local methods factories, the implementation developer for the Local Class implementation can disable the static `RegisterFactory` mechanism.

In this situation, the entire `RegisterFactory` class, as well as the static creation, should be removed from the `LocalMethodsFactoryImpl.cpp` file. Since the implementation will no longer support automatic registration of the `LocalClassFactory`, applications attempting to use a particular local methods implementation will need to add the following code **before** the application attempts to call `TENA::Middleware::Runtime::joinExecution`. A code example, shown below, is identical to the code contained in the corresponding `RegisterFactory` class.

```
boost::shared_ptr< Example::HomeStation::LocalMethodsFactory > factory(
    new SharedLocalImpl::Example_HomeStation::LocalMethodsFactoryImpl );

TENA::Middleware::LocalClassMethodsFactoryRegistry::
    getInstance().registerFactory( factory );
```

An expected use case for explicit local methods factory registration is when the application operator needs to select a particular local methods implementation at runtime. In this situation, the registration can't be automatically performed at static initialization time, unless the application was able to dynamically load the implementation library.

C++ API Reference and Code Examples

As mentioned above, application developers that are merely attempting to use a Local Class that requires an implementation (i.e., contains user defined constructor or method), the default registration behavior is to ensure that the appropriate `Impl.h` header files are included by the application and the build file links the appropriate implementation libraries. This mechanism relies on the static initialization registration mechanism.

The `Impl.h` header files are designed to be implementation independent, so if it is necessary for an application to change to a different local method implementation, only the build file needs to be updated to link the correct implementation library. In other words, there is no need to change application source code to have the application change from using one local method implementation to another implementation (provided that the implementations are using the same object model definition and have not customized the implementation to require access to application data — see [Accessing Application Data from a Local Class documentation page](#)).

For developers that need to actually provide object model implementations, the automatically generated implementation and registration code will satisfy most development needs. The generated "stub" code for the Local Class constructors and methods is installed in the SharedLocalImpl library directory (e.g., `src/Example-Tank-v4/Example-Tank-v4-SharedLocalImpl-v7.2.0`). The relevant implementation "stub" code is contained in a sub-directory structure based on the particular object mode type, and the implementation code that requires developer modification is named `LocalMethodsImpl`.

For example, the implementation code for the `HomeStation` Local Class from the `Example-Tank` object model is shown below for the file `src/Example-Tank-v4/Example-Tank-v4-SharedLocalImpl-v7.2.0/myImpl/SharedLocalImpl/Example_HomeStation/LocalMethodsImpl.cpp`. These `LocalMethodsImpl` classes contain all of the constructors and methods that need to be implemented for the particular Local Class type. In the case of the example `HomeStation` Local Class, there is a user defined constructor and a several methods named `setNameAndLocation` and `distanceFrom`.

An implementation developer would replace all of the `ArbitraryValue` code with desired values and behavior necessary for the constructor and method implementations. As noted, a constructor for the middleware to create a Local Class when an instance is received over the network is defined, as well as a destructor, but in most cases the implementation developer does not need to specialize the automatically generated code for these operations.

```
/*
 * \file SharedLocalImpl/Example_HomeStation/LocalMethodsImpl.cpp
 *
 * \brief Contains the definition of methods of
 * SharedLocalImpl::Example_HomeStation::LocalMethodsImpl.
 *
 * This file contains the definition of the methods of the
 * SharedLocalImpl::Example_HomeStation::LocalMethodsImpl class.
 */
#include "LocalMethodsImpl.h"
#include <TENA/Middleware/ArbitraryValue.h>
#include <Example/Location.h>
#include <iostream>
#include <stdexcept>
#include <string>
///////////////////////////////
SharedLocalImpl::Example_HomeStation::
LocalMethodsImpl::
LocalMethodsImpl( Example::HomeStation::State & rState,
    std::string const & /* name */, Example::Location::ImmutableLocalClassPtr const & /* location */ )
:
rState_( rState )
{
    // This constructor is invoked by the
    // SharedLocalImpl::Example_HomeStation::LocalMethodsFactoryImpl
    // as a result of a call to
    // Example::HomeStation::create(
    //     name, location )
    //
    // This is a "stub" example implementation. It is provided as a starting
    // point. This "stub" implementation should be replaced with the real
    // implementation.
    // Initialize all the non-optional attributes of
    // Example::HomeStation
this->rState_.set_name( TENA::Middleware::ArbitraryValue< std::string >() );
/// \todo In a real application, instances of \c ArbitraryValue<T>
/// should be replaced with values that are sensible for the
/// particular application.
Example::Location::LocalClassPtr location(
    Example::Location::create(
        /* xInMeters */ TENA::Middleware::ArbitraryValue< TENA::float64 >(),
        /* yInMeters */ TENA::Middleware::ArbitraryValue< TENA::float64 >() ) );
this->rState_.set_location( location );
}
///////////////////////////////
SharedLocalImpl::Example_HomeStation::
LocalMethodsImpl::
LocalMethodsImpl( Example::HomeStation::State & rState,
    Example::HomeStation::LocalMethodsFactory::StateAlreadyInitializedTag )
:
rState_( rState ) // Holds the values that were sent over the network
{
    // This constructor is invoked by the
    // SharedLocalImpl::Example_HomeStation::LocalMethodsFactoryImpl
    // when the Middleware receives an instance of
}
```

```

// Example::HomeStation over the network.
//
// Although it is possible, modifying the contents of the rState_ in this
// constructor is rarely wise and probably never necessary.
//
// Typically, this "do-nothing" example implementation is all that is needed.
// However, the implementation of this constructor can be customized if a
// particular implementation requires it for some reason.
}
////////////////////////////////////////////////////////////////
SharedLocalImpl::Example_HomeStation::
LocalMethodsImpl::
~LocalMethodsImpl()
{
    // Typically, this "do-nothing" example implementation is all that is needed.
}
////////////////////////////////////////////////////////////////
void
SharedLocalImpl::Example_HomeStation::
LocalMethodsImpl::
set_name( std::string const & /* name */ )
{
    // This is just an auto-generated "stub" example implementation. It is
    // provided merely as a convenient starting point. This "stub" implementation
    // should be replaced with the real implementation.
    TENA::Middleware::ArbitraryValue< void >();
}
////////////////////////////////////////////////////////////////
TENA::float64
SharedLocalImpl::Example_HomeStation::
LocalMethodsImpl::
distanceFromLocationInMeters( Example::Location::ImmutableLocalClassPtr const & /* location */ )
const
{
    // This is just an auto-generated "stub" example implementation. It is
    // provided merely as a convenient starting point. This "stub" implementation
    // should be replaced with the real implementation.
    return TENA::Middleware::ArbitraryValue< TENA::float64 >();
}

```

The automatically generated example applications for Local Class constructor and method implementations (specifically the SharedLocalImpl library) perform implementation registration implicitly, when the library containing the constructors and/or methods is loaded. Below is a concrete example for Example::HomeStation Local Class from the Example-Tank Object model.

```

/*
 * \file SharedLocalImpl/Example_HomeStation/LocalMethodsFactoryImpl.cpp
 *
 * \brief Contains the definition of methods of
 * SharedLocalImpl::Example_HomeStation::LocalMethodsFactoryImpl.
 *
 * This file contains the definition of the methods of the
 * SharedLocalImpl::Example_HomeStation::LocalMethodsFactoryImpl class.
 *
 * \attention
 * This software was written under US Government contracts 1435-04-01-CT-31085,
 * DASG60-02-D-0006-040, DASG60-02-D-0006-122, and W900KK-09-C-0013; and the US
 * Government retains unlimited rights to the software.
 */
#include "LocalMethodsFactoryImpl.h"
#include "LocalMethodsImpl.h"
#include <Example/Location.h>
#include <TENA/Middleware/ArbitraryValue.h>
////////////////////////////////////////////////////////////////
std::string const
SharedLocalImpl::Example_HomeStation::
LocalMethodsFactoryImpl::
getDescription() const
{
    // A unique description string should be used for each different version of
    // every distinct C++ implementation for the methods and constructors of the

```

```

// Example::HomeStation local class
// (which is defined in Example-Tank-v4.tdl).
//
// This string is used by the Middleware to provide the (optional) local class
// implementation consistency checking feature.
//
// When creating a new implementation (or new implementation version) be sure
// to change this string!
return TENA::Middleware::ArbitraryValue< std::string >(
    "Example-Tank-v4-SharedLocalImpl-vtrunk-7.2.0" );
}

////////////////////////////////////////////////////////////////
SharedLocalImpl::Example_HomeStation::
LocalMethodsFactoryImpl::
LocalMethodsFactoryImpl()
{
    // Typically, this "do-nothing" example implementation is all that is needed.
}

////////////////////////////////////////////////////////////////
SharedLocalImpl::Example_HomeStation::
LocalMethodsFactoryImpl::
~LocalMethodsFactoryImpl()
{
    // Typically, this "do-nothing" example implementation is all that is needed.
}

////////////////////////////////////////////////////////////////
Example::HomeStation::LocalMethodsPtr const
SharedLocalImpl::Example_HomeStation::
LocalMethodsFactoryImpl::
create( Example::HomeStation::State & rState,
        std::string const & name, Example::Location::ImmutableLocalClassPtr const & location )
{
    // This method is invoked by the Middleware as a direct result of a call to
    // Example::HomeStation::create(
    //     name, location )
    return Example::HomeStation::LocalMethodsPtr(
        new LocalMethodsImpl( rState,
            name, location ) );
}

////////////////////////////////////////////////////////////////
Example::HomeStation::LocalMethodsPtr const
SharedLocalImpl::Example_HomeStation::
LocalMethodsFactoryImpl::
create( Example::HomeStation::State & rState,
        StateAlreadyInitializedTag tag )
{
    // This method is invoked by the Middleware when an instance of
    // Example::HomeStation is received over the network.
    return Example::HomeStation::LocalMethodsPtr(
        new LocalMethodsImpl( rState, tag ) );
}

////////////////////////////////////////////////////////////////
#include <TENA/Middleware/LocalClassMethodsFactoryRegistry.h>
#include <TENA/Middleware/staticInitializationFailure.h>
#include <sstream>
////////////////////////////////////////////////////////////////
///! An anonymous namespace to keep its contents "hidden" inside this file.
namespace
{
    /*!\class RegisterFactory LocalMethodsFactoryImpl.cpp \
    myImpl/SharedLocalImpl/Example_HomeStation/LocalMethodsFactoryImpl.cpp
    *
    * \brief Registers this local class factory with the Middleware
    *
    * Every local class (or message) with one or more methods or constructors must
    * have a corresponding local class factory that is registered with the
    * Middleware. This registration must be completed *before* the call to
    * TENA::Middleware::Runtime::joinExecution(). This class exists solely as a
    * means to perform this registration.
    *
    * The idea is to declare an object instance of this class (named

```

```

* "registerFactory", below) that will, in turn, call the constructor, which
* then performs the registration.
*
* The result is that the act of loading the library containing this code
* causes the global registerFactory object (below) to be constructed.
* Constructing the registerFactory object, in turn, registers this
* LocalMethodsFactoryImpl.
*/
class RegisterFactory
{
public:
    RegisterFactory()
    {
        try
        {
            boost::shared_ptr< Example::HomeStation::LocalMethodsFactory > factory(
                new SharedLocalImpl::Example_HomeStation::LocalMethodsFactoryImpl );
            TENA::Middleware::LocalClassMethodsFactoryRegistry::
                getInstance().registerFactory( factory );
        }
        catch ( std::exception const & ex )
        {
            std::ostringstream out;
            out << __FILE__ << ':' << __LINE__ << ":" << ex.what() << std::endl;
            TENA::Middleware::reportStaticInitializationFailure( out.str() );
        }
    }
}; // End of class RegisterFactory
// Create an object, thereby invoking the RegisterFactory constructor
RegisterFactory registerFactory;
} // End of namespace
// As another wrinkle, on some platforms, e.g., OS X and Windows, a library
// will be completely ignored if no explicit references to something inside of
// it is made. Thus, here we create a "dummy" symbol that can be used to force
// the library to be linked.
//
// For more information, see:
//   Example/HomeStation/Impl.h
#include <include/Example/HomeStation/Impl.h>
int Example_Tank_v4_SharedLocalImpl_vtrunk_7_2_0_ExampleHomeStation_forceLinkage = 0;

```

Implementation developers would only need to modify the registration code in case some alternative registration mechanism was needed, such as a dynamic implementation configuration mechanism that allowed an application operator to select the particular object model implementations to be used, or if the Local Class implementation required access to application data. The [Accessing Application Data from a Local Class documentation page](#) provides information on these procedures.

Local Class Inheritance and Polymorphism

Local Class Inheritance and Polymorphism

A derived local class instance can be used wherever a base local class is called for. When a method is invoked on a derived local class object referenced using a base local class pointer, the derived implementation of the method is called. In C++ terminology, all local class methods are virtual, and all references to local classes are pointers that perform virtual call dispatching to the method belonging to the actual concrete type of the object instance.

Description

In the case of [Local Classes](#) with inheritance, a *derived* Local Class instance can be used anywhere a *base* Local Class is needed, be it:

- as an attribute of an SDO Servant,
- as an attribute of a Message,
- as an attribute of a Local Class,
- as a parameter of a remote or local method, or
- as a return value of a remote or local method.

In such cases, any Local Class method invoked by a subscriber or client application will result in the derived Local Class behavior.

Local Class Polymorphism + OM Subsetting => Slicing

⚠ – Not all applications are linked against identical object models, so derived Local Class objects may have to be "sliced" when being sent between applications.

The TENA Middleware has a feature called [OM subsetting](#), which allows different applications linked against different (possibly overlapping) object models to participate in the same execution. Therefore, one application in an execution may know about (i.e., be linked against) two Local Class types related by inheritance (e.g., Base and Derived), while another application in the same execution may know about only one of those Local Class types (e.g. Base).

Recall that Local Class instances can be sent from one application to another (e.g., as attributes of SDOs, as remote method parameters, etc.). Suppose that one application is calling a remote method on an SDO residing in another application. Suppose, further, that the method is declared to return a Base Local Class, the calling application is linked against Base but not Derived, the callee is linked against both Base and Derived, and the callee's remote method implementation returns a Derived instance. In this scenario, the Middleware does what you'd expect: it permits the calling application to see only the Base portion of the return value. (This is called "slicing" because the Derived portion of the object is "sliced" off (and thrown away).) If the calling application then invokes a method on the returned Local Class, it will use a Base method implementation, not a Derived method implementation (since the calling application doesn't even know about the Derived type), even though the object was originally a Derived object when returned by the remote method.

Downcasting

Users can perform a safe downcast from a *base* Local Class pointer to a *derived* Local Class pointer using the `TENA::Middleware::dynamicCast` function. The function will throw `std::bad_cast` if the cast was not successful. A code example is shown [below](#).

C++ API Reference and Code Examples

Consider the following TDL, which has a base local class, a derived local class, and an SDO with one attribute whose type is the base local class.

```
package Example
{
    local class BaseLocalClass
    {
        int16 i;
        string getLabel() const;
    };

    local class DerivedLocalClass : extends BaseLocalClass
    {
        int16 j;
    };

    class Test
    {
        BaseLocalClass base;
    };
}
```

Using DerivedLocalClass in place of BaseLocalClass

This code updates the SDO using a DerivedLocalClass for the BaseLocalClass:

```
// Get the updater, that is held in an auto_ptr
std::auto_ptr< Example::Test::PublicationStateUpdater >
p_TestUpdater(
    p_TestServant->createUpdater() );

// create a LocalClass instance for attribute DerivedLocalClass
Example::DerivedLocalClass::LocalClassPtr p_TestUpdater_derived_LocalClass(
Example::DerivedLocalClass::create(
/* i */ 11,
/* j */ 22) );

// Change the value of the p_TestUpdater's
// LocalClass attribute base, using a pointer to derived
p_TestUpdater->
    set_base(
        p_TestUpdater_derived_LocalClass );
// Pass the updater in to the servant to modify the state atomically
p_TestServant->commitUpdater(
    p_TestUpdater );
```

Polymorphic method behavior

This code calls a method on a BaseLocalClass pointer, which could actually point to a DerivedLocalClass.

```
Example::BaseLocalClass::ImmutableLocalClassPtr base_LocalClass_Ptr(
    pTest->get_base() );

os << base_LocalClass_Ptr->getLabel() << std::endl;
```

If the base_LocalClass_Ptr is in fact a DerivedLocalClass instance, the call to getLabel() will invoke the DerivedLocalClass LocalMethodsImpl behavior.

Dynamic Casting a BaseLocalClass to a DerivedLocalClass

This code converts a BaseLocalClass to a DerivedLocalClass using dynamicCast. If the object referenced by the BaseLocalClass pointer is not actually a DerivedLocalClass object, then the dynamic cast throws std::bad_cast. Otherwise, a DerivedLocalClass is returned.

```
Example::BaseLocalClass::ImmutableLocalClassPtr base_LocalClass_Ptr(
    pTest->get_base() );

try
{
    Example::DerivedLocalClass::ImmutableLocalClassPtr derived_LocalClass_Ptr(
        TENA::Middleware::dynamicCast
        <Example::DerivedLocalClass::ImmutableLocalClassPtr>
        ( base_LocalClass_Ptr ) );

    os << std::endl;
    os << std::setw(4) << " " << "derived.i ";
    os << derived_LocalClass_Ptr->get_i() << std::endl;
    os << std::setw(4) << " " << "derived.j ";
    os << derived_LocalClass_Ptr->get_j() << std::endl;
}
catch (std::bad_cast)
{
    os << std::setw(0) << " " << "Example::Test.base ";
    os << std::endl;
    os << std::setw(4) << " " << "base.i ";
    os << base_LocalClass_Ptr->get_i() << std::endl;
}
```

Using Local Method with SDO Pointer Argument

Using Local Method with SDO Pointer Argument

In some situations, a Local Class (or Message) method may need to use an [SDO Pointer](#) (a reference to a particular SDO instance). However, in order to obtain access to the SDO instance, the Local Method needs to obtain the current Session. The Session is obtained through the SessionID attribute that is part of the Local Class (or Message) class structure.

Since a Local Class can be created outside of a Session (and therefore, a TENA Execution), the Local Method implementation needs to check to ensure that the Local Class (or Message) is associated with a valid Session. Without access to the Session, an application is unable dereference an SDO Pointer and access the SDO instance.

Description

Under some situations, it may be helpful for a Local Class (or Message) to use an SDO Pointer that is passed in as a method argument. In order to dereference the SDO Pointer to access the SDO instance, the particular TENA::Session object is required. At a high level, the pseudo-code for dereferencing the SDO Pointer is:

```
sessionID = pSDO->rState_.getSessionID();
pSession = getSessionByID( sessionID );
```

The Session pointer (pSession) is needed in order to dereference the SDO Pointer which is documented on the [SDO Pointer wiki page](#).

Developing object model Local Class implementations is considered an advanced capability for TENA developers, and using Local Methods with SDO Pointers is a further advancement. The code shown below is associated with a middleware test program. A future version of the TENA Middleware is expected to improve the API support for this capability, but until that time the following code illustrates how to use an SDO Pointer in a Local Method.

 — Developers of Local Methods should create an MW helpdesk case (<https://www.tena-sda.org/helpdesk/browse/MW>) to coordinate with the TENA SDA project team.

The object model component is TENA,Tests-LcMethodWithSdoParameter,1.0.0,.ObjectModel,source,all. The relevant content of TENA-Tests-LcMethodWithSdoParameter.tdl is

```
package TENA
{
    package Tests
    {
        package LcMethodWithSdoParameter
        {
            class Sdo
            {
                int32 attr;
            };

            local class Lc
            {
                Sdo * getSdo();
                void setSdo( in Sdo * theSdo );
            };
        }; // package LcMethodWithSdoParameter
    }; // package Tests
}; // package TENA
```

The getSdo method is only and stub and not relevant at this point. The method of interest is setSdo which takes a pointer to an SDO as an input parameter. The implementation of setSdo performs a pullPublicationState using the supplied SDO pointer. This basically demonstrates the concepts involved.

The setSdo implementation is

```
TENA_MW_@MW_VERSION@:@GROUP_NAME_VERSION_AS_SYMBOL@::
TENA::Tests::LcMethodWithSdoParameter::Lc::
LocalMethodsImpl::
setSdo( ::TENA::Tests::LcMethodWithSdoParameter::Sdo::SDOPointerPtr const & theSdo )
{
    ::TENA::Middleware::SessionID sessionID( this->rState_.getSessionID() );
```

```

if( sessionID == ::TENA::Middleware::MW605::unknownSessionID )
{
    std::cout << "sessionID is unknown.";
    return;
}

::TENA::Middleware::SessionPtr pSession = getSessionByID( sessionID );
if( !pSession.isValid() )
{
    std::cout << "Session is not valid" << endl;
    return;
}

if( !theSdo.isValid() )
{
    std::cout << "LocalMethodsImpl::setSdo, theSdo is invalid." << std::endl;
    return;
}

::TENA::Tests::LcMethodWithSdoParameter::Sdo::PublicationStatePtr
pPubState(
    theSdo->pullPublicationState( pSession ) );
std::cout << "pullPublicationState returned: " << std::endl;
std::cout << pPubState;
//::TENA::Tests::LcMethodWithSdoParameter::Sdo::MetadataPtr pMetadata(
//    //pPubState->getMetadata() );
//std::cout << pMetadata << std::endl;
std::cout << "end pullPublicationState output." << std::endl;
}

```

The local class Lc has access to the sessionID through its _rState member. The sessionID is used to look up the SessionPtr, which is needed by pullPublicationState.

Looking up the SessionPtr from the sessionID is not currently a MiddlewareAPI method, but it probably should be added. Similar code is used in the MiddlewareRuntime, but it is not broken out as a separate method.

Here is the code to look up the SessionPtr based on the sessionID:

```

///////////
static
::TENA::Middleware::SessionPtr
getSessionByID( ::TENA::Middleware::SessionID sessionID )
{
    if( sessionID == ::TENA::Middleware::MW605::unknownSessionID )
    {
        std::cout << "sessionID is unknown.";
        return ::TENA::Middleware::SessionPtr();
    }

    ::TENA::Middleware::Utils::SmartPtr< ::TENA::Middleware::Runtime > pRuntime(
        TENA::Middleware::ExecutionManagement::Runtime::getRuntimePtr() );
    ::TENA::Middleware::ExecutionManagement::Runtime::ExecutionListPtr pExecutions( pRuntime->getExecutions() );
    BOOST_FOREACH( TENA::Middleware::ExecutionPtr pExecution,
        *pExecutions )
    {
        if ( pExecution.isValid() )
        {
            ::TENA::Middleware::Execution::SessionListPtr pSessions(
                pExecution->getSessions() );
            #if (_MSC_VER == 1900) // VS2015 compiler warns that declaration of '_foreach_col' hides previous local
            declaration
            #pragma warning(push)
            #pragma warning(disable: 4456)
            #endif
            BOOST_FOREACH( TENA::Middleware::SessionPtr pSession,
                *pSessions )
            {
            #if (_MSC_VER == 1900)
            #pragma warning(pop)
            #endif
                if( pSession.isValid() )

```

```
{  
    if( pSession->getID() == sessionID )  
    {  
        return pSession;  
    }  
}  
}  
  
return ::TENA::Middleware::SessionPtr();  
}
```

Message

Message

Messages are used to send a collection of attribute values to interested subscribers. Once a Message is received by the subscribing application, all interactions with that Message are local in nature. Message method calls operate on the local Message data and there is no communication with the original Message sender.

Description

A Message in the TENA Meta-Model is used to transmit to interested subscribing applications a collection of attribute values used to characterize a particular one-time event or action. The Message modeling construct is different from the Class (i.e., Stateful Distributed Object, SDO) construct in that a Class exists for a duration of time in which attribute values may be updated, while a Message is ephemeral, represented by attribute values that are defined for only the moment in which it was transmitted.

Messages support a "one to many" communication mechanism, like SDOs, in which all applications that have expressed interest in the particular message will receive the sent message. Once the message is received, the receiving application does not require any interaction with the sending application. TENA Messages do not support remote methods, as SDOs do. TENA Messages do support local methods, as Local Classes do. Local methods can be used to provide helper functions associated with the message attributes.

Typically, a Message is used to convey information about a particular event or action that occurs within the TENA execution. For example, a Message could be used to indicate a "rocket launch" to synchronize systems during a test. Generally speaking, if the item or information being modeled represents something that exists/evolves over time, then an SDO should be used, whereas if the item or information being modeled represents something that occurs at a particular moment in time, then a Message should be used.

Message Sending Overview

Applications that need to send Messages of a particular type need to compile and link against the appropriate object model definition and implementation. An object model will have a required implementation if any of the Messages or Local Classes declare constructors or methods.

For a C++ application, the necessary header file that needs to be included for a publishing application is the main Message header file: `TENA_HOME/TENA_VERSION/include/OMs/OM_NAME/PACKAGE_NAMES/MESSAGE_NAME.h`, such as `C:\TENA\6.0.0\include\OMs\Example-Vehicle-v1\Example\Notification.h`. There is also a header file associated with message implementation registration, if the message has an implementation. Refer to the [Message Implementation Registration documentation page](#) for additional details on this topic.

Inside the publishing application, a `MessageSender` is created, as shown in the code example below.

Applications can continue to modify the `Message` attribute values and resend the same `Message` if convenient, or a new `Message` can be created. On the other hand, the `MessageSender` is used to signify to the middleware that the application intends to send `Messages` of the particular type, and this may trigger network communication with other applications. Therefore, applications should maintain the `MessageSender` versus creating a new instance each time a `Message` needs to be sent.

Additional details on constructing and sending `Messages` can be found in the [Message Constructor](#) and [Message Sender](#) documentation pages.

Message Receiving Overview

When an application is interested in subscribing to a particular `Message` type, it is necessary to compile and link against the appropriate object model definition and implementation (just like when publishing `Messages`). An object model will have a required implementation if any of the Messages or Local Classes declare constructors or methods.

For a C++ application, the header file that needs to be included for a subscribing application is the same as for a publishing application: `TENA_HOME/TENA_VERSION/include/OMs/OM_NAME/PACKAGE_NAMES/MESSAGE_NAME.h`, such as `C:\TENA\6.0.0\include\OMs\Example-Vehicle-v1\Example\Notification.h`. There is also a header file associated with message implementation registration if the message has an implementation. Refer to the [Message Implementation Registration documentation page](#) for additional details on this topic.

In order for an application to be notified by the middleware when a matching `Message` is received, the application needs to implement code that is executed in response to the message receipt. This application specific code is implemented as a class that derives from the `AbstractObserver` interface that contains a `messageEvent` method that is invoked by the middleware. The definition of the `AbstractObserver` interface is shown below.

Since the application developer designs the class extending the `AbstractObserver`, the developer is free to pass application specific data into the class to support unique application needs. For example, if the `Observer` needed to perform database operations in response to receiving a `Message`, the constructor could require the application to specify the database connection information.

An application can register one or more `Observers` to a particular `Message` subscription, and each `Observer`'s `messageEvent` method will be invoked when a matching `Message` is received. The `messageEvent` implementation has access to the `ReceivedMessage`, which is a combines the actual `Message` sent with metadata associated with the messages transmission.

Subscribing is done by invoking the Message type-specific subscribe method. The appropriate Session is provided as an argument, with optional arguments to control "self reflection" (i.e., receiving one's own messages) and advanced filtering. A code example illustrating the subscribe operation is shown below for the Example::Notification Message.

Once the application has invoked the subscribe call, publishing applications whose Messages match the subscription request will be sent to the subscribing application. While the subscribing application is processing callbacks (through the middleware `evokeMultipleCallbacks` mechanism, see [Callback Framework documentation page](#) for more information), the `messageEvent` method for each registered Observer will be executed for each message.

C++ API Reference and Code Examples

Sending a Message

In this simple example, the only Message attribute value is set through the Message constructor. For more complex definitions, an application will need to use set methods on the Message to provide values for all the Message attributes used by the application.

```
TENA::Middleware::CommunicationProperties communicationProperties( TENA::Middleware::Reliable );  
  
Example::Notification::MessageSenderPtr p_ExampleNotificationSender;  
p_ExampleNotificationSender =  
    Example::Notification::MessageSender::create(  
        pSession,  
        communicationProperties );  
  
Example::Notification::MessagePtr pMessage;  
pMessage = Example::Notification::create( "This is a test." );  
  
pMessageSender->send( pMessage );
```

Release 6.0.2 Update — A preferred create method was added with release 6.0.2 of the middleware. The first argument is changed to a `SessionPtr` as shown above, versus a `Session` reference (i.e., `"*pSession"`) used in the previous releases. Users are encouraged to use the `SessionPtr` interface. See [MW-4286](#) for more details.

Receiving a Message

This code is from the automatically generated example applications.

```
Example::Notification::SubscriptionPtr  
pExampleNotificationSubscription  
    new Example::Notification::Subscription );  
  
Example::Notification::AbstractObserverPtr pPrintingObserver  
    new MyApplication::Example_Notification::PrintingObserver(  
        verbosity, std::cout ) );  
  
pExampleNotificationSubscription->addObserver( pPrintingObserver );  
  
// Declare the application's interest in Notification messages.  
Example::Notification::subscribe(  
    pSession,  
    pExampleNotificationSubscription,  
    selfReflection );
```

Release 6.0.2 Update — An alternative `subscribe` method was added with release 6.0.2 of the middleware in which the `SessionPtr` argument is used, versus the `Session` reference (i.e., `"*pSession"`) that was used in the previous releases. Users are encouraged to use the `SessionPtr` interface. See [MW-4286](#) for more details.

Java API Reference and Code Examples

Sending a Message

This shows a simple example sending a `Notification` message, defined in the Example-Vehicle object model.

```

// Create a MessageSender for the Example.Notification message defined by the Example-Vehicle OM
Example.Notification.MessageSender exampleNotificationSender =
    Example.Notification.MessageSender::create(session, TENA.Middleware.CommunicationProperties.Reliable );
// Create a "Notification" and set its "text" attribute.
Example.Notification.Message message = Example.Notification.create( );
message.set_text("This is a test." );
// Send the message
messageSender.send( message );

```

Receiving a Message

This example shows subscribing to a message using the example observer code (`ExampleNotificationPrintingObserver`) provided with a Java Distribution for an object model.

```

Example.Notification.Subscription exampleNotificationSubscription = new Example::Notification::Subscription();

Example.Notification.AbstractObserver printingObserver =
    new org.Example.Notification.Subscriber.ExampleNotificationPrintingObserver( verbosity );

// Use the example observer class to print each time a Notification is created, updated or destroyed.
exampleNotificationSubscription.addObserver( printingObserver );

// Declare the application's interest in Notification messages.
Example.Notification.subscribe( session, exampleNotificationSubscription, selfReflection );

```

Additional Message Documentation Pages

- [Accessing Application Data from a Message](#) — Application data can be passed to a Message method implementation via the local class methods factory.
- [Adding Message Support to Existing Application](#) — Description of the steps involved to add Message publication and subscription to an existing application.
- [Message Constructor](#) — TENA Messages may define specific Message constructors, instead of default constructors, which are all used to create Messages.
- [Message ID](#) — Uniquely identifies a Message instance.
- [Message Implementation Registration](#) — Applications that use Messages with defined constructors or methods are required to register an implementation factory with the middleware.
- [Message Inheritance and Polymorphism](#) — The TENA Meta-Model allows Message types to inherit from other Message types to extend /evolve existing constructs and/or support polymorphic behavior.
- [Message Sender](#) — A MessageSender provides an API for a publishing TENA Messages to subscribing applications.
- [Message Subscription](#) — Message subscription registers interest in receiving messages matching the subscribed type and optional Filter.
- [Message Unsubscribe](#) — Unsubscribing from a Message type removes interest in receiving messages matching the subscribed type and filter.

Accessing Application Data from a Message

Accessing Application Data from a Message

Sometimes, a Message method or constructor implementation requires access to some data other than the local class state and the method arguments. To meet this requirement, the Message methods factory (`LocalMethodsFactory`) implementation class's constructor can take any user-defined parameters. The factory can then pass any data along to the `LocalMethods` objects that it constructs and returns from its `create()` method.

⚠ When a Message implementation is required, it must be implemented in C++. The underlying requirement of the TENA Middleware is that a single shared implementation be used by all applications in an execution. Only a C++ implementation library is easily usable from C++, Java and .Net applications.

Description

Message objects may be constructed by the TENA Middleware, as well as by the user application. The Middleware may construct a Message instance in any of several situations, such as when a Message is received over the network or when the user application invokes the `create` function in the Message's namespace. Since the user application does not always construct the Message instance, if any of the Message methods need access to any of the user application's data, there needs to be some way for that data to get passed to the Message methods implementation. This is accomplished by having the user application supply a Message methods factory to the Middleware. The factory is constructed by the user application, and can therefore have a constructor that takes any application-specific data. The Middleware then calls upon this factory to generate a Message methods instance whenever the Middleware needs one.

The only requirements that the Middleware places on this factory are that it must have one or more `create` methods taking Message state and Message constructor parameters, and that the `create` methods must return a local methods object. As long as it does these things, the factory can do anything, such as taking application data in its constructor and passing that application data to the Message methods objects returned by its `create` methods. Experienced software engineers might recognize this as the "Factory Pattern".

An example may help to illustrate the concept better. Consider the `Example::Explosion` Message whose TDL looks like this:

```
message Explosion : extends Notification
{
    Explosion( Location reportedlocation );

    float64 distanceFromCenterOfExplosionInMeters( in Location here ) const;

    AreaOfEffect affectedArea;
};
```

Message Customizations for accessing Application Data

Suppose that your application has a logging system and needs to log information whenever the `distanceFromCenterOfExplosionInMeters` method is invoked. Suppose that any code wishing to use this logging system needs access to a "Logger" object. Your Message implementation object, which supplies the implementation for the `distanceFromCenterOfExplosionInMeters` method, will need access to the `Logger` object. The `distanceFromCenterOfExplosionInMeters` method itself cannot take the `Logger` object as a parameter because the `distanceFromCenterOfExplosionInMeters` method's parameters are predetermined by the object model. Instead, you will need to pass the `Logger` object to your `Example::Explosion::LocalMethodsFactory` implementation object, which will then pass it to the local methods implementation object.

Your local methods factory implementation looks (in relevant part) like the following (with the class declaration and definition combined). Note that there may be multiple `create` methods using different parameters that would all need to be changed to add the `Logger` argument.

```

class LocalMethodsFactoryImpl
: public Example::Explosion::LocalMethodsFactory
{
public:
    LocalMethodsFactoryImpl( Logger & rLogger )
    : rLogger_( rLogger );

    virtual
    Example::Explosion::LocalMethodsPtr const
    create( Example::Explosion::State & rState,
            Example::Location::ImmutableLocalClassPtr const & reportedlocation )
    {
        return Example::Explosion::LocalMethodsPtr(
            new LocalMethodsImpl( rLogger_, rState, reportedlocation ) );
    }
    ...
private:
    Logger & rLogger_;
};

```

Your local methods implementation looks (in relevant part) like this (with the class declaration and definition combined):

```

class LocalMethodsImpl
: TENA::Middleware::Utils::noncopyable,
  public virtual Example::Explosion::LocalMethods
{
public:
    LocalMethodsImpl( Logger & rLogger,
                      Example::Explosion::State & rState,
                      Example::Location::ImmutableLocalClassPtr const & reportedlocation )
    : rLogger_( rLogger )
    {
        ...
    }
    ...
private:
    Logger & rLogger_;
};

```

Release 6.0.5 Update! – Prior to middleware version 6.0.5, the type `boost::noncopyable` was used instead of the `TENA::Middleware::Utils::noncopyable` type.

Use of Arbitrary Value

The automatically generated example application code, including the user defined constructors and methods for Messages, will utilize the [ArbitraryValue](#) mechanism to automatically generate arbitrary values (e.g., value of "1" for an integer, value of "1.0" for a floating point number). This mechanism is used so that the automatically generated example application code will compile and run without modification.

When an implementation developer is customizing the behavior of a local methods implementation, all occurrences of `ArbitraryValue` should be replaced with proper values. Removing the default compiler directive "`TENA_MIDDLEWARE_ALLOW_ARBITRARY_VALUE`" from the implementation build file (Makefile under UNIX and Project file under Visual Studio) will ensure that the `ArbitraryValue` is not being used.

i — Remove compiler directive "`TENA_MIDDLEWARE_ALLOW_ARBITRARY_VALUE`" from implementation build file.

Implementation `getDescription`

All local method implementations are required to implement the method `getDescription` that provides a description name (*yes, `getName` would be a better name for this method*) for the particular implementation. Developers of a local method implementation need to modify all `getDescription` methods within their implementation library (if more than one exist) to provide this name intended to be unique from all other implementations.

The string returned from `getDescription` is used by the middleware to detect whether all applications joining an execution are using the same implementation for a particular Message type. Although enforcing that all applications use the same implementation is not a requirement, it is typical the appropriate behavior (and the default behavior for the middleware).

The descriptive implementation name should also include a version identifier, e.g., "fullConversions-v1", in order to differentiate between different implementation versions of the same software. Different versions are typically necessary to correct defects or improve functionality.

i — Provide descriptive name in all `LocalFactory::getDescription` methods.

Local Methods Factory Registration

Now that the local methods factory and local methods implementation has been written to use the application data (such as a `Logger` object), the appropriate application data needs to be provided by the application and the local methods factory needs to be registered with the middleware. The default (and automatically generated) local method factory registration process is done through a global initialization technique. In the default `localMethodsFactoryImpl.cpp` file (e.g., `myImpl/Example/Explosion/LocalMethodsFactoryImpl.cpp`), a `RegistryFactory` class is defined and is created in an anonymous namespace as shown in the code below for the `Example::Explosion` type.

```
///! An anonymous namespace to keep its contents "hidden" inside this file.
namespace
{
    class RegisterFactory
    {
public:
    RegisterFactory()
    {
        boost::shared_ptr< Example::Explosion::LocalMethodsFactory > factory(
            new Example::Explosion::SharedLocalImpl::LocalMethodsFactoryImpl );

        TENA::Middleware::LocalClassMethodsFactoryRegistry::
            getInstance().registerFactory( factory );
    }
}; // End of class RegisterFactory

// Create an object, thereby invoking the RegisterFactory constructor
RegisterFactory registerFactory;
} // End of namespace
```

The motivation of the static registration of the local methods implementation factory is that application developers only need to link the library containing the implementation code and the factory will be automatically registered with the middleware. Additional information related can be found on the [Message Implementation Registration documentation page](#).

Since the custom local methods implementation has added an argument (or arguments) for application data, the default `RegisterFactory` code will not compile. The above `RegisterFactory` code should be deleted because applications using the custom local methods implementation will need to manually register the local methods factory. Once the `RegisterFactory` code is removed from the implementation, application developers attempting to use this custom implementation will need to make the following invocation prior to attempting to join an execution.

```
boost::shared_ptr< Example::Explosion::LocalMethodsFactory > factory(
    new Example::Explosion::SharedLocalImpl::LocalMethodsFactoryImpl( logger );

TENA::Middleware::LocalClassMethodsFactoryRegistry::
    getInstance().registerFactory( factory );
```

Local methods implementation developers can also chose to define static methods within their `LocalMethodsFactoryImpl` class to perform this registration process and simplify the interface for users of the implementations. Although the `LocalMethodsFactoryImpl` class could define a static method to set application data, e.g., `setLogger`, this approach is not recommended because if the application forgets to invoke this method the application will encounter a runtime exception the first time a `LocalMethodsFactory` is needed, versus detecting the problem at join execution time.

Local method implementation developers are encouraged to review the [Sharing Object Model Implementations documentation page](#) for guidance on developing and sharing object model implementation code.

Usage Considerations

Thread Safety

Developers of Message implementations that utilize application data need to be aware of thread safety issues with the local method implementations. Since multiple middleware programming threads, as well as application threads, may be simultaneously invoking local methods, any use of application data needs to be done in a thread safe manner.

For example, consider a local methods implementation that utilized an `ostream` to log information about the invocation of a Message method. If every Message created is provided that same `ostream` object, there can be concurrent access to that stream resulting in undefined behavior. In this situation, the method implementation should also be provide access to a common thread locking primitive (e.g., `ACE::Mutex`) and this lock should be acquired before the `ostream` object is used and released when completed.

 — Thread safety required for application data used within the local methods implementations.

Adding Message Support to Existing Application

Adding Message Support to Existing Application

There are several steps involved in adding the capability to publish and subscribe to a TENA Message in an existing application. The description below represents the minimal requirements associated with adding Message publication or subscription support to an existing application. Additional details regarding Message publication and subscription can be found in the [Message documentation section](#).

Integrating publishing or subscribing to Messages using the TENA Middleware to an existing application

An existing application that needs to add support for publishing or subscribing to Messages of a particular type will need to modify the application source code and build files. The source code is modified to include the appropriate Message header files for necessary declarations. For Message publishing, Application source code is modified to create and publish Messages. For Message subscribing, Observer classes must be written to handle events.

The general requirements are to:

1. Include header files or definition libraries that define the desired Messages and APIs.
2. Add appropriate code to create a Runtime, connect to an Execution, and create a Session.
3. For publishing, add code to create desired MessageSenders, and create and send Messages using the created MessageSenders.
4. For subscribing, add code to define actions to respond to message events (Observers), connect them to a Subscription and subscribe to desired Message types.
5. For subscribing, decide on appropriate way to integrate calls to evoke callbacks.
6. Ensure that appropriate changes to build files are made: appropriate paths and include libraries, as required.

The specific details will vary based on requirements. For example, when adding publication or subscription to an existing TENA Middleware application, the existing Execution and Session objects already being used in the code may be sufficient.

C++ API Reference and Code Examples

Adding ability to publish Messages in an existing C++ application

An example is shown, starting with the Example-Vehicle-Publisher C++ application and adding support to publish Notification Messages. The following changes are needed to the Example-vehicle-Publisher.cpp file. In this case, because the starting point is already a TENA middleware-based application, the code to get a Runtime, Execution and Session is already present. Only the code to create a MessageSender, and create and send Messages need be added.

Add the Message type header file:

```
#include <Example/Notification.h>
```

Then instantiate a MessageSender used to send messages.

```
Example::Notification::MessageSenderPtr p_ExampleNotificationSender( Example::Notification::MessageSender::create( pSession, communicationProperties ) );
```

Finally, create and send a Message.

```
Example::Notification::MessagePtr pMessage(
    Example::Notification::create( "Hello!" ) );
p_ExampleNotificationSender->send( pMessage );
```

The build files (Makefile on UNIX and Project file for Microsoft Visual Studio) need to be modified to link the object model definition library and the implementation library, if one exists. Since the Notification Message is defined in the Example-Vehicle object model whose definition library is already linked into the existing application, there are no modifications required to add the object model definition library. If the object model definition library needed to be added, the build file would need to add the definition library to list of libraries used by the linker (e.g., libExample-Vehicle-v1-\$(TENA_PLATFORM)-v\$(TENA_VERSION).lib gets added to Linker:Input:Additional Dependencies configuration option in the Visual Studio project).

With respect to object model implementation, the Notification Message type does not have any constructors or methods defined in the object model, so there is no object model implementation necessary. If the Message type being added did have an implementation, then the build files would need to compile against the implementation header files and link against the appropriate implementation library. Information on Message implementations and registration is discussed on the [Message implementation registration documentation page](#).

Adding ability to subscribe to Messages in an existing C++ application

An existing application that needs to add support for subscribing to Messages of a particular type will need to modify the application source code and (possibly) the build files. The source code is modified to include the appropriate Message header file to include the necessary declarations. Application source code is also modified to perform the subscription operations to subscribe to the particular SDO type. In this case, because the starting point is already a TENA middleware-based application, the code to get a Runtime, Execution and Session is already present.

The illustration of the source code modifications are shown below, starting with the `Example-Vehicle-Subscriber` application and adding support to subscribe to `Notification` Messages.

For Message subscriptions, an application needs to define one or more `Observers` with code to define the behavior when a message event occurs. Additional information on the `Observer` pattern and subscription can be found on the [subscription services documentation page](#).

In this example, an existing `PrintingObserver` class is used. This is automatically generated for the example code generated for SDO subscribing applications. The implementation code for this `Observer` is not shown below, but can be copied from an example subscribing application.

```
class PrintingObserver
    : public virtual Example::Vehicle::AbstractObserver
{
public:
    PrintingObserver( TENA::uint32 verbosity, std::ostream & os );

    virtual
    void
    messageEvent(
        Example::Vehicle::ReceivedMessagePtr const & pReceivedMessage );
private:
    std::ostream & os_;
    TENA::uint32 verbosity_;
};
```

Now, the application source code must be modified. The following changes are made to the `Example-Vehicle-Subscriber.cpp` file. First, add the Message type header and the observer header files. Note that the Observer class files are placed in the `myImpl/myApplication/Example_Notification` directory, which follows the suggested convention of the automatically generated Example Application code, but users are free to locate their Observer class files in a different location if desired.

```
#include <Example/Notification.h>
#include "myImpl/MyApplication/Example_Notification/PrintingObserver.h"
```

The main application is then changed to instantiate the `Subscription` object. The `Subscription` object is used to hold subscription configuration information and attach `Observers`. In this example, only default subscription behavior is necessary.

```
Example::Notification::SubscriptionPtr
pNotificationSubscription( new Example::Notification::Subscription );
```

Then the application source code is modified to instantiate the `PrintingObserver` and added to the `Subscription` object.

```
Example::Notification::AbstractObserverPtr pPrintingObserver(
    new MyApplication::Example_Notification::PrintingObserver( verbosity, std::cout ) );

pNotificationSubscription->addObserver( pPrintingObserver );
```

Finally, the application subscribes to the `Vehicle` SDO type.

```
Example::Notification::subscribe(
    *pSession,
    pNotificationSubscription,
    false /* no self-reflection */ );
```

As with adding publication support, the build files (Makefile on UNIX and Project file for Microsoft Visual Studio) need to be modified to link the object model definition library. Since the `Notification` Message is defined in the `Example-Vehicle` object model whose definition library is already linked into the existing application, there are no modifications required to add the object model definition library. If the object model definition library needed to be added, the build file would need to add the definition library to list of libraries used by the linker (e.g., `libExample-Vehicle-v1-$(TENA_PLATFORM)-v$(TENA_VERSION).lib` gets added to `Linker:Input:Additional Dependencies` configuration option in the Visual Studio project).

The header and build files for the `PrintingObserver` also needs to be added to the application's build file. A complication with using the example `PrintingObserver` is that this code relies on a common helper class `printFunctions`. If the existing application already defined `printFunctions` code (such as in this example), the existing `printFunctions` code needs to merged with the `printFunctions` code copied into this application. For experimentation purposes, the copied `MyApplication::Example_Notification::PrintingObserver::print` method could be modified to just eliminate the "os_ << pPubState" code lines.

Java API Reference and Code Examples

For Java applications, the Java Binding libraries will have to be on your application classpath at build-time and run-time. Looking at the Ant `build.xml` and associated `dependencies.xml` for one of the generated example programs should give some idea of the required additions to the classpath. For example, in the example `Notification` publisher application in `$TENA_HOME/examples/Example-Vehicle-v1/Notification-Publisher_1.1.1_JavaApplication/`, the `dependencies.xml` lists two jars – `Bindings_Middleware_XXX_JavaBinding_XXX_MW6.0.XXX.jar` and `\{{Example_Vehicle_1_JavaBinding_XXX_MW6.0.XXX.jar}\}`. (The specific versions for these dependencies may change over time.)

The structure of the generated example code is intended to be somewhat reusable as-is. For each example program, an `Application` class provides the code to initialize `Runtime`, `Execution` and `Session`, to publish and update, or subscribe, as appropriate, and perform shutdown and cleanup. Once the appropriate jars are added to the classpath, the `Application` class code should be usable from within an arbitrary application. Using the Java Binding for the TENA Middleware generally requires the use of fully scoped classes, because Java does not provide the equivalent of C++ using statements. Since, for example, the message sender classes of `Tanks` and `Notifications` are both named `MessageSender`, only fully scoped names `Example.Explosion.MessageSender` and `Example.Notification.MessageSender` can be used.

 The discussion below is based on the java example code delivered for the `Example.Vehicle,1` object model. This can be downloaded using the JavaBinding download link for the [Example.Vehicle,1 Object Model](#).

Using example publisher code to add Message publication to an existing application.

The `org.Example.Notification_Publisher.Application` class, from the example code included with the JavaBinding distribution for the `Example.Notification,1` object model, provides a simple API:

```
class Application {
    public Application();
    public int microsecondsPerIteration;
    public int numberofIterations;
    public TENA.Middleware.EndpointVector endpoints;

    public int run(TENA.Middleware.Configuration config);
    public void shutdown();
}
```

Using this from some existing application could be done following the example of the Main class used to start an Application.

```
import org.Example.Notification_Publisher.Application;
...
String[] mwConfigArgs = ... // Arguments to configure middleware. Alternatively set configuration parameters directly on the Configuration
TENA.Middleware.EndpointVector endpoints = new TENA.Middleware.EndpointVector();
endpoints.add(new TENA.Middleware.Endpoint("myhostname:50000"));

Application app = new Application();
app.endpoints = endpoints;
app.numberofIterations = 50;
app.microsecondsPerIteration = 1000000;

TENA.Middleware.Configuration cfg = new TENA.Middleware.Configuration( mwConfigArgs );

app.run(cfg);
```

This would use the generated publisher Application class, to create a single `Notification MessageSender`, and create and publish `Message` instances 50 times. The endpoint vector is a required parameter to define where to connect to the execution manager. In example code provided, these are passed in to applications using an `-emEndpoints` argument. See [Execution Manager](#) for more information.

Looking into the example publisher `Application` class in a little more detail:

```

public synchronized int run(TENA.Middleware.Configuration configuration)
    throws NetworkError, Timeout, Exception
{
    // Load any implementation libraries required
    // None required for object model Example-Vehicle-v1

    // Initialize the runtime, execution, and session
    runtime = TENA.Middleware.Runtime.init( configuration );
    execution = runtime.joinExecution( endpoints ); // Join the execution
    session = execution.createSession( "Example_Notification-Publisher_v1.1.1_Session" );

    // Initialize the MessageSender
    exampleNotificationSender = Example.Notification.MessageSender.create(
        session,
        commProps );

    exampleNotificationServantFactory = Example.Notification.ServantFactory.create(session);
    exampleNotificationServant = ExampleNotificationUtility.createServant( exampleNotificationServantFactory,
TENA.Middleware.CommunicationProperties.Reliable);

    // Loop to update
    for ( int i = 1; i <= numberIterations; ++i ) {
        session.evokeMultipleCallbacks( microsecondsPerIteration ); // In this case, just being used to delay
        ExampleNotificationUtility.send( exampleNotificationSender );
    }
}

```

Publisher Application.run

```

public synchronized int run(TENA.Middleware.Configuration configuration)
    throws NetworkError, Timeout, Exception
{
    // Load any implementation libraries required
    // None required for object model Example-Vehicle-v1

    // Initialize the runtime, execution, and session
    runtime = TENA.Middleware.Runtime.init( configuration );
    execution = runtime.joinExecution( endpoints ); // Join the execution
    session = execution.createSession( "Example_Notification-Publisher_v1.1.1_Session" );
    // Initialize the MessageSender
    exampleNotificationSender = Example.Notification.MessageSender.create(
        session,
        commProps );

    exampleNotificationServantFactory = Example.Notification.ServantFactory.create(session);
    exampleNotificationServant = ExampleNotificationUtility.createServant( exampleNotificationServantFactory,
TENA.Middleware.CommunicationProperties.Reliable);
    // Loop to update
    for ( int i = 1; i <= numberIterations; ++i ) {
        session.evokeMultipleCallbacks( microsecondsPerIteration ); // In this case, just being used to delay
        ExampleNotificationUtility.send( exampleNotificationSender );
    }
}

```

Using the classes generated in the example code directly would simply publish a new `Example.Notification.Message` with updated attributes, as defined in the `ExampleNotificationUtility` class.. To change the code to publish data from some external source, it would be necessary to modify `org.Example.Notification_Publisher.ExampleNotificationUtility`. The `ExampleNotificationUtility` send method creates and sends a new Message each time it is invoked. The example below omits some miscellaneous code in the example to print debugging text to the screen.

```

public class ExampleNotificationUtility {

    public static void send(
        Example.Notification.MessageSender msgSender )
        throws Timeout, NetworkError
    {
        String updateLabel = new String("Update # " + ++updateCount);
        Example.Notification.Message message =
            Example.Notification.Message.create(
                /* text */ updateLabel
            );
        msgSender.send( message );
    }
}

```

Note that this is only example code showing the steps required. Some key things to note:

1. This implementation of run() blocks until all the updates have been sent. A more realistic application will need to define appropriate architecture to allow updates to happen concurrently with other application activities.
2. It is important to hold a reference to the Runtime, Execution, and Session. If these are garbage collected (or if releaseReference is explicitly called) then any created MessageSenders are also destroyed. The example Application holds these as member variables.
3. There is a certain amount of communication overhead to create a MessageSender. Usually a MessageSender is created and held as long as an application may publish a particular type of message.
4. The Runtime instance is a singleton for an application. Multiple Executions or Sessions may be created depending on application requirements.

Using example subscriber code to add Message subscription to an existing application.

This is very similar to the discussion above, which described adding Message publication to an existing application. The org.Example.Notification_Subscriber.Application class, from the example code included with the JavaBinding distribution for the Example, Notification,1 object model, provides a simple API:

```

class Application {
    public Application();
    public int microsecondsPerIteration;
    public int numberofIterations;
    public TENA.Middleware.EndpointVector endpoints;

    public int run(TENA.Middleware.Configuration config);
    public void shutdown();
}

```

Using this from some existing application could be done following the example of the Main class used to start an Application.

```

import org.Example.Notification_Subscriber.Application;
...
String[] mwConfigArgs = ... // Arguments to configure middleware. Alternatively set configuration parameters directly on the Configuration
TENA.Middleware.EndpointVector endpoints = new TENA.Middleware.EndpointVector();
endpoints.add(new TENA.Middleware.Endpoint("myhostname:50000"));

Application app = new Application();
app.endpoints = endpoints;
app.numberofIterations = 50;
app.microsecondsPerIteration = 1000000;

TENA.Middleware.Configuration cfg = new TENA.Middleware.Configuration( mwConfigArgs );

app.run(cfg)

```

This would use the generated subscriber Application class, to subscribe to Notification Messages, processing updates for a total of 50 seconds. The endpoint vector is a required parameter to define where to connect to the execution manager. In example code provided, these are passed in to applications using an -emEndpoints argument. See [Execution Manager](#) for more information.

Looking into the example subscriber Application class in a little more detail:

```

public synchronized int run(TENA.Middleware.Configuration configuration)
    throws NetworkError, Timeout, Exception
{
    // Load any implementation libraries required
    // None required for object model Example-Vehicle-v1

    // Initialize the runtime, execution, and session

    runtime = TENA.Middleware.Runtime.init( configuration );
    execution = runtime.joinExecution( endpoints ); // Join the execution
    session = execution.createSession( "Example_Notification-Publisher_v1.1.1_Session" );

    // Set up Subscription, adding observers, and then subscribe
    Example.Notification.Subscription notificationSubscription = new Example.Notification.Subscription();
    notificationSubscription.addObserver( new ExampleNotificationPrintingObserver() );

    Example.Notification.Subscription.subscribe(session, notificationSubscription);

    // Loop for specified time, handling any delivered message events
    for ( int i = 1; i <= numberOfIterations; ++i ) {
        session.evokeMultipleCallbacks( microsecondsPerIteration );
    }
}

```

This code uses the example code class `ExampleNotificationPrintingObserver` to handle message events for `Notification` Messages.

```

public class ExampleNotificationPrintingObserver extends Example.Notification.AbstractObserver {
    public ExampleNotificationPrintingObserver() {
    }

    public void messageEvent( Example.Notification.ReceivedMessage receivedMessage )
    {
        // Use utility class to print the attributes of the message that was sent
        PrintUtility.display(receivedMessage.getSentMessage());
        // Use utility class to print the message metadata, such as when the message ws sent
        PrintUtility.display(receivedMessage.getMetadata());
    }
}

```

By passing resources to a suitable Observer constructor, the event methods could be modified to update displays, send data out through other APIs, log into databases, etc. Multiple Observers may be added to a Subscription, and they may be added or removed while the Subscription is in use.

Message Constructor

Message Constructor

The TENA Meta-Model supports definition of user specific Message constructors, which along with default constructors are used to create Messages. User specific Message constructors can be used to implement range checking of attribute values or perform other initialization operations when a Message is created. If user specific Message constructors are not defined in the object model, then default constructors will be defined for the Messages.

 For object models with constructors defined, Message and LocalClass implementations must be implemented using C++. This allows the use of the resulting libraries from any of the TENA Middleware supported languages.

Description

Release 6 of the TENA Middleware provides the ability to explicitly define constructors for both [Local Classes](#) and [Messages](#). Constructors are used to initialize attributes, either directly from parameters or by performing some computation, when an object is instantiated.

Within the TENA Meta-Model, Message constructors are optional. User defined Message constructors can be used when it is necessary to provide specific implementation behavior to check the Message attribute values or perform specific implementation behavior at construction time.

For example, if it is necessary to ensure that a Message attribute value is non-zero, a constructor can be defined that includes the attribute value (as well as other attributes, as appropriate) to ensure that the particular attribute value is "range checked" at construction time. This attribute can be defined as `readonly` in the object model definition to prevent the generation of a `set` method that would allow the attribute value to be set after construction. Another use case for user-defined constructors would be when there are dependencies between multiple attribute values that can only be effectively checked together – using a constructor with these dependent attribute values to perform this checking and marking these attributes as `readonly` would satisfy this need.

User defined Message constructors can also be necessary to perform implementation specific initialization operations. For example, a Message implementation that was required to connect to an external database may need to establish that database connection at construction time. Note that in this particular example, the `LocalClassFactory` could be used with static methods for the database operations if that provides a better design for the needs of the implementation.

As mentioned, Message constructors are optional and if no constructors are specified in the object model, a default constructor with no arguments and a constructor with an argument for each non-optional attribute are automatically generated. Otherwise, only the user specified constructors are available.

For the Message constructors, whether user-defined or the default generated, the API (application programming interface) uses static `create` functions that match the argument syntax defined by the constructors. These `create` functions are defined in the namespace defined by the object model Package structure.

A simple auto-generated `create` function for the `Example::Notification` Message that does not define any user specified constructors is shown in the code fragment below. This `Example::Notification` Message has a single string attribute whose value can be provided in the `create` invocation as shown.

```
// Notification TDL syntax
message Notification
{
    string text;
};

// Notification message creation
Example::Notification::MessagePtr pMessage(
    Example::Notification::create( "This is a test notification." ) );
```

Message instances are held within a reference counted pointer (so called "smart" pointer), as indicated by the type suffix `ptr`. This is because multiple references to the same Message instance may exist within an application's process space (including middleware references) and only when all references are released will the underlying Message instance be destroyed and memory released.

An example of a Message that has a user defined constructor is illustrated in the code fragment below. The constructor defines a `Location` to be specified, which is used in the `create` invocation. `Location` is Local Class that is defined within the `Example-Vehicle` object model. As illustrated, Local Classes follow a similar `create` construction process as Messages.

```

// Explosion TDL syntax
message Explosion : extends Notification
{
    Explosion( Location reportedlocation );
    ...
};

// Explosion message creation
TENA::float64 xLocation( 2.0 );
TENA::float64 yLocation( 3.0 );
Example::Location::LocalClassPtr pLocation(
    Example::Location::create( xLocation, yLocation ) );

Example::Explosion::MessagePtr pMessage(
    Example::Explosion::create( pLocation ) );

```

Once a Message instance is created there are no restrictions on how many times the Message instance can be sent. The Message instance will exist until all pointers referencing it are released. This makes it possible for an application to cache, modify, and resend a Message instance, if appropriate. Within the subscriber application, the received message is a copy of the original sent message and there are no direct connection to the original sent message.

Range Checking

It may be useful for constructors to range-check attributes when a Message is initialized. In order to properly range-check an attribute at construction, the local class needs to be modified as follows:

1. Mark each attribute to be range-checked as `readonly`. This causes the OMC (Object Model Compiler) to skip generation of a `set_AttributeName` method to set the attribute directly.
2. Add an explicit constructor that assigns the values for all of the attributes that need to be range-checked.

Additionally, if an attribute requiring range-checking needs to be able to be changed after construction, you should add a `set_AttributeName` method to the local class or message, and implement it with the appropriate range checking logic.

Constructor Implementation Development

When designing an object model with Messages that declare constructors, an implementation for these constructors must be developed to be used applications using the object model. The TENA Object Model Compiler (OMC) will automatically generate the source code for a "stub" implementation of any user defined Message constructors (as well as any Message methods). This generated implementation code is associated with a shared implementation library that contains all of the Message and Message implementation code for the particular object model.

The generated shared implementation library is installed in the `TENA_HOME/TENA_VERSION/src/OM_NAME/OM_NAME-SharedLocalImpl-v1/myImpl` directory. Users can modify the "SharedLocalImpl" name and the "1" version if desired. For each Message or Message that contains a constructor and/or local method, a sub-directory will exist based on the Message or Message name. Under this directory will be a `SharedLocalImpl/LocalMethodsImpl.cpp` file that contains the "stub" code for the Message constructor implementation.

Object model implementation developers can begin with the automatically generated shared implementation library code and modify to support the specific implementation needs. Details on the generated Message constructor "stub" code is described in the API section below.

Once the object model implementation code has been built and tested, the implementation developer can package up the necessary files to share with other users of that object model. The [sharing object model implementations documentation page](#) provides additional information on sharing the developed implementation code.

Additionally, applications that use object models that require implementations need to register the appropriate implementation (since multiple object model implementations are possible). The [Message implementation registration documentation page](#) provides information on the registration process.

C++ API Reference and Code Examples

When an application is just **using** a Message with defined constructors, the necessary API of concern is the default `create` function(s) discussed above. This function allows an application to create a particular Message instance.

If it is necessary to **develop** the Message constructor implementation, then the developer needs to modify the auto-generated shared implementation library code associated with the particular object model.

As an example, consider the `Example-Tank` object model `Explosion` Message type (object model TDL shown above). The generated code that needs to be modified for the constructor implementation is defined in the source code file `src/Example-Tank-v1/Example-Tank-v1-SharedLocalImpl-v1/myImpl/Example/Explosion/SharedLocalImpl/LocalMethodsImpl.cpp`. Note that the version number of the `Example-Tank` object model is expected to change, so the "-v1" in this directory location may be different.

This auto-generated `LocalMethodsImpl.cpp` "stub" implementation code is shown below. The code will build and run without modification, but the behavior uses arbitrary values and default behavior that needs to be replaced with the necessary behavior needed by the implementation developer.

```

Example::Explosion::SharedLocalImpl::
LocalMethodsImpl::
LocalMethodsImpl( Example::Explosion::State & rState,
    Example::Location::ImmutableLocalClassPtr const & /* reportedlocation */ )
:
rState_( rState )
{
    // This constructor is invoked by the
    // Example::Explosion::SharedLocalImpl::LocalMethodsFactoryImpl
    // as a result of a call to
    // Example::Explosion::create(
    //     reportedlocation )
    //
    // This is a "stub" example implementation. It is provided as a starting
    // point. This "stub" implementation should be replaced with the real
    // implementation.

    // Initialize all the non-optional attributes of
    // Example::Explosion
    this->rState_.set_text( TENA::Middleware::ArbitraryValue< std::string >() );

    /// \todo In a real application, instances of \c ArbitraryValue<T>
    /// should be replaced with values that are sensible for the
    /// particular application.
    Example::Location::LocalClassPtr affectedArea_0(
        Example::Location::create(
            /* xInMeters */ TENA::Middleware::ArbitraryValue< TENA::float64 >(),
            /* yInMeters */ TENA::Middleware::ArbitraryValue< TENA::float64 >() ) );
    Example::AreaOfEffect::LocalClassPtr affectedArea(
        Example::AreaOfEffect::create(
            /* centerOfArea */ affectedArea_0,
            /* radius */ TENA::Middleware::ArbitraryValue< TENA::float64 >() ) );
    this->rState_.set_affectedArea( affectedArea );
}

```

Specifically, the implementation developer should remove all occurrences of `ArbitraryValue` and set the appropriate `Message` attribute values using the constructor arguments. Note that the name of the constructor arguments, `reportedLocation` in this example, are commented out to avoid compiler warnings associated with unused arguments. Implementation developers should remove these comment markers and use the constructor arguments to set the attribute values, contained in the `State` object, appropriately.

As an example, the "stub" implementation can be changed as illustrated below. The `text` attribute value is set to "BOOM!" and the `affectedArea` is set to use the `reportedLocation` variable that the user of this `Message` is required to pass into the constructor.

```

LocalMethodsImpl( Example::Explosion::State & rState,
    Example::Location::ImmutableLocalClassPtr const & reportedLocation )
: rState_( rState )
{
    this->rState_.set_text( "BOOM!" );

    Example::AreaOfEffect::LocalClassPtr affectedArea(
        Example::AreaOfEffect::create(
            /* centerOfArea */ reportedLocation,
            /* radius */ 10.0 ) );
    this->rState_.set_affectedArea( affectedArea );
}

```

The `radius` is not (at least with the "v1" version of `Example-Tank` object model) provided to the `Explosion` constructor, even though it is used to define the `AreaOfEffect` attribute value. This is probably a modelling error, but had the `radius` been passed into the constructor, a user specific constructor implementation would probably want to check that the `radius` value is non-negative.

In normal range checking operations, the implementation code should throw an exception if an invalid value was provided. Typically, a `std::invalid_argument` exception is used with a text description of the error along with the provided value.

Note that within the automatically generated `LocalMethodsImpl` classes that an additional constructor is defined to be used when the middleware needs to create a `Message` from information sent across the network (i.e., middleware receives a message instance from a publishing application). A destructor is also automatically generated for the `LocalMethodsImpl` classes. Neither the network constructor or destructor will typically need to be specialized by the implementation developer and the automatically generated code will suffice.

Java API Reference and Code Examples

When an application is just **using** a Message with defined constructors, the necessary API of concern is the default `create` function(s) discussed earlier. This function allows an application to create a particular Message instance.

If it is necessary to **develop** the Message constructor implementation, then the developer needs to modify the auto-generated shared implementation library code associated with the particular object model. **⚠ For the current TENA Middleware, all object model library implementation code must be implemented in C++.** Because a single Message implementation library must be shared among all applications, developed in any supported language, that publish or subscribe to a given Message type, the library must be implemented in a language that is usable for any language supported by the TENA Middleware. There is no efficient way to access a Java implementation or .Net implementation from a C++ program. In contrast, most of the access for the Java and .Net bindings is already accomplished by wrapping existing C++ libraries for use by Java and .Net.

In order to develop an implementation for Messages in an object model, a developer would have to download a C++ Object Model distribution, develop the implementation code using as described above. Then simply use `java.lang.System.loadLibrary` to load the resulting library, before creating an Execution in the application.

Message ID

Message ID

Every sent Message is assigned a corresponding ID composed of the `MessageSenderID` and an integer count that is unique to the send invocation. The `MessageID` is returned to a publisher from the `MessageSender::send` method. A subscriber can access this ID by calling the `getID` method of a `ReceivedMessage` instance. The ID is also available from Message Metadata.

Description

All TENA [Middleware IDs](#) derive from a template ID class that establishes a common interface. This template class defines an underlying value representation. Additional information on the capabilities can be found on the [Middleware IDs documentation page](#).

For `MessageIDs`, the underlying ID value type is a `MessageIDType` class containing a `MessageSenderID` and an unsigned integer value. The `MessageSenderID` identifies the `MessageSender` that sent the Message and the integer is an integer counter value generated by the sender that uniquely identifies each message send. The ID class is designed to hide the underlying data values and provide a common interface for copying, comparing, and displaying IDs. If these values are desired for a `MessageID`, an application can call the Message Metadata methods `getSenderId` and `getMessageCount`.

Message Implementation Registration

Message Implementation Registration

Messages can be created by an application or by the middleware in response to receiving an SDO update or a message from another application. When a Message has a user defined constructor or method, a `LocalMethodsFactory` is needed so that whenever the Message is created the appropriate implementation code is utilized. The factory must be registered with the middleware, which is automatically done with the automatically generated example applications by simply including the appropriate implementation header file and linking with the associated library. Application developers are allowed the freedom to customize this registration mechanism for explicit control if necessary.

⚠ All Message and LocalClass implementations must be implemented in C++. This allows the use of the resulting libraries from any of the MW supported languages.

Description

Message Implementation Development

When designing an object model with Messages that declare constructors or methods, an implementation for these constructors must be developed to be used applications using this Message type. The TENA Object Model Compiler (OMC) will automatically generate the source code for a "stub" implementation of any user defined Message constructors and methods. This generated implementation code is associated with a shared implementation library that contains all of the Message and Local Class implementation code for the particular object model.

The generated shared implementation library is installed in the `TENA_HOME/TENA_VERSION/src/OM_NAME/OM_NAME-SharedLocalImpl-v1/myImpl` directory. Users should modify the "SharedLocalImpl" name if a custom implementation is being developed. Changing the name of the implementation requires finding all occurrences of SharedLocalImpl in the generated source example code, and replacing it with the desired name. This includes renaming any directories named SharedLocalImpl. Once done, the resulting source code will generate an implementation with a different name.

For each Message or Local Class within the object model that contains a constructor and/or local method, a `SharedLocalImpl` sub-directory will exist followed by a directory name based on the Message or Local Class name (e.g., `SharedlocalImpl/Example_Explosion`). Under this directory will be a `LocalMethodsImpl.cpp` file that contains the "stub" code for the Local Class constructor implementation.

Object model implementation developers can begin with the automatically generated shared implementation library code and modify to support the specific implementation needs. In the case of TENA standard object models, and perhaps other object models in the future, the implementations have already been developed and are included with the object model packages, so applications only need to register the object model implementations needed for their application.

Message Implementation Registration

A Message has two parts: `State` and `LocalMethods`. If a Message type specifies (in the object model definition) neither constructors nor methods, a `LocalMethods` implementation (and therefore `LocalMethodsFactory`) is not necessary. If a constructor or method is declared, an application must supply a `LocalMethods` and `LocalMethodsFactory` implementation that satisfies the interface specified in the definition header file.

ℹ — Note that the Object Model Compiler will create default constructors for Messages in which no user defined constructors are defined. In this situation, the implementation code for the default constructors is included with the object model definition code and a `LocalMethods` and `LocalMethodsFactory` is not needed. Only user defined constructors require a separate implementation.

The `LocalMethodsFactory` is called by the Middleware to create the `LocalMethods` implementation objects for any Messages that are created. Messages can be created explicitly by the application, or they can be created by the middleware in response to receiving a message from some other publishing application. Therefore, a `LocalMethodsFactory` implementation must be registered for each Message type (to be used by the application) that has a user defined constructor or method before an application can join the execution.

When does the application pass the `LocalMethodsFactoryImpl` object to the Middleware? It needs to be done early on, before any Message can be created. Therefore, the TENA Middleware has this rule: If an application (potentially) uses a message with constructors or methods defined, a `LocalMethodsFactory` must be registered in the application before that application calls `TENA::Middleware::Runtime::joinExecution`.

How does the application pass the `LocalMethodsFactoryImpl` object to the Middleware (i.e., How does the application register the local class methods factory implementation with the Middleware)? The technical answer: The application calls the `registerFactory()` method on the `TENA::Middleware::LocalClassMethodsFactoryRegistry` as follows:

```
boost::shared_ptr< Example::Explosion::LocalMethodsFactory > factory(
    new SharedLocalImpl::Example_Explosion::LocalMethodsFactoryImpl );

TENA::Middleware::LocalClassMethodsFactoryRegistry::
    getInstance().registerFactory( factory );
```

But, it would be somewhat onerous for the application developer to have to write this code manually for every single Local Class. To assist with this, the automatically generated example application code provides the `RegisterFactory` in the `LocalMethodsFactoryImpl.cpp` source file. Not only does this file implement the `LocalMethodsFactory`, but it also declares an instance of `RegisterFactory` that performs the automatic registration of the implementation **during global initialization**. Through this mechanism, registration occurs by including a single header file (e.g., `Example/Explosion/Impl.h`) and linking the associated implementation library (e.g., `libExample-Tank-v3-SharedLocalImpl-v2-xp-vc80-v6.0.0.lib`).

Working Around Defective Linkers

Some compilers and linkers, such as Microsoft Visual C++, have a design flaw by which they fail to link code that has no references to it. (This is incorrect behavior because that code, despite having no references to it, may be invoked at static initialization time, and may do important things. If the linker, in a well-intended attempt to be efficient, omits that code, the application can fail to do things that it is supposed to do.) Because of this defect, the static initialization code described above that performs local methods factory registration fails to get called with certain linkers unless extra steps are taken to deceive the defective linker into performing correctly.

To work around this linker design flaw, the automatically generated example application code defines a dummy "forceLinkage" variable at the bottom of the `LocalMethodsFactoryImpl` definition file, alongside the code that performs the registration via static initialization. The generated code also contains an `Impl.h` header file (one for each Local Class or Message) that references the dummy "forceLinkage" variable. That reference, as the name implies, deceives the defective linker into linking the local methods factory registration code into the executable so that it gets invoked upon static initialization. To simplify even further, the example application code uses a `localClassImpls.h` header file that #includes all of the `Impl.h` header files for all the Local Classes and Messages in the object model. The application has only to #include that one `localClassImpls.h` header file to cause all of the local class methods factories to be registered automatically with the TENA Middleware.

What if more fine-grained control is needed?

The `exampleApplication` plugin used by the Object Model Compiler generates example applications that assume that all the local methods in a given object model are implemented in a single library. If, for some reason, the local methods implementation for individual Local Classes and/or Messages needs to be separated into multiple libraries, that is very possible. In fact, as described above, the `localClassImpls.h` header file is simply an aggregation of header files for each Local Class or Message with a method or constructor defined. For example, the `Example-Tank-v3-SharedLocalImpl localClassImpls.h` header file aggregates these "fine-grained" header files:

- `Example/AreaOfEffect/Impl.h`
- `Example/ComplexAreaOfEffect/Impl.h`
- `Example/Explosion/Impl.h`
- `Example/HomeStation/Impl.h`

If multiple implementation libraries exist, the application build files just need to be adjusted to include the appropriate fine-grained `Impl.h` header files and link with the necessary implementation libraries.

Explicit LocalMethodsFactory Registration

The `RegisterFactory` class is intended to be used to support automatic registration during static initialization time. If it is necessary to support applications being able to explicitly register the appropriate local methods factories, the implementation developer for the Message implementation can disable the static `RegisterFactory` mechanism.

In this situation, the entire `RegisterFactory` class, as well as the static creation, should be removed from the `LocalMethodsFactoryImpl.cpp` file. Since the implementation will no longer support automatic registration of the `LocalClassFactory`, applications attempting to use this particular local methods implementation will need to add the following code **before** the application attempts to call `TENA::Middleware::Runtime::joinExecution`. A code example, shown below, is identical to the code contained in the corresponding `RegisterFactory` class.

```
boost::shared_ptr< Example::Explosion::LocalMethodsFactory > factory(
    SharedLocalImpl::Example_Explosion::LocalMethodsFactoryImpl );
TENA::Middleware::LocalClassMethodsFactoryRegistry::
    getInstance().registerFactory( factory );
```

An expected use case for explicit local methods factory registration is when the application operator needs to select a particular local methods implementation at runtime. In this situation, the registration can't be automatically performed at static initialization time, unless the application was able to dynamically load the implementation library.

C++ API Reference and Code Examples

As mentioned above, application developers that are merely attempting to use a Message that requires an implementation (i.e., contains user defined constructor or method), the default registration behavior is to ensure that the appropriate `Impl.h` header files are included by the application and the build file links the appropriate implementation libraries. This mechanism relies on the static initialization registration mechanism.

The `Impl.h` header files are designed to be implementation independent, so if it is necessary for an application to change to a different local method implementation, only the build file needs to be updated to link the correct implementation library. In other words, there is no need to change application source code to have the application change from using one local method implementation to another implementation (provided that the implementations are using the same object model definition and have not customized the implementation to require access to application data — see [Accessing Application Data from a Message documentation page](#)).

For developers that need to actually provide object model implementations, the automatically generated implementation and registration code will satisfy most development needs. The generated "stub" code for the Message constructors and methods is installed in the SharedLocalImpl library directory (e.g., src/Example-Tank-v3/Example-Tank-v3-SharedLocalImpl-v2). The relevant implementation "stub" code is contained in a sub-directory structure based on the particular object mode type, and the implementation code that requires developer modification is named LocalMethodsImpl.

For example, the implementation code for the Explosion Message from the Example-Tank object model is shown below for the file src/Example-Tank-v3/Example-Tank-v3-SharedLocalImpl-v2/myImpl/SharedLocalImpl/Example_Explosion/LocalMethodsImpl.cpp. These LocalMethodsImpl classes contain all of the constructors and methods that need to be implemented for the particular Message type. In the case of the example Explosion Message, there is a user defined constructor and a single method named distanceFromCenterOfExplosionInMeters.

An implementation developer would replace all of the ArbitraryValue code with desired values and behavior necessary for the constructor and method implementations. As noted, a constructor for the middleware to create a Message when an instance is received over the network is defined, as well as a destructor, but in most cases the implementation developer does not need to specialize the automatically generated code for these operations.

```
////////// -*- C++ -*-  
/*!  
 * \file SharedLocalImpl/Example_Explosion/LocalMethodsImpl.cpp  
 *  
 * \brief Contains the definition of methods of  
 * SharedLocalImpl::Example_Explosion::LocalMethodsImpl.  
 *  
 * This file contains the definition of the methods of the  
 * SharedLocalImpl::Example_Explosion::LocalMethodsImpl class.  
 */  
  
#include "LocalMethodsImpl.h"  
#include <TENA/Middleware/ArbitraryValue.h>  
#include <Example/AreaOfEffect.h>  
#include <Example/Location.h>  
#include <iostream>  
#include <stdexcept>  
#include <string>  
  
//////////  
SharedLocalImpl::Example_Explosion:::  
LocalMethodsImpl:::  
LocalMethodsImpl( Example::Explosion::State & rState,  
    Example::Location::ImmutableLocalClassPtr const & /* reportedLocation */, TENA::float64 /* reportedRadius */ )  
:  
rState_( rState )  
{  
    // This constructor is invoked by the  
    // SharedLocalImpl::Example_Explosion::LocalMethodsFactoryImpl  
    // as a result of a call to  
    // Example::Explosion::create(  
    //     reportedLocation, reportedRadius )  
    //  
    // This is a "stub" example implementation. It is provided as a starting  
    // point. This "stub" implementation should be replaced with the real  
    // implementation.  
  
    // Initialize all the non-optional attributes of  
    // Example::Explosion  
    this->rState_.set_text( TENA::Middleware::ArbitraryValue< std::string >() );  
  
    /// \todo In a real application, instances of \c ArbitraryValue<T>  
    /// should be replaced with values that are sensible for the  
    /// particular application.  
    Example::Location::LocalClassPtr areaOfEffect_0(  
        Example::Location::create(  
            /* xInMeters */ TENA::Middleware::ArbitraryValue< TENA::float64 >(),  
            /* yInMeters */ TENA::Middleware::ArbitraryValue< TENA::float64 >() ) );  
    Example::AreaOfEffect::LocalClassPtr areaOfEffect(  
        Example::AreaOfEffect::create(  
            /* centerOfArea */ areaOfEffect_0,  
            /* radius */ TENA::Middleware::ArbitraryValue< TENA::float64 >() ) );  
    this->rState_.set_areaOfEffect( areaOfEffect );  
}  
  
//////////  
SharedLocalImpl::Example_Explosion:::
```

```

LocalMethodsImpl::
LocalMethodsImpl( Example::Explosion::State & rState,
    Example::Explosion::LocalMethodsFactory::StateAlreadyInitializedTag )
:
rState_( rState ) // Holds the values that were sent over the network
{
    // This constructor is invoked by the
    // SharedLocalImpl::Example_Explosion::LocalMethodsFactoryImpl
    // when the Middleware receives an instance of
    // Example::Explosion over the network.
    //
    // Although it is possible, modifying the contents of the rState_ in this
    // constructor is rarely wise and probably never necessary.
    //
    // Typically, this "do-nothing" example implementation is all that is needed.
    // However, the implementation of this constructor can be customized if a
    // particular implementation requires it for some reason.
}

///////////////////////////////
SharedLocalImpl::Example_Explosion::
LocalMethodsImpl::
~LocalMethodsImpl()
{
    // Typically, this "do-nothing" example implementation is all that is needed.
}

///////////////////////////////
TENA::float64
SharedLocalImpl::Example_Explosion::
LocalMethodsImpl::
distanceFromCenterOfExplosionInMeters( Example::Location::ImmutableLocalClassPtr const & here ) const
{
    // This is just an auto-generated "stub" example implementation. It is
    // provided merely as a convenient starting point. This "stub" implementation
    // should be replaced with the real implementation.

    return TENA::Middleware::ArbitraryValue< TENA::float64 >();
}

```

The automatically generated example applications for Message constructor and method implementations (specifically the `SharedLocalImpl` library) perform implementation registration implicitly, when the library containing the constructors and/or methods is loaded. Below is a concrete example for `{Example::Explosion}` Message from the `Example-Tank` object model.

```

/////////////////////////////-*- C++ -*-/
/*!
* \file SharedLocalImpl/Example_Explosion/LocalMethodsFactoryImpl.cpp
*
* \brief Contains the definition of methods of
* SharedLocalImpl::Example_Explosion::LocalMethodsFactoryImpl.
*
* This file contains the definition of the methods of the
* SharedLocalImpl::Example_Explosion::LocalMethodsFactoryImpl class.
*/
#include "LocalMethodsFactoryImpl.h"
#include "LocalMethodsImpl.h"
#include <Example/AreaOfEffect.h>
#include <Example/Location.h>
#include <TENA/Middleware/ArbitraryValue.h>

/////////////////////////////
std::string const
SharedLocalImpl::Example_Explosion::
LocalMethodsFactoryImpl::
getDescription() const
{
    // A unique description string should be used for each different version of
    // every distinct C++ implementation for the methods and constructors of the

```

```

// Example::Explosion local class
// (which is defined in Example-Tank-v3.tdl).
//
// This string is used by the Middleware to provide the (optional) local class
// implementation consistency checking feature.
//
// When creating a new implementation (or new implementation version) be sure
// to change this string!

return TENA::Middleware::ArbitraryValue< std::string >(
    "Example-Tank-v3-SharedLocalImpl-v2" );
}

///////////////////////////////
SharedLocalImpl::Example_Explosion::
LocalMethodsFactoryImpl::
LocalMethodsFactoryImpl()
{
    // Typically, this "do-nothing" example implementation is all that is needed.
}

///////////////////////////////
SharedLocalImpl::Example_Explosion::
LocalMethodsFactoryImpl::
~LocalMethodsFactoryImpl()
{
    // Typically, this "do-nothing" example implementation is all that is needed.
}

///////////////////////////////
Example::Explosion::LocalMethodsPtr const
SharedLocalImpl::Example_Explosion::
LocalMethodsFactoryImpl::
create( Example::Explosion::State & rState,
        Example::Location::ImmutableLocalClassPtr const & reportedLocation, TENA::float64 reportedRadius )
{
    // This method is invoked by the Middleware as a direct result of a call to
    // Example::Explosion::create(
    //     reportedLocation, reportedRadius )

    return Example::Explosion::LocalMethodsPtr(
        new LocalMethodsImpl( rState,
            reportedLocation, reportedRadius ) );
}

///////////////////////////////
Example::Explosion::LocalMethodsPtr const
SharedLocalImpl::Example_Explosion::
LocalMethodsFactoryImpl::
create( Example::Explosion::State & rState,
        StateAlreadyInitializedTag tag )
{
    // This method is invoked by the Middleware when an instance of
    // Example::Explosion is received over the network.

    return Example::Explosion::LocalMethodsPtr(
        new LocalMethodsImpl( rState, tag ) );
}

///////////////////////////////
#include <TENA/Middleware/LocalClassMethodsFactoryRegistry.h>
#include <TENA/Middleware/staticInitializationFailure.h>
#include <sstream>

///////////////////////////////
/// An anonymous namespace to keep its contents "hidden" inside this file.
namespace
{
    /*!\class RegisterFactory LocalMethodsFactoryImpl.cpp \
    myImpl/SharedLocalImpl/Example_Explosion/LocalMethodsFactoryImpl.cpp
    */

```

```

* \brief Registers this local class factory with the Middleware
*
* Every local class (or message) with one or more methods or constructors must
* have a corresponding local class factory that is registered with the
* Middleware. This registration must be completed *before* the call to
* TENA::Middleware::Runtime::joinExecution(). This class exists solely as a
* means to perform this registration.
*
* The idea is to declare an object instance of this class (named
* "registerFactory", below) that will, in turn, call the constructor, which
* then performs the registration.
*
* The result is that the act of loading the library containing this code
* causes the global registerFactory object (below) to be constructed.
* Constructing the registerFactory object, in turn, registers this
* LocalMethodsFactoryImpl.
*/
class RegisterFactory
{
public:
    RegisterFactory()
    {
        try
        {
            boost::shared_ptr< Example::Explosion::LocalMethodsFactory > factory(
                new SharedLocalImpl::Example_Explosion::LocalMethodsFactoryImpl );

            TENA::Middleware::LocalClassMethodsFactoryRegistry::
                getInstance().registerFactory( factory );
        }
        catch ( std::exception const & ex )
        {
            std::ostringstream out;
            out << __FILE__ << ':' << __LINE__ << ":" << ex.what() << std::endl;
            TENA::Middleware::reportStaticInitializationFailure( out.str() );
        }
    }
}; // End of class RegisterFactory

// Create an object, thereby invoking the RegisterFactory constructor
RegisterFactory registerFactory;
} // End of namespace

// As another wrinkle, on some platforms, e.g., OS X and Windows, a library
// will be completely ignored if no explicit references to something inside of
// it is made. Thus, here we create a "dummy" symbol that can be used to force
// the library to be linked.
//
// For more information, see:
//   Example/Explosion/Impl.h

#include <include/Example/Explosion/Impl.h>
int Example_Tank_v3_SharedLocalImpl_v2_ExampleExplosion_forceLinkage = 0;

```

Implementation developers would only need to modify the registration code in case some alternative registration mechanism was needed, such as a dynamic implementation configuration mechanism that allowed an application operator to select the particular object model implementations to be used, or if the Message implementation required access to application date. The [Accessing Application Data from a Message documentation page](#) provides information on these procedures.

Message Inheritance and Polymorphism

Message Inheritance and Polymorphism

The TENA Meta-Model supports the concept of inheritance for Messages. By "deriving" from a base Message type, all of the base class attributes and methods are available for use by the derived class. This is useful for extending the base Message without breaking existing application code. It also allows several similar but distinct types (e.g., GunFire, MissileFire, and BombDrop) to share common attributes and behavior (e.g., WeaponRelease). Additionally, the primary purpose of inheritance in software programming, polymorphism, also applies in the design of TENA object models using inheritance.

Description

A generalized discussion of inheritance within the TENA Meta-Model can be found on the [Meta-Model inheritance documentation page](#). This page describes some important limitations of inheritance as implemented for TENA. In summary:

- Multiple inheritance (inheriting from multiple classes) is not supported.
- Derived classes cannot add attributes with the same name or methods with the same signature as a base class.
- Automatic delegation of method implementation to base classes is not performed.

Another way of stating the last issue is that derived classes must supply implementation code for all methods, including inherited ones. The derived class implementation is free to delegate any operations to its base class.

Messages support the concept of substitutability, meaning a derived type can be used where a base type is expected. The smart pointer mechanism employed throughout the TENA Middleware API (application programming interface) allows for automatic conversion from derived to base type when required. In cases where a derived type is used instead of a base type, the derived type implementation will be used unless the application was not linked against the derived object model definition. If no derived implementation is available, the Message will be "sliced" to exhibit only the base class behavior. This is a feature of [OM subsetting](#).

It is also possible to perform a safe downcast from a *base* Message pointer to a *derived* Message pointer using the `TENA::Middleware::dynamicCast` function. This function will throw `std::bad_cast` if the cast was not successful. A code example is given in the API section below.

Usage Considerations

A relevant change for TENA Middleware Release 6 involves [discovery order](#), or receive order with respect to messages. This potentially affects the way an application handles Message receipt if an application subscribes to multiple levels of an inheritance hierarchy.

C++ API Reference and Code Examples

An example of using dynamic casting for a received Message is illustrated in the following code fragments. First, a publishing application sends an `Example::Explosion` Message.

```
Example::Explosion::MessageSenderPtr p_MessageSender(
    Example::Explosion::MessageSender::create( pSession, communicationProperties ) );

Example::Location::LocalClassPtr pLocation( Example::Location::create( 10, 10 ) );

Example::Explosion::MessagePtr pMessage(
    Example::Explosion::create( pLocation ) );

pMessageSender->send( pMessage );
```

Release 6.0.2 Update — An alternative `create` method was added with release 6.0.2 of the middleware in which the `SessionPtr` argument is used, versus the `Session` reference (i.e., `*pSession`) that was used in the previous releases. Users are encouraged to use the `SessionPtr` interface. See [MW-4286](#) for more details.

Next, the subscriber receives a `Notification` Message and attempts to cast it to an `Explosion` Message type.

```

Example::Notification::SubscriptionPtr
pExampleNotificationSubscription( new Example::Notification::Subscription );

Example::Tank::AbstractObserverPtr pCastingObserver(
    new MyApplication::Example_Notification::CastingObserver );

pExampleNotificationSubscription->addObserver( pCastingObserver );

Example::Notification::subscribe( pSession, pExampleNotificationSubscription );
...

void
MyApplication::Example_Notification::
CastingObserver::
messageEvent( Example::Notification::ReceivedMessagePtr const & pReceivedMessage )
{
    try {
        Example::Explosion::MessagePtr pExplosion =
            TENA::Middleware::Utils::dynamicCast<Example::Explosion::MessagePtr>(
                pReceivedMessage->getSentMessage() );

        Example::Location::LocalClassPtr pLocation( Example::Location::create( 0, 0 ) );
        pExplosion->distanceFromCenterOfExplosionInMeters( pLocation );
    }
    catch ( std::bad_cast )
    {
        std::cout << "ReceivedMessage not an Example::Explosion" << std::endl;
    }
}

```

Release 6.0.2 Update — An alternative `subscribe` method was added with release 6.0.2 of the middleware in which the `SessionPtr` argument is used, versus the `Session` reference (i.e., `*pSession`) that was used in the previous releases. Users are encouraged to use the `SessionPtr` interface. See [MW-4286](#) for more details.

Since the Message type `Explosion` does inherit from type `Notification`, this `dynamicCast` will work correctly.

Java API Reference and Code Examples

Subscribing to a base type of a message, and receiving a instance of derived message is illustrated in the following code fragments. First, a publishing application sends an `Example.Explosion.Message`.

```

Example.Explosion.MessageSender messageSender = Example.Explosion.MessageSender.create( session,
communicationProperties );

Example.Location.LocalClass location = Example.Location.create( 10.0d, 10.0d );

Example.Explosion.Message message = Example.Explosion::create( "Boom!", location, 1.0d );

messageSender.send( message );

```

Next, the subscriber receives a `Notification` Message. Because an `Explosion` is-a `Notification`, the subscriber receives the `Notification`.

```

Example.Notification.Subscription notificationSubscription = new Example.Notification.Subscription();

Example.Tank.AbstractObserver myObserver = new org.Example.Notification_Subscriber.PrintingObserver();

notificationSubscription.addObserver( myObserver );

Example.Notification.Subscription.subscribe( session, notificationSubscription );
...

```

```

package org.Example.Notification_Subscriber;
class PrintingObserver extends Example.Notification.AbstractObserver {
    public void messageEvent( Example.Notification.ReceivedMessage receivedMessage ) {
        System.out.println( receivedMessage.getSentMessage().get_text() );
    }
}

```

⚠ As of release 1.0.1 of the Java Binding, there is no equivalent of the C++ TENA::Middleware::dynamicCast ability to *downcast* a Message to a derived type. Since the subscribing application is *aware* of the derived type (in order to be able to downcast) an equivalent approach would be to simply subscribe to the (one or more) derived types in addition to or instead of subscribing to the base type. There is a case to add the ability to construct a derived Message or LocalClass from a base type when a derived type has been sent. See [Java-276](#)

Message Sender

Message Sender

The purpose of the `MessageSender` class is to send Messages constructed by the application to interested subscribing applications. A `MessageSender` is type specific and only publishes a particular Message type. When an application creates a `MessageSender` it enables the middleware to establish the necessary communication infrastructure needed to send messages to interested subscribing applications. The communication properties (e.g., reliable versus best effort transport) are specified through the `MessageSender` constructor arguments.

Description

A `MessageSender` performs the publishing activities necessary for an application that intends to send TENA Messages. By instantiating a type specific `MessageSender`, an application declares its intention to send Messages of that type. The creation of a `MessageSender` class is used by the TENA Middleware to establish the communication infrastructure used to connect the message publisher with interested message subscribers.

 The `MessageSender` construction process has overhead to communicate with any interested applications. Applications should create and hold needed `MessageSenders` versus constructing a new `MessageSender` each time a message needs to be sent.

Two static methods named `create` are included in the `MessageSender` class generated for each Message type which can be used to create a `MessageSender` object. When created, the `MessageSender` must be associated with a `Session` object (obtained when an application joins an Execution). The `CommunicationProperties` argument specifies whether reliable or best-effort communication is used to deliver the messages. The default communication behavior is to use reliable transport.

An optional `Tag` can be specified to limit what Message subscriptions would match Messages sent by this `MessageSender`. (This is part of the feature known as [Advanced Filtering](#).)

Applications may need to create multiple `MessageSender` objects for the same Message type in order to support different communication types or `Tags`.

In C++, the static `create` methods return a reference counted "managed" pointer to the `MessageSender`. This pointer mechanism allows the application (and middleware) to hold onto multiple references to the same object. The `MessageSender` will be destroyed when the application releases all pointers to the `MessageSender` or when the `Session` upon which it depends goes out of scope.

Sending a message is accomplished by invoking the `MessageSender` `send` method. This method accepts a smart pointer to a `Message` object that has been created by the application. A `MessageID` object is returned that uniquely identifies the resulting sent Message.

Note that the `MessageSender` does not cache any message state after a message has been sent. If desired an application may save a `Message` object so that it can be modified and resent. Subscribing applications will see each received message as independent from any other message, though.

C++ API Reference and Code Examples

The code example below shows the creation of a `MessageSender` for the `Example::Notification` message type. This `MessageSender` is using reliable communication. An `Example::Notification` message is then created and sent using the `MessageSender`.

```
TENA::Middleware::CommunicationProperties  
communicationProperties( TENA::Middleware::Reliable );  
  
Example::Notification::MessageSenderPtr p_ExampleNotificationSender;  
p_ExampleNotificationSender  
= Example::Notification::MessageSender::create(  
    pSession,  
    communicationProperties );  
  
Example::Notification::MessagePtr pMessage;  
pMessage = Example::Notification::create( "This is a test." );  
  
pMessageSender->send( pMessage );
```

Release 6.0.2 Update — An alternative `create` method was added with release 6.0.2 of the middleware in which the `SessionPtr` argument is used, versus the `Session` reference (i.e., `*pSession`) that was used in the previous releases. Users are encouraged to use the `SessionPtr` interface. See [MW-4286](#) for more details.

Java API Reference and Code Examples

The code example below shows the creation of a `MessageSender` for the `Example.Notification` message type. This `MessageSender` is using reliable communication. An `Example.Notification` message is then created and sent using the `MessageSender`.

```
Example.Notification.MessageSender notificationSender =
    Example.Notification.MessageSender.create( session, TENA.Middleware.CommunicationProperties.Reliable );

notificationSender.send( Example.Notification.Message.create( "This is a test." ) );
```

Message Communication Properties

Message Communication Properties

The TENA Middleware supports two forms of communication transport when exchanging Messages between publishing and subscribing applications. The **Reliable** transport is based on the TCP communication protocol that is designed to support reliable point-to-point communication. The **Best Effort** transport is based on the UDP Multicast communication protocol that is designed to support efficient communication from one publisher to many subscribers.

Description

The MessageSender determines which communication transport is used for delivering Messages. Reliable transport is recommended for sending Messages in situations where lost Messages cannot be tolerated. For more information concerning communication transports, see the [reliable](#) and [best-effort](#) documentation.

Message Subscription

Message Subscription

An application must subscribe before it can receive Messages of a particular type. The subscription process includes attaching one or more Observers to the subscription that provide the application specific processing behavior associated with the received messages. In addition to the type-based filtering, an optional Filter may be specified to define additional filtering criteria as part of the middleware [Advanced Filtering capability](#).

Description

Subscribing to a Message type communicates an application's interest in receiving Messages from all current and future applications that send Messages matching that type. Since the TENA Middleware uses "send-side" filtering, an application's Message interest must be communicated to all publishing applications to determine whether there is a match in what the publishing application is publishing and what the subscribing application is interested in receiving. Obviously, this interest communication is not instantaneous and there is a communication delay between when the application defines the subscription interest and the publisher begins sending matching Messages.

Applications use the concept of a Session to internally organize publication and subscription actions, so Message subscriptions are associated with a particular [Session](#) object. See the [Execution Management Services documentation page](#) for additional information on Sessions.

When subscribing to a particular Message type, there is a `subscribe` function in the namespace of the Message type. The `subscribe` function accepts a Session, an instance of a class extending `SubscriptionInterface`, a self-reflection flag, and optionally, a Filter.

The `SubscriptionInterface` argument is typically provided in the form of a `Subscription` instance. ⚠ The use of `SubscriptionInterface` is a more generic capability for backwards compatibility, only available in the C++ binding. All the language bindings provide a concrete `Subscription` class that can be used for `subscribe`. Additional information regarding the `SubscriptionInterface` and attaching Observers to control application behavior with receiving messages is covered in the subsequent section.

The `selfReflection` argument is used to control whether the subscribing application wants to receive all Messages, including those published through the specified Session. If `selfReflection` is set to `true`, the default, then the application will receive those Messages published by the specified Session. In many situations, an application is not interested in receiving its own messages. The default behavior was set to enable self reflection, in order to be compatible with previous releases of the TENA Middleware.

The optional `Filter` argument allows for additional filtering criteria of Messages. The default is a "no tag" Filter that matches publishers not using advanced filtering. This capability is described in detail by the [advanced filtering documentation page](#).

Message Observers

An application that subscribes to Messages needs to provide application specific behavior to be executed when a Message is received. This behavior may be to write information to a database, update a display, or perform some other custom logic needed by the application. In this situation, the TENA Middleware receives the information (typically from the network) and constructs a Message object. The middleware then needs to notify the application that there is a new Message that needs to be acted upon.

Internally to the middleware, Messages received by a subscribing application are dispatched via a callback mechanism. A `SubscriptionInterface` derived class provides the `CallbackFactory` that creates `Callback` objects that placed in a callback queue. Automatically generated code associated with an object model definition will define a `Subscription` class (derived by `SubscriptionInterface`) that can be used by the subscribing application to attach `Observer` classes that provide the application specific behavior that needs to be invoked when a Message is received.

When the middleware performs the callback processing (see [callback framework documentation page](#) for additional details), any received Messages will be dispatched to the `messageEvent` method of all registered Observers for that particular subscription. The `messageEvent` method is written by the application developer to provide the appropriate behavior that the application should perform in response to receiving a Message.

An application developer can create as many Observers as necessary and attach appropriate Observers to particular Subscriptions. For example, an application may have created a "logging" observer, `LoggingObserver` class, and a "calculate damage" observer, `CalculateDamageObserver` class. The `LoggingObserver` could be attached to all Message Subscriptions independent of the Message type, and the `CalculateDamageObserver` would only be attached to the Subscription associated with the `Explosion` Message type.

Observers are attached to the Message specific Subscription object through the `addObserver` method. A code example for attaching a `PrintingObserver` to an `Example::Notification` Message type from the `Example-Vehicle` object model is shown below.

Details concerning the Observer mechanism can be found on the [Observer Mechanism documentation page](#). It should be noted that `callbacks` as implemented for TENA Middleware release 5 are still supported. However, the Observer mechanism allows callback behavior to be packaged into separate components that can be dynamically and selectively enabled, and are believed to provide a superior user API.

As mentioned, application developers are responsible for developing the `Observer` classes that are needed for their particular needs. When a Message is received that corresponds to registered `Observer`, the middleware will invoke the `messageEvent` method on the `Observer` class. The argument provided to the `messageEvent` method is a `ReceivedMessage` pointer. The `ReceivedMessage` class composes the actual Message instance and the associated Metadata instance. Users obtain these instances from the `ReceivedMessage` pointer through the methods `getSentMessage` and `getMetadata`.

ℹ — Note that `ReceivedMessage` pointer provided to the `messageEvent` method does not support polymorphism, but the polymorphic Message object can be accessed using the `getSentMessage` method. The resulting Message may be dynamically cast, modified, and forwarded by a `MessageSender`. More information concerning inheritance and polymorphism is available in the [Message Inheritance documentation page](#).

⚠ — As previously mentioned, the subscribing application is required to use the middleware evoke callback services (e.g., `evokeMultipleCallbacks`) in order for the middleware to process the callback objects that have been placed on an internal queue. In turn, these callback implementations will invoke the `messageEvent` method associated with registered Observers. Since the application is free to use multiple programming threads with the evoke callback services, application developers are responsible to ensure that the `Observer::messageEvent` methods support concurrency.

Duplicate Subscriptions

Subscribing to the same Message type (in the same Session) will replace the previous Subscription with the new Subscription and associated Observers. If two independent subscriptions are needed within an application, separate Sessions can be utilized, although attaching multiple Observers to the same Subscription may be more appropriate.

C++ API Reference and Code Examples

subscribe API

The `subscribe` declaration for `Example::Notification` Message type from the `Example-Vehicle` object model is shown below.

```
static
void
subscribe(
    ::TENA::Middleware::SessionPtr session,
    ::Example::Notification::SubscriptionInterfacePtr const & subscription,
    bool enableSelfReflection = true,
    ::TENA::Middleware::Filter const & filter =
        ::TENA::Middleware::Filter() );
```

Release 6.0.2 Update — An alternative `subscribe` method was added with release 6.0.2 of the middleware in which the `SessionPtr` argument is used, versus the `Session` reference (i.e., `"Session &"`) that was used in the previous releases. Users are encouraged to use the `SessionPtr` interface. See [MW-4286](#) for more details.

```
Example::Notification::AbstractObserverPtr pCountingObserver(
    new MyApplication::Example_Notification::CountingObserver( verbosity, std::cout ) );
pExampleNotificationSubscription->addObserver( pCountingObserver );
```

Observer Implementation

When an application needs to subscribe to a particular Message type, it is necessary for the developer to provide the implementation code to be used when a matching Message is received by the application. This implementation code is packaged in the form of an `Observer` class. In order to register the `{Observer}` with the TENA Middleware, the user defined `Observer` class must extend the appropriate `AbstractObserver` class. Each Message type has a corresponding `AbstractObserver` class whose declaration is provided in the object model definition, specifically the `messageType.h` header file.

The generated `AbstractObserver` class for the `Example::Notification` Message type from the `Example-Vehicle` object model is shown below. As the name applies, this class is abstract and is intended for the user to write a derived class that implements the `messageEvent` method, along with the constructor(s) of the user defined derived class.

```

class AbstractObserver
{
public:
    ///! Default constructor with a no-op implementation
    AbstractObserver() {}

    ///! Virtual destructor with a no-op implementation
    virtual ~AbstractObserver() {}

    /*! \brief Invoked each time a Message is received
     *
     * Observer classes that derive from this class should implement this
     * method to perform whatever actions are desired each time a
     * Message is received.
     *
     * \param[in] pReceivedMessage A reference-counted "smart" pointer to
     * the ReceivedMessage
     */
    virtual
    void
    messageEvent( ReceivedMessagePtr const & pReceivedMessage )
        = 0;

}; // End of class Notification_Message::AbstractObserver

```

The example applications include a simple printing and printing Observer. An abbreviation of the `Example::Notification::PrintingObserver` class declaration is shown below.

```

class PrintingObserver
    : public Example::Notification::AbstractObserver
{
public:
    PrintingObserver(int verbosity, std::ostream os);

    virtual
    void
    messageEvent(
        Example::Notification::ReceivedMessagePtr const & pReceivedMessage );

private:
    int verbosity_;
    std::ostream os_;
};

```

The code declaration above illustrates how an Observer could be defined to print information about each received message for the `Example::Notification Message` type.

The corresponding `messageEvent` method implementation is shown below for the `Example::Notification Message` type from the `Example-Vehicle` object model. This example is from the generated example application code. In this simple example, several `ostream` operators have been defined (*not shown*) to simplify printing the `Message` attribute and metadata values.

```

void
MyApplication::Example_Notification::PrintingObserver::
messageEvent( Example::Notification::ReceivedMessagePtr const & pReceivedMessage )
{
    if ( verbosity_ > 1 )
    {
        os_ << "Message: Example::Notification " << std::endl;
        if ( verbosity_ > 2 )
        {
            os_ << pReceivedMessage << std::endl;
            if ( verbosity_ > 3 )
            {
                os_ << pReceivedMessage->getMetadata() << std::endl;
            }
        }
    }
    else if ( verbosity_ == 1 )
    {
        os_ << 'r' << std::flush;
    }
}

```

In the above `messageEvent` example, the `PrintingObserver` makes use of two application variables (`verbosity`, `os`) that were passed into the constructor. Since the application developer designs the `Observer` classes, providing access to application specific information is easily handled. Developers can provide access to application data structures by defining constructors for the implemented `Observer` class that store appropriate application as class members to be used by the `messageEvent` method implementation.

Application developers may create `Observer` derived classes that can be shared among multiple applications. For example, a collection of `Observer` classes (for the different object model types) can be defined to support application logging activities where the information related to each received Message is logged to a database or file for diagnostic purposes.

Observer Instantiation and Registration

An application attempting to subscribe to a particular Message type is required to instantiate and register the appropriate `Observer`(s). Without an `Observer` registered to a Message subscription, there is no mechanism for an application to be notified when a matching Message is received.

`Observers` are attached to a Message type-specific `Subscription` object that is defined in the object model definition. Applications create a `Subscription` object, then create and attach one or more `Observers`. The example below illustrates attaching a `PrintingObserver` and a `CountingObserver` for a subscription to `Example::Notification` Message type defined in the `Example-Vehicle` object model.

```

Example::Notification::SubscriptionPtr pExampleNotificationSubscription(
    new Example::Notification::Subscription );

Example::Notification::AbstractObserverPtr pPrintingObserver(
    new MyApplication::Example_Notification::PrintingObserver( verbosity, std::cout ) );

MyApplication::Example_Notification::CountingObserverPtr pCountingObserver(
    new MyApplication::Example_Notification::CountingObserver( 1024 ) );

pExampleNotificationSubscription->addObserver( pPrintingObserver );
pExampleNotificationSubscription->addObserver( pCountingObserver );

```

`Observers` can also be removed from a `Subscription` using the `removeObserver` method. This method requires a `SubscriptionPtr`, similar to the `addObserver` method.

subscribe Invocation

Once the `Subscription` object has been created and `Observers` have been attached, the application invokes the `subscribe` function to perform the Message subscription. These `subscribe` functions are defined for each Message type within the appropriate namespace. The declaration of the `subscribe` method is shown below.

```

void
subscribe(
    ::TENA::Middleware::SessionPtr session,
    Example::Notification::SubscriptionInterfacePtr const & subscription,
    bool enableSelfReflection = true,
    ::TENA::Middleware::Filter const & filter = ::TENA::Middleware::Filter() );

```

Release 6.0.2 Update — The `subscribe` method shown here was added with release 6.0.2 of the middleware in which the `SessionPtr` argument is used, versus the `Session` reference (i.e., `*pSession`) that was used in the previous releases. Users are encouraged to use the `SessionPtr` interface. See [MW-4286](#) for more details.

As with all subscriptions, they are associated with a particular `Session` that the application has created with the middleware. Typically an application will only have a single `Session`, although there may be use cases in which multiple `Sessions` within an application is beneficial. Additional information on Sessions can be found in the [Execution Management Services documentation page](#).

The second argument to the `subscribe` function is named `enableSelfReflection` and used to indicate whether the application needs to receive messages that are sent through this `Session`. In many cases, the applications doesn't care about Messages sent by this application `Session`, but the default argument value is defined to be `true` to be consistent with previous versions of the middleware.

The last argument provided to the `subscribe` function is an optional `Filter` that is used to provide additional non-type based filtering criteria. More information on this topic can be found in the [Advanced Filtering documentation page](#).

An example invocation of the `subscribe` method is shown below for the `Example::Notification` Message defined in the `Example-Vehicle` object model.

```

// Declare the application's interest in Notification messages.
Example::Notification::subscribe(
    pSession,
    pExampleNotificationSubscription,
    selfReflection );

```

Release 6.0.2 Update — The `subscribe` method shown here was added with release 6.0.2 of the middleware in which the `SessionPtr` argument is used, versus the `Session` reference (i.e., `*pSession`) that was used in the previous releases. Users are encouraged to use the `SessionPtr` interface. See [MW-4286](#) for more details.

Like many objects used in the TENA Middleware API, the `Subscription` and `Observer` objects are wrapped in smart pointers. These pointers assist destruction and memory management activities since the objects are referenced by both the application and the middleware. Therefore, the application can "let go" (i.e., allow to go out of scope or be released) of the `SubscriptionPtr` used the `subscribe` call and the middleware will still hold a reference that prevents the `Subscription` object from being destroyed.

Since the application and middleware can hold a reference (through the smart pointer) to the same `Subscription` object, the application is permitted to add and remove `Observers` to that subscription dynamically. The middleware will handle thread safety issues related to these operations.

Java API Reference and Code Examples

subscribe API

The `subscribe` declaration for `Example.Notification` Message type from the `Example-Vehicle` object model is shown below.

```

package Example.Notification;
class Subscription {
    ...
    static void      subscribe(Session session, Subscription subscription) ;
    static void      subscribe(Session session, Subscription subscription, boolean enableSelfReflection) ;
    static void      subscribe(Session session, Subscription subscription, boolean enableSelfReflection, Filter
filter);
    ..
}
{code:Java|title=Attaching an Observer|borderStyle=solid|linenos=false}
Example.Notification.AbstractObserver observer = new org.Example.Notification_Subscription.
ExampleNotificationPrintingObserver( verbosity );
exampleNotificationSubscription.addObserver( observer );

```

Observer Implementation

When an application needs to subscribe to a particular Message type, it is necessary for the developer to provide the code to be used when a matching Message is received by the application. This code is packaged in the form of an `Observer` class. In order to register the `{Observer}` with the TENA Middleware, the user defined `Observer` class must extend the appropriate `AbstractObserver` class. Each Message type has a corresponding `AbstractObserver` class whose declaration is provided in the jva binding jar for the particular object model.

The generated `AbstractObserver` class for the `Example::Notification` Message type from the `Example-Vehicle` object model is shown below. As the name applies, this class is abstract and is intended for the user to write a derived class that implements the `messageEvent` method, along with the constructor(s) of the user defined derived class.

```
package Example.Notification;
abstract class AbstractObserver {
    public abstract void messageEvent(ReceivedMessage rcvdMessage);
}
```

The example applications include two simple Observer classes: simple printing and counting Observer. The `PrintingObserver` class for `Example.Notification` is shown below.

```
package org.Example.Notification_Subscriber;

class ExampleNotificationPrintingObserver extends Example.Notification.AbstractObserver {
    private int verbosity_;

    public ExampleNotificationPrintingObserver( int verbosity ) {
        verbosity_ = verbosity;
    }

    public void messageEvent( Example.Notification.ReceivedMessage receivedMessage ) {
        if ( verbosity_ > 1 ) {
            System.out.println("Message: Example.Notification");
            if ( verbosity_ > 2 ) {
                PrintUtility.display(receivedMessage.getSentMessage()); // Display all the message attributes
                if ( verbosity_ > 3 ) {
                    PrintUtility.display(receivedMessage.getMetaData()); // Display the message metadata
                }
            }
        } else if ( verbosity_ == 1 ) {
            System.out.print("r"); System.out.flush();
        }
    }
}
```

The code declaration above illustrates how an Observer could be defined to print information about each received message for the `Example.Notification` type. `PrintUtility` is a generated helper class that simply prints all the defined attributes for a Message or its metadata.

In the above `messageEvent` example, the `PrintingObserver` makes use of an application variable (`verbosity`) that is passed into the constructor. Since the application developer designs the `Observer` classes, providing access to application specific information is easily handled. Developers can provide access to application data structures by defining constructors for the implemented `Observer` class that store appropriate application structures as class members to be used by the `messageEvent` method implementation.

Application developers may create `Observer` derived classes that can be shared among multiple applications. For example, a collection of `Observer` classes (for the different object model types) can be defined to support application logging activities where the information related to each received Message is logged to a database or file for diagnostic purposes.

Observer Instantiation and Registration

An application attempting to subscribe to a particular Message type is required to instantiate and register the appropriate `Observer`(s). Without an `Observer` registered to a Message subscription, there is no mechanism for an application to be notified when a matching Message is received.

Observers are attached to a Message type-specific `Subscription` object that is defined in the object model definition. Applications create a `Subscription` object, then create and attach one or more Observers. The example below illustrates attaching a `PrintingObserver` and a `CountingObserver` for a subscription to `Example::Notification` Message type defined in the `Example-Vehicle` object model.

```

Example.Notification.Subscription notificationSubscription = new Example.Notification.Subscription();

Example.Notification.AbstractObserver printingObserver = new org.Example.Notification_Subscriber.
PrintingObserver( verbosity );
Example.Notification.AbstractObserver countingObserver = new org.Example.Notification_Subscriber.
CountingObserver();

notificationSubscription.addObserver( printingObserver );
notificationSubscription.addObserver( countingObserver );

```

Observers can also be removed from a `Subscription` using the `removeObserver` method. This requires holding the reference to the added observer.

subscribe Invocation

Once the `Subscription` object has been created and `Observers` have been attached, the application invokes the `subscribe` function to perform the Message subscription. These `subscribe` functions are defined for each Message type within the appropriate namespace, as shown earlier.

All subscriptions are associated with a particular `Session`, that the application has created with the middleware. Typically an application will only have a single `Session`, although there are use cases in which multiple `Sessions` within an application are beneficial. Additional information on `Sessions` can be found in the [Execution Management Services documentation page](#).

The (optional) third argument to the `subscribe` function is named `enableSelfReflection` and used to indicate whether the application needs to receive messages that are sent through this `Session`. In many cases, the applications doesn't care about Messages sent by this application `Session`, but the default argument value is defined to be `true` to be consistent with previous versions of the middleware.

The last argument provided to the `subscribe` function is an optional `Filter` that is used to provide additional non-type based filtering criteria. More information on this topic can be found in the [Advanced Filtering documentation page](#).

An example invocation of the `subscribe` method is shown below for the `Example.Notification.Message` defined in the `Example-Vehicle` object model.

```

// Declare the application's interest in Notification messages.
Example.Notification.Subscription.subscribe(session, notificationSubscription, false /* no self reflect */ );

```

Applications are permitted to add and remove `Observers` to a subscription dynamically. The middleware will handle thread safety issues related to these operations.

Message Unsubscribe

Message Unsubscribe

In order for an application to stop receiving Messages to which it has subscribed, it may invoke the `unsubscribe()` function. This function in the `Message` type namespace will remove interest in receiving Messages of that class type matching the `Filter`, if specified.

Description

When an application is no longer interested in receiving messages of a particular Message type, it should invoke the `unsubscribe` static method. The `unsubscribe` method accepts a Session and optionally a Filter (if [Advanced Filtering](#) is used). If multiple subscriptions exist for a Message type using different Filters, only the one matching the Filter specified in the `unsubscribe` call will be affected.

Several items that applications need to consider when using the Message unsubscribe mechanism:

- Calling `unsubscribe` with a Filter that is different from that used to subscribe will not unsubscribe from the subscription.
- For C++, the `SubscriptionInterface` smart pointer passed in to the `subscribe` function is copied and saved internally, so releasing the `SubscriptionInterface` pointer will not cause the application to unsubscribe.

C++ API Reference and Code Examples

An example `unsubscribe` declaration is shown in the code fragment below. As mentioned above, the `Filter` argument is optional and should only be used for Advanced Filtering subscriptions.

```
/// If you subscribe to Message type T1 with Filter F1, then
/// unsubscribe from Message type T1 with Filter F2, where F1 != F2,
/// you will NOT undo the subscription to T1 with Filter F1.
///
static
void
unsubscribe(
    ::TENA::Middleware::SessionPtr session,
    ::TENA::Middleware::Filter const & filter =
        ::TENA::Middleware::Filter());
```

Release 6.0.2 Update — An alternative `create` method was added with release 6.0.2 of the middleware in which the `SessionPtr` argument is used, versus the `Session` reference (i.e., `"Session &"`) that was used in the previous releases. Users are encouraged to use the `SessionPtr` interface. See [MW-4286](#) for more details.

Applications using advanced filtering with multiple subscriptions can determine the active set of `Filters` for a particular type with the following `Session` method.

```
/*! \brief Retrieve the subscription filters currently in use for a given type
*
* \param[in] typeIDpath The type Filters for subscriptions to which this method returns. (In plain English:
"This method returns Filters for subscriptions to this type.")
*
* \return A set of subscription Filters currently in use by all the
*         (non-[instance-based]) subscriptions to SDOs and Messages
*         of the given type in this Session. A subscription to
*         a type that is derived from the type given by typeIDpath
*         doesn't count. Instance subscriptions don't count (since
*         instance subscriptions don't have filters). If there
*         are N subscriptions to the given type in this Session,
*         then the return value has N (corresponding) elements.
*         The Filters in the return value are copies. The caller
*         may modify them. Doing so has no effect on the
*         associated subscriptions.
*/
std::set< ::TENA::Middleware::Filter > const
getFilters( ::TENA::Middleware::TypeIDpath const & typeIDpath ) const;
```

Java API Reference and Code Examples

An example `unsubscribe` declaration is shown in the code fragment below. As mentioned above, the `Filter` argument is optional and should only be used for Advanced Filtering subscriptions.

```
package Example.Notification;
class Subscription {
    static void unsubscribe( TENA.Middleware.Session session);
    static void unsubscribe( TENA.Middleware.Session session, TENA.Middleware.Filter filter);
```

Applications using advanced filtering with multiple subscriptions can determine the active set of `Filters` for a particular type with the following `Session` method.

```
/*! \brief Retrieve the subscription filters currently in use for a given type, specified by a typeIDPath
 */
class Session {
...
    public TENA.Middleware.FilterSet getFilters(TENA.Middleware.TypeIDpath typeIDpath );
```

Middleware Introduction

Middleware Introduction

The TENA Middleware is a key component of the Test and Training Enabling Architecture (TENA), forming the software foundation for enabling effective collaboration between distributed and independently developed systems. TENA applications use a "publish-subscribe" paradigm with a distributed shared memory model to exchange information, combined with the ability to define remote and local services. The middleware utilizes a formal meta-model to define modeling contracts enforced with automatic code generation technology to promote interoperability and reduce integration costs in test and training events. In addition to the middleware, the TENA project defines an encompassing architecture for procedures and tools for pre and post event activities.

Background

Foundation Initiative (FI) 2010 is a joint interoperability initiative from the Director of Operational Test and Evaluation (DOT&E). The FI 2010 vision is to enable interoperability across ranges, facilities, and simulations in a quick and cost-efficient manner and to encourage reuse of current range assets and future range systems. The referenced FI 2010 project has evolved into the TENA Software Development Activity (SDA), which continues to support the development and maintenance of TENA.

To achieve this vision, FI 2010 is developing and validating a common architecture, a core set of tools, inter-range communication capabilities, interfaces to existing range assets, interfaces to weapon systems, and recommended procedures for conducting test events or training exercises. The common architecture under development is called the Test and Training Enabling Architecture (TENA), and it will be synergistic with the Department of Defense (DoD) High Level Architecture (HLA) for Modeling and Simulation. TENA addresses test requirements not supported by HLA and improves the ability of ranges to interact with simulations. The FI 2010 project is coordinating with the Range Commanders Council (RCC) to ensure the range community, as a whole, reviews and adopts the TENA architecture, Application Programmer's Interface (API), and TENA Object Model. TENA products can function over both organic, DoD wide-area networks and commercial communication services. The FI 2010 project will develop interfaces necessary for existing range resources to become TENA-compliant. The interfaces will enable current infrastructures to be adapted to this common architecture in a cost-effective manner, rather than require that replacement systems be developed. FI 2010 is also defining TENA interfaces for tactical systems, allowing weapon systems under test to be stimulated by a simulation.

Several military transformations impacting the test and training range community are underway within the military domain. Joint Vision 2020 (JV 2020), an evolutionary enhancement to the original Joint Vision 2010, provides a road map and timeline for "creation of a force that is dominant across the full spectrum of military operations - persuasive in peace, decisive in war, pre-eminent in any form of conflict."

As information technologies transform the demands on the military, so they also transform its acquisition processes. The goal of Simulation Based Acquisition (SBA) is to substantially reduce the time, resources, and risk associated with acquisition, while producing higher quality products by adopting a "model-simulate-fix-test-iterate approach" to acquisition.

These transformations within the military coalesce directly in the Test and Evaluation (T&E) community. Test and training ranges must implement the JV 2020 vision through joint testing, must serve as an SBA enabler, must foster integration of testing and training, and must do all this in ways that minimize future range operations costs. To accomplish these goals, ranges must become more integrated and interoperable, allowing range assets to be procured and used efficiently, to be reused effectively, and to be combined to create a scale and scope of capabilities that will meet the challenges of implementing JV 2020. The FI 2010 project is the range community's mechanism for creating the infrastructure needed to meet these challenges.

TENA, the FI 2010 Common Architecture

The FI 2010 common architecture, referred to as the Test and Training Enabling Architecture (TENA), is illustrated in Figure 1. TENA enhances software interoperability and reuse throughout the range community by giving developers guidance on how to design range software applications that can interact easily with other TENA applications in support of range events. The architecture also specifies a common TENA Object Model, akin to a common set of interface definitions, that enables this interoperability through a community-wide understanding of range-related information. TENA applications (also called TENA resources), tools, repositories, and gateways all use the TENA Middleware as their communication medium. A copy of the TENA Middleware software is linked into every TENA application and is the mechanism for all execution-time communication between TENA applications. TENA's primary benefit is its ability to allow exercise planners to rapidly compose TENA-compliant applications into a geographically distributed "logical range" for a range event.

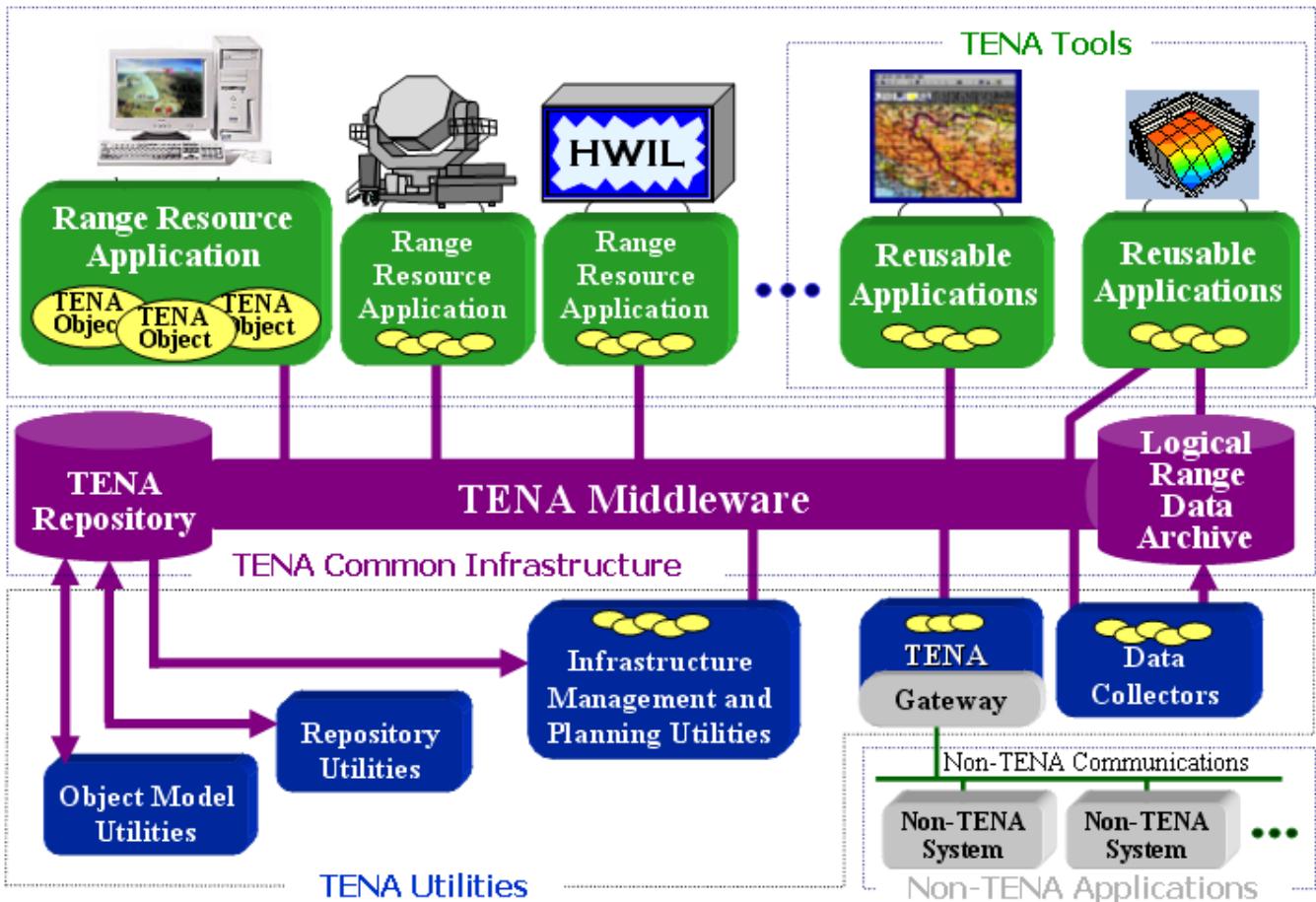


Figure 1: The TENA Architecture

A logical range is a suite of TENA resources, sharing a common object model, that work together for a given range event. In this context, TENA resources are:

- Compiled range applications that interact using TENA Middleware services,
- Gateway applications that bridge TENA resources to legacy systems or other protocols or architectures,
- TENA tools and databases, configured for a particular event.

TENA and its constituent parts are documented in a Baseline Report and a number of requirements documents. Technical decisions are made by the TENA Architecture Management Team (AMT), a group of TENA stakeholders who are also range engineers, that meets every six to eight weeks to consult on TENA's progress and on the development of the TENA Middleware prototype. Requirements for the TENA Middleware are derived from six primary sources:

- The FI 2010 Joint Overarching Requirements Document (JORD),
- The FI 2010 Technical Capabilities Requirements Document (TCRD),
- The TENA Baseline Report Volume III: Requirements,
- The IKE 2 Statement of Objectives (SOO),
- The TENA Architecture Management Team (AMT) proceedings,
- Specific direction from the TENA SDA Project Office.

Key Driving Requirements

The following discussion concerning the driving requirements of TENA has been extracted from the TENA Architecture Reference document that can be obtained from the FI 2010 website.

The JORD emphasizes that cost-effective interoperability is the primary reason for creating TENA. Interoperability is thus the most important technical driving requirement. Many of the technical driving requirements listed in the JORD are simply alternative ways of naming the same technical capability (sharability and reusability, for instance). One could successfully argue that there is only a single technical driving requirement for TENA - interoperability - since the other listed requirements may be derived directly from this one. For example, if one system is interoperable with a suite of systems, it should be able to be reused in the different places these other systems are used. Such a system is obviously flexible as well. Nevertheless, it is important to emphasize certain alternative aspects of interoperability by including them in the list of driving requirements, because each aspect has a slightly different thrust that is important for the architecture to capture. For this reason, the following three items have been chosen as the TENA technical driving requirements:

1. Interoperability - the characteristic of a suite of independently- developed components, applications, or systems that implies that they can work together, as part of some business process, to achieve the goals defined by a user or users.

2. Reusability - the characteristic of a given component, application, or system that implies that it can be used in arrangements, configurations, or in enterprises beyond those for which it was originally designed.

3. Composability - the ability to rapidly assemble, initialize, test, and execute a logical range from members of a pool of reusable, interoperable software elements (e.g., components or applications).

Though interoperability is the primary requirement elaborated in the JORD, most of the JORD describes user-oriented operational issues either related to interoperability or to creating logical ranges. Thus, a number of key operational issues are emphasized in the JORD text. Analysis of these yields the six most fundamental operational driving requirements. These are:

A. TENA must support the implementation of logical ranges, including the management of both software and data throughout the entire event lifecycle, including the requirements definition, planning, preparation, execution, and post-execution phases.

B. TENA must support the Joint Vision 2010/2020 by providing the foundation for testing and training in a net-work-centric warfare environment.

C. TENA must support rapid application and logical range development, testing, and deployment in a cost-effective manner.

D. TENA must support easy integration with modeling and simulation to advance the DoD's simulation-based acquisition joint distributed engineering plant concepts.

E. TENA must be gradually deployable and interact with non-TENA systems without interrupting current range operations.

F. TENA must support a wide variety of common range systems by meeting their operational performance requirements, including sensors, displays, control systems, safety systems, environment representations, data processing systems, communication systems, telemetry systems, analysis tools, data archives, and others.

Middleware Services

Middleware Services

The TENA Middleware provides a collection of services that can be used by applications to support their particular needs.

The various middleware services are organized into the following categories:

Service Category	Description
Configuration Mechanism	Mechanism used to define configuration parameters and allows users to set parameter values.
Execution Management Services	TENA Middleware uses concepts such as Runtime, Execution, and Session to support the necessary application Execution Management Services.
Publication Services	Middleware publication services enable an application to share object and messages with other interested applications.
Subscription Services	Middleware subscription services enable an application to receive SDO and messages from other applications.
Advanced Filtering	Advanced filtering provides applications the ability to provide finer control of subscription interests.
Middleware Threading Model	The TENA Middleware uses multiple internal programming threads and can support both single threaded and multi-threaded applications.
Middleware IDs	Unique numerical identifiers are provided for many of the middleware API elements to support application programming needs.
Middleware Metadata	Various middleware classes provide application developers with access to internal information (i.e., metadata) related to the operation of the middleware.

Advanced Filtering

Advanced Filtering

Nominally, the publish-subscribe behavior of the middleware operates on matching the object model type being published with the type of interest for the subscriber. Advanced filtering extends this type-based filtering to allow publishers and subscribers to provide additional criteria to provide finer control of subscription interests to minimize the amount of unwanted information that would be received with a pure type-based filtering system.

Description

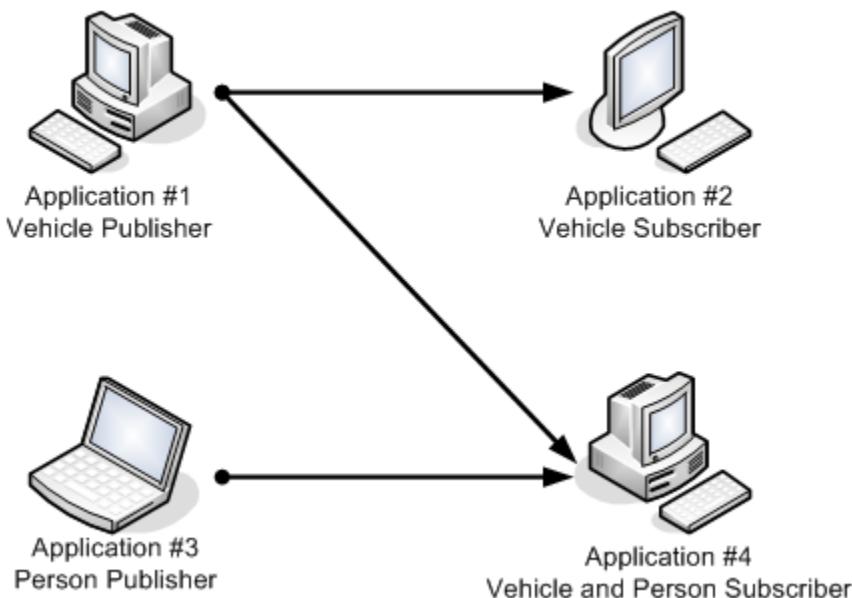
The TENA Middleware provides a peer-to-peer publish and subscribe capability in which the underlying software connects publishers with interested subscribers. Once connected, a publisher will attempt to provide the data of interest to the subscriber. The [TENA Meta-Model](#) supports [Stateful Distributed Objects \(SDOs\)](#) and [Messages](#) that can be exchanged between publishers and subscribers. A key objective in many distributed publisher-subscriber systems is to minimize the unwanted data that is sent to the subscribers, for the purpose of improving network utilization and minimizing unnecessary computer processing. If the subscribing application has to perform "receive-side" filtering to throw away a large percentage of received network messages, there is wasted computer resources in sending and processing the unwanted network messages.

Tip: Note that the term "object" will be used in the following discussion to represent either an [SDO Servant](#) or an [SDO Proxy](#), depending on whether the object exists in the publishing or subscribing application, respectively.

A primary technique that is used to support publisher-subscriber filtering is based on the particular data types defined in the [object model](#). Type-based filtering ensures that a subscriber will only receive data associated with either an SDO or Message type that matches a subscription request made by the subscribing application. For example, if an object model includes an SDO type named `Vehicle`, and if a subscribing application only subscribes to the `Vehicle` SDO type, then any other data that is published in the execution will not be delivered to the subscribing application.

Tip: When objects are published using the Best Effort protocol, the middleware must do the filtering on the subscriber side. When the Reliable protocol is used, the middleware is able to do the filtering on the publisher side. In either case the middleware only delivers data to the subscribing application which it has expressed interest in.

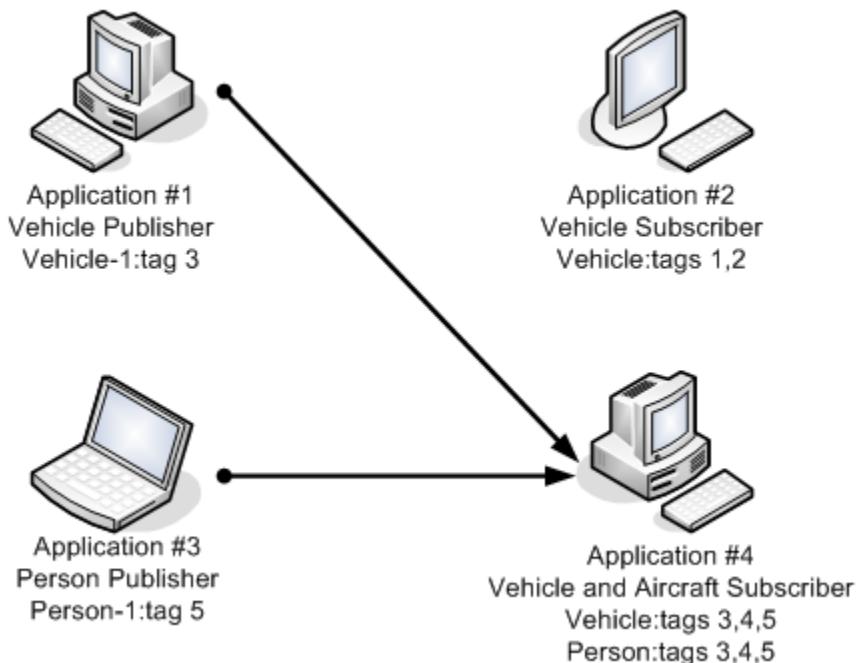
A simple illustration of type-based filtering is shown in the figure below. In this hypothetical scenario, Application #2 only subscribes to type `Vehicle` and will not receive any of the `Person` data published by Application #3. Application #4 subscribes to both type `Vehicle` and type `Person`, so it will receive data from both Application #1 and Application #3.



TENA Type-based Filtering Illustration

An extension to the type-based filtering supported by the TENA Middleware is referred to as "Advanced Filtering", which allows publishing and subscribing applications to provide finer specification of the publish-subscribe interest criteria than just the data types. Under Advanced Filtering a publishing application can dynamically associate a **tag** (an integer number) with a particular object or message. Likewise, a subscribing application can subscribe to object and message types with the additional constraint of requiring a match to one or more tags. When using tag-based filtering, the middleware will ensure that only data that matches with respect to the type and the tag will be delivered to the subscribing application.

The figure below illustrates tag-based filtering using Vehicle and Person publishers and subscribers. In this example, the Vehicle publisher (Application #1) is updating a Vehicle object (Vehicle-1) with the tag value of "3". Application #2 has subscribed to type Vehicle using the tag values of "1" and "2". Since there is not a tag match between publishing application #1 and subscribing application #2, the data for Vehicle-1 is not provided to application #2. Application #4 has subscribed to type Vehicle with the tag values of "3", "4", and "5", so there is a match and the data for Vehicle-1 will be delivered to application #4.



TENA Tag-based Filtering Illustration

When using tag-based filtering, the publishing and subscribing applications can change the tag specifications dynamically. For example, a publishing application may associate an object instance with a particular tag value for some period of time and then change to a different tag value.

Since messages are ephemeral (i.e., only exist for the moment that they are sent), the tag-based filtering concept as applied to messages is straightforward: If a publisher publishes messages of a given type with a given tag, and if that publisher has been informed of a subscriber's interest in that message type and tag value, then the publisher will attempt to send the message to the subscriber.

Since SDOs and their corresponding Proxies persist (unlike messages), the concept of tag-based filtering on SDOs is slightly more complicated than for messages.

When the SDO of a proxy in a subscribing application does not match any of the subscribing application's interests, the proxy is said to be "out of scope". This can occur if:

- the subscribing application discovers an SDO, then unsubscribes (or otherwise changes its interest not to match the SDO anymore) while holding the proxy, or
- the publishing application changes the SDO's tag so that it no longer matches the subscribing application's interest.

An "out of scope" proxy does not receive updates (in accordance with the subscribing application's interest). By holding a proxy, a subscribing application signals interest in being notified of the object's destruction. The scope of a proxy is available to a subscribing application by calling the `isInScope()` method on the proxy.

In addition to providing the ability for the subscribing application to get the current scope value for an object, the middleware will generate `EnteredScopeCallback` and `LeftScopeCallback` notifications that are delivered to the subscribing application through the normal `Observer` and/or `callback` mechanisms. An `EnteredScopeCallback` is delivered when a proxy transitions from "out of scope" to "in scope". (Thus, it does not occur upon discovery, since the proxy did not exist prior to discovery.) A `LeftScopeCallback` is delivered when a proxy transitions from "in scope" to "out of scope". (Thus, it does not occur upon destruction, since the proxy does not exist after destruction.)

The cases shown in the table below illustrate the possible scenarios involving dynamic tags with respect to object (SDO) instances. These cases indicate the possible transitions with a subscribing application in response to either a publishing application or the subscribing application tag change.

Simple Dynamic Object Tag Behavior

Scenario Description	Resulting Middleware Behavior
Publisher of undiscovered object changes object's tag causing a subscription to match object.	Middleware provides <code>DiscoveryCallback</code> to subscribing application.

Subscriber changes subscription tag causing subscription to match previously undiscovered object.	Middleware provides <code>DiscoveryCallback</code> to subscribing application.
Publisher changes object tag that breaks an existing subscription match with previously discovered object.	Middleware provides <code>LeftScopeCallback</code> to subscribing application. The object ceases to receive updates, but still receives destruction notifications.
Subscriber changes subscription tag that breaks an existing subscription match with previously discovered object.	Middleware provides <code>LeftScopeCallback</code> to subscribing application. The object ceases to receive updates, but still receives destruction notifications.
Publisher changes object tag causing a subscription to match previously discovered but out-of-scope object.	Middleware provides <code>EnteredScopeCallback</code> to subscribing application. The object begins receiving updates.
Subscriber changes object tag causing subscription to match previously discovered, but out of scope object.	Middleware provides <code>EnteredScopeCallback</code> to subscribing application. The object begins receiving updates.

It is important to note that when an object leaves scope (i.e., there is no longer a subscription match), the object will cease to receive state change updates, though it will receive a destruction notification if the publishing application destroys the object. If a subscriber application wishes to receive state change updates for a particular SDO instance that does not match the subscriber application's interest, the subscriber application must add that SDO to its interest by making an instance subscription to the SDO.

Applications are not required to utilize advanced filtering mechanisms. The middleware allows applications to operate consistently when there are applications using both purely-type-based subscriptions and advanced filtering operations. When an application uses type-based subscription, any advanced filtering publication (i.e., one with a tag) will result in a subscription match because a type-based subscription indicates that the application is interested in all objects or messages of that type, regardless of any advanced filtering specification. In the case of a purely-type-based publication (i.e., a publication not using a tag), a subscriber using advanced filtering (i.e., using a tag) will **not** receive the type-based object or message because the subscriber has explicitly provided additional subscription criteria that needs to be matched. The table below highlights the two cases that arise with mixing type-based and advanced filtering applications.

Scenarios involving type-based and advanced filtering applications

Scenario Description	Resulting Middleware Behavior
Publisher publishes with an advanced filtering tag and Subscriber subscribes to only a type (no tag).	Subscriber receives the data.
Publisher publishes without an advanced filtering tag and Subscriber subscribes to a type qualified by an advanced filtering tag.	Subscriber does not receive the data.

Note that inheritance-based type matching still works when advanced filtering tags are used. For example, a subscriber application subscribed to a Base type with a given tag will match a Derived type published with that same tag.

Subscribers wishing to use advanced filtering are required to pass a `Filter` to the subscription function.

Publishers wishing to use advanced filtering are required to pass a `Tag` to the method that creates the published object.

Assuming the published and subscribed-to types match, the following chart shows how the middleware matches the various kinds of Filters and Tags:

Matching of Filters and Tags

	Tag: <code>hasTagValue</code>	Tag: <code>noTagValue</code>
Filter: Wildcard	Match	Match
Filter: Tag	Match if Filter and Tag values are equal	No match
Filter: No tag	No match	Match

Usage Considerations

Tag Semantics

The current implementation of Advanced Filtering uses numerical integer values to represent the tags used to specify interests related to matching publishers and subscribers. There is no semantic meaning of the tag values that is understood or enforced by the middleware. It is the responsibility of the TENA application developers and event coordinators to define the semantic meaning of the tag values and ensure that the tags are applied consistently.

As an example, an event may decide to define tag values associated with SDO type `Aircraft` to correspond to the Aircraft's altitude discretized into 1,000 foot steps (a tag value of 0 indicates that the Aircraft is on the ground, and a tag value of 32 indicates that the Aircraft altitude is between 31,000 and 32,000 feet). Provided that all of the TENA applications participating in the event follow this tag value mapping algorithm, this approach will work. But consider the insidious problem that can result if one developer incorrectly assumed that the altitude steps were 10,000 feet instead.

Event coordinators are encouraged to clearly document the semantic meaning and algorithms associated with tag values. Additionally, simple test programs can be constructed to evaluate the usage of advanced filtering tags in order to detect incorrect usage. A future implementation of Advanced Filtering intends to allow event coordinators to develop software interest objects that can be shared among applications and designed to enforce consistent semantics.

Performance Considerations

When using advanced filtering, application developers need to be sensitive to how rapidly object or subscription tag values change. There are network communication requirements between the publishers and subscribers that need to take place whenever there is a change in the tag values. Additionally, there is a computational cost for the affected publishers and subscribers when there are tag value changes.

Another performance consideration in determining how rapidly an application should change tag values is the resulting subscription latency associated with any tag value change. As mentioned, there is network communication and middleware processing whenever there are changes in the tag values. That communication and processing does not happen instantaneously. Any time a tag value changes, there will be a delay (subscription latency) between when the tag was changed and when the publisher and subscriber will be connected (with respect to a publish-subscribe paradigm). If tags are changed rapidly in comparison to the subscription latency, the system may not be able to keep up with the tag changes to deliver interested data to the subscriber.

Multiple Discoveries

Multiple subscriptions, whether using advanced filtering or not, can result in multiple discoveries of the same SDO servant. For example, an application can perform a type-based subscription to `Tank` and then have a tag-based subscription to `Tank` for a tag value that represents the color "green". If a publishing application creates a green `Tank`, the subscribing application will receive two "discovery" events that result in two SDO proxies. This is useful because the subscribing application can supply different callback behavior for each subscription. This behavior is similar to multiple subscriptions within an inheritance hierarchy. Subscribing applications can check the `SDOids` of discovered proxies to help determine when two proxies are for the same servant.

Filter

Filter

A `Filter` is an object that a subscriber uses to specify which published objects of a given type it wants based on certain filtering criteria. Every published object has a `Tag`, (known as a "FilteringContext" before Release 6.0.4) against which the `Filter` matches.

Note that although the longer term plan is to provide more complex capabilities for `Filter` objects, at present `Filters` essentially provide an integer value (or an "anything" wildcard) that is used to match against the analogous integer in the `Tag` associated with the `MessageSender` or `Servant`. The actual filtering is generally done at the sender to minimize network traffic.

Description

Subscribers wishing to use advanced filtering are required to pass a `Filter` to the subscription function.

The `Filter` object can be any one of three kinds:

- A "wildcard" filter — which matches any published object of the given type, regardless of whether the published object has an advanced filtering tag, and if so, regardless of the tag's value. This can be thought of as an all-pass filter. It is analogous to a fully open valve.
- A "tag" filter — which matches only published objects of the given type having the given advanced filtering tag.
- A "no tag" filter — which matches only published objects of the given type having no advanced filtering tag value (i.e., a `Tag` object with no value set).

The default `Filter` is the wildcard filter. A subscription using the wildcard filter is a purely type-based subscription. When a subscriber does not explicitly pass a `Filter` to the `subscribe` function, it implicitly uses a wildcard filter.

The "no tag" filter is useful for subscriber applications that wish to subscribe to published objects that do not use advanced filtering (i.e., whose `Tags` do not have values). To create a "no tag" filter specify `TENA::Middleware::Filter::MatchUntagged` as the filter value.

C++ API Reference and Code Examples

SDO Subscribing with Advanced Filtering

When subscribing to object types, an application can specify an advanced filtering tag in the `subscribe` function. The code fragment below demonstrates subscribing to `Example::Vehicle` SDOs that have a `Tag` value of 4. It does this by passing a `Filter` with value 4 to the `subscribe` function.

⚠ — Although the current release of the Advanced Filtering capability does not support the ability to provide a set of filtering tags in a subscription request, users can perform multiple subscriptions if necessary.

```
Example::Vehicle::SubscriptionPtr pExampleVehicleSubscription(
    new Example::Vehicle::Subscription( wantPruning ) );

TENA::Middleware::Filter subFilter( 4 );

Example::Vehicle::subscribe(
    pSession,
    pExampleVehicleSubscription,
    false /* no self-reflection */ ,
    subFilter );
```

Release 6.0.2 Update — An alternative `subscribe` method was added with release 6.0.2 of the middleware in which the `SessionPtr` argument is used, versus the `Session` reference (i.e., "`*pSession`") that was used in the previous releases. Users are encouraged to use the `SessionPtr` interface. See [M-W-4286](#) for more details.

Message Subscribing with Advanced Filtering

Subscribing to messages behaves in a manner identical to that of objects, through the message type's `subscribe` function. The code fragment below demonstrates subscribing to `Example::TrafficReport` messages that have a `Tag` value of 5. It does this by passing a `Filter` value of 5 to the `subscribe` function.

```

TENA::Middleware::Filter messageFilter( 5 );

Example::TrafficReport::SubscriptionPtr
pExampleTrafficReportSubscription(
    new Example::TrafficReport::Subscription );

Example::TrafficReport::subscribe(
    pSession,
    pExampleTrafficReportSubscription,
    false /* no self-reflection */ ,
    messageFilter );

```

Release 6.0.2 Update — An alternative `subscribe` method was added with release 6.0.2 of the middleware in which the `SessionPtr` argument is used, versus the `Session` reference (i.e., `"*pSession"`) that was used in the previous releases. Users are encouraged to use the `SessionPtr` interface. See [M W-4286](#) for more details.

Java API Reference and Code Examples

SDO Subscribing with Advanced Filtering

When subscribing to object types, an application can specify an advanced filtering tag in the `subscribe` function. The code fragment below demonstrates subscribing to `Example.Vehicle` SDOs that are published with a tag value of 4. It does this by passing a `Filter` with a tag value 4 to the `subscribe` function.

```

Example.Vehicle.Subscription vehicleSubscription = new Example.Vehicle.Subscription( wantPruning );

Example.Vehicle::Subscription.subscribe( session, vehicleSubscription,
    false /* no self-reflection */ ,
    new TENA.Middleware.Filter( TENA.UnsignedInteger.asUnsigned(4) );

```

Message Subscribing with Advanced Filtering

Subscribing to messages behaves in a manner identical to that of objects, through the message type's `subscribe` function. The code fragment below demonstrates subscribing to `Example.TrafficReport` messages that are published with a tag value of 5. It does this by passing a tag value of 5 to the `subscribe` function.

```

Example.TrafficReport.Subscription trafficSubscription =
    new Example.TrafficReport.Subscription();

Example.TrafficReport.Subscription.subscribe( session, trafficSubscription,
    false /* no self-reflection */ ,
    new TENA.Middleware.Filter( TENA.UnsignedInteger.asUnsigned(5) );

```

Tag

Tag

The Tag (formerly known as "Filtering Context") defines the criteria associated with a published object or message that can be used to match a subscriber's [interest filters](#). SDOs and MessageSenders have associated Tags. Subscribing applications have Filters that the Middleware advanced filtering services use to match publishers and subscribers with respect to SDOs and messages. Tags and Filters are each implemented using a single integer value used to represent semantic meaning based on the filtering needs of the applications.

 – **Historical Note:** Tag was known as "FilteringContext" prior to release 6.0.4. The old name can still be used for backward compatibility.

Description

The Tag of a published object (SDO or MessageSender) is what can be filtered upon by a subscriber. Subscribing applications can use a "Filter" that matches certain Tags of published SDOs or Messages. Both Tag and Filter are implemented using a simple integer value. The tag values correspond to particular filtering criteria (e.g., value of 1 may correspond to the vehicle color being red, value of 2 may correspond to blue). It is critical for applications to implement their code to calculate Tag values and Filters consistently — it is recommended that applications using common advanced filtering schemes to share this implementation code.

Every publisher object, whether an SDO (Servant) or a MessageSender, contains a Tag. When a subscriber application subscribes to a type, it specifies a Filter. If the Filter matches the Tag and the subscribed-to type matches the published type, then the published data is delivered to the subscriber.

Publishers wishing to use advanced filtering are required to pass a Tag object to the method that creates the SDO or MessageSender. If no Tag is passed, a default is used.

Effect of Tag on Multicast Address

If an SDO or Message is published using Best Effort (IP/UDP Multicast), the object's Tag affects the multicast address on which that object is published. This feature enables the Middleware to take advantage of low-level filtering provided by the operating system of the subscriber for maximum efficiency. By listening only on the appropriate multicast addresses, the subscriber can receive only the multicast messages and updates that it wants without having to waste CPU cycles discarding unwanted packets. The multicast address used by a given publisher is computed using an algorithm explained on the [multicast address assignment documentation page](#).

Changing Tags

The Middleware allows publishers to change the object's or message's tag in either of two ways:

- simultaneously, with an SDO state change, and
- independent of an SDO state change.

A publishing application wishing to change an SDO's Tag independent of an SDO state change can simply call the `setTag()` method on the Servant or the Servant's `ServantPublicationStateView`. A MessageSender's Tag value is immutable, so applications wanting to publish Messages with different Tags need to create separate MessageSenders.

For more information on when you would want to, and how to, change an SDO's Tag when the SDO's state changes, see the [computing tag based on state change documentation page](#).

C++ API Reference and Code Examples

An application wishing to publish an object that can be filtered by a subscriber using advanced filtering must create a `TENA::Middleware::Tag` and pass it to the `createServant` or `createMessageSender` method. The Tag constructor takes an integer tag value.

The TENA Middleware supports the capability for an SDO's Tag object to have its tag values recomputed automatically at each state change. To that end, when an SDO (Servant) using advanced filtering is constructed, the caller must pass in an instance of `SDOTag` (a subclass of Tag having a virtual `computeTag()` method). If the publishing application does not require the Tag value to be recomputed upon state changes, then the default `SDOTag::computeTag()` implementation can be used. Otherwise, the `computeTag()` method should be overridden in a user-supplied class that derives from `SDOTag`.

The code fragment below illustrates the use of an `SDOTag` for an SDO whose tag does not change when the SDO's state changes.

```

#include <Example/Vehicle.h>

Example::Vehicle::ServantPtr pVehicleServant;

Example::Vehicle::ServantFactoryPtr pServantFactory(
    Example::Vehicle::ServantFactory::create( pSession ) ;

std::unique_ptr< Example::Vehicle::PublicationStateInitializer >
    initializer( pServantFactory->createInitializer() ) ;

... // initialization code omitted

TENA::uint32 objectTag( 3 );
std::unique_ptr< Example::Vehicle::SDOTag > pTag(
    new Example::Vehicle::SDOTag( objectTag ) );

pVehicleServant = pServantFactory->createServant(
    *initializer,
    std::move( pTag ),
    communicationProperties );

```

Release 6.0.2 Update

— An alternative `create` method was added with release 6.0.2 of the middleware in which the `SessionPtr` argument is used, versus the `Session` reference (i.e., `"*pSession"`) that was used in the previous releases. Users are encouraged to use the `SessionPtr` interface. See [MW-4286](#) for more details.

If it is necessary to change an object's advanced filtering tag after the object has been created, a new Tag can be passed in as an argument to the `setTag` method. The code fragment below shows changing an SDO's Tag independent of an update.

 — Users are discouraged from changing the object's tag value too frequently due to the subscription latency and performance issues.

```

TENA::uint32 objectNewTag( 4 );
std::unique_ptr< Example::Vehicle::SDOTag > pNewTag(
    new Example::Vehicle::SDOTag( objectNewTag ) );

pVehicleServant->setTag( std::move( pNewTag ) );

```

The advanced filtering operations for Messages behave in a similar manner as objects (SDOs). When sending messages, a `MessageSender` is created with the necessary tag value. The code fragment below illustrates this programming interface.

```

TENA::uint32 messageTagValue( 5 );
TENA::Middleware::Tag messageTag( messageTagValue );

Example::TrafficReport::MessageSenderPtr
pMessageSender(
    Example::TrafficReport::MessageSender::create(
        pSession,
        messageTag,
        communicationProperties ) );

```

Release 6.0.2 Update

— An alternative `create` method was added with release 6.0.2 of the Middleware in which the `SessionPtr` argument is used, versus the `Session` reference (i.e., `"*pSession"`) that was used in the previous releases. Users are encouraged to use the `SessionPtr` interface. See [MW-4286](#) for more details.

Java API Reference and Code Examples

An application wishing to publish an object that can be filtered by a subscriber using advanced filtering must create a `TENA::Middleware::Tag` and pass it to the `createServant` or `createMessageSender` method. The `Tag` constructor takes an integer tag value.

For SDOs, the TENA Middleware supports the capability for Tags to be recomputed automatically at each state change. Therefore, instead of passing a constant Tag when creating an SDO Servant using advanced filtering, the caller passes in an instance of `SDOTag`. This class adds a method `computeTag()`, which is specific to the SDO type being used. If the publishing application does not require the tag to be recomputed based on the SDO's state, then the `SDOTag` can be used directly with a constant `UnsignedInteger` value. If the tag may change based on the SDO's state, then the developer should extend the `SDOTag` class and override the `computeTag()` method to set the tag based on the SDO PublicationState.

```
Example.Vehicle.Servant vehicleServant;

Example.Vehicle.ServantFactory vehicleFactory = Example.Vehicle.ServantFactory.create( session );
Example.Vehicle.PublicationStateInitializer = vehicleFactory.createInitializer();
... // initialization code omitted

Example.Vehicle.SDOTag friendTag = new Example.Vehicle.SDOTag(TENA.UnsignedInteger.asUnsigned(3));

vehicleServant = servantFactory->createServant(initializer, friendTag, TENA.Middleware.CommunicationProperties.
BestEffort );
```

If it is necessary to change an object's tag after it is created, a new `SDOTag` may be set using the `setTag` method. The code fragment demonstrates this.

 — Users are discouraged from changing the object's tag value frequently due to the subscription latency and performance issues.

```
Example.Vehicle.SDOTag foeTag = new Example.Vehicle.SDOTag(UnsignedInteger.asUnsigned(4));

vehicleServant.setTag( foeTag );
```

The advanced filtering operations for Messages are similar. When sending messages, a `MessageSender` is created with the necessary tag value. The code fragment below illustrates this.

```
TENA::Middleware::Tag importantTag = new TENA::Middleware::Tag(5);

Example.TrafficReport.MessageSender msgSender = Example.TrafficReport.MessageSender.create(
    session, importantTag, TENA.Middleware.CommunicationProperties.BestEffort );
```

Computing Tag based on State Update

Computing Tag based on State Update

There are two ways to change an SDO's [Tag](#):

- Independent of a state change, and
- Simultaneous with a state change.

This page explains the second way: changing an SDO's Tag when the SDO's publication state changes.

– Historical Note: Prior to release 6.0.4, a Tag was referred to as a "FilteringContext". For backward compatibility, one can still use the old name, which has become a typedef to the new name.

Computing Tag based on State Update

The ability to change an SDO's tag simultaneously with an SDO state change is important for generating the proper scope callbacks in subscribing applications. If the tag change associated with a state change were not atomic with the state change, a subscriber leaving scope because of the tag change could get an invalid state change callback, while a subscriber entering scope because of the tag change could fail to get a state change callback that it should have gotten.

The Middleware enables a publishing application to make a tag change simultaneously with an SDO state change by letting the publishing application specify a `Tag` object that computes the integer tag value automatically. For this purpose, the Middleware defines (for each SDO type) a class called `SDOTag`. `SDOTag` is a `Tag` that has a pure virtual `computeTag` method, which the user implements. This `computeTag()` method can recompute the SDO's tag based on the SDO's state or anything else it wishes. The Middleware invokes this `computeTag` method automatically upon each state change.

Publishing applications wishing to make a tag change simultaneously with an SDO state change should provide a C++ implementation class that derives from the `SDOTag` base class and implements the virtual `computeTag` method. (The default implementation of `computeTag` is a no-op.)

C++ API Reference and Code Examples

To change an SDO's Tag simultaneously with a state update, the application developer must create a custom `SDOTag` (formerly known as "TagComputing FilteringContext") and implement its `computeTag` method. The code fragment below demonstrates how to do this:

```
class MyTag
    : public Example::Vehicle::SDOTag
{
    void computeTag( Example::Vehicle::PublicationStatePtr const & pPubState )
    {
        set_tag( pPubState->get_displayColor() );
    }
};
```

Then, when creating the Vehicle servant, you'd pass in the custom `SDOTag` as follows:

```
std::unique_ptr< Example::Vehicle::SDOTag > pTag(
    new MyTag() )

pVehicleServant = pServantFactory->createServant(
    std::move( pRemoteMethods ),
    *initializer,
    std::move( pTag ),
    communicationProperties );
```

Java API Reference and Code Examples

To change an SDO's Tag simultaneously with a state update, the application developer must create a derived `SDOTag` class and override the `computeTag` method. The code fragment below demonstrates how to do this:

```
class MyDynamicTag extends Example.Vehicle.SDOtag {  
    void computeTag( Example.Vehicle.PublicationState pubState )  
    {  
        set_tag( pubState.get_displayColor() ); // Set the integer "tag" in some fashion from the sdo state  
    }  
};
```

Then, when creating the Vehicle servant, you'd pass in the custom SDOtag as follows:

```
Example.Vehicle.Servant vehicleServant = pServantFactory->createServant(  
    remoteMethods, initializer, new MyDynamicTag(), communicationProperties );
```

Configuration Mechanism

Configuration Mechanism

The TENA Middleware uses a mechanism for defining configuration parameters that control the runtime behavior of the software. Users are able to specify the parameter values through command line arguments, configuration files, and environment variables. The configuration mechanism is used for the middleware and the Execution Manager process. The automatically generated Example Applications utilize the same configuration mechanism to define configuration parameters associated with the application behavior. Application developers can utilize the same configuration mechanism for their configuration requirements.

Description

The TENA Middleware uses a configuration mechanism to allow application developers and operators to control the configuration parameter values associated with the operation of the middleware. Although the configuration mechanism was developed to support the middleware, application developers can utilize the same mechanism for application specific configuration control.

Each configuration parameter is controlled by a `Setting` (C++ object) that controls the semantics for assigning and retrieving the parameter setting value. The collection of `Settings` is managed by a `Configuration` (another C++ object) that supports adding settings, controlling the parsing of settings values from multiple input sources, and retrieving the setting values.

Input sources supported by the configuration mechanism include command line arguments, configuration files, environment variables, and key value pair list. The key value pair list is a low level data structure that uses a character string for the parameter name and corresponding string for the parameter value.

Configuration Input Parsing

By default, the configuration mechanism supports multiple standard input sources that can be used to define the parameter setting values: (1) command line, (2) configuration file, (3) environment variable. Developers can also define the parameter values programmatically through the Configuration API (application programming interface). By default, these input sources are processed according to a precedence order with command line being the highest precedence and environment variables being the lowest. In the situation in which there are multiple occurrences of the same setting from a particular input source, the last value specified will be used.

Command Line Processing — Configuration constructors support the standard `argc`, `argv` convention of command line arguments that are used when an application is started through an operating system shell. Command line arguments can specify configuration settings using a single dash '-' before the setting name followed by the parameter value (if a value is associated with the setting). Nominally, a case sensitive query on the Configuration object is used to determine if the specified parameter is found. A `std::runtime_error` exception will occur if the parameter requires a value and it was not specified in the command line argument list. Recognized command line arguments will be removed from the `argv` vector and `argc` will be decremented accordingly.

Configuration File Processing — When the Configuration object is provided the name and location of a configuration file, every line within the file will be processed according to the syntax [`<prefix>.|<parameterName>[=value]`]. Only a single parameter may be specified per line. The prefix can be used to organize parameters for multiple applications within the same file. Prefixes may contain letter, digit, hyphen ('-'), and underbar ('_') characters. Parameters without a prefix will be applied to all configurations, although a prefixed parameter will take precedence over an entry for the same unprefixed parameter. Environment variable values can be substituted into a configuration file using the syntax `$(envVarName)`. The hash (#) can be used for comments within the configuration file in which anything to the right of the hash will be ignored by the parser.

Environment Variable Processing — Environment variables can be used to specify configuration parameter settings. A prefix can be used to organize parameters associated with different applications, with the prefix used for the TENA Middleware being "TENA_MW_". When searching the configuration parameters for a match to a particular environment variable, the prefix is removed, any underbar ('_') characters are removed, and the resulting letters are converted to lower case. Thus, the environment variable "TENA_MW_CONNECTION_TIMEOUT_IN_MILLISECONDS" can be used to set the middleware parameter "connectionTimeoutInMilliseconds".

The underlying configuration mechanism utilizes key value pair lists to store configuration parameters and values. Application developers may utilize the `TENA::Middleware::Utils::BasicConfiguration::KeyValueList` to set parameter values within their application code, versus relying on the standard input sources. The key for this list is the name of the parameter. As an example, if an application utilizes a GUI (graphical user interface) for the operator to define configuration values, these values can be inserted into a `KeyValueList` and this list can then be provided to an `ApplicationConfiguration` either through the constructor or using the `parse` method. This mechanism will also work if the application wants to statically or programmatically define configuration values within the application software.

`ApplicationConfiguration` objects provide methods to identify any parameters specified in the configuration file or environment variables matching the prefix that are unknown to the configuration's defined settings. These methods allow the application to warn the operator in case there was a misspelling or similar problem with an unknown configuration parameter. Parameters specified through the command line `argc`, `argv` mechanism are consumed when the parameter is processed, so any resulting command line arguments specified in the `argv` vector are unknown to the configuration's defined settings.

Configuration Classes

There are several Configuration classes that can be used by TENA applications for the middleware and application configuration needs:

- **`TENA::Middleware::Utils::BasicConfiguration`** — Base configuration class that provides the basic services associated with configuration parameters. This class is not recommended to be used directly in typical situations, instead the `ApplicationConfiguration` class is recommended to be used for application configuration needs.

- **TENA::Middleware::Configuration**— Class derived from `BasicConfiguration` that contains the parameters associated with the operation of the TENA Middleware. A `TENA::Middleware::Configuration` object must be provided to the `TENA::Middleware::init()` function to properly initialize the middleware. This class is not inheritable and prevents adding application specific parameter settings. ⚠ Note that this is the only configuration class exposed via language bindings other than C++. Other languages have a variety of libraries suitable for the more general problem of passing application settings to an application. The underlying middleware library still requires construction of a `TENA::Middleware::Configuration`, which will be wrapped in an equivalent proxy class for other language bindings.
- **TENA::Middleware::ApplicationConfiguration**— Class derived from `BasicConfiguration` and used for application configuration classes. The `TENA::Middleware::ApplicationConfiguration` class contains a `TENA::Middleware::Configuration` that utilizes common parsing of the input sources, with a `tenaConfiguration()` method to obtain the contained middleware configuration.
- **ApplicationConfiguration**— Automatically generated configuration class that is provided with object model specific example applications. The `ApplicationConfiguration` class derives from `TENA::Middleware::ApplicationConfiguration` and defines configuration parameter settings in the `defineSettings()` methods for parameters such as `emEndpoints`, `bestEffort`, and `verbosity`.

The automatically generated `ApplicationConfiguration` class is recommended to be used for all TENA applications because it simplifies the support for middleware configuration parameters and enables developers to easily support application specific configuration parameters, such as a parameter which specifies the Execution Manager (EM) endpoint list ("`emEndpoints`").

Why is the parameter "emEndpoints" not in `TENA::Middleware::Configuration`?

Although it seems as though the configuration parameter associated with the Execution Manager (EM) endpoint list belongs in the middleware configuration object, it is actually an argument to the `TENA::Middleware::RuntimePtr::joinExecution()` method and not an internal middleware configuration parameter. Applications may utilize various techniques to obtain and manage the EM endpoint list, which would be constrained if the EM endpoint list configuration parameter was maintained internal to the middleware.

Parameter Setting Class

Individual configuration parameter settings are characterized by the following items:

- Parameter Name,
- Description,
- Default Value (if applicable),
- Required Indicator (whether a user supplied value is necessary),
- Password Indicator (whether value should not be displayed in clear text),
- Hidden Indicator (whether parameter should be displayed in usage/help message),
- Source Identifier (indicates the input source, e.g., command line, file, of the value used).

Parameter names may only contain letter, digit, and hyphen ('-') characters. A runtime exception will occur if others characters are attempted to be used for the parameter name.

Application developers can define parameter settings for a particular configuration object by using the `addSetting()` method. An example parameter setting is illustrated in the code fragment below for a parameter named "numberOflterations" that has a default value of 30.

```
appConfig.addSetting( TENA::Middleware::Utils::Setting( "numberOfIterations",
    TENA::Middleware::Utils::Value< TENA::uint32 >().setDefault( 30 ),
    "The number of iterations (updates and/or callback intervals) the "
    "application will run." ) );
```

Additional details related to the `Setting` class and guidance on advanced usage may be found in the [API Reference section](#).

Usage Considerations

ApplicationConfiguration Definition

If an application does not need to use the TENA configuration mechanism for application specific configuration parameters, then a `TENA::Middleware::Configuration` object can be constructed and used as an argument to the `TENA::Middleware::init()` function. As noted in the "Configuration Input Processing" section above, applications are required to construct a vector of Execution Manager (EM) endpoints to provide to the `TENA::Middleware::RuntimePtr::joinExecution()` method. Therefore, the recommended approach to manage the middleware configuration parameters and the EM endpoint vector is to use the automatically generated `ApplicationConfiguration` class that inherits from `TENA::Middleware::ApplicationConfiguration` and defines an "emEndpoints" configuration parameter. Additional configuration parameters defined in the automatically generated `ApplicationConfiguration` class can be removed if not needed for the particular application.

The example `ApplicationConfiguration` class defines a constructor which accepts arguments to support the standard input sources (i.e., command line, configuration file, and environment variables), although users may want to modify this definition to define additional constructors if necessary. For example, if the application used a GUI to specify configuration values, a constructor for a `KeyValueList` could be added to the `ApplicationConfiguration` declaration.

It is appropriate for all of the `ApplicationConfiguration` constructors to check the validity of parsed application specific configuration values and handle unknown options. The middleware configuration parameters are checked by the `TENA::Middleware::Configuration` class. Additionally, an application can check if all required middleware and application specific configuration parameters have been properly set within the application before there is an attempt to initialize the middleware and join an execution. A `validate()` method on the Configuration classes can be invoked for this purpose (additional details can be found in the [API Reference section](#)).

Therefore, the recommended steps in developing a configuration object is to utilize the automatically generated `ApplicationConfiguration` class and customize to the particular needs of the application. The `ApplicationConfiguration` class files (`ApplicationConfiguration.h` and `ApplicationConfiguration.cpp`) can be located in the `TENA_HOME/mwVersion/src/omName/appName/myHelpers` directory. If the standard input sources (i.e., command line, configuration file, and environment variables) are being used by the application, then only the `ApplicationConfiguration` constructor and `defineSettings()` method need to be customized for the application's requirements. An illustration of the `defineSettings()` methods for just the "emEndpoints" configuration parameter is shown below.

```
void
ApplicationConfiguration::
defineSettings()
{
    // Add arguments that this application supports to control
    // its operations

    this->addSettings()
        ( "emEndpoints",
            TENA::Middleware::Utils::Value< std::vector< TENA::Middleware::Endpoint > >()
                .setInitializer( &TENA::Middleware::parseEndpointString ),
                "The endpoint(s) on which the primary execution manager and EM "
                "replicas, if any, are listening for requests e.g., "
                "-emEndpoints iiop://<hostname>:50000 to contact an EM running on the "
                "host <hostname> listening on port 50000. "
                "Multiple EM endpoints must to be separated by commas (if using the "
                "same protocol) or semicolons. If semicolons are used, the string "
                "may be enclosed in double quotes or the semicolon escaped with a "
                "backslash (i.e. \"\\;\\\") to avoid shell interpretation issues." )

    ; // End of the addSetting() statement
}
```

Although the above interface may look unfamiliar (due to advanced C++ programming techniques), the pattern to add additional application configuration patterns is easily followed. Between the `addSettings()` statement and the closing semicolon (';'), additional parameters can be added with the following structure, where `parameterName` and `parameterDescription` are `std::strings`.

```
( parameterName,
    TENA::Middleware::Utils::Value< TYPE >(),
    parameterDescription
)
```

The `TYPE` used for the parameter setting can be any built-in or `TENA` data type (e.g., `std::string`, `TENA::unit32`). If the `TENA::Middleware::Utils::Value` line is omitted, then the setting type will default to a "flag" setting which does not have a corresponding value, but implicitly uses a `bool` value to indicate whether the flag type parameter was set. User defined parameter types can be defined and are covered in the API Reference section of this documentation.

If the parameter uses a default value, then the inserter syntax is appended to invoke the `setDefault()` method on the `Value` object, as shown below:

```
( parameterName,
    TENA::Middleware::Utils::Value< TYPE >().setDefault( defaultValue ),
    parameterDescription
)
```

Additionally, configuration parameters can use initializer functions that can be used to check that a particular configuration value is legal. For example, a configuration value for a duration may need to check that a negative value is not supplied. Built-in functions for specifying a "min-max" value range can be accomplished with the template class `TENA::Middleware::Utils::RangeChecker` that can be used for arbitrary types that support inequality operators '<' and '>'. An example of specifying a range checking initializer is shown in the code fragment below. Additional information on the use of an initializer function can be found in the API section of this documentation.

```
( "connectionTimeoutInMilliseconds",
TENA::Middleware::Utils::Value< TENA::uint32 >().setDefault( 10000 )
.setInitializer(
    TENA::Middleware::Utils::RangeChecker< TENA::uint32 >(1, INT_MAX ) ),
"Timeout duration (in milliseconds) used by the middleware when "
"establishing new network connections. Users need to take into "
"consideration the expected network latency when setting this value "
"to ensure that the value is not too small, as well as recognizing that "
"too large of a value may introduce application delays in attempting to "
"establish connections when there are communication faults. The default "
"value is 10000 (10 seconds)." )
```

A similar `addSetting()` method can also be used to add a single parameter setting at a time. The syntax of the `addSetting()` method is shown below where if the `TENA::Middleware::Utils::Value` line is omitted, the parameter setting is defined to be a "flag" setting which does not have a corresponding value, but implicitly uses a `bool` value to indicate whether the flag type parameter was set. The `isOptional` argument is a `bool` value that indicates that the parameter setting is optional, and this argument defaults to `false` if not specified. The `isPassword` argument is a `bool` value that indicates the parameter value is a password and shouldn't be displayed.

```
this->addSetting( parameterName,
    TENA::Middleware::Utils::Value< TYPE >(),
    parameterDescription,
    isOptional,
    isPassword);
```

In the constructor of `ApplicationConfiguration`, any configuration parameters whose values require range or validity checking should be examined and a `TENA::Middleware::ConfigurationError` exception should be thrown if an error is encountered. A method on the `Setting` object, `getShortUsageString()` can be used in the exception/error message to list the usage syntax with respect to the configuration parameters.

If the application utilizes non-standard input sources or wants to limit the input sources, then additional (or alternative) constructors for `ApplicationConfiguration` need to be added, such as a `KeyValueList` constructor in the case of an application that programmatically defines configuration values or utilizes a GUI for the operator to define the values. In this situation, the `ApplicationConfiguration` constructor should invoke the appropriate `TENA::Middleware::ApplicationConfiguration` constructor in the declaration. The constructor should follow the processing performed in the default constructor that is automatically generated.

ApplicationConfiguration Parsing and Value Retrieval

Once an application has defined the appropriate `ApplicationConfiguration` class, it is necessary to instantiate an `ApplicationConfiguration` object within the application using the appropriate constructor. An `ApplicationConfiguration` contains a `TENA::Middleware::Configuration` and both objects, by default, perform the necessary parsing of input sources to define all of the configuration values during construction. If the configuration objects have been customized to defer parsing to outside of the constructor, then the application is responsible for invoking the appropriate parsing methods (e.g., `parse configuration file, parse KeyValueList`) prior to retrieving the parameter values.

Since the default behavior is to perform the parameter value range and validity checking in the `Configuration` constructor, if parsing is performed outside of the constructor, the parameter values need to be re-examined with respect to range or validity checking. Checking for unknown parameters after performing parsing outside the constructor may also be prudent to detect parameter spelling errors or similar problems.

An illustration of creating an `ApplicationConfiguration` object and using it to initialize the middleware and join an execution is shown in the code fragment below. Note that the `TENA::Middleware::Configuration` is obtained through the `tenaConfiguration()` method and used in the `TENA::Middleware::init()` function to initialize the middleware with the middleware configuration parameter values.

```

ApplicationConfiguration appConfig(
    argc,
    argv,
    "", // Config file ("Example-Person-Publisher-vRC.1-1.config" to use example)
    "Example-Person-Publisher-v1", // Config file prefix
    "" ); // Application environment variable prefix

TENA::Middleware::RuntimePtr pTENAmiddlewareRuntime(
    TENA::Middleware::init( appConfig.tenaConfiguration() ) );

std::vector< TENA::Middleware::Endpoint > endpointVector(
    appConfig["emEndpoints"].getValue<
        std::vector< TENA::Middleware::Endpoint >>() );

// Join the Execution. Hold onto the ExecutionPtr to indicate
// participation in the Execution. Let go of it to leave the Execution.
TENA::Middleware::ExecutionPtr pExecution(
    pTENAmiddlewareRuntime->joinExecution( endpointVector ) );

```

As shown in the code fragment above, the application can obtain the value of a particular configuration parameter by using the vector ('[]') operator with the string name of the parameter and the `getValue()` method template. Templates are used to ensure that the correct data type is used for the parameter. Additional examples of retrieving the parameter values are shown in the code fragment below.

```

TENA::uint32 const microsecondsPerIteration( 1000 *
appConfig["millisecondsPerIteration"].getValue< TENA::uint32 >() );

TENA::uint32 verbosity(
appConfig["verbosity"].getValue< TENA::uint32 >() );

std::string publisherSelections(
appConfig["publish"].getValue< std::string >() );

std::string subscriberSelections(
appConfig["subscribe"].getValue< std::string >() );

```

Additional details on the use of the Configuration and Setting classes is provided in the following section, along with access to complete working examples based on automatically generated code for example object models.

Copying a Configuration

The `BasicConfiguration`, `Configuration`, and `ApplicationConfiguration` classes all define protected copy constructors. In addition, `BasicConfiguration` declares a private assignment operator. Doing so eliminates inadvertent copying of configurations and reduces the risk of accidentally "slicing" the polymorphic configuration class.

Instead, `TENA` configuration follows the Prototype pattern. Each specialization of `ApplicationConfiguration` (e.g., `MyConfiguration`) may define a private implementation of the virtual `cloneImpl()` method as shown below.

```

MyConfiguration * cloneImpl() const
{
    return new MyConfiguration( *this );
}

```

The compiler generated copy constructor for `MyConfiguration` will suffice in most cases. To make a copy of `MyConfiguration`, an application can now do the following.

```

MyConfiguration originalConfig( argv, argc );
...
std::unique_ptr<MyConfiguration> apMyConfigCopy(
    boost::polymorphic_downcast< MyConfiguration * >( originalConfig.clone().release() ) );
// Construct a "smart pointer" from the unique_ptr
TENA::Middleware::Utils::SmartPtr< myConfiguration > myConfigCopyPtr( std::move( apMyConfigCopy ) );

```

Note that `boost::polymorphic_downcast` performs a static cast to the specified type in release mode when `NDEBUG` is defined. For debug builds, it is a dynamic cast that will fail an assertion if the cast fails. While this syntax is verbose, it provides assurance that the classes are being cast appropriately. The `MyConfiguration` class can also overload the non-virtual `clone()` method as shown below.

```

std::unique_ptr< MyConfiguration >
MyConfiguration:::
clone() const
{
    std::unique_ptr< MyConfiguration > result( this->cloneImpl() );
    return result;
}

```

Doing so eliminates the need for casting the result of the `clone()` invocation for `MyConfiguration`.

C++ API Reference and Code Examples

Example Applications

Any automatically generated example application will include a working example of an `ApplicationConfiguration`. The following files illustrate the key classes associated with the example application for publishing `Example::Vehicle` objects. Developers are encouraged to review the operation of these classes within a working example application, such as `Example-Vehicle-Publisher`. A number of automatically generated example applications for the Example and TENA standard object models are included with the TENA Middleware distribution package and installed in the `TENA_HOME/version/src` directory structure.

Filename Link	Description
<code>applicationConfiguration</code>	Automatically generated <code>ApplicationConfiguration</code> class that includes application specific parameters.
<code>Example-Vehicle-v1-Vehicle-Publisher-v2.cpp</code>	Automatically generated main program that uses <code>ApplicationConfiguration</code> object.
<code>TENA/Middleware/ApplicationConfiguration</code>	Base class used for defining application specific <code>ApplicationConfiguration</code> classes.
<code>TENA/Middleware/Configuration</code>	Middleware configuration class.
<code>TENA/Middleware/Utils/BasicConfiguration</code>	Base configuration class that provides basic capabilities for derived configuration classes. Not recommended for direct use.
<code>TENA/Middleware/Utils/Setting</code>	Class used to define a particular configuration parameter (aka setting) characteristics.

ApplicationConfiguration Constructors

The automatically generated `ApplicationConfiguration` constructor declaration is shown below, along with the declaration of the `TENA::Middleware::ApplicationConfiguration` base class constructor. The arguments associated with these constructors are described below as well.

```

ApplicationConfiguration(
    int & argc,
    char * argv[],
    std::string const & configFilePathname,
    std::string const & configFilePrefix,
    std::string const & envPrefix,
    std::string const & usageHeader,
    std::string const & usageFooter = "",
    TENA::Middleware::Utils::Setting::SourceType source =
        TENA::Middleware::Utils::Setting::SourceTypeCommandLine,
    bool configFileRequired = true );

```

```

ApplicationConfiguration(
    std::string const & configFilePathname,
    std::string const & configFilePrefix,
    std::string const & envPrefix,
    std::string const & usageHeader,
    std::string const & usageFooter = "",
    bool configFileRequired = true );

```

```

ApplicationConfiguration(
    TENA::Middleware::Utils::BasicConfiguration::KeyValueList & optionsList,
    std::string const & usageHeader,
    std::string const & usageFooter = "",
    TENA::Middleware::Utils::Setting::SourceType source =
        TENA::Middleware::Utils::Setting::SourceTypeApplication );

```

The `argc` and `argv` arguments are associated with the command line arguments used to invoke the application and typically can be obtained through an operating system or compiler provided function. In some cases, an application may not have access to the command line arguments and `argc` can be set to a value of zero and `argv` can be an empty `char *` vector.

Argument `configFilePathname` defines the name and location of the configuration file, if one is being used. An empty string is used to indicate that no configuration file should be used for parsing. If the `configFilePathname` does not provide an absolute file location, it is assumed to be relative to the location of the executable.

The `configFilePrefix` is used to indicate the configuration file prefix used for the application specific parameters defined in the configuration file. For example, a single configuration file can be used to define the configuration parameters for an Execution Manager and several applications. In order to ensure that the appropriate values are used for the appropriate application, a prefix can be used to apply parameter values with the same name to the correct application.

An example configuration file supporting the Execution Manager, App1, and App2 is shown below. Execution Manager configuration parameters need to use the prefix "executionManager" when defined within a configuration file. The `ApplicationConfiguration` created for App1 will need to use the "App1" prefix and App2 will need to create an `ApplicationConfiguration` using the "App2" prefix. Note that in the example configuration file, the parameter "emEndpoints" is not qualified with a prefix. This allows a single specification of this parameter that will be used by both App1 and App2. Also note that the "noErrorLog" parameter does not specify any value. That is because `noErrorLog` is a flag type parameter which doesn't take a value. Including an equal sign after a flag type parameter name is also valid.

```

# Configuration parameter values for EM, App1, and App2
executionManager.listenEndpoints = machine0:50000
App1.listenEndpoints = machine1:50001
App2.listenEndpoints = machine2:50002
emEndpoints = machine0:50000
noErrorLog

```

Constructor argument `envPrefix` is used when the application uses a prefix for application specific configuration parameters that can be defined through environment variables. An environment variable prefix can be used to organize parameters associated with different applications, with the prefix used for the TENA Middleware being "TENA_MW_". When searching the configuration parameters for a match to a particular environment variable, the prefix is removed, any underbar ("_") characters are removed, and the resulting letters are converted to lower case. Thus, the environment variable "TENA_MW_CONNECTION_TIMEOUT_IN_MILLISECONDS" can be used to set the middleware parameter "connectionTimeoutInMilliseconds". The default prefix for application environment variables is an empty string which implies no prefix is used for the environment variable names, so the environment variable name "NUMBER_OF_ITERATIONS" could be used for the configuration parameter named "numberOflterations".

The base class `TENA::Middleware::Configuration` constructor has several additional constructor arguments, `usageHeader`, `usageFooter`, and `source`. The usage header and footer strings are displayed before and after the usage statement associated with the application configuration parameters. A default usage header of "Application Options:" is used to separate middleware configuration parameters from application parameters in the usage statement.

The `source` argument is used to indicate the input source of the `argc`, `argv` parameters, which defaults to `SourceTypeCommandLine`. The `source` type is an enumeration with possible values of `SourceTypeUnknown`, `SourceTypeDefault`, `SourceTypeCommandLine`, `SourceTypeFile`, `SourceTypeEnvironment`, `SourceTypeApplication`, `SourceTypeOther`, and `SourceTypeAll`. The `source` argument is used in conjunction with the TENA Console to allow an operator to determine the source of a particular configuration value.

TENA::Middleware::Configuration Constructors

Several constructors for the `TENA::Middleware::Configuration` class are defined to permit flexibility in the input sources used for obtaining middleware configuration values as shown below. These constructors can be used as examples if developers need to define similar `ApplicationConfiguration` constructors.

- **Standard Input Sources**—Used to parse the standard input sources (i.e., command line, configuration file, and environment variables) for the values for the middleware configuration parameters (in the precedence order listed). The `prefix` is used for the configuration file processing and the `SourceType` indicates the source of the `argv` values, which is generally the command line. `Source type` is used support TENA Console operations in determining the particular input source of a configuration value.

```

Configuration(
    int & argc,
    char** argv,
    std::string const & filename = "",
    std::string const & prefix = "",
    TENA::MiddlewareUtils::Setting::SourceType =
        TENA::MiddlewareUtils::Setting::SourceTypeCommandLine );

```

- **Key Value List**— Used when the application manually defines the middleware configuration parameters using the `TENA::Middleware::BasicConfiguration::KeyValueList` mechanism. This mechanism may be used when the application developer uses an alternative input mechanism for the operator, such as a GUI.

```

Configuration(
    TENA::MiddlewareUtils::BasicConfiguration::KeyValueList & optionsList,
    TENA::MiddlewareUtils::Setting::SourceType =
        TENA::MiddlewareUtils::Setting::SourceTypeCommandLine );

```

- **Configuration File Only**— Used when only a configuration file should be used for obtaining middleware configuration values.

```

Configuration(
    std::string const & filename,
    std::string const & prefix );

```

- **Environment Variables Only**— Used when only environment variables should be used for obtaining middleware configuration values.

```
Configuration();
```

Key `TENA::Middleware::ApplicationConfiguration` Methods

The `TENA::Middleware::ApplicationConfiguration` class is used as a base class for application specific configuration classes. The automatically generated example applications use such a class named `ApplicationConfiguration`. Users are encouraged to begin with this class for customizing for their particular needs.

Two parse methods are defined to allow an application to parse the input source for values associated with the configuration parameter settings. The two parse methods are defined below, where the first method uses the `argc/argv` input source, and the second method uses a `KeyValueList` (which is a list of string pairs).

Applications that don't use the standard input sources, or need to augment those sources, can use the `KeyValueList` to create a list of "key value" pairs. The key is a `std::string` containing the configuration parameter name and the value of the parameter is captured in a `std::string`. The `KeyValueList` provides a convenient mechanism to exchange configuration parameters and values between a configuration object and alternative mechanism for managing the parameters, such as software that allows an application operator to set configuration parameter values through a graphical user interface.

```

virtual void
parse(
    int & argc,
    char * argv[],
    TENA::MiddlewareUtils::Setting::SourceType source =
        TENA::MiddlewareUtils::Setting::SourceTypeCommandLine );

typedef std::pair< std::string, std::string > KeyValuePair;
typedef std::list< KeyValuePair > KeyValueList;

virtual void
parse(
    BasicConfiguration::KeyValueList & optionList,
    TENA::MiddlewareUtils::Setting::SourceType source =
        TENA::MiddlewareUtils::Setting::SourceTypeApplication );

```

 — Whenever parsing is performed to obtain parameter values, any range and validity checking needed for the parameter values should be performed. Additionally, checking for unknown parameters (as discussed below) should be performed.

Applications can obtain the contained `TENA::Middleware::Configuration` object that is contained within the `TENA::Middleware::ApplicationConfiguration` object. The middleware configuration object is contained within the application configuration to simplify the parsing of both configuration parameters using the same input sources. Applications will use this method to obtain the middleware configuration to pass into the middleware initialization function.

```
TENA::Middleware::Configuration & tenaConfiguration();
```

When configuration files or environment variables are parsed to match the parameters associated with a configuration, it may be beneficial to use a configuration specific prefix for the file entries and environment variables. Using a prefix permits separation of parameters that may be used for different applications. When the parsing finds a configuration file entry or environment variable with a matching non-blank prefix, and that parameter is not specified by the configuration, then the parameter is marked as being unknown. The unknown parameters may indicate that an operator has made a spelling error or similar problem. The methods to obtain a `KeyValueList` of these unknown parameters are shown below. In addition to the unknown configuration file and environment variable prefixed parameters, the `argv` vector can be examined after the configuration parsing has been completed since any remaining (i.e., unconsumed) command line arguments may indicate a spelling error or similar problem.

```
TENA::MiddlewareUtils::BasicConfiguration::KeyValueList & unknownEnvironmentOptions();
TENA::MiddlewareUtils::BasicConfiguration::KeyValueList & unknownFileOptions();
```

The automatically generated `ApplicationConfiguration` constructor illustrates how this unknown parameter processing can be performed, as shown in the code below.

```
// Check for unused arguments.
if ( argc > 1 ) // Allow 1 argument for the program name
{
    showUsage = true;
    std::cerr << "Warning: The following command line options were not "
    "recognized:\n" << TENA::MiddlewareUtils::ARGV( argc-1, &(argv[1]) ) << std::endl;
}

if ( this->unknownEnvironmentOptions().size() > 0 )
{
    showUsage = true;
    TENA::MiddlewareUtils::ARGV argVector;
    TENA::MiddlewareUtils::toArgv( argVector, this->unknownEnvironmentOptions() );
    std::cerr << "Warning: The following environment variable options "
    "were not recognized:\n" << argVector << std::endl;
}

if ( this->unknownFileOptions().size() > 0 )
{
    showUsage = true;
    TENA::MiddlewareUtils::ARGV argVector;
    TENA::MiddlewareUtils::toArgv( argVector, this->unknownFileOptions() );
    std::cerr << "Warning: The following file options "
    "were not recognized:\n" << argVector << std::endl;
}

if ( showUsage )
{
    std::cerr << "\nUsage: Example-Person-Publisher-vRC.1-1 "
    << this->getShortUsageString()
    << "\n\nUse \"-help\" for a more detailed usage message\n"
    << std::endl;
}
```

Key `TENA::Middleware::Configuration` Methods

The `TENA::Middleware::Configuration` class encapsulates the configuration parameters and processing related to the middleware configuration parameters. In addition to the constructors discussed above, this class also supports the `unknownEnvironmentOptions` and `unknownFileOptions` methods that can be used to detect spelling or similar errors with the middleware parameters.

```
TENA::MiddlewareUtils::BasicConfiguration::KeyValueList & unknownEnvironmentOptions();
TENA::MiddlewareUtils::BasicConfiguration::KeyValueList & unknownFileOptions();
```

Key TENA::MiddlewareUtils::BasicConfiguration Methods

TENA::MiddlewareUtils::BasicConfiguration is the base class for all configuration options. It is not intended to be used directly for application purposes because the automatically generated ApplicationConfiguration class attempts to provide an easier to use structure for most application configuration needs, although applications can use BasicConfiguration directly if they need to provide low-level configuration control within their application.

The BasicConfiguration class manages parameter Setting objects for a particular configuration. Each Setting contains characteristics about the parameter and the value, if one exists. Methods on the TENA::MiddlewareUtils::BasicConfiguration class for working with Settings are shown below. Note that a copy of the Setting is made with the addSetting() method, so the application does not need to hold onto the Setting after it has been added to the configuration. Likewise, getSetting() and vector operator ('[]') returns a copy of the Setting, if found. Method getSetting() and vector ('[]') operator will throw a std::runtime_error exception if the parameter setting is not found.

```
void
addSetting( Setting const & s );

Setting const &
getSetting( std::string const & settingName,
    TENA::uint32 flags = CASE_SENSITIVE ) const;

Setting const &
operator[] ( std::string const & settingName ) const;
```

The getSetting() method, as well as other BasicConfiguration methods utilize several constants as method arguments to control the processing behavior. Flags CASE_SENSITIVE and CASE_INSENSITIVE can be used to indicate whether the string used for comparison when assigning and accessing parameters should be case sensitive or not. By default, the parameter names are case sensitive. Parameter settings can be defined as visible or hidden (VISIBLE, HIDDEN) to control whether the parameter is displayed in the usage print statements. A parameter can also be designated as SETTING_OPTIONAL to indicate that it is not required for an operator to provide a value for that parameter. The validate() method checks to ensure that values exist for all required parameters.

```
static const TENA::uint32 CASE_SENSITIVE    = 0x0000;
static const TENA::uint32 CASE_INSENSITIVE = 0x0001;

static const TENA::uint32 VISIBLE = 0x0000;
static const TENA::uint32 HIDDEN   = 0x0002;

static const TENA::uint32 SETTING_OPTIONAL = 0x0004;
```

When adding multiple parameter settings to a configuration, the addSettings() method returns a SettingInserter that can be used to define the settings in a more compact manner. The addSettings() method takes the previously discussed processing flags that are applied to all settings added using the returned inserter. Operation of the addSettings() method is illustrated in the Usage Considerations section of this documentation.

```
SettingInserter
addSettings( TENA::uint32 flags = VISIBLE );

SettingInserter &
operator() (std::string name,
    AbstractValue const & valueSemantic,
    std::string documentation);

SettingInserter &
operator() (std::string name,
    AbstractValue const & valueSemantic,
    std::string documentation,
    bool passwordValue);

SettingInserter &
operator() (std::string name,
    std::string documentation);
```

Developers can modify the values of parameter Settings that have already been added to the configuration through the setSettingValue() and setsettingValues() methods. As mentioned elsewhere, developers are responsible for ensuring that the parameter values adhere to range and validity constraints. The setSettingValue() will throw a std::runtime_error exception if the named parameter is not found. Method setSettingValues() will return any key value pairs from the valueList argument that are not found, so successfully processed key value pairs will be consumed from the keyValueList. Developers can check the KeyValueList after the setSettingsValues() method has been invoked for unknown parameters.

```

virtual void
setSettingValue( std::string name,
    std::string value,
    Setting::SourceType source =
        Setting::SourceTypeUnknown,
    TENA::uint32 flags = CASE_SENSITIVE );

virtual void
setSettingValues( KeyValueList & valueList,
    Setting::SourceType source =
        Setting::SourceTypeUnknown,
    TENA::uint32 flags = CASE_SENSITIVE );

```

Developers can check if a configuration has a value defined for all of the required parameters by using the `validate()` method. This method returns a boolean value of `true` if the configuration is properly set (with respect to required parameters).

```

virtual
bool
validate( std::string & errMsg ) const;

```

Developers can obtain the existing parameter names and values as a vector of strings, or as a single string. This format follows the command line convention with a hyphen ('-') before the parameter name and followed by the value (if applicable).

```

virtual
std::vector< std::string >
getAsCommandLine() const;

virtual
std::string
getCommandLineString () const;

```

Several print related methods are available with the `BasicConfiguration`. The `printUsage()` method lists each parameter with the full description, constrained to the character width specified. Method `getShortUsageString()` only provides a single line usage message for the parameter settings.

```

virtual
void
printUsage( std::ostream & os,
    size_t width = 79 ) const;

virtual
std::string
getShortUsageString() const;

```

All of these methods are available to the derived configuration classes (e.g., `TENA::Middleware::Configuration`, `TENA::Middleware::ApplicationConfiguration`), although developers may want to specialize some of the behavior for their application specific configuration class.

Key Setting Methods

As mentioned in the description of the base configuration class `BasicConfiguration`, the underlying representation for parameter settings is captured in the `Setting` class. The constructors for this class are shown below in which the two versions differ with the `AbstractValue` argument.

```

Setting(std::string name,
    AbstractValue const & valueSemantic,
    std::string documentation,
    bool isSpecifiedAsOptional = false,
    bool isPasswordValue = false );

Setting(std::string name,
    std::string documentation,
    bool isSpecifiedAsOptional = false,
    bool isPasswordValue = false );

```

Basic "setters" and "getters" for this class are shown below. The `getDefaultValue()` method throws an `std::runtime_error` exception if a default value does not exist. Note that the `isRequired()` and `requiresValues()` methods are independent because a "flag" parameter does not have a value, but it may be required.

```
std::string const & getName() const;

bool isSet() const;
template< class T > T getValue() const;

bool hasDefault() const;
template< class T > T getDefaultValue() const;

std::string const & getDocumentation() const;

boolisRequired() const;
bool requiresValue() const;

Setting & setHidden( bool isHidden = true );
bool isHidden() const;

bool isPassword() const;

std::string getUsageString() const;
std::string getTypeString();
```

Key value Methods

A template class, `Value`, is used to store the value of the parameter setting. This class ensures that the string representation is properly processed to obtain the specific parameter type. When a `Setting` is created, the appropriate parameter type is used in the `Value` constructor as shown in the declaration statement below.

The `setDefault()` method illustrates the use of a default value that is associated with the `Value`. If this `Value` is used with a parameter and the user does not specify a parameter value, then the default value will be used for that parameter.

```
TENA::MiddlewareUtils::Value< TENA::uint32 > someValue;
someValue.setDefault(2);
```

The `Value` class is capable of supporting the necessary conversions between the string representation and type representations for built-in data types (e.g., long, double, `TENA::uint32`). In some situations, the string representation of the value may require specialized processing to convert between the type representation and the string representation.

For example, the configuration parameter `emEndpoints` used in the automatically generated `ApplicationConfiguration` class uses the type `std::vector< TENA::Middleware::Endpoints >`. The `Value` class needs an initializer function to convert from a string representation the type `std::vector< TENA::Middleware::Endpoints >`. These initializer functions follow the function signature shown below, where `T` is the particular `Value` type.

```
void functionName( std::string const & stringValue, T & typeValue );
```

As an example, the function declaration for parsing the EM endpoint string is shown below along with the use of this function in the `setInitializer()` method on the `Value` class.

```
TENA_Middleware_EXPORT
void
parseEndpointString(
    std::string const & endpointString,
    std::vector< TENA::MiddlewareUtils::Endpoint > & endpoints );

TENA::MiddlewareUtils::Value< std::vector< TENA::MiddlewareUtils::Endpoint > > emEndpointsValue;

emEndpointsValue.setInitializer( &parseEndpointString );
```

The initializer function will be used when the `initializeValue()` method is invoked on the `Value`, which is done whenever the parameter `Setting` is parsed from a input source. A `std::invalid_argument` exception will be thrown if a string parsing error occurs within the initializer function. Users are permitted to define their own initializer functions to ensure that a particular configuration parameter value is legal.

A template class that can be used for "min-max" range checking has been defined within the middleware. Users can use this class for any data type that support the inequality '<' and '>' operators. The `TENA::MiddlewareUtils::RangeChecker` class is defined in the code below.

```
template < class T >
class RangeChecker
{
public:
    RangeChecker( T const & min,
                  T const & max );

    void
    operator() ( std::string const & in,
                 T & _out );

private:
    T _min;
    T _max;
};

template< class T >
inline
void
TENA::Middleware::Utils::RangeChecker< T >::
operator() ( std::string const & in,
              T & _out )
{
    T value( boost::lexical_cast< T >( in ) );

    if (_min > value || _max < value)
    {
        std::stringstream ss;
        ss << "The given value of \\" " << in
             << "\\ is outside of allowed range ["
             << _min << "," << _max << "]";

        throw std::invalid_argument( ss.str() );
    }

    _out = value;
}
```

Java API Reference and Code Examples

`TENA::Middleware::Configuration Constructors`

As mentioned above, for the Java language binding, only the `TENA.Middleware.Configuration` is provided. It is required by the method `Runtime.init`.

Several constructors for the `TENA.Middleware.Configuration` class are defined to permit flexibility in the input sources used for obtaining middleware configuration values as shown below. These correspond to the constructors available in C++, with slight differences because of language features. See also the [javadoc](#).

- **Standard Input Sources**— Used to parse the standard input sources (i.e., command line, configuration file, and environment variables) for the values for the middleware configuration parameters (in the precedence order listed). The `prefix` is used for the configuration file processing and the `SourceType` indicates the source of the `argv` values, which is generally the command line. `Source type` is used support TENA Console operations in determining the particular input source of a configuration value.

```
// Initialize a configuration from environment variables. For every configuration setting, an environment
// variable TENAMW_<setting>, may be used to provide the value.
TENA.Middleware.Configuration()

// Initialize a configuration from strings in same form as command line arguments.
TENA.Middleware.Configuration(String[] argv)

// Initialize a Configuration from strings and from a file. {{prefix}} allows for a single configuration
// file to be shared among applications. Settings in the file with "<prefix>." prepended to the setting
// will only be processed into a configuration with the matching prefix (without the ".").
// specifying "appl" as the prefix, those settings will only be
TENA.Middleware.Configuration(String[] argv, String filename)
TENA.Middleware.Configuration(String[] argv, String filename, String prefix)

// Initialize a configuration from a file only.
TENA.Middleware.Configuration(String filename, String prefix)
```

Related Topics and Links

- [Middleware Configuration Parameters](#)
- [Execution Manager Configuration Parameters](#)

Middleware Configuration Parameters

Middleware Configuration Parameters

The TENA Middleware linked with every TENA application supports a collection of configuration parameters that can be adjusted by the application operator to control the runtime behavior of the middleware. Where possible, the default value for the middleware configuration parameters have been selected to provide optimal behavior for the most typical operating environments.

Description

The configuration parameters available for the TENA Middleware are defined in the following table. Refer to the [configuration mechanism](#) documentation for how these parameters values can be set.

— The default middleware parameter values are usually appropriate for typical environments in which TENA applications are distributed across a wide-area network. If, however, all TENA applications will be connected on a single local area network (LAN) and the network is configured correctly, then parameters can be adjusted (typically by shortening timeouts and reducing numbers of retry attempts) for optimal performance. See documentation for [recommended settings for a stable, working, LAN environment](#).

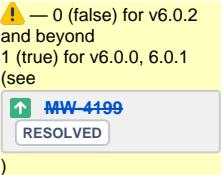
Middleware Configuration Parameters

Configuration Parameter	Description	Default Value	Data Type
listenEndpoints	The endpoint(s) on which this process should listen for requests, e.g., use -listenEndpoints <hostname>:55100 to listen on port 55100 on the host <hostname>. The port value is optional and it is recommended to allow the middleware to select the port value. It may be helpful to specify the allowable port range using the portspan option, such as "192.168.1.1:55100/portspan=10". Note: IANA (www.iana.org) recommends using ports in the range 49152 - 65535 for private uses.	User required.	std::vector< TENA::Middle ware::Utils: :Endpoi nt >
configFile	The name of the configuration file to use for obtaining configuration values.	No file used.	std::string
optionalConfigFile	The name of a configuration file to be used for obtaining configuration values, but unlike the configFile parameter, the optionalConfigFile is not required to exist.	No optional file is used.	std::string
configFilePrefix	The configuration file prefix to use for parameters associated with this application.	No prefix used.	std::string
applicationName	This configuration option is used to specify the application name as it appears in log file names and the TENA console.	Defaults to argv[0] if provided.	std::string
logDir	The directory for diagnostic and error log files.	The directory is \$HOME/TENA/log except on Windows where it is %APPDATA%\TENA\log. If the directory is not writable or if the \$HOME (or %APPDATA%) environmental variable is not set, then the logDir is required to be set by the user. ! — Prior to TENA Middleware version 6.0.9, the default was \$TENA_HOME/\$TENA_VERISON/log or the current working directory if that directory is not available.	std::string

noErrorLog	Disable error log file (messages will only go to std::cout).	not set	Parameter is a "flag" type, in which there is not an explicit data type used; the parameter is either set or not set.
multicastInterface	Interface to use for sending and receiving \"best effort\" (UDP multicast) data. On UNIX machines, the interface device name (e.g., eth0) can be used, and on Windows, the hostname or IP address for the desired interface should be used.	Undefined.	std::string
multicastTTL	The multicast Time-to-Live for \"best effort\" data. When using TENA::Middleware::BestEffort, i.e., IP multicast, to transmit data across multiple LANs, the routers that control the exchange of data between the LANs must be configured to pass IP multicast. As a protection against mistaken router configurations (e.g., router loops), every IP multicast datagram is tagged with a Time to Live (TTL) value. Legal values are 0 <= multicastTTL <= 255.	16 ! — Changed in TENA Middleware release 6.0.5.1 (previously defaulted to value of 1).	TENA::uint16
multicastSendBufferSize	Size, in bytes, of the send buffer for the \"best effort\" multicast socket. This controls the size of the OS buffer used to hold data to be transmitted on an IP multicast socket. Since IP multicast is \"best effort\" (i.e., unreliable), attempts to transmit data faster than the network can support will result in a buffer overrun and data will be lost in some OS-specific way (most OSes will discard the entire buffer). Legal values are 1024 <= multicastSendBufferSize <= 8388608, i.e., from 1 KB to 8 MB.	131072, i.e., 128 KB.	size_t
multicastReceiveBufferSize	Size, in bytes, of the receive buffer for the \"best effort\" multicast socket. This controls the size of the OS buffer used to hold data received on an IP multicast socket. Since IP multicast is \"best effort\" (i.e., unreliable), if data arrives faster than it is processed, then the buffer will be overrun and data will be lost in some OS-specific way (most OSes will discard the entire buffer). Legal values are 1024 <= multicastReceiveBufferSize <= 8388608, i.e., from 1 KB to 8 MB.	131072, i.e., 128 KB.	size_t
bestEffortFragmentSize	Size, in bytes, at which the middleware will fragment best effort traffic. Must be less than 65536, since IP will fail above that limit.	62000 (bytes)	size_t
connectionTimeoutInMilliseconds	Timeout duration (in milliseconds) used by the middleware when establishing new network connections. Users need to take into consideration the expected network latency when setting this value to ensure that the value is not too small, as well as recognizing that too large of a value may introduce application delays in attempting to establish connections when there are communication faults. A value of 0 disables the timeout, effectively setting the timeout to infinity (which is not recommended).	10000 (10 seconds)	TENA::uint32
twoWayTimeoutInMilliseconds	Timeout duration (in milliseconds) used by the middleware when invoking a two-way method. This does not include two-way SDO method invocations. It only applies to various internal middleware methods. A value of 0 disables the timeout, effectively setting the timeout to infinity (which is not recommended).	120000 (2 minutes)	TENA::uint32
enableStructuredExceptionTranslation	Enable translation of structured exceptions on Windows to std::exception. This option is useless except under Windows. This option is ignored for any non-Microsoft application. In a Microsoft application, several types of application runtime errors (e.g., de-referencing an invalid pointer) will raise what Microsoft calls \"structured exceptions\". These exceptions are incompatible with standard C++ exceptions (e.g., they can only be caught in a catch (...) block) and must be translated into a meaningful C++ exception. This option enables the translation of Microsoft structured exceptions into a C++ exception class named TENA::Middleware::Utils::StructuredException. Unfortunately, translating Microsoft structured exceptions hampers debugging a C++ application with Microsoft's debugger. ! — NOT available in TENA Middleware version 6.0.3 and beyond.	not set	Parameter is a "flag" type, in which there is not an explicit data type used; the parameter is either set or not set.

enableTENAdebug	Enable TENA_DEBUG messages that may be useful for investigating application or middleware problems. TENA_DEBUG messages are only displayed when running a debug build application and not in opt builds.	not set	Parameter is a "flag" type, in which there is not an explicit data type used; the parameter is either set or not set.
enableDIMEtrace	Enable DIME trace messages that may be useful for investigating application or middleware problems related to low level middleware processing.	not set	Parameter is a "flag" type, in which there is not an explicit data type used; the parameter is either set or not set.
disableAlertOnError	Disables sending of alert error messages to consoles.	not set	Parameter is a "flag" type, in which there is not an explicit data type used; the parameter is either set or not set.
disableAlertOnWarning	Disables sending of alert warning messages to consoles.	not set	Parameter is a "flag" type, in which there is not an explicit data type used; the parameter is either set or not set.

disableAlertOnNote	Disables sending of alert note messages to consoles.	not set	Parameter is a "flag" type, in which there is not an explicit data type used; the parameter is either set or not set.
popupWindowsDisabled	Used to disable the Widows error popup mechanism which can interfere with the operation of a distributed system. ⚠ — Available in TENA Middleware version 6.0.3 and beyond.	Windows error popups are disabled for "release" builds (1) and enabled for "debug" builds (0).	TENA::bool
windowsPeriodicTimerResolutionInMilliseconds	Windows, by default, performs timer interrupts at 64 Hz, so the standard timer resolution is 15.625 milliseconds. Setting this value to between 1 and 15 increases the timer resolution. The timer resolution defaults to 1 which allows increases the timer resolution. Setting this value to 0 leaves the Windows system timer untouched. The timer resolution defaults to 1 which allows accurate timing but can degrade overall system performance. On non-Windows platforms, this option is ignored. ⚠ — Available in TENA Middleware version 6.0.5 and beyond.	1	TENA::uint32
dispatchDurationInSeconds	Duration (in seconds) to dispatch middleware processing during creation of an SDO servant (or messageSender) to allow subscription interests to propagate before initial servant updates (or messages) can be sent to subscribers. When a subscribing application declares their interest in a type of object, there is no means to determine how long it will take for that interest information to be reflected back to an application that publishes that type. Generally, only a "very small" amount of time is required. Regardless of the time required, a race condition exists. Starting the subscribing application before starting the publishing application does not guarantee that the subscriber's interest will be reflected in the publishing application before it creates and updates an object (or sends a message) of the given type. When this happens, the subscriber will "miss out" on the initial update(s) or message(s). This parameter exists to mitigate this race condition by allowing more time for subscriber interests to arrive at the publishing application before it creates the first (and only the first) object instance or message of a given type.	0.0	TENA::float64
corbaTransientRetries	Number of times to retry sending data to a remote application when a CORBA::Transient exception occurs. Transient communication errors are an unfortunate occurrence in any networked computer system. When a transient error occurs during a network operation, a decision must be made about how many times to re-try the operation. ⚠ — Deprecated in TENA Middleware version 6.0.3 and beyond. Use the transientCommunicationAttempts parameter instead.	3	long
transientCommunicationAttempts	Number of attempts that are made to send data to a remote application when a CORBA::Transient exception occurs. Transient communication errors are an unfortunate occurrence in any networked computer system. When a transient error occurs during a network operation, a decision must be made about how many times to re-try the operation. ⚠ — Available in TENA Middleware version 6.0.3 and beyond.	3	long
announceTransientRetries	Number of times to retry sending data to a remote application when an exception occurs, typically, as a result of abnormally terminated application. This option is similar to corbaTransientRetries but is specific to the number of re-tries to be made when network errors arise due to abnormally terminated applications (as opposed to momentary network glitches). In this specific case, it is reasonable to not retry. ⚠ — Deprecated in TENA Middleware version 6.0.3 and beyond. Use the transientAnnounceAttempts parameter instead.	0	TENA::int32
transientAnnounceAttempts	Number of attempts that are made to send data to a remote application when an exception occurs, typically, as a result of abnormally terminated application. This option is similar to transientCommunicationAttempts, but is specific to the number of attempts made when network errors arise due to abnormally terminated applications (as opposed to momentary network glitches). In this specific case, it is reasonable to make no additional attempts. ⚠ — Available in TENA Middleware version 6.0.3 and beyond.	0	TENA::int32
cannotContactEMretryTimeLimitInSeconds	Amount of time to retry sending data to an execution manager event channel when a CORBA::Transient exception occurs. Transient communication errors are an unfortunate occurrence in any networked computer system. When a transient error occurs while sending an execution event, a CannotContactExecutionManager exception will be thrown after attempting to send for an amount of time specified by this value.	60	unsigned long
requestUserOneWayAck	Whether to request an acknowledgment on oneway SDO method invocations.	0 (false)	TENA::int32

requestPublisherAck	Whether to request an acknowledgment on the following kinds of messages: SDO discovery, SDO destruction, SDO tag change, MessageSender creation, and MessageSender destruction.		TENA::int32
requestDataAck	Whether to request an acknowledgment on SDO updates and messages. Note that this only applies to reliable transport. Note that enabling this option is will impose a high runtime cost.	0 (false)	TENA::int32
waitForMissingEventsIntervalInSeconds	The interval (in seconds) during which the middleware will wait for a missing event before giving up and dealing with events that have arrived in the meantime.	11 (seconds)	long
numORBthreads	The number of threads to devote to running the ORB (Object Request Broker) within the middleware.	16	TENA::uint32
ftPrimaryIDwaitIntervalInSeconds	Duration (in seconds) in which an application will wait to be informed of a new primary EM after encountering a communication fault attempting to contact the current primary EM and sending an alert to the secondary EMs.	10 (seconds)	TENA::uint32
ftPrimaryIDmaxWaitCount	Number of attempts that an application will report a communication fault in attempting to contact the primary EM. When an application encounters a communication fault with the primary EM, it sends an alert in an attempt to have a secondary EM take over as the primary. If there is no response to that alert, the application will re-send the alert the number of times indicated by this parameter.	12	TENA::uint32
mwPushTimeoutInMilliseconds	Timeout duration (in milliseconds) used by the middleware when invoking various internal middleware methods.	20000 (20 seconds)	TENA::uint32
disconnectTimeoutInMilliseconds	Timeout duration (in milliseconds) used by the middleware when disconnecting and/or deactivating internal middleware objects.	5000 (5 seconds)	TENA::uint32

The data types for the configuration parameters are shown in case an application needs to get or set the parameter values programmatically.

Hidden Parameters

This section is only visible to members of the TENA "team" (i.e., permission group `TENA-team`).

The following parameters are defined as "hidden" and are not visible to users when configuration parameters are listed. The justification for hiding the visibility of these parameters may be to avoid user confusion for parameters that should not typically be modified, or the parameters may negatively affect the behavior of the application. Please contact the TENA Development Team for additional information or consideration in using these parameters.

Hidden Configuration Parameter	Description	Default Value	Data Type
clientListenEndpoints	The endpoint(s) on which client ORB of this process should listen for requests. See help for <code>listenEndpoints</code> .	Same as <code>listenEndpoints</code> value.	<code>std::vector<TENA::Middleware::Utils::Endpoint></code>
disableAlerts	Disable sending of alerts.	not set	Parameter is a "flag" type, in which there is not an explicit data type used;the parameter is either set or not set.
disableDiagnostics	Disable embedded diagnostics.	not set	Parameter is a "flag" type, in which there is not an explicit data type used;the parameter is either set or not set.
diagnosticsProcessStats	Enable specified embedded diagnostics process statistics.	not set	Parameter is a "flag" type, in which there is not an explicit data type used;the parameter is either set or not set.
diagnosticsRuntimeStats	Enable specified embedded diagnostics runtime statistics.	not set	Parameter is a "flag" type, in which there is not an explicit data type used;the parameter is either set or not set.
diagnosticsExecutionStats	Enable specified embedded diagnostics execution statistics.	not set	Parameter is a "flag" type, in which there is not an explicit data type used;the parameter is either set or not set.

diagnosticsExecutionTypeStats	Enable specified embedded diagnostics execution by type statistics.	not set	Parameter is a "flag" type, in which there is not an explicit data type used;the parameter is either set or not set.
diagnosticsExecutionSubscriberStats	Enable specified embedded diagnostics execution by subscriber statistics.	not set	Parameter is a "flag" type, in which there is not an explicit data type used;the parameter is either set or not set.
diagnosticsSessionStats	Enable specified embedded diagnostics session statistics.	not set	Parameter is a "flag" type, in which there is not an explicit data type used;the parameter is either set or not set.
diagnosticsSessionTypeStats	Enable specified embedded diagnostics session by type statistics.	not set	Parameter is a "flag" type, in which there is not an explicit data type used;the parameter is either set or not set.
diagnosticsSessionServantStats	Enable specified embedded diagnostics session by servant statistics.	not set	Parameter is a "flag" type, in which there is not an explicit data type used;the parameter is either set or not set.
diagnosticsRemoteMethodStats	Enable specified embedded diagnostics remote method statistics.	not set	Parameter is a "flag" type, in which there is not an explicit data type used;the parameter is either set or not set.
disableDiagnosticsArchiving	Disable archival of embedded diagnostics information for deleted objects.	not set	Parameter is a "flag" type, in which there is not an explicit data type used;the parameter is either set or not set.
diagnosticsDumpIntervalInSeconds	Time interval in seconds to dump the collected embedded diagnostics data.	0	TENA::uint32
requestHandshakeAck	Whether to request an ack on internal Middleware oneway invocations.	⚠ — true for 6.0.4 and beyond. 1 (true) for earlier versions.	⚠ — bool for 6.0.4 and beyond. TENA::uint32 for earlier versions.
requestHeartbeatAck	Whether to request an ack on heartbeat oneway invocations. Default is 1 (true); set to 0 to disable.	1	TENA::uint32
requestAlertAck	Whether to request an ack on alerts. Default is 1 (true); set to 0 to disable.	1	TENA::uint32
enablePacketDecoration	Enable writing of extra information to end of TENA event network packets. Primarily useful for wireshark debugging. Only supported for debugged builds.	not set	Parameter is a "flag" type, in which there is not an explicit data type used;the parameter is either set or not set.
connectionLoggingVerbosity	Used to detect middleware operations associated with cached network connections. A value of 0 means disable entirely; 1 means send alerts on serious problems; 2 sends alerts on interesting events; and 3 sends alerts for any activity.	1	TENA::uint32

Usage Considerations

All applications must define the value for the `listenEndpoints` argument. This parameter requires the hostname or IP address associated with the network interface that should be used to communicate with other TENA applications. The port number for the `listenEndpoints` is not required to be specified, but may need to be used if there are firewall devices that have specific port numbers opened for communication with remote sites. Refer to the [Network Address page](#) for additional information.

When Best Effort (UDP multicast) communication is used, the parameter `multicastInterface` will default to the same interface as the `listenEndpoints` argument, and therefore does not typically need to be set. The `multicastTTL` parameter does need to be modified if the multicast traffic needs to be transmitted beyond the local area network (LAN). Additional information on Best Effort can be found on the [Multicast Support page](#).

The `connectionTimeoutInMilliseconds` and `twoWayTimeoutInMilliseconds` parameters may need to be adjusted based on specific network and application behavior. The default connection timeout value is set to 10 seconds and on networks that are excessively slow, this value may need to be increased. Simple ICMP ping measurements can be used to provide rough guidance on the network latency that contributes to the time required to establish a connection, although an overburdened machine may also contribute to slow connection times. The two-way timeout value is used to determine how long the application is going to wait for a response from another application when invoking a remote method. This time includes the processing time used by the server and the default value is two minutes.

Related Topics and Links

- [Configuration Mechanism](#)
- [Execution Manager Configuration Parameters](#)

Recommended Parameters for Working LAN Environment

Recommended TENA Parameters for a Working LAN Environment

The TENA Middleware configuration parameters may require tuning for a given operating environment. The default parameter values are usually appropriate for typical environments in which TENA applications are distributed across a wide-area network. If, however, all TENA applications will be connected on a single local area network (LAN) and that LAN is operating properly, then it is recommended to adjust certain Middleware configuration parameters (typically by shortening timeouts and reducing numbers of retry attempts). This page describes the recommended settings for a TENA Execution running entirely on a single properly operating high data-rate, low-latency LAN.

Middleware Configuration Parameters

The following Middleware parameters should be applied for every TENA application and confirmed by a TENA Console operator.

Middleware Parameter	Recommended Setting	Notes
connectionTimeoutInMilliseconds	1000	Default value of 10 seconds was used for problematic WANs, but value of 1 second is sufficient for a properly configured LAN.
twoWayTimeoutInMilliseconds	10000	Default values is 2 minutes, but if your SDO Remote Method Implementations take less than 2 seconds or so, then 10 seconds should be sufficient for this timeout.
corbaTransientRetries	2	Default value is 3, but 2 should be sufficient for a functioning network. ⚠ DEPRECATED in TENA MW version 6.0.3 and beyond. Use the <code>transientCommunicationAttempts</code> parameter instead.
transientCommunicationAttempts	2	Default value is 3, but 2 should be sufficient for a functioning network. ⚠ AVAILABLE in TENA MW version 6.0.3 and beyond.
requestPublisherAck	0	The default value was changed from 1 to 0 in release 6.0.2 of the middleware.
waitForMissingEventsIntervalInSeconds	2	This can occur when TCP sockets are broken and re-established. The default value is 11 (one second more than the default connection timeout).
mwPushTimeoutInMilliseconds (hidden parameter)	2000	The default value is 20 seconds, but is not necessary for a working LAN.
disconnectTimeoutInMilliseconds (hidden parameter)	1000	The default is 5 seconds, but is not necessary for a working LAN.

TENA applications can be told to read a config file using the `-configFile <file>` option. The following entries can be put into an application configuration file used by TENA applications to set the above parameters.



Config File Entries for Applications

```
connectionTimeoutInMilliseconds=1000
twoWayTimeoutInMilliseconds=10000
corbaTransientRetries=2
requestPublisherAck=0
waitForMissingEventsIntervalInSeconds=2
mwPushTimeoutInMilliseconds=2000
disconnectTimeoutInMilliseconds=1000
```

The Middleware configuration parameters are defined in the User Guide, [Middleware Configuration Parameters](#).

Execution Manager Configuration Parameters

The recommended executionManager (EM) configuration parameters for a working LAN environment are defined in the following table. These parameters should be confirmed by a TENA Console operator.

EM Parameter	Recommended Setting	Notes
rmiTimeoutInMilliseconds	2000	2 seconds is sufficient for a properly configured LAN. ⚠ OBSOLETE in TENA MW version 6.0.3 and beyond. Use the <code>twoWayTimeoutInMilliseconds</code> parameter instead.

twoWayTimeoutInMilliseconds	2000	2 seconds is sufficient for a properly configured LAN. ⚠ AVAILABLE in TENA MW version 6.0.3 and beyond. Use the rmiTimeoutInMilliseconds parameter for 6.0.2 and below.
heartbeatIntervalInSeconds	15	15 seconds is the shortest permitted interval, and should be sufficient on a LAN.
connectionTimeoutInMilliseconds	1000	Default value of 10 seconds was used for problematic WANs, but value of 1 second is sufficient for a properly configured LAN.
corbaTransientRetries	2	Default value is 3, but 2 should be sufficient for a functioning network. ⚠ DEPRECATED in TENA MW version 6.0.3 and beyond. Use the transientCommunicationAttempts parameter instead.
transientCommunicationAttempts	2	Default value is 3, but 2 should be sufficient for a functioning network. ⚠ AVAILABLE in TENA MW version 6.0.3 and beyond.
mwPushTimeoutInMilliseconds (hidden parameter)	2000	The default value is 20 seconds, but is not necessary for a working LAN.
disconnectTimeoutInMilliseconds (hidden parameter)	1000	The default is 5 seconds, but is not necessary for a working LAN.

The TENA executionManager can be told to read a config file using the `-configFile <file>` option. The following EM configuration file can be used by an EM to set the above parameters:

Config File Entries for executionManager

```
twoWayTimeoutInMilliseconds=2000
heartbeatIntervalInSeconds=15
connectionTimeoutInMilliseconds=1000
corbaTransientRetries=2
mwPushTimeoutInMilliseconds=2000
disconnectTimeoutInMilliseconds=1000
```

TENA Console Configuration Parameters

The recommended TENA Console configuration parameters for a working LAN are defined in the following table.

View menu -> Options -> General tab Parameter	Recommended Setting
Timeout (in milliseconds) for Establishing a TCP Connection	1,000
Timeout (in milliseconds) for a Two-way (Request/Acknowledge) Network Operation	2,000
Number of Times to Try Re-Invoking a Network Operation after a Network Error	1
Audible Beep Alert on Errors	(checked)
Audible Beep Alert on Warnings	(checked)
Start New EM tab Parameter	Recommended Setting
Heartbeat Interval: (sec)	15
Connection Timeout: (ms)	1,000
TwoWay Timeout: (ms)	2,000
Start New EM tab Advanced Execution Manager Options	Recommended Setting
Total CORBA Transient Tries	2
Push Timeout (ms)	2,000
Disconnect Timeout (ms)	1,000

Network Address

Network Address

An application's Network Address (also called Endpoint) is used to define the computer hostname or IP address and port number that should be used by other applications to communicate with that application. Using IP (Internet Protocol) based communication protocols, the IP address and port number is needed to establish communication sockets for sending and receiving network packets.

Description

The Network Address, also referred to as an Endpoint, is used to define the necessary information for one application to establish IP-based communication with another networked application. The TENA Middleware uses network addresses for the various applications and related processes to establish communication sockets between these processes for the purpose of exchanging information.

A network address is composed of two parts: IP address (which can typically be determined by the computer's hostname), and port number. The port number is an integer value and used by the operating system to segment network traffic, associating the communication stream with the appropriate process. The Internet Assigned Numbers Authority (IANA, <http://www.iana.org>) specifies that port number values in the range 0-1023 are reserved for specific purposes (such as port 80 is reserved for web servers). Users can use any port number in the range 1024 through 65535 that are not already in use.

i — If a port number is not specified, the middleware will automatically (through the operating system) obtain an available port number for the application to use. Not specifying a port number is recommended, unless the network utilizes firewall devices in which only certain ports are open for use by the TENA applications.

When an [Execution Manager \(EM\)](#), application, or [TENA Console](#) is started, it is necessary to define the `listenEndpoints` value which is the network address in which the process will use to listen for communication requests from other applications. The syntax for the `listenEndpoints` command line parameter is "`hostname:portNumber`", or "`ipAddress:portNumber`". This parameter name is plural because, generally speaking, an application can listen to multiple network interfaces, although there is little practical benefit in using this for TENA executions.

If an application attempts to use a port number already in use, the operating system will prevent the socket from being established and an error message will be reported. If necessary, an application can specify a span for the port number that causes the middleware to attempt to successively use a range of port numbers until a socket connection is established. The notation for port span is "`ipAddress:portNumber/portspan=span`", such as "`192.168.1.1:55100/portspan=10`".

Currently the TENA Middleware adheres to the IPv4 standard, although it is expected that IPv6 support will be added in the future as more networks begin to transition to a mixed IPv4/IPv6 environment.

C++ API Reference and Code Examples

The TENA Middleware provides a helper class that can be used to store and operate on the network address information. This class is called `TENA::Middleware::Endpoint` and the key elements of the class definition is shown below. Developers are encouraged to use this class when it is necessary to work with network addresses/endpoints in their application.

```

TENA_Middleware_EXPORT
void
parseEndpointString(
    std::string const & endpointString,
    std::vector< TENA::Middleware::Endpoint > & endpoints );

class Endpoint
{
public:

    Endpoint();

    explicit Endpoint(
        std::string const & endpointStr);

    Endpoint(
        std::string const & hostname,
        unsigned short port);

    ///! Returns true if this Endpoint is valid.
    bool isValid() const;

    ///! Returns a string representation of this endpoint.
    std::string toString() const;

    ///! If the address is in IP format, converts to a hostname.
    bool convertToHostname();

    ///! If the address is hostname, converts it to IP address format.
    bool convertToIPAddress();

    /**
     * \brief Sets the value for this endpoint from parsing a single-profile
     * endpoint specification string.
     *
     * Sets the value for this endpoint from parsing a single-profile
     * endpoint specification string.
     *
     * \param endpointString TAO format endpoint string; hostname:port also
     * accepted (iiop assumed).
     *
     * \exception - std::invalid_argument is raised if the endpoint
     * input string is improperly formatted or the port number out of range.
     */
    void fromString( std::string const & endpointString );

    std::string const & getProtocol() const;
    std::string const & getVersion() const;
    std::string const & getAddress() const;
    std::string const & getOptions() const;
    std::string const & getHostname() const;
    TENA::uint16 getPort() const;
};


```

Execution Management Services

Execution Management Services

A TENA Execution is a collection of applications that collaborate through the sharing of information and services. The middleware services associated with supporting application participation in TENA Executions are referred to Execution Management Services. These services are supported with several API classes: Runtime, Execution, and Session.

Execution Management Services

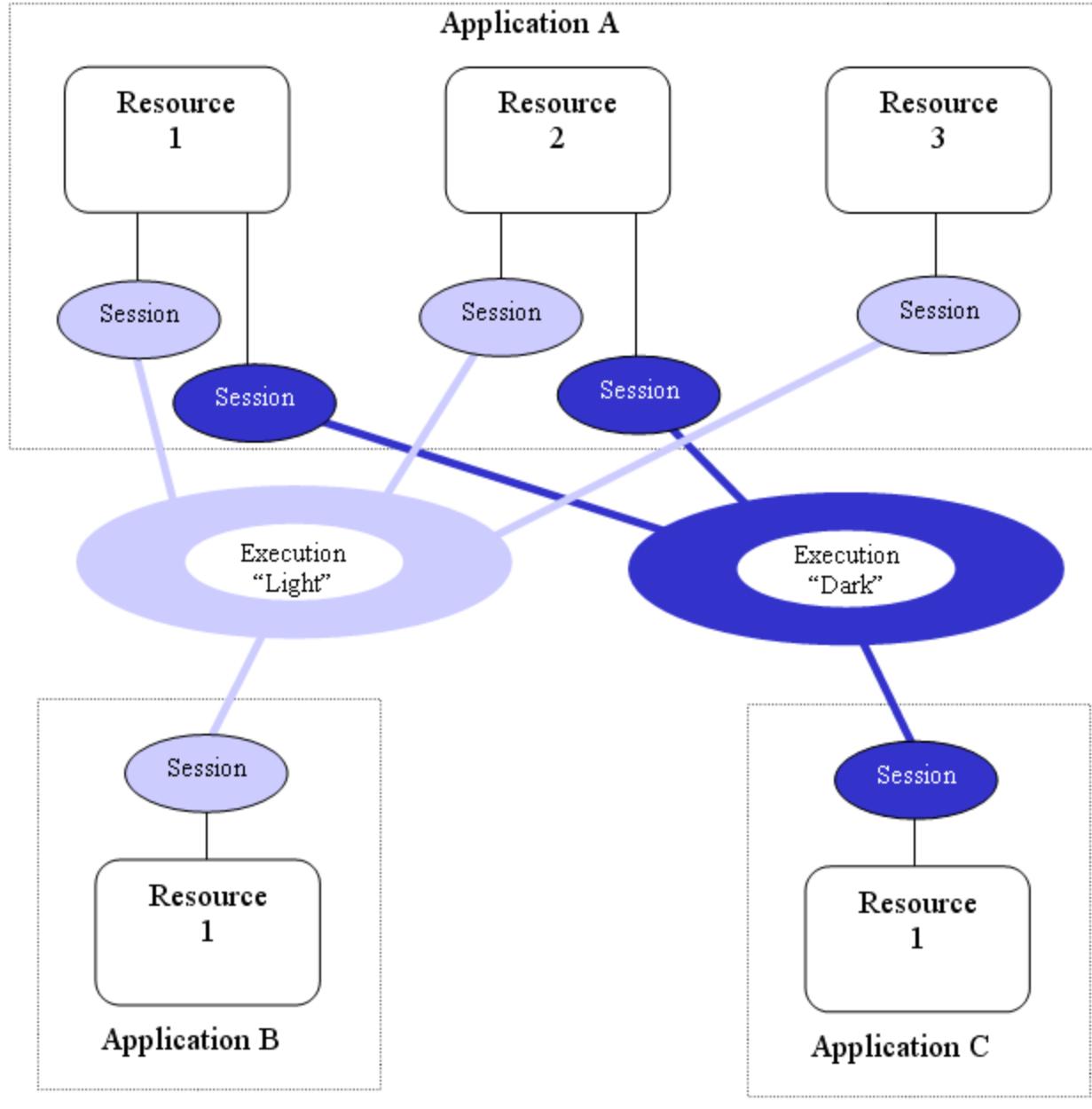
This documentation introduces the TENA Middleware services that support the management of TENA Executions. The execution management services currently focus on supporting operations associated with joining and resigning from executions. As discussed in [TENA Execution Manager documentation](#), users create executions manually by launching an Execution Manager process on a particular computer system. An application using the TENA Middleware connects to this particular execution in the context of a session.

Sessions and Executions

It is important to understand the different ways in which applications may interact with executions. When we speak of an application, we refer to a single executable running on a single computer. An application may be divided up into sub-application units that behave as if they were independent. These sub-application units are referred to as "resources". An application, therefore, consists of one or more resources. Each resource publishes and subscribes to information independently from every other resource. It is envisioned that most applications will consist of only a single resource, but the flexibility to accommodate multiple resources per application has been built into the TENA Middleware.

Having multiple resources per application can aid execution and application designers in creating more flexible and extensible systems. There are complex applications that have multiple distinct segments, each of which might be implemented by different developers. Each segment might be interested in publishing and subscribing to information independently from what the other segments are doing. For example, one segment of an application might display certain user-designated information about TENA objects while another segment might be interested in data logging. Both segments would exist in the same application in the same process. The capability to have multiple sessions per application, each with their own independent publication and subscription information, allows these independent segments of code, now called resources when joined with their session, to act independently. Each resource, in turn, may interact with one or more executions.

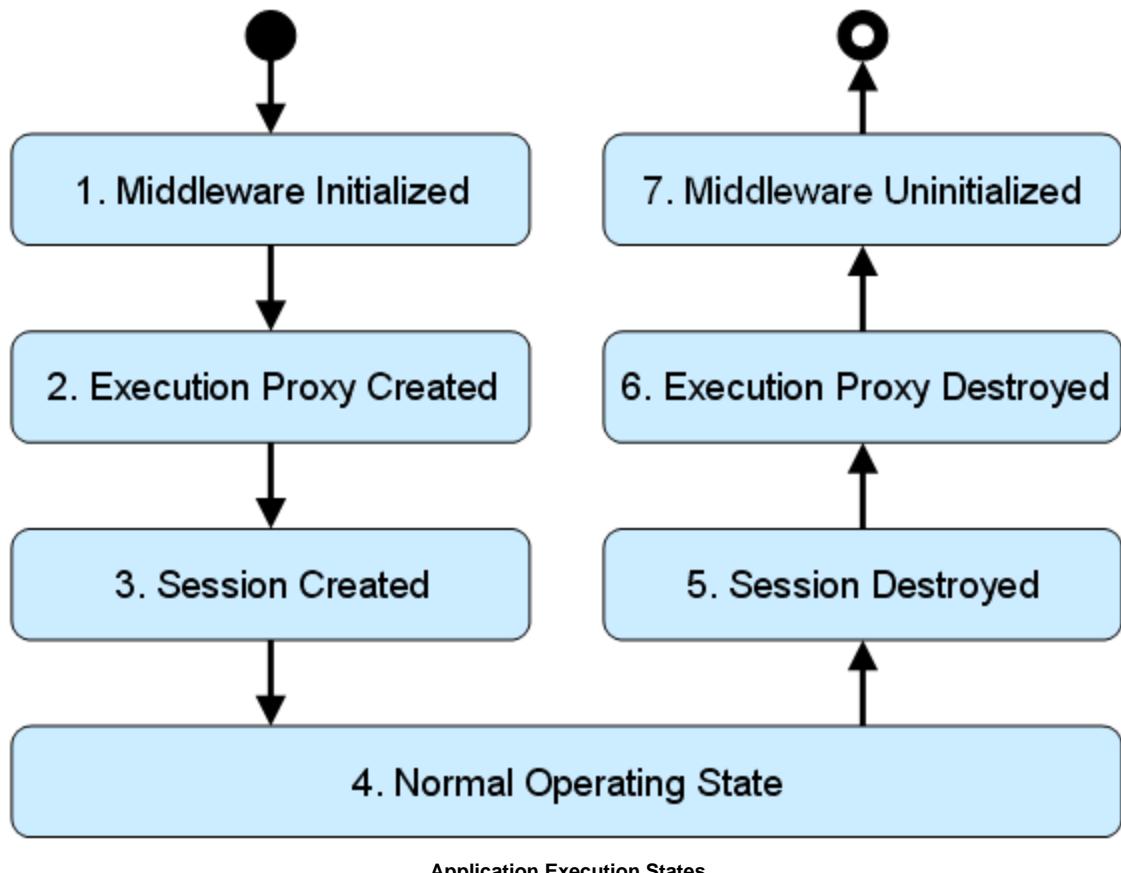
For example, a resource that bridges between two different object models would need one session for the first execution (and its object model), and a second session for the second execution. The relationship between sessions and executions is illustrated graphically in the figure below. Application A is a complex, multi-resource, multi-execution application that interacts with Applications B and C using different executions. Application A's resources also interact with each other in both executions.



As mentioned previously, the typical situation will be a single application joined to a single execution with a single session. There is no advantage with introducing multiple execution and sessions within a typical application. Multiple executions within an application are most commonly needed when the application must act as a "gateway" between two or more independent executions. Multiple sessions within an application are commonly used when there are separately developed software components that operate completely independently with respect to the publication and subscription of information and services. Often these separate components are superficially linked to an application to use an independent session that operates in a parasitic and autonomous manner. Applications can also use multiple sessions if it is necessary to provide fine-grained control of the processing associated with subscriptions. For example, if the application needed to subscribe to "cars" and "boats", but wanted to use higher priority (or more) application processing threads associated with processing the received "cars" information, a separate session could afford that control. Nominally speaking though, there is not an advantage (e.g., performance, thread safety) of introducing multiple sessions to isolate different subscriptions.

Classes Used in Execution Management

After the user initializes the Execution Manager for the desired execution, the system is prepared for applications to join the execution. Each application's software must go through a series of steps to join the execution. These steps are depicted in the figure shown below.



There are several classes and methods/functions that are used by applications to join an execution. An application is required to follow an initialization sequence in which different programming objects are constructed with the necessary state information to support an execution. The programming objects are shown in the table below.

Execution Management Programming Objects

Execution Management Programming Object	Description
TENA::Middleware::Configuration	Holds the TENA Middleware configuration information associated with the application. Users are permitted to add user-defined configuration parameters to this object.
TENA::Middleware::RuntimePtr	Provides access to common communication machinery used within the middleware for that particular process.
TENA::Middleware::ExecutionPtr	Contains the information related to the particular execution joined by the application. An application may utilize multiple ExecutionPtr objects.
TENA::Middleware::SessionPtr	Represents the particular publication and subscription state of an execution participant. Similar to the ExecutionPtr, an application may use multiple SessionPtr objects within a process.

Following standard object-oriented programming practices, an object is held by the application to indicate its continued interest. For example, when an application joins a particular execution, an `ExecutionPtr` object is used to represent that execution. Internally it contains the necessary information and machinery to allow the application to connect to the execution. When the application is no longer interested in that particular execution, the `ExecutionPtr` object can be deleted.

Initially, the user application creates the `Configuration` object and then invokes the `init` function to obtain the `RuntimePtr` object. Then the application obtains an `ExecutionPtr` that represents a particular execution. Finally from the `ExecutionPtr` a `SessionPtr` object is created that binds an application resource with a particular execution. In the examples below, the variable `endpointVector` refers to a vector of endpoints specifying the network addresses to be used to contact the running Execution Manager. How applications will coordinate to connect to the correct host/port where an execution manager is running will depend on the details of the environment. In example code provided, these are passed in to applications using the `-emEndpoints` argument. See [Execution Manager](#) for more information.

The sequence of operations to create and join an execution from within an application are show in the C++ code fragment below:

```
ApplicationConfiguration appConfig(
    argc,
    argv,
    "", // Config file ("Example-Vehicle-v1-AllPublishSubscribe-v1.config" to use example)
    "Example-Vehicle-v1-AllPublishSubscribe-v1", // Config file prefix
    "" ); // Application environment variable prefix

...
EndpointVector endpoints; // One or more endpoints added according to details of site infrastructure
...
// Initialize the middleware using the TENA::Middleware::Configuration
// object. Hold onto the RuntimePtr to enable this process to communicate
// with one or more Executions. Release it when this process no longer
// needs to communicate with any Executions.
TENA::Middleware::RuntimePtr pTENAmiddlewareRuntime(
    TENA::Middleware::init( appConfig.tenaConfiguration() ) );

...
// Join the Execution. Hold onto the ExecutionPtr to indicate
// participation in the Execution. Let go of it to leave the Execution.

TENA::Middleware::ExecutionPtr pExecution(
    pTENAmiddlewareRuntime->joinExecution( endpointVector ) );

// Create a Session in the Execution. Any name for the Session may be
// used, but a unique name within this application is required if multiple
// application sessions exist for the same execution.
TENA::Middleware::SessionPtr pSession( pExecution->createSession(
    "Example-Vehicle-v1-AllPublishSubscribe-v1_Session" ) );
```

An equivalent example in Java would be:

```

// Class member variables. In java, these are often held as member variables in a class.
TENA.Middleware.Runtime mwRuntime;
TENA.Middleware.Execution mwExecution;
TENA.Middleware.Session mwSession;
...

TENA.Middleware.Configuration mwConfig =
    new TENA.Middleware.Configuration(argv,
        "", // Config file ("Example-Vehicle-v1-AllPublishSubscribe-v1.config" to use example)
        "Example-Vehicle-v1-AllPublishSubscribe-v1");

...
// One or more endpoints added according to details of site infrastructure
TENA.Middleware.EndpointVector endpoints = new TENA.Middleware.EndpointVector() ;
...

// Initialize the middleware using the TENA.Middleware.Configuration
// object. Hold a reference to the Runtime to enable this process to communicate
// with one or more Executions.
mwRuntime = TENA.Middleware.init( appConfig.tenaConfiguration() ) ;

...

// Join the Execution. Hold onto the ExecutionPtr to indicate
// participation in the Execution. Let go of it to leave the Execution.

mwExecution = mwRuntime.joinExecution( endpointVector ) ;

// Create a Session in the Execution. Any name for the Session may be
// used, but a unique name within this application is required if multiple
// application sessions exist for the same execution.
mwSession = mwExecution.createSession("subscriber_session");
...

// At shutdown, explicitly release the objects. This will happen via garbage-collection,
// but for some applications it is desirable to free resources and disconnect communication
// connections at a predictable time.
mwSession.releaseReference();
mwExecution.releaseReference(); // If there are no other sessions from the execution
mwRuntime.releaseReference(); // If there are no other executions.

```

In some cases, the application may not have access to the command line arguments from `main()`. In this situation, the user application can define the necessary arguments through a configuration file or environment variables (see [Middleware Configuration Mechanism](#) for more information).

Note that the execution management objects (`RuntimePtr`, `ExecutionPtr`, and `SessionPtr`) are used to indicate an application's interest in using the middleware, execution, and session, respectively. When an application is no longer interested in these elements the associated object can be deleted.

These "Ptr" objects are implemented as **managed pointers** that perform reference counting to destroy the underlying object when it is appropriate. This allows multiple parts of the software (including the middleware) to hold onto a reference to the same pointer and only when all of the references are removed will the object be actually destroyed.

Typically, the object pointers are effectively deleted by letting them "go out of scope" - in other words, the object pointer is created within a particular scope indicated by braces `{ ... }`, and the compiler will automatically destroy the pointer at the closing brace.

```

TENA::Middleware::ExecutionPtr pExecution =
    pTENAMiddlewareRuntime->joinExecution( );
...
{
    ...
    TENA::Middleware::SessionPtr pSession =
        pExecution->createSession( sessionName );
    ...
} // pSession goes out of scope and is destroyed!

```

Additionally, the application can invoke the `reset()` method on the managed pointer to let go of the reference. This technique is illustrated in the code fragment below. Once `reset()` is invoked, any use of the pointer object will result in a runtime exception. Therefore, use of `reset()` should be done with extreme care.

```
// Application manually resets pExecution
pExecution.reset();
```

In the other language bindings, these pointer objects are proxied through garbage-collected references. The added method `releaseReference()` allows the control that in C++ is accomplished by dropping the reference.

Refer to [Smart Pointer documentation](#) for more information on the use and operation of managed pointers.

Join Execution Options

Release 6.0.4 and later of the middleware added an additional `joinExecution()` API method for the `TENA::Middleware::Runtime` class that contains the argument of type `TENA::Middleware::ExecutionJoinOptions`. The `joinOptions` argument allows the application to specify a user SiteID and ApplicationID, as well as explicitly stating which object models that the application will use with the execution.

User Site and Application IDs

A common TENA standard object model that has been used for identification purposes is the [TENA-UniqueID object model](#). This object model relies on application developers to properly utilize the attribute values to ensure uniqueness of this identifier across the execution. As an initial step in having the middleware support improved management of user-defined identifiers, the `ExecutionJoinOptions` permits the application developer to specify `userSiteID` and `userApplicationID` values at join execution time. These values are added to the application's [Middleware Metadata](#) and available to event operators through the [TENA Console](#) tool. Console operators can manually monitor the `userSiteID` and `userApplicationID` values to ensure applications are following the proper guidance related to the particular event procedures.

When an application uses the `joinExecution` method that does not include the `ExecutionJoinOptions`, or if the `ExecutionJoinOptions` does not specify these values, the middleware will set the `UserSiteID` value to be zero and the `userApplicationID` value will be set to be the same as the middleware generated `applicationID` (which is guaranteed to be unique for the execution, see [Middleware IDs](#) for more information).

Recommended guidance for application development with respect to these identifiers is the following:

1. Make `UserSiteID` and `UserApplicationID` values be configuration options for the application.
2. When these values are set, use the `joinExecution` method with the values specified in the `ExecutionJoinOptions` argument.
3. When using the TENA-UniqueID object model within the application, utilize the `UserSiteID` and `UserApplicationID` values. If necessary, the values can be obtained through the `Execution::Metadata`, see [Middleware Metadata](#) for details.

Explicit Object Model Specification

The `ExecutionJoinOptions` argument is used when an application needs to join multiple executions with different versions of the same object model. Since a single execution does not permit the use of multiple versions of the same object model, the use must occur in separate executions. Normally, the middleware automatically detects the object models linked into a particular application, and communicates information related to these object model definitions to the Execution Manager during the join process to perform [Object Model Consistency Checking](#). In the multiple object model version scenario, the application needs to explicitly specify what object models will be used in which execution. More details regarding this capability are discussed in the [Multiple Version Support](#) documentation page.

Middleware IDs

Middleware IDs

The Middleware API uses a number of objects to represent different elements of an execution. Many of these objects contain a unique numerical identifier that can be used for application programming needs. These IDs present a common interface to support comparison operations and mechanisms to convert to numerical and text representations.

Description

The TENA Middleware provides several programming objects for the purpose of providing unique (within some scope) identifiers for particular TENA application components. The following table lists these types along with their underlying representations:

Middleware Identifiers

ID Type	Representation
RuntimeID	struct of 4 uint32
ExecutionID	struct of 4 uint32
SessionID	uint32
ObjectID	struct of 4 uint32
MessageSenderId	struct of 4 uint32
MessageID	struct of 4 uint32 and uint32
TypeID	uint64
ApplicationID	uint32

These ID types are "distinct" in C++, they are not aliases for other types and there is no implicit conversion from other types. This feature is deliberate; the idea is to make it difficult to inadvertently mix IDs that correspond to different TENA entities.

Despite being distinct types, the ID types have some operations in common:

- They can be constructed from an instance of their underlying type or from an array of bytes (literally, a `boost::array<uint8>`).
- They have a method `getValue`, which returns the underlying representation.
- They have a method `getBytes`, which returns the value as a byte array.
- They have a method `toHex`, which returns a string containing the ID represented as a single hexadecimal quantity.

IDs also support the full complement of comparison operators (i.e., `operator==`, `operator<`, etc.) and can be output to streams (i.e., `std::ostream`).

Usage Considerations

There are two ways application developers will commonly acquire an ID:

- from its associated Middleware entity (e.g., getting a `MessageSenderId` from a `MessageSender`),
- from a value defined in an object model.

The TENA Middleware classes that have a corresponding ID type have a `getID` method that returns the ID for an instance:

```
TENA::Middleware::RuntimeID id = myRuntime->getID();
```

ID Accessors

ID Type	Accessor(s)
RuntimeID	<code>TENA::Middleware::Runtime->getID()</code>
ExecutionID	<code>TENA::Middleware::Execution->getID()</code>
SessionID	<code>TENA::Middleware::Session->getID()</code>
ObjectID	<code>TENA::Middleware::SDO::ServantBase->getSDOid()</code>

MessageSenderID	TENA::Middleware::Messages::MessageMetadata->getSenderID()
TypeID	e.g., Example::Person::typeID
ApplicationID	TENA::Middleware::Execution::Metadata->getApplicationID()

Applications can use IDs to keep track of instances of various TENA entities. Because they support comparison, they are appropriate for use in sorted C++ standard library containers like `std::set` and `std::map`.

Incorporating IDs into object models

While it is possible that some future release of the Middleware will incorporate knowledge of IDs directly into TDL (TENA Definition Language), this capability is not currently available. So, for the purpose of incorporating IDs into an object model, alternative representations of IDs must be used.

For the IDs whose underlying representation is an integer type (`SessionID`, `ApplicationID`, and `TypeID`), it is straightforward to use the corresponding integer type in the object model:

```
package MyPackage
{
    local class MyLocalClass
    {
        uint64 someTypeID;
    };
}
```

In C++ application code, we can simply construct a `TypeID` from the `uint64` value retrieved from the local class' accessor:

```
TENA::Middleware::TypeID someTypeID( myLocalClass->get_someTypeID() );
```

For IDs that cannot be represented with a single integer value (`RuntimeID`, `ExecutionID`, `ObjectID`, and `MessageSenderID`), this is slightly more complicated. It is recommended that these IDs be represented by a vector of bytes in the object model:

```
package MyPackage
{
    local class MyLocalClass
    {
        vector< uint8 > someObjectID;
    };
}
```

In addition to being constructable from their underlying value type, IDs can be constructed from a `boost::array` of `uint8` (i.e., bytes). For convenience, each ID type has a member typedef, `byte_array_type`, that corresponds to the `boost::array` template instance used for the byte array. It is straightforward to convert the `std::vector<uint8>` gotten from the local class' accessor into a `boost::array<uint8, N>`:

```
// 
// The empty aggregate initializer initializes the array elements to 0.
//
TENA::Middleware::ObjectID::byte_array_type someObjectIDbytes = {};
std::vector< TENA::uint8 > byteVec = someLocalClass->get_someObjectID();
assert(byteVec.size() == someObjectIDbytes.size());
std::copy(byteVec.begin(), byteVec.end(), &someObjectIDbytes[0]);
TENA::Middleware::ObjectID someObjectID(someObjectIDbytes);
```



boost::array is statically sized; std::vector is not.

`boost::array` is statically sized, meaning that the size of the array (that is, the number of elements) is known at compile time and cannot be changed at run time. In contrast to this, the size of `std::vector` is determined at runtime and can be changed arbitrarily. This difference means there is some potential for error when converting between `boost::array` and `std::vector`.

In the above example, we `assert` that the vector and the array are the same size. Like any assertion, this isn't strictly necessary. But in the event of an error, it will ensure that your application fails in an obvious way.

C++ API Reference and Code Examples

The middleware ID classes are template classes and support the following methods and functions. The definition has been abbreviated to focus on the key items.

```
template < typename T >
class ID
{
public:

    value_type const &
    getValue()
        const
    {
        return this->_value;
    }

    byte_array_type const
    getBytes()
        const
    {
        return TENA::Middleware::getBytes( this->_value );
    }

    std::string const
    toHex()
        const
    {
        return TENA::Middleware::toHex( this->_value );
    }
};

template < typename T >
bool
operator==( ID< T > const & lhs, ID< T > const & rhs );

template < typename T >
bool
operator!=( ID< T > const & lhs, ID< T > const & rhs );

template < typename T >
bool
operator<( ID< T > const & lhs, ID< T > const & rhs );

template < typename T >
bool
operator<=( ID< T > const & lhs, ID< T > const & rhs );

template < typename T >
bool
operator>( ID< T > const & lhs, ID< T > const & rhs );

template < typename T >
bool
operator>=( ID< T > const & lhs, ID< T > const & rhs );

template < typename T >
std::ostream &
operator<<( std::ostream & out, ID< T > const & id );

template < typename T >
std::string const
toHex( T const & val );

template < >
TENA_Middleware_EXPORT
std::string const
toHex< DIME::Identity >( DIME::Identity const & val );

template < typename T >
boost::array< uint8, GetValueBytes< T >::value > const
getBytes( T const & val );
```

```

template <>
TENA_Middleware_EXPORT
boost::array<
    uint8,
    GetValueBytes< DIME::Identity::Identity >::value > const
getBytes< DIME::Identity::Identity >(
    DIME::Identity::Identity const & val );

template < typename T >
T const
fromBytes(
    boost::array< uint8,
    GetValueBytes< T >::value > const & bytes );

template <>
TENA_Middleware_EXPORT
DIME::Identity::Identity const
fromBytes< DIME::Identity::Identity >(
    boost::array< uint8,
    GetValueBytes< DIME::Identity::Identity >::value > const &
    bytes );

```

Java API Reference and Code Examples

While the Java API provides wrapper classes for all these ID types, they are represented as opaque types, that do not allow access to underlying byte representation. Every ID type in java provides only an `equals()` method and a `tohex()` that allows extracting a hex representation String.

Middleware Metadata

Middleware Metadata

Applications often benefit from having access to operational information related to the status of the middleware and middleware elements. This information, referred to as **Metadata**, is associated with the various C++ objects that correspond to the middleware API. There is metadata related to the state of the application, as well as information related to specific objects and messages.

⚠ Note: The same metadata is available with the alternate language bindings. See for example the various metadata classes at the [Middleware are Java Binding](#) documentation. Utilities for dealing with int64 (long) timestamps are not available, as of the 1.0.0 release, however.

Description

Associated with various TENA classes, users are able to access metadata information related to the class to aid in diagnostics or application specific processing. There is metadata available for the following TENA classes:

- `TENA::Middleware::Runtime` — Information related to the overall application configuration.
- `TENA::Middleware::Execution` — Information related to a particular execution that the application has joined.
- `TENA::Middleware::Session` — Information related to a particular session that the application has established.
- `TENA::Middleware::SDO` — Information related to a particular SDO servant (published object) or SDO proxy (discovered object).
- `TENA::Middleware::MessageSender` — Information related to a class used to send messages.
- `TENA::Middleware::Message` — Information related to a particular message.

Note on time represented as a `TENA::int64`

Several Metadata methods return a `TENA::int64` timestamp. A `TENA::int64` timestamp value represents the number of nanoseconds since epoch (midnight starting 1/1/1970 UTC). Despite the fact that the value is in nanoseconds, the value may not have nanosecond-level accuracy. (The resolution depends on the computer's clock.) The `TENA::Middleware::Utils::toTimestamp()` function provides a way to convert the `TENA::int64` timestamp to a human-readable string.

Alternatively, one can use the `TENA::Time` local class from the [TENA-Time Standard Object Model](#) to convert a `TENA::int64` time to other time formats (e.g., UnixTime, UTCTime, GPSTime, or YTDTime). Of course, this requires linking your application with the TENA-Time object model definition and object model implementation libraries.

For example, if you want to convert a `TENA::int64` nanoseconds time to a struct showing seconds, minutes, hours, dayOfMonth, month, year, and dayOfWeek, then you can do the following:

- Pass the `TENA::int64` nanoseconds time value to the `TENA::Time` local class constructor.
- Pass the resulting `TENA::Time::LocalClassPtr` to the `TENA::UnixTime` constructor.
- Store the `TENA::UnixTime`'s seconds attribute in a `time_t` variable.
- Pass the address of that `time_t` variable to the `gmtime_r()` function, which produces a `tm` struct having the desired elements.

Runtime Metadata

The attributes for Runtime Metadata are shown in the following table. The returned data type for the metadata methods indicate if the value is non-changing (i.e., only set once) by appending the type field with the "(non-changing)" qualifier.

Runtime Metadata Methods

Runtime Metadata Method	Returned Data Type	Description
<code>getID()</code>	<code>TENA::Middleware::RuntimeID</code> (non-changing)	Provides an ID to represent a particular <code>TENA::Middleware::Runtime</code> and is designed to be universally unique.. Only one <code>TENA::Middleware::Runtime</code> can exist in a single TENA application.
<code>getTimeLastUpdated()</code>	<code>TENA::int64</code>	Provides the time (as specified by the operating system) in which the middleware last updated the metadata structure. See note on int64 time .
<code>getMiddlewareVersion()</code>	<code>std::string</code> (non-changing)	Provides a string representation of the version of the middleware (e.g., "6.0.0").
<code>getApplicationEndpointsString()</code>	<code>std::string</code> (non-changing)	Provides a string representation of the network endpoints (i.e., computer network names or IP addresses and the port numbers) that the application uses to listen for incoming middleware communication.
<code>getProcessID()</code>	<code>TENA::uint32</code> (non-changing)	Provides a numeric identifier used by the operating system to represent the process associated with the running TENA application.

Runtime Metadata Usage Considerations

Users can obtain the Runtime Metadata object and invoke the available methods as shown in the code fragment below. The Runtime metadata is updated when the application joins or resigns from an execution.

```
TENA::Middleware::RuntimePtr pTENAmiddlewareRuntime(
    TENA::Middleware::init( programConfig ) );

TENA::Middleware::Runtime::MetadataPtr const
    pRuntimeMetadata( pTENAmiddlewareRuntime->getMetadata() );

std::cout << pRuntimeMetadata->getMiddlewareVersion() << std::endl;
```

Execution Metadata

The attributes for Execution Metadata are shown in the table below. The returned data type for the metadata methods indicate if the value is non-changing (i.e., only set once) by appending the type field with the "(non-changing)" qualifier.

Execution Metadata Methods

Execution Metadata Method	Returned Data Type	Description
getID()	TENA::Middleware::ExecutionID (non-changing)	Provides an ID to represent a particular execution to which participants may join and is a counter (starting at value of 1) managed by the Execution Manager process.
getTimeLastUpdated()	TENA::int64	Provides the time (as specified by the operating system) in which the middleware last updated the metadata structure. See note on int64 time.
getApplicationID()	TENA::Middleware::ApplicationID (non-changing)	Provides an ID to represent the current application that is joined to the logical range execution and is a counter (starting at value of 1) managed by the Execution Manager process.
getUserSiteID()	TENA::uint32	The application developer may specify the userSiteID value at join execution time using the <code>ExecutionJoinOptions</code> . When an application uses the <code>joinExecution</code> method that does not include the <code>ExecutionJoinOptions</code> , or if the <code>ExecutionJoinOptions</code> does not specify this value, the middleware will set the userSiteID value to be zero. Available in TENA Middleware version 6.0.4 and beyond.
getUserApplicationID()	TENA::uint32	The application developer may specify the userApplicationID value at join execution time using the <code>ExecutionJoinOptions</code> . When an application uses the <code>joinExecution</code> method that does not include the <code>ExecutionJoinOptions</code> , or if the <code>ExecutionJoinOptions</code> does not specify this value, the middleware will set the userApplicationID value to be the same as the middleware-generated applicationID (which is guaranteed to be unique for the execution). Available in TENA Middleware version 6.0.4 and beyond.
getSpecifiedObjectModels()	std::vector<std::string>	Provides a vector containing the object model types that were used for this particular execution (as specified in the <code>ExecutionJoinOptions</code> that was provided in the <code>joinExecution</code> invocation). Available in TENA Middleware version 6.0.4 and beyond.
getExecutionManagerEndpointsString()	std::string	Provides a string representation of the Execution Manager network endpoints (i.e., computer network names or IP addresses and the port numbers) used by the application to communicate with the Execution Manager process.
getExecutionManagerEndpointsVector()	std::vector<TENA::Middleware::Utils::Endpoint>	Provides a vector of <code>TENA::Middleware::Utils::Endpoint</code> entries that can be used by the application to communicate with the Execution Manager process.
getMulticastProperties()	std::string (non-changing)	Provides a string representation of the multicast address/port range used by the execution. If the Execution Manager was started without the <code>multicastProperties</code> option, this method will return the string ":0:0"

Execution Metadata Usage Considerations

Users can obtain the Execution Metadata object and invoke the available methods as shown in the code fragment below.

```
std::vector< TENA::Middleware::Endpoint > endpointVector(
    appConfig[ "emEndpoints" ].getValue<
        std::vector< TENA::Middleware::Endpoint > >() );

// Join the Execution. Hold onto the ExecutionPtr to indicate
// participation in the Execution. Let go of it to leave the Execution.
TENA::Middleware::ExecutionPtr pExecution(
    pTENAMiddlewareRuntime->joinExecution( endpointVector ) );

TENA::Middleware::Execution::MetadataPtr const pExecutionMetadata( pExecution->getMetadata() );

std::cout << pExecutionMetadata->getExecutionManagerEndpointsString() << std::endl;
```

Session Metadata

The attributes for Session Metadata are shown in the following table. The returned data type for the metadata methods indicate if the value is non-changing (i.e., only set once) by appending the type field with the "(non-changing)" qualifier.

Session Metadata Methods

Session Metadata Method	Returned Data Type	Description
getTimeLastUpdated()	TENA::uint32	Provides the time (as specified by the operating system) in which the middleware last updated the metadata structure.
getID()	TENA::Middleware::SessionID (non-changing)	Provides an ID to represent the session and designed to be universally unique.
getName()	std::string (non-changing)	Provides the name of the session (if provided).
getServantCountByType()	TypeCountMap ¹	Provides a std::map container that indicates the number of servants for each type.
getProxyCountByType()	TypeCountMap ¹	Provides a std::map container that indicates the number of proxies for each type.
getMessagesSentByType()	TypeCountMap ¹	Provides a std::map container that indicates the number of messages sent for each type.
getMessagesReceivedByType()	TypeCountMap ¹	Provides a std::map container that indicates the number of messages received for each type.

¹ The type TypeCountMap is a std::map with std::string as the key type and TENA::uint32 as the data type.

⚠ — Note that the getProxyCountByType and getMessagesReceivedByType methods are not reliable mechanisms to determine active application type subscriptions. There is currently no API mechanism to support this operation and applications are required to maintain this information if needed.

Session Metadata Usage Considerations

Users can obtain the Session Metadata object and invoke the available methods as shown in the code fragment below.

```

TENA::Middleware::SessionPtr pSession( pExecution->createSession(
    "Example-Vehicle-Subscriber-v1_Session" ) );

// Create SDO servants
...

TENA::Middleware::SessionMetadataPtr const pSessionMetadata( pSession->getMetadata() );

TENA::Middleware::SessionMetadata::TypeCountMap servantTypes(
    pSessionMetadata->getServantCountByType() );

for ( TENA::Middleware::SessionMetadata::TypeCountMap::const_iterator iter = servantTypes.begin();
      iter != servantTypes.end();
      ++iter )
{
    std::cout << " Type: " << (*iter).first
        << " Count: " << (*iter).second << std::endl;
}

```

SDO Metadata

The attributes for SDO Metadata are shown in the following table. The returned data type for the metadata methods indicate if the value is non-changing (i.e., only set once) by appending the type field with the "(non-changing)" qualifier.

SDO Metadata Methods

SDO Metadata Method	Returned Data Type	Description
getID()	TENA::Middleware::ObjectID (non-changing)	Provides an ID to represent the particular Stateful Distributed Object (SDO) and designed to be universally unique.
getReactivationCount()	TENA::uint32	Provides an integer counter that starts with a value of one and is incremented each time the SDO is reactivated .
getPublisherApplicationID()	TENA::Middleware::ApplicationID (non-changing)	Provides an ID to represent the publishing application and designed to be universally unique.
getPublisherEndpointsString()	std::string (non-changing)	Provides a string representation of the network endpoints (i.e., computer network names or IP addresses and the port numbers) that the application publishing the SDO uses to listen for incoming middleware communication.
getPublisherSessionID()	TENA::Middleware::SessionID (non-changing)	Provides an ID to represent the publishing application's session that is unique to the publishing application's process.
getTypeIDPath()	std::vector<TENA::Middleware::TypeID> (non-changing)	Provides a vector of TENA::Middleware::TypeIDs for each type in the inheritance hierarchy for the SDO (order of most-derived to least-derived). The TypeID is a numeric identifier that can be used to look up the character string name of the SDO type.
getMulticastAddressString()	std::string	Provides a string representation of the multicast address used to send updates for this SDO (or empty string if updates are sent reliably).

getTimeOfCreation()	TENA::int64	Provides a timestamp (as specified by the operating system of the publishing application) when the SDO was created. See note on int64 time.
getTimeOfDiscovery()	TENA::int64	Provides a timestamp (as specified by the operating system of the subscribing application) when the SDO was discovered. Throws std::logic_error if invoked on publisher side. See note on int64 time.
is_TimeOfDiscovery_set()	bool	Indicates whether the TimeOfDiscovery metadata value is set.
getTimeOfCommit()	TENA::int64	Provides a timestamp (as specified by the operating system of the publishing application) when the SDO was updated (including when the SDO was first initialized by the construction process). If called on a Proxy's Metadata, the return value indicates the time of the last commit reflected in the Proxy's publication state. See note on int64 time.
getTimeOfReceipt()	TENA::int64	Provides a timestamp (as specified by the operating system of the subscribing application) when the update corresponding to the metadata was received by the subscribing application. Throws std::logic_error if invoked on publisher side. See note on int64 time.
is_TimeOfReceipt_set()	bool	Indicates whether the TimeOfReceipt metadata value is set.
getStateVersion()	TENA::uint32	Provides an integer counter that starts with a value of one and is incremented each time the servant's state is updated.
getTag()	TENA::Middleware::Tag	Provides the advanced filtering information provided by the publishing application in which the update corresponding to the metadata was performed. TENA Middleware version 6.0.4 deprecated FilteringContext in place of Tag.
getCommunicationProperties()	TENA::Middleware::CommunicationProperties	Provides the enumeration value which represents the particular type of communication used for updating the SDO (e.g., TENA::Middleware::Reliable or TENA::Middleware::BestEffort)
isTerminated()	bool	Indicates if the SDO is associated with an SDO servant that has been marked as "terminated". SDOs are terminated through the application removal process, typically when the application has abnormally terminated or disconnected from the execution. Note that the terminology of this capability was changed from "application termination" to "application removal", although this method name was maintained to support API compatibility. In TENA v6.0.3 and above, isTerminated() is deprecated and isRemoved() should be used.

SDO Metadata Usage Considerations

Users can obtain the SDO Metadata object and invoke the available methods as shown in the code fragment below.

```
Example::Vehicle::ServantPtr pServant(
    pServantFactory->createServant( *initializer, communicationProperties ) );

Example::Vehicle::MetadataPtr pMetadata( pServant->getPublicationState()->getMetadata() );

std::cout << "Example::Vehicle::Metadata"
    << "\n\tTimeOfCommit = "
    << TENA::Middleware::Utils::toTimestamp( pMetadata->getTimeOfCommit() )
    << std::endl;
```

MessageSender Metadata

The attributes for MessageSender Metadata are shown in the following table. The returned data type for the metadata methods indicate if the value is non-changing (i.e., only set once) by appending the type field with the "(non-changing)" qualifier.

MessageSender Metadata Methods

MessageSender Metadata Method	Returned Data Type	Description
getID()	TENA::Middleware::MessageSenderId (non-changing)	Provides an ID to represent the MessageSender and designed to be universally unique.

getEndpointsString()	std::string (non-changing)	Provides a string representation of the network endpoints (i.e., computer network names or IP addresses and the port numbers) that the current application uses to listen for incoming middleware communication.
getApplicationID()	TENA::Middleware::ApplicationID (non-changing)	Provides an ID to represent the current application and designed to be universally unique.
getSessionID()	TENA::Middleware::SessionID (non-changing)	Provides an ID to represent the current application's session that is unique to this application's process.
getTypeIDpath()	std::vector<TENA::Middleware::TypeID> (non-changing)	Provides a vector of TENA::TypeID for each type in the inheritance hierarchy for the message sender (order of most-derived to least-derived). The TypeID is a numeric identifier that can be used to look up the character string name of the object type.
getTag()	TENA::Middleware::Tag	Provides the advanced filtering information provided by the sending application. TENA Middleware version 6.0.4 deprecated FilteringContext in place of Tag.
getMessageCount()	TENA::uint32	Provides the counter value for the number of messages sent by this MessageSender.
getCommunicationProperties()	TENA::Middleware::CommunicationProperties	Provides the enumeration value which represents the particular type of communication being used when sending messages with this MessageSender (e.g., TENA::Middleware::Reliable or TENA::Middleware::BestEffort)

MessageSender Metadata Usage Considerations

Users can obtain the MessageSender Metadata object and invoke the available methods as shown in the code fragment below.

```
Example::Notification::MessageSenderPtr pMsgSender
    Example::Notification::MessageSender::create(
        pSession, communicationProperties );

TENA::Middleware::Messages::MessageSenderMetadataPtr pMetadata(
    pMsgSender->getMetadata() );

std::cout << "Example::Notification::MessageSender::Metadata"
    << "\n\tMessageCount = " << pMetadata->getMessageCount() << std::endl;
```

 Release 6.0.2 Update — An alternative `create` method was added with release 6.0.2 of the middleware in which the `SessionPtr` argument is used, versus the `Session` reference (i.e., `*pSession`) that was used in the previous releases. Users are encouraged to use the `SessionPtr` interface. See [MW-4286](#) for more details.

Message Metadata

The attributes for Message Metadata are shown in the following table. The returned data type for the metadata methods indicate if the value is non-changing (i.e., only set once) by appending the type field with the "(non-changing)" qualifier.

Message Metadata Methods

Message Metadata Method	Returned Data Type	Description
getID()	TENA::Middleware::MessageID (non-changing)	Provides a universally unique ID to represent this Message. Throws <code>std::logic_error</code> if invoked on an unsent message.
getSenderId()	TENA::Middleware::MessageSenderID (non-changing)	Provides an ID to represent the MessageSender that sent this particular message and designed to be universally unique. Throws <code>std::logic_error</code> if invoked on an unsent message.
getMessageCount()	TENA::uint32	Provides the message count of the MessageSender that sent this message. Throws <code>std::logic_error</code> if invoked on an unsent message.

getSenderApplicationID()	TENA::Middleware::ApplicationID (non-changing)	Provides an ID to represent the sending application and designed to be universally unique. Throws <code>std::logic_error</code> if invoked on an unsent message.
getSenderEndpointsString()	std::string (non-changing)	Provides a string representation of the network endpoints (i.e., computer network names or IP addresses and the port numbers) that the application sending the message uses to listen for incoming middleware communication. Throws <code>std::logic_error</code> if invoked on an unsent message.
getSenderSessionID()	TENA::Middleware::SessionID (non-changing)	Provides an ID to represent the sending application's session that is unique to the sending application's process. Throws <code>std::logic_error</code> if invoked on an unsent message.
getTypeIDPath()	std::vector<TENA::Middleware::TypeID> (non-changing)	Provides a vector of TENA::Middleware::TypeIDs for each type in the inheritance hierarchy for the message (order of most-derived to least-derived). The TypeID is a numeric identifier that can be used to look up the character string name of the object type.
getMulticastAddressString()	std::string (non-changing)	Provides a string representation of the multicast address used to send this message (or empty string if sent reliably).
getTimeOfTransmission()	TENA::int64	Provides a timestamp (as specified by the operating system of the publishing application) when the message was sent by the sending application. Throws <code>std::logic_error</code> if invoked on an unsent message. See note on int64 time.
getTimeOfReceipt()	TENA::int64	Provides a timestamp (as specified by the operating system of the subscribing application) when the update corresponding to the metadata was received by the subscribing application. See note on int64 time.
getTag()	TENA::Middleware::Tag	Provides the advanced filtering information provided by the sending application. TENA Middleware version 6.0.4 deprecated FilteringContext in place of Tag.
getCommunicationProperties()	TENA::Middleware::CommunicationProperties	Provides the enumeration value which represents the particular type of communication used for sending the message (e.g., TENA::Middleware::Reliable or TENA::Middleware::BestEffort)

Message Metadata Usage Considerations

Users can obtain the Message Metadata object and invoke the available methods as shown in the code fragment below.

```
// Processing an Example::Notification::ReceivedMessagePtr named pMessage...

Example::Notification::MetadataPtr pMetadata( pMessage->getMetadata() );

std::cout << "Example::Notification::Metadata"
<< "\n\tTimeOfTransmission = "
<< TENA::Middleware::Utils::toTimestamp( pMetadata->getTimeOfTransmission() )
<< std::endl;
```

Application-Specific Clock

Application-Specific Clock

The TENA Middleware provides a mechanism to utilize an arbitrary clock for creating timestamps associated with logging, diagnostics and meta-data. By default, the middleware will utilize the operating system clock for timestamps, but application developers can create their own clock class for use by the middleware.

 — Available for TENA Middleware version 6.0.3 and later.

Description

The TENA Middleware uses timestamps when recording entries in log files, diagnostics, and meta-data. Nominally, the middleware will utilize the operating system clock to obtain these timestamps. In some cases, the computer systems associated with TENA applications will have access to other clocks, e.g., IRIG, GPS, etc. Having the middleware utilize these other clocks for timestamps, rather than using the system clock, may assist with correlating events in the distributed system.

Application developers can create their own clocks by creating a C++ class that inherits from the `TENA::Middleware::Utils::MiddlewareTimeGenerator` class. The derived class only needs to implement the `getTime()` method that returns the current time represented as nanoseconds since midnight January 1, 1970 using a 64-bit integer (`TENA::uint64`), following the UNIX time convention. Note well that the [UNIX time representation](#) is complicated by the use of leap second adjustments to Coordinated Universal Time (UTC). The UNIX time representation only accounts for a fixed number of seconds in a day (86,400) which results in a leap second offset when converting between UNIX time and other time representations that do not account for leap seconds (e.g., GPS time). Refer to the vendor information associated with the external time source with regard to converting to UNIX time representation.

The implementation of the `getTime()` method is required to be thread safe. No internal thread synchronization is performed for the `getTime()` method. This avoids unnecessary overhead for the default `MiddlewareTimeGenerator` implementation that is based upon the thread-safe `gettimeofday()` system function. The simple example code example below shows the use of a `boost::mutex` for addressing thread safety since it prevents multiple threads from invoking `getTime` simultaneously.

A `TENA::Middleware::Utils::MiddlewareTime` class provides a static method, `setTimeGenerator()`, for applications to register their user-defined `MiddlewareTimeGenerator` class. Additionally, there is a static method, `getTime()`, that the application can utilize to obtain the current time used by the middleware. To ensure that the same clock source is used by the middleware, the `setTimeGenerator()` method should ideally be invoked before TENA is initialized, i.e., before calling `TENA::Middleware::init()`.

 — Users implementation an application-specific clock are encouraged to open a Middleware (MW) [helpdesk case](#) to collaborate with the TENA SDA development team. It is expected that user-defined time source implementations may be reusable across the user community.

C++ API Reference and Code Examples

The header file `TENA/Middleware/Utils/MiddlewareTime.h` defines the `MiddlewareTimeGenerator` and `MiddlewareTime` classes as shown below. The code has been simplified and re-formatted.

TENA/Middleware/Utils/MiddlewareTime.h

```
class MiddlewareTimeGenerator // By "TimeGenerator" we mean "clock"
{
public:
    MiddlewareTimeGenerator();

    virtual ~MiddlewareTimeGenerator();

    virtual TENA::int64 getTime();
};

typedef SmartPtr< MiddlewareTimeGenerator > MiddlewareTimeGeneratorPtr;

class MiddlewareTime
{
public:
    MiddlewareTime();

    static TENA::int64 getTime( );

    static void setTimeGenerator( MiddlewareTimeGeneratorPtr const & generator );
};
```

Example Source Code for Creating an Application-Specific Clock

The following is a simple example user-defined `MiddlewareTimeGenerator` that uses a fictional Atomic clock. The function `getAtomicTimeInNanoseconds()` would be provided by a user-supplied library.

A Fictional AtomicClock Class

```
class AtomicClock
    : public TENA::Middleware::Utils::MiddlewareTimeGenerator
{
public:
    AtomicClock() {}

    TENA::int64 getTime()
    {
        return getAtomicTimeInNanoseconds(); // Function provided by the fictional Atomic clock driver
    }
};

typedef TENA::Middleware::Utils::SmartPtr< AtomicClock > AtomicClockPtr;
```

The application registers the application-specific clock with the static `TENA::Middleware::Utils::MiddlewareTime::setTimeGenerator()` method as shown in the code fragment below.

Application Code to Make Use of the Fictional AtomicClock Class

```
AtomicClockPtr pAtomicClock( new AtomicClock );

TENA::Middleware::Utils::MiddlewareTime::setTimeGenerator( pAtomicClock );

...

TENA::Middleware::init( ... );
```

Reverting to the default `MiddlewareTimeGenerator` may be accomplished by passing an invalid `MiddlewareTimeGeneratorPtr` to `setTimeGenerator()`.

```
TENA::Middleware::Utils::MiddlewareTime::setTimeGenerator( TENA::Middleware::Utils::MiddlewareTimeGeneratorPtr(
    0 ) );
```

IRIG and GPS Clocks Using a BC63x Card

In the range community, it is not uncommon to find IRIG and/or GPS clocks used, specifically via the BC637 or BC635 cards from Microsemi (formerly Symmetricom, formerly Bancomm). Release 6.0.4 (and later) of the TENA Middleware SDK includes support for these BC63x cards as a library and header file bundled with the the MiddlewareSDK. In addition to using the clock in the BC63x card, the library will report any problems with the card in the TENA Console.

The header file, `TENA/Middleware/Clock/BC63x.h` is shown below (somewhat simplified and re-formatted).

TENA/Middleware/Clock/BC63x.h

```
namespace TENA {
    namespace Middleware {
        namespace Clock {
            /// A Clock using a Symmetricom BC637 (or BC635) Card
            class BC63x
                : public TENA::Middleware::Utils::MiddlewareTimeGenerator
            {
                public:
                    enum Mode { GPS, IRIGA_AM, IRIGB_AM, IRIGA_DC, IRIGB_DC };

                    BC63x(
                        Mode mode,
                        TENA::int64 numberOfWarningsToSuppress = 0 ); // Oftentimes the default value of zero is desirable
                    TENA::int64 getTime(); // Reads the time from the BC637 (or BC635) Card
                };
                typedef TENA::Middleware::Utils::SmartPtr< BC63x > BC63xPtr;
            } // End of namespace Clock
        } // End of namespace Middleware
    } // End of namespace TENA
```

Using the BC63x clock in an application is quite easy, e.g.,

A Simple Example of Using the BC63x clock in an Application

```
#include <TENA/Middleware/Clock/BC63x.h>

...
TENA::Middleware::Utils::MiddlewareTime::setTimeGenerator(
    TENA::Middleware::Utils::MiddlewareTimeGeneratorPtr
    new TENA::Middleware::Clock::BC63x(
        TENA::Middleware::Clock::BC63x::Mode::IRIGB_AM ) );
// From this point on, all of the Middleware's timestamps (including the default TENA::Time::create(), etc.)
will use the clock in the BC63x card
//
// For maximum effect, the clock should be set before calling TENA::Middleware::init()
```

A more flexible usage example, making use of the Middleware's application configuration mechanisms, is shown below:

A More Flexible Example of Using the BC63x clock in an Application

```
///////////
// Begin ApplicationConfiguration.cpp like the below:
/////////

#include "ApplicationConfiguration.h"
#include <TENA/Middleware/Clock/BC63x.h>
#include <limits>

void
ApplicationConfiguration::
defineSettings()
{
    this->addSettings( TENA::Middleware::Utils::BasicConfiguration::SETTING_OPTIONAL )
        ( "bc63xClock",
        TENA::Middleware::Utils::Value< TENA::Middleware::Clock::BC63x::Mode >(),
        "If set, then a BC63x timing board will be used in the specified mode, "
        "which is one of {GPS, IRIGA_AM, IRIGB_AM, IRIGA_DC, IRIGB_DC}. If not "
        "set, then the default system clock is used." )
```

```

( "bc63xNumberOfWarningsToSuppress",
  TENA::Middleware::Utils::Value< TENA::int64 >()
  .setInitializer(
    TENA::Middleware::Utils::RangeChecker< TENA::int64 >(
      0, std::numeric_limits< TENA::int64 >::max() ) ),
  "If set, then the corresponding number of warnings from a BC63x timing "
  "board will be suppressed. Note that the TENA::Middleware::Clock::BC64x "
  "implementation automatically increases the number of warnings suppressed "
  "using Fibonacci sequence, so oftentimes leaving this unset or set to "
  "zero is desirable." )

; // End of the optional addSettings() statement

// Add configuration options that this application recognizes

this->addSettings()
( "emEndpoints",
  TENA::Middleware::Utils::Value< std::vector< TENA::Middleware::Endpoint > >()
  .setInitializer( &TENA::Middleware::parseEndpointString ),
  "The endpoint(s) on which the primary executionManager (EM) and "
  "EM replicas, if any, are listening for requests e.g., "
  "-emEndpoints <hostname>:55100 to contact an EM running on the "
  "host <hostname> listening on port 55100. "
  "Multiple EM endpoints must be separated by commas (if using the "
  "same protocol) or semicolons. If semicolons are used, the string "
  "may be enclosed in double quotes or the semicolon escaped with a "
  "backslash (i.e. \"\\;\\\") to avoid shell interpretation issues." )

////////////////////////////////////////////////////////////////////////
//
// And then begin the main() program like the below
//
////////////////////////////////////////////////////////////////////////

ApplicationConfiguration appConfig(
  argc,
  argv,
  "", // Using -configFile option better than typing configFile name here
  "", // Using -configFilePrefix option better than typing configFile prefix
  "" ); // Using -envVarPrefix option better than typing env var prefix here

if ( appConfig["bc63xClock"].isSet() )
{
  std::cout << "Time as reported by Middleware before switching to BC63x clock: "
  << TENA::Middleware::Utils::toTimestamp( TENA::Middleware::Utils::MiddlewareTime::getTime() )
  << std::endl;

TENA::int64 bc63xNumberOfWarningsToSuppress =
  ( appConfig["bc63xNumberOfWarningsToSuppress"].isSet()
    ? appConfig["bc63xNumberOfWarningsToSuppress"].getValue< TENA::int64 >()
    : 0 );

TENA::Middleware::Utils::MiddlewareTime::setTimeGenerator(
  TENA::Middleware::Utils::MiddlewareTimeGeneratorPtr(
    new TENA::Middleware::Clock::BC63x(
      appConfig["bc63xClock"].getValue< TENA::Middleware::Clock::BC63x::Mode >(),
      bc63xNumberOfWarningsToSuppress ) ) );

std::cout << "Time as reported by Middleware after switching to BC63x clock: "
  << TENA::Middleware::Utils::toTimestamp( TENA::Middleware::Utils::MiddlewareTime::getTime() )
  << std::endl;
}

TENA::Middleware::RuntimePtr pTENAmiddlewareRuntime
= TENA::Middleware::init( appConfig.tenaConfiguration() );

```

Middleware Threading Model

Middleware Threading Model

In order to effectively support the network communication activities the TENA Middleware utilizes a configurable thread pool that is able to concurrently handle the processing of incoming network packets. These activities involve either internal middleware processing, subscription callback generation, or remote method invocations. Other than the remote method invocations, the middleware threads are not used in the application programming "space" and operate independently from the application processing.

Description

The TENA Middleware needs to dedicate one or more programming threads to reading incoming network packets. If the middleware is not responsive in reading the incoming network packets, the TCP (Transmission Control Protocol) communication may begin to perform poorly and negatively effect remote applications attempting to communicate with the application not processing the network packets. In the case of UDP (User Datagram Protocol) Multicast communication, if the middleware does not read the network packets fast enough, the network devices can just discard large numbers of the multicast packets.

In an attempt to ensure that the middleware is effectively handling the incoming network packets a pool of programming threads is used to process the incoming packets. The number of threads used by the middleware for this processing is controlled by the configuration parameter `numORBthreads`, which has a default value of 16. It is not recommended that users adjust this configuration parameter without understanding the consequences.

The incoming network packets can be associated with three different activities:

1. Internal middleware coordination requests and responses.
2. Subscription events (e.g., SDO discovery, message receipt).
3. Remote method invocations.

The internal middleware activities are handled by the middleware and there is no interaction with the user application code. The subscriptions events are quickly packaged into callbacks that are placed on a queue until the user application code indicates that the callback queue can be processed. When the user application code invokes the `evokeMultipleCallbacks` method, the user application programming thread used for that invocation is used to process the next callback on the queue which results in executing any user provided code defined in the `Observers` associated with the particular subscription. Multiple simultaneous invocations of `evokeMultipleCallbacks` can be made by the user application code, although the developer is responsible to ensure that the `Observer` code is written in a thread-safe manner with respect to thread concurrency and reentrancy.

In the third activity involving remote method invocations, the middleware thread will immediately be used to invoke the user application code associated with the particular Stateful Distributed Object remote method implementation. Therefore, the user remote method implementation code needs to be written thread-safe – it is possible that multiple programming threads could attempt to execute the remote method implementation code concurrently. Typically, a programming "lock", such as a `mutex` can be used to guard against more than one programming thread from entering the method implementation simultaneously.



All of the middleware API constructs that are used by applications are written to be thread-safe, meaning that applications can use multiple programming threads that simultaneously invoke the same or different middleware API methods. There is no need to guard against concurrent middleware access, or use separate middleware objects (e.g., `Session`) for thread safety issues.

Publication Services

Publication Services

A fundamental architectural element of the TENA Middleware is the use of the "publish-subscribe" paradigm in which applications indicate the types of Stateful Distributed Objects (SDOs, or objects) and Messages that they are going to produce and what types they are interested in receiving. The middleware Publication Services are used by applications to state their intentions of producing objects or messages.

Description

TENA applications can publish Stateful Distributed Objects (SDOs, or objects) and Messages for other applications to receive. SDOs and messages are characterized by a collection of attributes that describe the particular SDO or message instance. The SDOs can support remote methods that provide services that other applications can utilize. SDOs have defined period of existence in which they are created, updated one or more times, and then destroyed. Messages are ephemeral (i.e., exist only for a moment).

In a "publish-subscribe" system, applications indicate what types of data they intend to publish and what types they want to subscribe. The infrastructure then matches the appropriate publishers with the interested subscribers of the same type. Note that although the terms publishers and subscribers are used to describe the behavior of these systems, it is often the situation in which an application is both a publisher and a subscriber.

In the TENA Middleware, when the publishers and subscribers are matched, the communication path is a direct link between the applications to minimize latency. This is in contrast to some publish-subscribe that use a common storage location (aka "blackboard") in which there is a separate mediator process between the publisher and subscriber.

When an application indicates that it intends to publish a particular SDO or message type, the middleware uses that publication action to establish the appropriate communication path between the publisher and all interested subscribers. Subsequent to the publication action, if another application becomes interested in the published type then the middleware will then establish the communication path for this new subscribing application.

The two publication actions performed by applications are: (1) creation of an SDO servant factory, and (2) creation of a message sender. Both of these actions are done within the context of a particular Session. The Session manages the publication and subscription information within an application (see [Execution Management Services documentation](#) for additional information on Sessions).

When an application deletes the servant factory or message sender for a particular SDO model type, this action indicates to the middleware that the application no longer intends to create SDOs or send messages of that type, and the middleware will then no longer establish communication paths with additional subscribers of that type.



When an application creates or destroys a servant factory or message sender, there is distributed communication and processing across the execution to adjust publication and subscription information and communication paths for all of the middleware nodes. Applications should not use servant factories and message senders in a transient manner, e.g., create message sender, send message, then destroy message sender each time a message needs to be sent.

The TENA Middleware supports two different communication mechanisms for delivering SDO and message information from publishers to subscribers: Reliable and Best Effort. These transport types indicate the underlying communication protocol used to deliver the information with TCP ([Transmission Control Protocol](#)) used for Reliable transport and UDP ([User Datagram Protocol](#)) Multicast used for Best Effort transport. Additional information on the transport mechanisms can be found on the [Reliable Transport](#) and [Best Effort Transport](#) documentation pages.

The publishing application indicates what protocol is going to be used for a particular SDO or message. Applications are not required to publish all SDOs or messages using the same transport. In the case of SDOs, the initial *discovery* message and *destruction* message are always delivered using the Reliable transport to ensure that applications don't miss discoveries and destructions since the Best Effort transport is inherently unreliable.

Additional publication details associated with SDOs and messages are defined in the specific documentation pages related to SDOs and Messages. Some of the key pages are:

- [SDO Creation](#)
- [SDO Update](#)
- [Message Sender](#)

C++ API Reference and Code Examples

An example for creating an SDO servant factory is shown in the code fragment below. In this example, the SDO type is `Example::Vehicle` and a static method for this class is used to create the servant factory. A reference to the `Session` is provided to the `create` method since all publications and subscriptions are done in the context of a session.

```
// Create a ServantFactory to create Example::Vehicle SDOs.  
Example::Vehicle::ServantFactoryPtr p_ExampleVehicleServantFactory(  
    Example::Vehicle::ServantFactory::create( pSession ) );
```

Release 6.0.2 Update — An alternative `create` method was added with release 6.0.2 of the middleware in which the `SessionPtr` argument is used, versus the `Session` reference (i.e., `*pSession`) that was used in the previous releases. Users are encouraged to use the `SessionPtr` interface. See [M W-4286](#) for more details.

Once the servant factory has been created, the application can create specific SDO instances of the SDO type, `Example::Vehicles` in this example.

A similar code fragment for creating a message sender (for message type `Example::Notification` in this example) is shown below. The static `create` method requires the application to provide the `Session` reference, which is similar to the SDO servant factory. Note that, unlike the servant factory, the message sender `create` method also requires the `communicationProperties` argument, which is used to define whether the message sender will be using Reliable or Best Effort transport (`TENA::Middleware::Reliable` or `TENA::Middleware::BestEffort`, respectively). The `communicationProperties` specification for SDOs is deferred until the individual SDO instance is created.

```
p_ExampleNotificationSender
= Example::Notification::MessageSender::create(
    pSession,
    communicationProperties );
```

Release 6.0.2 Update — An alternative `create` method was added with release 6.0.2 of the middleware in which the `SessionPtr` argument is used, versus the `Session` reference (i.e., `*pSession`) that was used in the previous releases. Users are encouraged to use the `SessionPtr` interface. See [M W-4286](#) for more details.

Java API Reference and Code Examples

An example for creating an SDO servant factory is shown in the code fragment below. In this example, the static method `Example.Vehicle.ServantFactory.create` used to create the servant factory, associated with a session. The `Session` is provided to the `create` method since all publications are done in the context of a session.

```
// Create a ServantFactory to create Example::Vehicle SDOs.
Example.Vehicle.ServantFactory srvFactory =
    Example.Vehicle.ServantFactory.create( session ) ;
```

Once the servant factory has been created, the application can create specific SDO instances of the SDO type, `Example.Vehicle.Servants` in this example.

A similar code fragment for creating a message sender (for message type `Example.Notification.Servant` in this example) is shown below. The static `create` method requires the application to provide a `Session` reference, similar to the SDO servant factory. Unlike the servant factory, the message sender `create` method also requires the `communicationProperties` argument, which is used to define whether the message sender will be using Reliable or Best Effort transport (`TENA.Middleware.CommunicationProperties.Reliable` or `TENA.Middleware.CommunicationProperties.BestEffort`).

```
Example.Notification.MessageSender notificationSender =
    Example.Notification.MessageSender::create( pSession, communicationProperties );
```

Best Effort Transport

Best Effort Transport

The TENA Middleware supports two forms of communication transport when exchanging SDO updates or Messages between publishing and subscribing applications. The **Best Effort** transport is based on the UDP Multicast communication protocol that is designed to support efficient communication from one publisher to many subscribers. In return for this increased scalability, the UDP Multicast protocol is inherently unreliable and application designers need to consider the consequences of unreliable communication with the SDO updates or messages that may be using this transport.

Description

TENA applications that publish Stateful Distributed Objects (SDOs, or objects) and Messages can choose to use either a Reliable or Best Effort transport for the communication. In the case of SDOs, only the updates of the attribute values are controlled by the transport setting — the discovery and destruction messages sent from the publisher to subscribers are always done using the Reliable transport.

The Best Effort transport is based on the [UDP \(User Datagram Protocol\)](#) which is designed to be a one-to-many communication protocol in which the network devices are used to replicate the network packets to delivery them to the appropriate destinations. In order to provide increased performance and scalability, the UDP Multicast protocol is unreliable and network devices and applications are free to silently drop UDP packets if deemed necessary. Since there is no reliability with the underlying protocol, TENA applications using the Best Effort transport are unable to guarantee that a particular SDO update or message was successfully delivered to the intended destinations.

The UDP Multicast protocol makes use of reserved multicast IP addresses that are used for applications to either send or receive multicast network packets. The defined range of multicast addresses is between 239.192.0.0 and 239.255.255.255. When using Best Effort transport, the Execution Manager (EM) for the execution needs to be configured with the appropriate multicast address range used for the Best Effort traffic. See [EM configuration parameters](#) for a description of the `multicastProperties` configuration parameter.

Usage Considerations

Since TENA Messages are ephemeral, the Reliable transport mechanism is the recommended protocol for sending all Messages. Using Best Effort transport could result in messages being dropped and neither the publisher or subscriber would know about the missing message. It is possible that Best Effort for messages could be used if the the applications employed some other technique for detecting or handling lost messages.

In the case of SDO updates, the Reliable transport is the recommended protocol unless performance issues are encountered with testing. Typically, in a local area network (LAN) environment there is no appreciable performance gain with Best Effort transport. As the network becomes larger with more geographically dispersed sites and applications, the Best Effort transport will provide increased scalability for sending SDO updates. Although event designers must consider the unreliable consequences of Best Effort.

When using UDP Multicast, the more multicast addresses that are used by the execution, the better the segmentation of traffic will be to minimize the amount of unwanted traffic received by applications. For example, if only a single multicast address was specified, all Best Effort traffic would be sent to this one multicast address and all subscribers would receive all of the traffic and there would be no segmentation. On the other hand, if every SDO model type had its own multicast address, then subscribers would only receive the types of interest and there would be perfect segmentation. Therefore, more multicast addresses provides better data segmentation, but many network devices can only support a finite number of multicast addresses until there is performance degradation. The degradation is typically such that the network device will no longer perform multicast address filtering and all multicast traffic will be treated as broadcast traffic. Testing is the only reliable means to evaluate the performance of network devices with respect to the number of multicast addresses used.

Note that the middleware does not implement send-side filtering with Best Effort communication and will transmit network packets independent of the subscription state of other applications. The intent of multicast communication is to rely on the network devices to performing filtering to prevent network traffic from flowing where it is not needed.

The middleware does not currently expose a user API to control the multicast assignment algorithm (e.g., make "Tank" updates go to multicast address 239.239.100.27). Instead the middleware uses a hashing algorithm based on the type name, number of available addresses, and whether advanced filtering tags are utilized. Users can determine the particular multicast address associated with an SDO update or message by examining the [metadata](#) of the received update or message. Additionally the [TENA Console](#) provides the ability to calculate the multicast address for a particular type. If users are interested in the details of the multicast assignment algorithm, it is defined in the [multicast address assignment documentation page](#).

C++ API Reference and Code Examples

The communication transport for SDO updates and messages is defined by an enumeration named `TENA::Middleware::CommunicationProperties` which has a value of either `TENA::Middleware::Reliable` or `TENA::Middleware::BestEffort`. Applications define the communication transport when an SDO is created, or when a message sender is created, as shown in the code fragment below.

```

TENA::Middleware::CommunicationProperties communicationProperties( TENA::Middleware::BestEffort );

Example::Vehicle::ServantPtr pServant(
    pServantFactory->createServant(
        *initializer,
        communicationProperties ) );

p_ExampleNotificationSender
= Example::Notification::MessageSender::create(
    pSession,
    communicationProperties );

```

Release 6.0.2 Update — An alternative `create` method was added with release 6.0.2 of the middleware in which the `SessionPtr` argument is used, versus the `Session` reference (i.e., `"*pSession"`) that was used in the previous releases. Users are encouraged to use the `SessionPtr` interface. See [M W-4286](#) for more details.

Java API Reference and Code Examples

The communication transport for SDO updates and messages is defined by an enumeration named `TENA::Middleware::CommunicationProperties` which has a value of either `Reliable` or `BestEffort`. Applications define the communication transport when an SDO is created, or when a message sender is created, as shown in the code fragment below.

```

vehicleServant =
    servantFactory->createServant( initializer, TENA.Middleware.CommunicationProperties.BestEffort );

notificationSender =
    Example.Notification.MessageSender.create( session, TENA.Middleware.CommunicationProperties.BestEffort );

```

Multicast Address Assignment

Multicast Address Assignment

When using the TENA Middleware's Advanced Filtering mechanism with Best Effort (i.e., UDP Multicast) communication, the middleware needs to calculate the particular multicast address for a particular object model type and the particular advanced filtering tag. In some cases, application developers are interested in knowing the actual algorithm used to calculate this multicast address.

Description

Some TENA users have asked how the TENA Middleware computes the multicast address on which Messages or SDO Updates are transmitted.

! These Implementation Details Could Change in a Future Middleware Release, e.g., Release 6.1

The algorithm is internal and may change in future releases, so it should not be relied upon. For Release 6.0.x, the algorithm's pseudocode is as follows:

```
Let baseAddr = the base multicast IPv4 address as given to the execution manager, represented as a 32-bit
integer in host byte order
Let numMulticastAddresses = the number of multicast addresses as given to the execution manager
Let typename = the string name of the type of the SDO being published
Let tag = the 32-bit integer advanced filtering tag that the SDO is using

Let typeID = typename run through java.util.zip.CRC32
Let offset = typeID ^ tag // where "^" means bitwise XOR
Let offset = offset % numMulticastAddresses // where % means "modulo"
Let addr = baseAddr + offset
```

This resulting "addr" value is the multicast IPv4 address (as a 32-bit integer in host byte order) on which the SDO's updates will be published.

One advantage of the above function is that it allows the application to choose a tag to achieve a desired multicast offset. To do so, one must take advantage of the fact that the bitwise XOR function is the inverse of itself.

Since (in pseudocode):

```
offset = type ^ tag
```

one can simply choose a tag equal to `(type ^ desiredOffset)`.
Plugging that in, you get:

```
offset = type ^ type ^ desiredOffset
```

or equivalently,

```
offset = desiredOffset
```

Here is some actual code that implements the above pseudocode and publishes an Example::Vehicle SDO on a given multicast offset:

```

static TENA::uint32 const servantMulticastOffset = programConfig["servantMulticastOffset"].getValue< TENA::uint32 >(); // This is the desired offset
static TENA::uint32 const servantTypeID_low32 = static_cast< TENA::uint32 >( Example::Vehicle::typeID.value() & 0x00000000ffffffffULL );
static TENA::uint32 const servantTypeID_high32 = static_cast< TENA::uint32 >( ( Example::Vehicle::typeID.value() & 0xffffffff00000000ULL ) >> 32 );
TENA::uint32 servantTag = servantMulticastOffset ^ servantTypeID_low32 ^ servantTypeID_high32;

std::cout << "Publishing BaseSDO servant with tag " << servantTag << " to achieve a multicast offset of " <<
servantMulticastOffset << "." << std::endl;

Canonical::BaseSDO::ServantPtr servantPtr =
    servantFactPtr->createServant(
        pRemoteMethods,
        *initializer,
        std::auto_ptr< Canonical::BaseSDO::SDOTag >(
            new Canonical::BaseSDO::SDOTag(
                servantTag ) ),
        TENA::Middleware::BestEffort ) );

```

Note well that the multicast group used to transmit the updates to the Example::Vehicle SDO created above will be equal to the base multicast group given to the executionManager, plus the servantMulticastOffset (shown above), modulo the total number of multicast groups given to the executionManager.

Reliable Transport

Reliable Transport

The TENA Middleware supports two forms of communication transport when exchanging SDO updates or Messages between publishing and subscribing applications. The **Reliable** transport is based on the TCP communication protocol that is designed to support reliable point-to-point communication. In the case of the middleware having one publisher sending information to many subscribers, the Reliable transport mechanism performs separate individual network write operations to each subscribing application. This results in a reliable communication mechanism (i.e., the application knows if an SDO update or message is not successfully delivered), but scalability can be a consideration that needs to be examined by the event designers.

Description

TENA applications that publish Stateful Distributed Objects (SDOs, or objects) and Messages can choose to use either a Reliable or Best Effort transport for the communication. In the case of SDOs, only the updates of the attribute values are controlled by the transport setting — the discovery and destruction messages sent from the publisher to subscribers are always done using the Reliable transport.

The Reliable transport is based on the [TCP \(Transmission Control Protocol\)](#) which attempts to provide reliable communication protocol between two nodes. The underlying protocol is designed to re-transmit packets that are not properly delivered, so the protocol can tolerate some packet loss across the network link (although performance will degrade). When TCP is unable to successfully transmit network packets to the intended destination, the operating system will notify the middleware of the error, which then will be reported as an [Alert](#) to the application and event operators.

When an application attempts to use Reliable transport for SDOs or messages between a publisher and subscriber, the middleware must first establish a TCP connection between the two "endpoints" (if a connection does not already exist). An endpoint is the [network address](#) that defines the IP address and port number that is used for IP ([Internet Protocol](#) based protocols, such as TCP).

With a mis-configured or faulty network, as well as in the case of an incorrect destination endpoint, the TCP connection request will timeout waiting for a response from the remote endpoint. The connection timeout value used by the middleware is configurable and obviously needs to be set to be larger than the expected connection time required for the various network links used by the application. The default value for the connection timeout configuration parameter, `connectionTimeoutInMilliseconds`, is set to 10 seconds to support large wide area networks in which TENA is commonly used. A downside to using too large of a connection timeout value is that the middleware can encounter a delay in waiting for the timeout in the case of a communication fault. See [middleware configuration parameters](#) documentation page for a description of the middleware configuration parameters, including `connectionTimeoutInMilliseconds`.

The TENA Middleware, by default, will make multiple attempts to perform a failed network write operation. The intent of this behavior is to try to make the middleware more resilient to transient network problems. The number of retries with a transient network write error is controlled by a configuration parameter, `corbaTransientRetries`, and the default value is set to three.

As mentioned previously, TCP is a point-to-point communication protocol, so a TENA publisher is required to perform multiple TCP network writes for each subscriber when using the Reliable transport. When there are many subscribing applications, the publishing application can become burdened with performing all of the network write operations and using Reliable transport can become a scalability limitation for large exercises. The alternative is to use the Best Effort transport which uses the UDP Multicast communication protocol that relies on network devices to replicate the network packets to all of the interested applications across the network. Refer to the Usage section in this page and the [Best Effort transport page](#) for considerations regarding the use of Reliable and Best Effort.

Usage Considerations

Since TENA Messages are ephemeral, the Reliable transport mechanism is the recommended protocol for sending all Messages. Using Best Effort transport could result in messages being dropped and neither the publisher or subscriber would know about the missing message. It is possible that Best Effort for messages could be used if the the applications employed some other technique for detecting or handling lost messages.

In the case of SDO updates, the Reliable transport is the recommended protocol unless performance issues are encountered with testing. Typically, in a local area network (LAN) environment there is no appreciable performance gain with Best Effort transport. As the network becomes larger with more geographically dispersed sites and applications, the Best Effort transport will provide increased scalability for sending SDO updates. Although event designers must consider the unreliable consequences of Best Effort.

C++ API Reference and Code Examples

The communication transport for SDO updates and messages is defined by an enumeration named `TENA::Middleware::CommunicationProperties` which has a value of either `TENA::Middleware::Reliable` or `TENA::Middleware::BestEffort`. Applications define the communication transport when an SDO is created, or when a message sender is created, as shown in the code fragment below.

```

TENA::Middleware::CommunicationProperties communicationProperties( TENA::Middleware::Reliable );

Example::Vehicle::ServantPtr pServant(
    pServantFactory->createServant(
        *initializer,
        communicationProperties ) );

p_ExampleNotificationSender
= Example::Notification::MessageSender::create(
    pSession,
    communicationProperties );

```

Release 6.0.2 Update

— An alternative `create` method was added with release 6.0.2 of the middleware in which the `SessionPtr` argument is used, versus the `Session` reference (i.e., `*pSession`) that was used in the previous releases. Users are encouraged to use the `SessionPtr` interface. See [Mw-4286](#) for more details.

Java API Reference and Code Examples

The communication transport for SDO updates and messages is defined by an enumeration named `TENA::Middleware::CommunicationProperties` which has a value of either `Reliable` or `BestEffort`. Applications define the communication transport when an SDO is created, or when a message sender is created, as shown in the code fragment below.

```

vehicleServant =
    servantFactory->createServant( initializer, TENA.Middleware.CommunicationProperties.Reliable );

notificationSender =
    Example.Notification.MessageSender.create( session, TENA.Middleware.CommunicationProperties.Reliable );

```

Subscription Services

Subscription Services

A fundamental architectural element of the TENA Middleware is the use of the "publish-subscribe" paradigm in which applications indicate the types of Stateful Distributed Objects (SDOs, or objects) and Messages that they are going to produce and what types they are interested in receiving. The middleware Subscription Services are used by applications to indicate what types of SDOs or messages that they are interested in receiving from other applications.

Description

TENA applications can subscribe to receive Stateful Distributed Objects (SDOs, or objects) and Messages that are produced by other applications. SDOs and messages are characterized by a collection of attributes that describe the particular SDO or message instance. The SDOs can support remote methods that provide services that other applications can utilize. SDOs have defined period of existence in which they are created, updated one or more times, and then destroyed. Messages are ephemeral (i.e., exist only for a moment).

In a "publish-subscribe" system, applications indicate what types of data they intend to publish and what types they want to subscribe. The infrastructure then matches the appropriate publishers with the interested subscribers of the same type. Note that although the terms publishers and subscribers are used to describe the behavior of these systems, it is often the situation in which an application is both a publisher and a subscriber.

When an application indicates that it is interested in receiving SDOs or messages of a particular type, the middleware will convey that interest information to all active publishers in the execution to determine if there is a match between what the subscriber is interested in receiving and what the publisher is producing. If there is a match, the publisher will establish the appropriate communication path to be used for sending SDO and message information that match the subscriber's interest.

An application indicates its interest in a particular SDO or message type by invoking the appropriate subscribe method. The subscribe methods are defined as static methods for the class associated with the particular SDO model type. For example, if the SDO type of interest is `Example::Vehicle`, then the appropriate subscribe method would be defined in the `Example::Vehicle` class. A more complete example of these method invocations are shown in the API section of this documentation.

Observers

Applications that are subscribed to a particular SDO or message type can receive notification from the middleware when information is received from matching SDOs or messages. The notification mechanism is based on the Observer Design Pattern in which an application is able to register one or more observer classes that can perform some application specific function when an event associated with a subscription occurs. The possible subscription events are described in the table below.

Subscription Events

Event Type	Observer Method	Description
SDO Discovery	<code>discoveryEvent</code>	Indicates that a new SDO has been discovered.
SDO State Change	<code>stateChangeEvent</code>	Indicates that a state change has been received for an existing SDO.
SDO Destruction	<code>destructionEvent</code>	Indicates that a destruction notification has been received for an existing SDO.
SDO Left Scope	<code>leftScopeEvent</code>	Indicates that the subscription has ceased to be able to receive updates for the SDO.
SDO Entered Scope	<code>enteredScopeEvent</code>	Indicates that the subscription has resumed being able to receive updates for the SDO.
Message	<code>messageEvent</code>	Indicates that a new message has been received.

As an example, consider a display application that draws a map with the location of all tanks operating within the execution. When the display application subscribes to the SDO type `Tank`, it can attach an observer with custom code that implements behavior when tanks are discovered, updated, or destroyed so that the map display indicates the current location of all tanks. The observer code for handling the SDO Discovery event may flash an icon at the initial location of the particular tank. When an SDO State Change event is handled, the application's observer code could check if the position of the tank changed, and if so, redraw the icon in the correct position. An SDO Destruction event may be handled with the application's observer code by removing the icon from the map.

User defined observers are attached to an instance of a type specific `Subscription` class (e.g., `Example::Vehicle::Subscription`). These `Subscription` instances are used by the application to represent particular subscription requests. In the case of SDO subscriptions, the `Subscription` instance also maintains a list of SDO proxies that have been discovered based on the subscription request. Applications can iterate through the list of discovered SDO proxies when necessary, so there is no need for applications to maintain a separate list of discovered proxies.

In summary, subscribing applications write custom observer code that provides application specific behavior in response to a subscriptions event (e.g., SDO discovery, message receipt). Multiple independent observers can be attached to the same `Subscription` to provide different capabilities, such as updating a display, performing data logging. The code for these observers can be packaged and shared among applications to promote code reuse for common subscription capabilities.

Additional subscription observer information can be found in the [Observer Mechanism documentation page](#).

Callbacks and Callback Processing

Underneath the observer mechanism is a callback framework (which was exposed in Release 5 of the middleware). The callback framework handles the same subscription events supported by the Observer Mechanism, but there are several advantages of the Observer pattern over the Callback pattern:

- Multiple independent observers can be attached to a subscription, whereas under the callback pattern all of the event handling code needs to be combined together in a single method. This pattern allows a developer to create a common observer class that can easily be integrated and used by multiple applications, regardless of what other observers the application may be using.
- Callback pattern required specific event code for a particular type to be scattered in multiple class files (i.e., discovery callback class in one set of files, state change callbacks in another set of files). Observer event handling code for a particular type is contained in a single class (i.e., discovery, state change, etc. code for a particular type and particular observer are contained in single class).

With respect to the subscription events, the middleware queues up the events (in the form of callbacks) that when processed will invoke the appropriate observer event methods. The middleware does not asynchronously process these events because there may need to be coordination with the application related to processing thread control. Instead the application needs to provide a processing thread (or threads) to perform the processing for the subscription events. The application provides this thread (or threads) by invoking the `evokeMultipleCallbacks` method on the `Session`.

The primary purpose of this structure is to accommodate different application threading models. If the application wants to only use a single programming thread, then the application will need to balance any application specific processing with the middleware callback processing in a time sharing manner. If the application wants to use multiple programming threads, then one or more threads can be dedicated to processing of subscription events. In the multiple application thread scenario, the developer is required to ensure that their subscription processing code (i.e., the observer event methods) is thread safe with respect to other application threads.

Additional information related to the callback mechanism can be found in the [callback documentation page](#).

Unsubscribe

If an application is no longer interested in a particular subscription request, the application can unsubscribe. This will indicate to the middleware that the application is no longer interested in receiving discoveries or updates for SDOs of the given type or Messages of the given type.

After an application has unsubscribed from a particular SDO type, the application may still hold onto proxies for discovered SDOs of that SDO type. These proxies, though no longer receiving updates, will continue to support remote methods and receive destruction notifications. Applications can destruct the proxies of the discovered SDOs if they are no longer interested in those SDOs.

Additional information on the unsubscribe procedures can be found in either the [SDO unsubscribe documentation page](#) or the [message unsubscribe documentation page](#).

Additional subscription details associated with SDOs and messages are defined in the specific documentation pages related to SDOs and Messages. Some of the key pages are:

- [SDO Subscription](#)
- [SDO Proxy](#)
- [SDO Unsubscribe](#)
- [Message Subscription](#)
- [Message Unsubscribe](#)

C++ API Reference and Code Examples

The basic operations associated with subscribing to SDOs (Stateful Distributed Objects, or objects) is illustrated in the C++ code fragment below. A subscription is represented within the application software as an instance of a type specific `Subscription` class, `Example::Vehicle::SubscriptionPtr` in this example. The Subscriptions are contained within a managed pointer (see documentation in [Smart Pointers page](#)) which is used to properly manage multiple references to this `Subscription`. An application constructs a new `Subscription` (using default constructor arguments in this example), attaches any user defined `Observers` needed for the subscription event processing, and then invokes the type specific `subscribe` method. In this simple example, no `Observers` are attached and default arguments are used for the `subscribe` method.

```
Example::Vehicle::SubscriptionPtr pExampleVehicleSubscription(
    new Example::Vehicle::Subscription );

// Declare the application's interest in Vehicle SDOs.
Example::Vehicle::subscribe(
    pSession,
    pExampleVehicleSubscription );
```

Release 6.0.2 Update — An alternative `subscribe` method was added with release 6.0.2 of the middleware in which the `SessionPtr` argument is used, versus the `Session` reference (i.e., `"*pSession"`) that was used in the previous releases. Users are encouraged to use the `SessionPtr` interface. See [M W-4286](#) for more details.

After the application invokes this `subscribe` method, the other applications in the execution will begin to provide information about any vehicle SDOs that those applications have created. Since there are no `Observers` attached to this subscription, the subscribing application will not receive any notification of vehicle discoveries, updates, destructions, etc. Instead, the application will need to operate in a "polling" manner in which it can periodically check if the subscription has discovered any vehicles. The code fragment below shows how the application can access the list of discovered SDOs from this subscription.

The `SDOList` returned from the `getDiscoveredSDOList` method is a copy of the discovered SDO list at the time that the method was invoked. New SDOs may be discovered or existing SDOs in this list may be destroyed while the application is iterating through this list.

```
Example::Vehicle::SDOList discoveredList(
    pExampleVehicleSubscription->getDiscoveredSDOList() );

std::cout << "There are " << discoveredList.size()
    << " discovered Example::Vehicle SDOs on the list.\n"
    << std::endl;
```

Additional details on the API related to SDO subscriptions can be found in the API section of the [SDO subscription documentation page](#).

An example for subscribing the messages is shown in the code fragment below. An `Example::Notification::SubscriptionPtr` is created as is with SDO subscriptions. Since messages only exist as they are delivered, it is necessary for an application to attach an `Observer` in order to do anything useful with message subscriptions. In the example below, a `CountingObserver` is attached to the `Subscription`. The message `CountingObserver` is a simple observer that simply counts the received messages. **⚠ This `CountingObserver` is not part of the middleware API, but is a simple example of an observer implementation that is delivered in the example code.**

```
Example::Notification::SubscriptionPtr
pExampleNotificationSubscription(
    new Example::Notification::Subscription );

MyApplication::Example_Notification::CountingObserverPtr
pExampleNotificationCountingObserver(
    new MyApplication::Example_Notification::CountingObserver() );

pExampleNotificationSubscription->addObserver( pExampleNotificationCountingObserver );

Example::Notification::subscribe(
    pSession,
    pExampleNotificationSubscription );
```

Release 6.0.2 Update — An alternative `subscribe` method was added with release 6.0.2 of the middleware in which the `SessionPtr` argument is used, versus the `Session` reference (i.e., `*pSession`) that was used in the previous releases. Users are encouraged to use the `SessionPtr` interface. See [M W-4286](#) for more details.

As with the SDO specific `subscribe` method invocation, the subscription is associated with the application's previously created `Session`. Additional details related to message subscription API can be found in the [Message Subscription documentation page](#).

All subscribing applications are required to call the `evokeMultipleCallbacks` method on the appropriate `Sessions` in order for the `Observer` event handling code to be invoked by the middleware. In a single threaded application, the main loop of the application will typically time slice between application processing and callback processing. Multi-threaded applications can dedicate one or more programming threads to the callback processing, but care is required to ensure that the `Observer` event handling code is "thread safe" (i.e., handle thread concurrency with the `Observer` event method code and handle reentrancy if multiple callback threads are active). The code fragment below shows a single threaded application that has a main processing loop that calls the `evokeMultipleCallbacks` method for a configurable period of time each loop iteration.

```
for ( TENA::uint32 i = 1; i <= numberIterations; ++i )
{
    pSession->evokeMultipleCallbacks( microsecondsPerIteration );
    ...
    // perform other processing
}
```

Additional details on the API related to the callback processing can be found in the API section of the [callback processing documentation page](#).

Java API Reference and Code Examples

The basic operations associated with subscribing to SDOs (Stateful Distributed Objects, or objects) is illustrated in the Java code fragment below. A subscription is represented within the application software as an instance of a type specific `Subscription` class, `Example.Vehicle.Subscription` in this example. An application constructs a new `Subscription` (using default constructor arguments in this example), attaches any user defined `Observer`s needed for the subscription event processing, and then invokes the type specific `subscribe` method. In this simple example, no `Observers` are attached and default arguments are used for the `subscribe` method.

```
Example.Vehicle.Subscription vehicleSubscription = new Example.Vehicle.Subscription();

// Declare the application's interest in Vehicle SDOs.
Example.Vehicle.Subscription.subscribe( pSession, vehicleSubscription );
```

After the application invokes this `subscribe` method, the other applications in the execution will begin to provide information about any `Vehicle` SDOs that those applications have created. Since there are no `Observers` attached to this subscription, the subscribing application will not receive any notification of `Vehicle` discoveries, updates, destructions, etc. Instead, the application will need to operate in a "polling" manner in which it can periodically check if the subscription has discovered any `Vehicles`. The code fragment below shows how the application can access the list of discovered SDOs from this subscription.

The method `{getDiscoveredSDOlist}` on the `Subscription` returns a `Collection<Vehicle.Proxy>`. This collection contains the discovered SDOs at the time of the call. New SDOs may be discovered or existing SDOs in this list may be destroyed while the application is iterating through this list.

```
Collection<Example.Vehicle.Proxy> discoveredSDOs =
    vehicleSubscription->getDiscoveredSDOlist() );

System.out.println("There are " + discoveredSDOs + " Vehicle SDOs");
```

Additional details on the API related to SDO subscriptions can be found in the API section of the [SDO subscription documentation page](#).

An example for subscribing to messages is shown in the code fragment below. An `Example.Notification.Subscription` is created as with SDO subscriptions. Since messages are ephemeral it is necessary for an application to attach an `Observer` in order to do anything useful with message subscriptions. Since messages only exist as they are delivered, it is necessary for an application to attach an `Observer` in order to do anything useful with message subscriptions. In the example below, a `CountingObserver` is attached to the `Subscription`. The message `CountingObserver` is a simple observer that simply counts the received messages. ***⚠ This CountingObserver is not part of the middleware API, but is a simple example of an observer implementation that is delivered in the example code.***

```
Example.Notification.Subscription notificationSubscription =
    new Example.Notification.Subscription();

org.Example.Example_Notification.CountingObserver countingObserver =
    new org.Example.Example_Notification.CountingObserver( 1024 ) );

notificationSubscription.addObserver( countingObserver );

Example.Notification.subscribe( session, notificationSubscription );
```

As with the SDO specific `subscribe` method invocation, the subscription is associated with the application's previously created `Session`. Additional details related to message subscription API can be found in the [Message Subscription documentation page](#).

All subscribing applications are required to call the `evokeMultipleCallbacks` method on the appropriate `Sessions` in order for the `Observer` event handling code to be invoked by the middleware. In an application with a single thread for interacting with middleware events, the thread will typically time slice between application processing and callback processing. Multi-threaded applications can dedicate one or more programming threads to the callback processing, but care is required to ensure that the `Observer` event handling code is "thread safe" (i.e., handle thread concurrency with the `Observer` event method code and handle reentrancy if multiple callback threads are active). The code fragment below shows an application with a single thread allocated for processing.

```
for ( int i = 1; i <= numberIterations; ++i )
{
    session.evokeMultipleCallbacks( microsecondsPerIteration );
    ...
    // perform other processing
}
```

Additional details on the API related to the callback processing can be found in the API section of the [callback processing documentation page](#).

Best Match

Best Match

The Best Match feature permits applications to subscribe to multiple types within an SDO or Message hierarchy, but only discover the SDO or receive the Message on the most matching subscription. Since subsequent subscriptions can modify the resulting best match of an SDO, applications need to be prepared for handling object "scope" events typically associated with Advanced Filtering.

⚠ — Available for TENA Middleware version 6.0.3 and beyond.

Description

TENA applications often make subscriptions with overlapping interests. If a published object matches multiple subscriptions in a single application, then by default, all matching subscriptions receive the published data. However, if a subscribing application uses BestMatch, then only the most specific matching (i.e., best-matching) subscription receives the published data.

For example, suppose an application subscribes to all Tank SDOs and to all Vehicle SDOs. Since a Tank is a Vehicle (i.e., Tank inherits from Vehicle), the two subscriptions overlap. If a Tank is published, the subscribing application would normally receive two discovery callbacks, one in the Tank subscription and one in the Vehicle subscription. Similarly, if the Tank's state is changed, the subscribing application would normally receive two state change callbacks, one in each subscription. However, if the subscribing application prefers to receive data for the Tank in only one subscription, it can use the BestMatch feature.

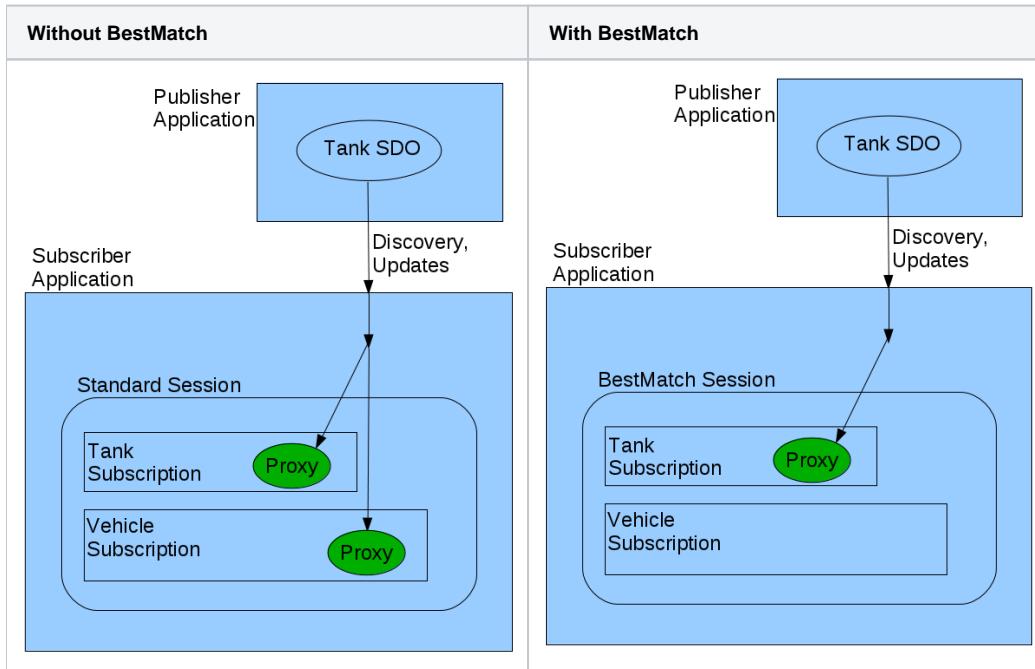
Each TENA Middleware Session has a `bestMatch` attribute, which is set when the Session is created. By default, Sessions are created with `bestMatch` disabled. Passing `true` for the optional second parameter (`bestMatch`) to the `createSession()` method creates the Session with `bestMatch` enabled. If `bestMatch` is enabled in a Session, then whenever SDO or Message data is delivered to subscriptions in the Session, only the most specific matching subscriptions will receive the data. Subscription specificity is defined by the following two rules, in decreasing order of applicability:

1. A more derived subscription is more specific than a less derived subscription.
2. A non-wildcard (i.e., uses [Advanced Filtering](#)) subscription is more specific than a wildcard subscription.

Examples (a Tank with advanced filtering "tag 7" is published):

- An application subscribed to Tank and Vehicle in a BestMatch Session will receive Tank data in the Tank subscription (by Rule 1).
- An application subscribed to Tank and Tank[tag 7] in a BestMatch Session will receive Tank[tag 7] data in the Tank[tag 7] subscription (by Rule 2).
- An application subscribed to Tank and Vehicle[tag 7] in a BestMatch Session will receive Tank[tag 7] data in the Tank subscription (since Rule 1 trumps Rule 2).

System Diagrams Without and With BestMatch



Usage Considerations

A user can create a BestMatch Session when constructing the Session:

```
TENA::Middleware::SessionPtr pSession
pExecution->createSession( "my bestMatch session", true /* bestMatch */ ) );
```

The BestMatch status of a Session can be queried as follows:

```
if ( pSession->isBestMatchEnabled() ) {
    std::cout << "Best match is enabled!";
}
```

Subscription Changes Affecting Proxy Scope

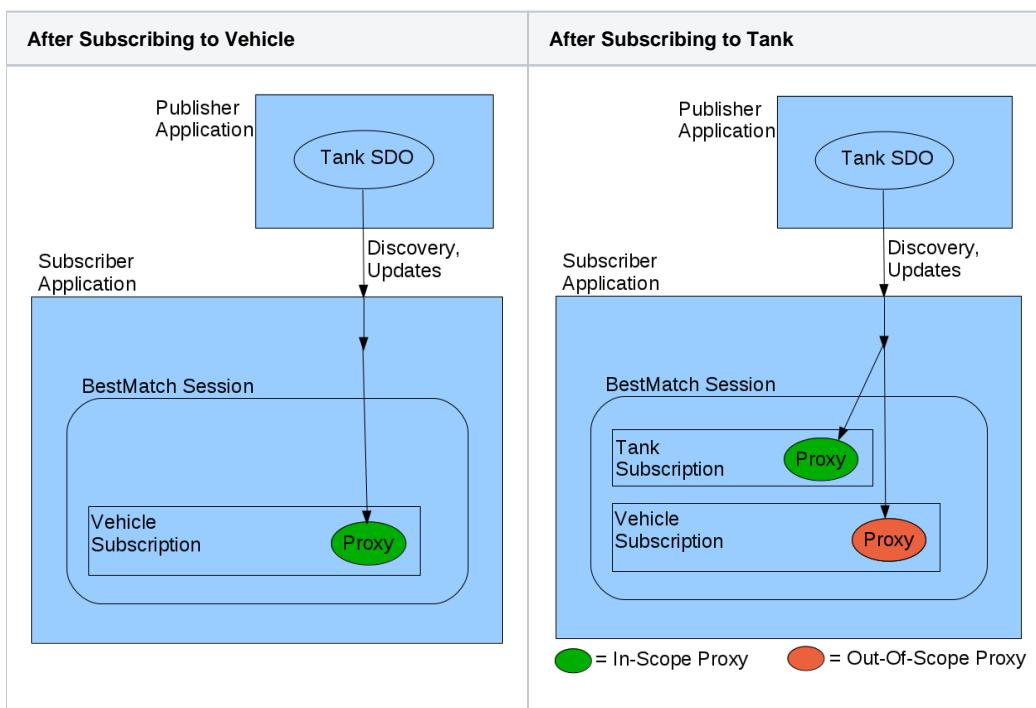
Subscription changes in a BestMatch Session can affect whether existing subscriptions in that Session receive published data.

Example 1:

A publishing application is repeatedly modifying the state of a Tank.

A subscribing application uses a BestMatch Session to subscribe to Vehicle, then to Tank.

Upon making the Vehicle subscription, the subscribing application discovers the Tank SDO (as a Vehicle) and begins receiving updates for it. Then, upon making the Tank subscription, the subscribing application discovers the Tank SDO (as a Tank) and begins receiving updates for it. But since the subscribing application is using a BestMatch subscription, it can't be receiving updates for the Tank SDO on both the Vehicle and Tank subscriptions, so the Vehicle subscription (which is no longer the best match for the Tank SDO) stops receiving updates. At that point, since the Vehicle Proxy is no longer receiving updates, the Middleware delivers it a LeftScope Observer event (see [Observer Mechanism](#)).



i — Users are encouraged to perform their BestMatch subscriptions in most derived to least derived order. This will prevent the initial discovery or message receipt of proxies or messages that are not the best match.

Example 2:

A publishing application is repeatedly modifying the state of a Tank.

A subscribing application uses a BestMatch Session to subscribe to Vehicle, then to Tank.

At this point, the subscribing application has an out-of-scope Vehicle Proxy and an in-scope Tank Proxy, as described in Example 1.

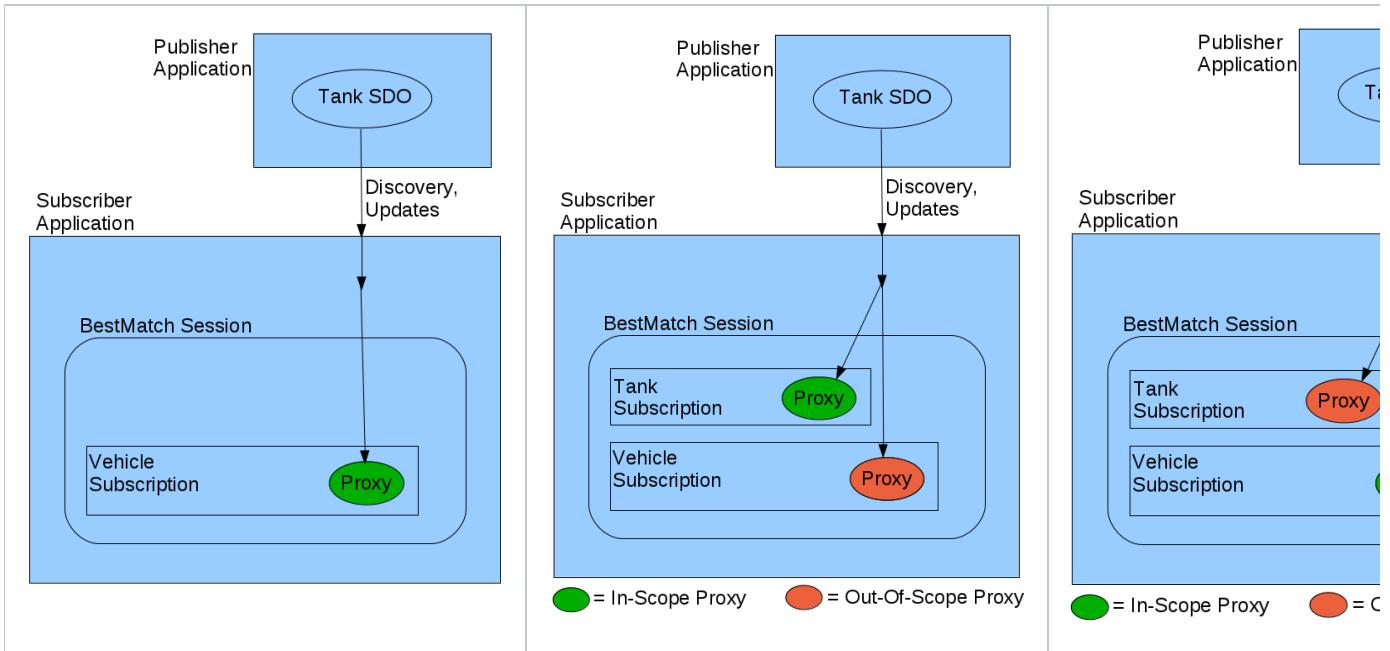
The subscribing application then unsubscribes from Tank.

At this point, the application is subscribed to Vehicle but not to Tank, so it must stop receiving updates for the Tank.

This means that Vehicle is now the best-matching subscription for the remote Tank.

Therefore, the out-of-scope Vehicle Proxy comes back into scope, receives an EnteredScope callback, and resumes receiving updates.

After Subscribing to Vehicle	After Subscribing to Tank	After Unsubscribing from Tank
------------------------------	---------------------------	-------------------------------



Alternatives

If an application developer finds it undesirable for subscription changes to affect the scope of Proxies in other subscriptions, the application developer can do any of the following:

- refrain from using the BestMatch feature;
- use multiple Sessions to isolate subscriptions, thereby preventing interactions between them; or
- subscribe in order of decreasing specificity (i.e., derived-then-base or tagged-then-wildcard), and unsubscribe in order of increasing specificity (i.e., base-then-derived or wildcard-then-tagged).

Instance Subscriptions

Instance subscriptions are not treated specially with regard to BestMatch. If the BestMatch specificity rules (defined above) dictate that an instance subscription is not the best-matching subscription for the given SDO, then the Proxy resulting from the instance subscription does not receive updates.

Example 1:

An application publishes a Tank SDO and updates it repeatedly.

Another application uses a BestMatch Session to perform:

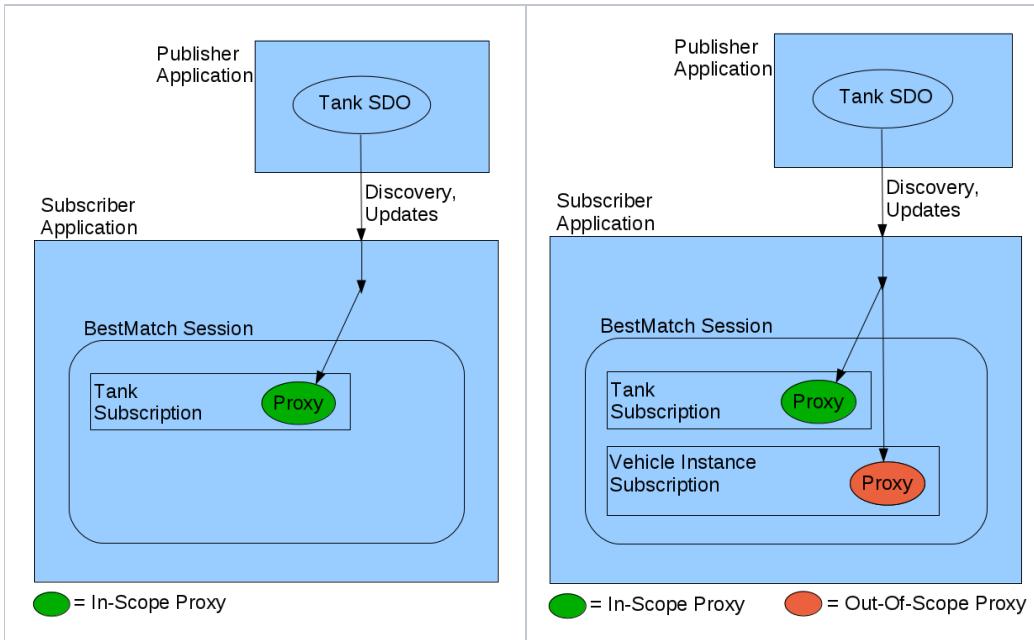
- a Tank subscription, and
- an instance subscription to the published Tank SDO using a Vehicle SDOpointer .

Since the instance subscription is performed at the Vehicle level, the type-based Tank subscription is the better match.

Therefore, the Vehicle Proxy resulting from the instance subscription is out-of-scope from birth, and receives no updates.

The subscribing application instead discovers the Tank via the Tank subscription. The Proxy resulting from *that* subscription receives the Tank updates.

After Subscribing to Tank	After Instance-Subscribing to Tank as a Vehicle
----------------------------------	--



Example 2:

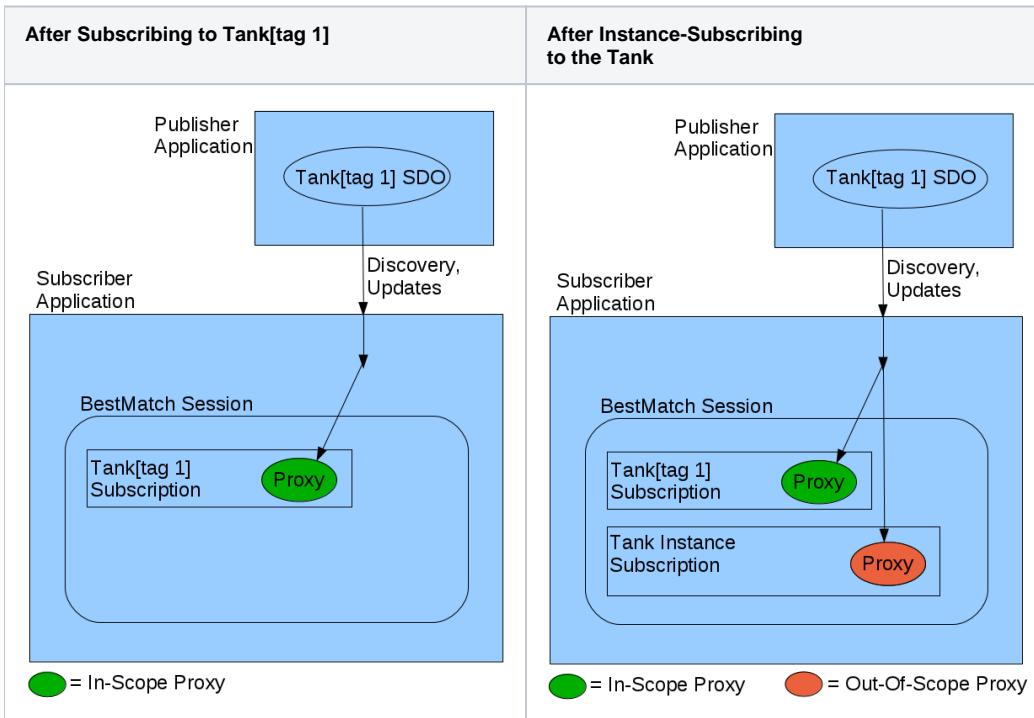
An application publishes a Tank SDO with Advanced Filtering tag 1 and updates it repeatedly.
Another application uses a BestMatch Session to perform:

- a Tank subscription with advanced filtering tag 1, and
- an instance subscription to the published Tank SDO.

Since the instance subscription is performed using a wildcard subscription (as all instance subscriptions necessarily do), the Tank[tag 1] subscription is a better match.

Therefore, the Tank Proxy resulting from the instance subscription is out-of-scope from birth, and receives no updates.

The subscribing application instead discovers the Tank via the Tank[tag 1] subscription. The Proxy resulting from *that* subscription receives the Tank updates.



i — The interaction between BestMatch and instance subscriptions can cause, as shown in the examples above, the proxy associated with the instance subscription to go out of scope (i.e., no longer receive updates). This behavior may be undesirable with certain applications, so it is recommended that instance subscriptions are not made in a BestMatch Session, and instead use a separate Session.

Known Deficiencies

BestMatch Does Not Take Into Account Blacklisting

When a subscribing application has a subscription to a given type, then drops a Proxy to an SDO discovered using that type subscription, the Middleware considers that SDO "blacklisted" with respect to that subscription. The result is that updates for that SDO are no longer delivered to the application (since there is no Proxy on which to receive the updates).

As of Release 6.0.3, the BestMatch matching logic does not take into account blacklisted SDOs. So, if a BestMatch Session subscribes to Tank and discovers a bunch of Tanks, then subscribes to Vehicle, then drops the Proxy for one of the Tanks (blacklisting it), the Vehicle subscription will **not** discover the dropped Tank. Effectively, the Middleware still considers the Tank subscription to be the "best match" for the given Tank, even though the given Tank was blacklisted by that subscription.

This behavior may change with a future Middleware release, so application developers should not rely on it.

BestMatch Does Not Create Proxies Where They Didn't Already Exist

Example 1:

One application repeatedly updates a Tank.

Another application, using a BestMatch Session, subscribes to Tank, then subscribes to Vehicle, then unsubscribes from Tank.

Subscribing to Tank produces a Discovery. Subscribing to Vehicle produces nothing. Unsubscribing from Tank *should* cause the Vehicle subscription to discover the published Tank (since there is no longer a Tank subscription). But, as of Release 6.0.3, the Vehicle subscription does **not** discover the Tank as a result of unsubscribing from Tank.

Example 2:

One application repeatedly updates a Tank.

Another application, using a BestMatch Session, subscribes to the Tank instance, then subscribes to Vehicle, then drops the Tank Proxy.

Subscribing to the Tank instance produces a Discovery. Subscribing to Vehicle produces nothing. Dropping the Tank Proxy *should* cause the Vehicle subscription to discover the published Tank (since no other subscription matches the Tank). But, as of Release 6.0.3, the Vehicle subscription does **not** discover the Tank as a result of dropping the Tank Proxy.

This behavior may change with a future Middleware release, so application developers should not rely on it.

API Reference and Code Examples

TENA::Middleware::Execution has this method for creating Sessions:

```
SessionPtr const  
createSession( std::string const & name, bool bestMatch = false );
```

Note that the second parameter is optional, and tells whether the Session should have BestMatch enabled.

TENA::Middleware::Session has the following method related to BestMatch:

```
bool isBestMatchEnabled() const;
```

Java API

TENA.Middleware.Execution has this method for creating a Session:

```
Session createSession( String name, boolean bestMatch);
```

A TENA.Middleware.Session has the following method related to BestMatch:

```
boolean isBestMatchEnabled();
```

Callback Framework

Callback Framework

When an TENA application subscribes to SDO or message types, the notification regarding the receipt of information related to discovered SDOs, received messages, etc. occur asynchronously within the middleware with respect to any application processing that may be happening. The middleware and application need to share a framework that allows the middleware to notify the application to perform processing in response to these subscription events. This underlying callback framework is based on the [command design pattern](#). A similar framework based on the [observer design pattern](#) is built upon the callback framework and is the recommended structure for TENA application subscription event handling (see [observer documentation page](#) for more details), although an understanding of the lower level callback framework is useful for understanding the middleware operations with respect to callback processing.

Description

In a middleware system that requires bi-directional method/function invocations (i.e., a user application invokes methods/functions on the middleware and the middleware invokes methods/functions on the user application) a control mechanism is necessary to coordinate these invocations. The purpose of the control mechanism is to address the programming threading and concurrency issues that may arise in such a middleware system.

The TENA Middleware system attempts to provide a flexible control mechanism via a unified callback evocation interface. When the middleware is required to make invocations that must be handled by the user application, a Callback object is created and placed in a queue. The user application is able to provide a thread (or threads) of control to the middleware for executing these callbacks (waiting in the queue) when it is appropriate for the application. This interface will support different application threading models (single threaded, multi-threaded) and different concurrency models (non-reentrant, reentrant).

In a TENA application, the user code can invoke methods/functions on the middleware and the middleware can invoke methods/functions in the user application "space". Several threading and concurrency issues can arise when crossing the application-middleware boundary. For example, if the middleware asynchronously invokes code associated with the user application, there can be reentrancy problems if the user software was not written to handle such situations.

The TENA Middleware is responsible for properly handling invocations from the user application (independent of threading and concurrency issues), so the user application may invoke operations on the middleware whenever necessary. Invocations in the reverse direction from the TENA Middleware to the user application **that are packaged as Callback objects** (see note below) will not occur unless instructed by the user application itself. This instruction is achieved through the callback evocation services described below.

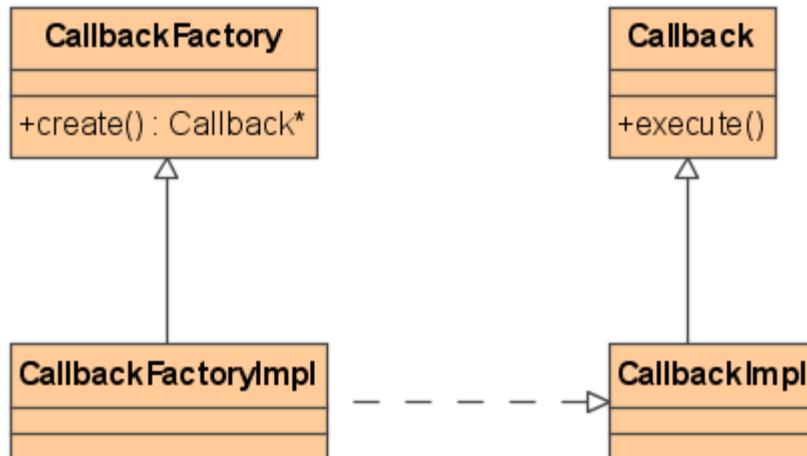
! An important caveat to the invocations from the middleware to the user application code that are **not** packaged as callback objects is SDO remote methods. When an application creates SDOs with remote methods, client applications may invoke those methods asynchronously and the remote method implementation code will be executed by the middleware independent of the user application actions. Therefore, SDO remote method implementation code should be written in a thread-safe manner. Additional information regarding the remote method implementation developer guidance is provided in the [SDO remote method documentation page](#).

There are multiple types of callback objects that are supported by the middleware (as shown in the table below). The callback framework requires a separate class for each callback type for each SDO or Message type.

Types of Middleware Callback Objects

Callback Type	Description
SDO Discovery	Provides notification of an SDO discovery.
SDO State Change	Provides notification of an SDO publication state change.
SDO Destruction	Provides notification of an SDO destruction.
SDO Entered Scope	Provides notification that an SDO entered the Advanced Filtering interest specification.
SDO Left Scope	Provides notification that an SDO left the Advanced Filtering interest specification.
Message Receipt	Provides notification of receiving a message.

All of these callbacks follow a similar pattern that involves a callback factory class with a `create()` method and a callback class with an `execute()` method. The user application registers the appropriate factory and the middleware invokes the `create()` method when required. A callback object is returned and the middleware places the object on a queue until the user is ready to handle the processing of the callbacks. A simplified UML class diagram of the callback mechanism is shown in figure below.



Simplified Diagram of Callback Mechanism

The middleware uses a common queue to hold the `Callback` objects that await processing. Each `Callback` object inherits from a common interface that defines an `execute()` method. Therefore, after the `Callback` object is placed in the queue the middleware does not need to know about the type of callback, but rather just invokes the `execute` method to begin processing that callback when appropriate. This software design is referred to as the "Command" pattern (see [Command Design Pattern](#) for more information).

The user application is responsible for writing the specific behavior of the callback operations when the callback is processed. For example, the user may want to update a graphical display when a new SDO is discovered. The callback class for this behavior will inherit from the middleware abstract base class and define the specific behavior to update the display.

But where have the callbacks gone?

The auto-generated example applications that are based on user object models have been changed with Release 6 of the middleware to present the user with an "Observer" framework, as opposed to the callback framework. The observers are built on top of the callback framework, and are expected to better support user application needs and simplify application development. Observers are very similar to callbacks – they contain user processing code that should be executed in response to a subscription related event. Although the observer mechanism is similar to the callback framework, there are several advantages that are discussed in the [observer documentation page](#).

Users with application code from release 5 of the middleware who are interested in preserving their application callback code can utilize the `SubscriptionInfo` class associated with the SDO or message type of interest. The constructor for the `SubscriptionInfo` class accepts the callback factories that were used in release 5. Since the `SubscriptionInfo` class inherits from `SubscriptionInterface`, the `SubscriptionInfo` class can be used in the `subscribe` methods where a `Subscription` is required.

i — Users with release 5 applications are strongly encouraged to migrate their callback implementation code into the observer mechanism promoted with release 6 of the middleware.

Callback Evocation Services

At the core of the callback services, the TENA Middleware provides a collection of callback evocation methods. These methods allow the user application to inform the middleware that any pending callbacks can be executed, resulting in operations using code that is defined by the user application. There are three variations of the callback evocations services, shown below. These services exist on the `TENAMiddleware::Session` class. The different evocation services are described in the table below.

Callback Evocation Services

Callback Evocation Service Method Signature	Description
<code>size_t evokeCallback() ;</code>	To invoke a single callback from the middleware (if one is pending), the user application can use the <code>evokeCallback()</code> method without any arguments. This variation of the callback evocation service (without any arguments) instructs the middleware to execute a callback if one exists in the queue. The middleware will execute only a single callback at a time and will return immediately if no callbacks exist in the queue. The return value indicates the number of callbacks remaining on the queue.
<code>size_t evokeCallback(unsigned int waitTimeInMicroseconds)</code>	The second variation of <code>evokeCallback</code> will also process a single callback or will wait a maximum duration of time for a callback if one does not exist initially. The middleware will return after the specified time if no callback exists, otherwise the method will return after the completion of the callback. The return value indicates the number of callbacks remaining on the queue.

<pre>size_t evokeMultiple Callbacks (unsigned int waitTimeInMic roseconds)</pre>	<p>The third version of the callback evocation services is the most commonly used method to obtain callbacks. It is the most efficient means to drain the callback queue until the duration is exceeded. Unlike the first two evocation services, this method allows the middleware to execute multiple callbacks. User applications may want to use this variation of the callback evocation service if they operate in a time-based loop in which they can provide any excess time for each loop to the middleware for handling callbacks. The return value indicates the number of callbacks remaining on the queue.</p>
--	---

Release 6.0.5 Update  – Prior to middleware version 6.0.5, the parameter name for the `evokeCallback` and `evokeMultipleCallbacks` methods was named `maxWaitInMicroseconds`, which has been changed to `waitTimeInMicroseconds`.

The first two variations above only process a single callback at a time and may be useful for applications that need fine control over callback operations. It is anticipated (and recommended) that typical applications will use the multiple callback evocation service. The user application should provide to the middleware the amount of time the middleware should wait for callback processing in an attempt to avoid starvation of the callback processing.

In case of callback starvation, the middleware will continue to service the network and any internal events using the middleware processing thread(s), but the pending callback queue will continue to grow. This may cause excessive memory growth and result in a delay with the timely processing of callback information.

An adaptive approach can be used where the application evaluates the callback queue length at the end of an `evokeMultipleCallbacks` invocation and adjusts the amount of processing time provided depending on the size of the queue. Developers are encouraged to build some diagnostic capability into their application that can monitor the callback queue length, perhaps alerting the operator with a warning message if the queue length continues to grow over time.

Note that the time-limited callback methods cannot guarantee they will honor the time exactly but will always return after the specified time has elapsed. This is because execution control can be passed to the application for an indeterminate amount of time and the nature of multi-threaded applications running on a non-real-time operating system.

Multi-threaded applications can choose to dedicate one or more programming threads in the `evokeMultipleCallbacks` call. In this scenario, the application developers need to manage the thread safety of the application specific observer/callback code. In particular, the user defined observer event handling code needs to address thread concurrency issues with other application threads, including reentrancy when there are concurrent threads in the `evokeMultipleCallbacks` call.

 — Note that applications that do not subscribe to SDOs or Messages are not required to perform callback processing.

Usage Considerations



What time interval should be used for the `evoke callback` method?

The default value used for the `evokeMultipleCallbacks` argument in the example applications is 1 second (1,000,000 microseconds). Whether this is a reasonable value for your application depends on the needs for your application.

Does your application have multiple programming threads and if so, do any of these threads get dedicated to calling `evokeMultipleCallbacks`? If so, then the thread(s) can `evokeMultipleCallbacks` for as long as is reasonable to break out and check if the application needs to shut down (1-3 seconds).

If the application only uses a single programming thread, then the amount of time spent in `evokeMultipleCallbacks` depends on the how much of a time window should be spent with processing the callbacks versus the time spent doing other application processing. For example, an application may operate at a one hertz rate — do application processing and then give the remainder of the 1 second time window to `evokeMultipleCallbacks`. There are two considerations with this approach though:

1. The time the middleware spends in `evokeMultipleCallbacks` may be longer than the time provided because of the time spent in user code processing the last callback.
2. Avoid using small time intervals in `evokeMultipleCallbacks` because it may cause context thrashing and hurt performance. For example, typical operating systems are unable to distinguish time values less than 10 milliseconds with respect to task/thread management, so using a smaller value for the `evokeMultipleCallbacks` argument will be ignored.

Object Model Subsetting

Object Model Subsetting

TENA applications are compiled against one or more object models that are used across the particular event. Applications are only required to be compiled against the SDOs and Messages that are used by the application for either publication or subscription, versus all of the object models associated with the event. The TENA Middleware will automatically handle the receipt of object model types that are unknown to the application, even if the unknown type is derived from a known type. This object model subsetting behavior allows object models to evolve independently of applications, as long as the particular definitions used by the application are not changed.

Description

A TENA Object Model is used by an application to formally define the information that is exchanged with other TENA applications. This information is modeled as SDOs (Stateful Distributed Objects) and messages. TENA applications can *publish* and/or *subscribe* to the different types of SDOs and messages defined in the object model(s) that are used to build the application. Additional information concerning object models and the object model compiler, which is used to automatically generate software that binds an application to object model definitions, can be found at the following links:

- [TENA Object Model](#)
- [TENA Object Model Compiler](#)

Previous releases of the TENA Middleware required that all applications participating in a particular distributed event were required to use the same (complete) set of object models. Unfortunately, this creates unnecessary application maintenance to upgrade the application software to newer object model versions whenever there is any object model change, even though the particular application may not be impacted by the object model change. For example, if an application only subscribes to the SDO type "Vehicle", then changes to other object model types should not require the application to be updated.

The primary objective behind the Object Model Subsetting capability is to allow object model designers to develop separate object models that attempt to isolate applications from independent object model changes. In the previous example, the model definition for the Vehicle object can be placed in a separate object model that only contains items related to the Vehicle model. Other (non-Vehicle related) object model types that are used in a particular event can be placed in separate object models, and changing those object models will not impact applications that only use the unchanged Vehicle object model.

An important consideration with the middleware implementation of object model subsetting support that is worth noting is in the situation where the object model change involves the addition of a derived class. For example, consider an object model with type "Vehicle" which is subsequently extended through inheritance to define the type "Taxi". A subscribing application that is built with only the knowledge of type Vehicle needs to discover SDOs created by applications that publish Taxi SDOs, although these SDOs would be *sliced* to only provide the subscribing application the Vehicle portion of the SDO.

As part of object model subsetting, the middleware needs to evaluate the complete inheritance hierarchy for any received communication from a publishing application to determine if there is a subscription match at any level in the inheritance hierarchy, even when the subscribing application was not compiled against one or more of the derived types. This middleware inheritance inspection behavior associated with the object model subsetting capability applies for SDOs, messages, and local classes. Local classes operate polymorphically because they are accessed through a "pointer" mechanism and therefore, a publishing application can supply a derived local class in place of the type defined in an SDO or message definition.

Usage Considerations

There are no application software modifications needed to take advantage of the TENA Middleware object model subsetting capabilities. Object model designers may want to decompose their object models into separate models to provide isolation with respect to future modifications, rather than developing a single monolithic object model. From an application developer perspective, only those object models that contain types that are used by the application should be included in the building of the application.

When a TENA execution involves applications using different object models and different versions of the same object model, the middleware must ensure that every application is using a consistent set of object models. The [Object Model Consistency Checking](#) capability is used to ensure that applications joining a TENA execution do not introduce inconsistent object model types.

Release 6 of the Middleware introduces additional type introspection capabilities which can be helpful in understanding when SDOs are being sliced by OM subsetting. See [Type Registry](#) for more details.

C++ API Reference and Code Examples

Example object models have been designed to illustrate the operation of object model subsetting. The [Example-Vehicle](#) and [Example-Tank](#) object models contain the Vehicle and Tank SDO definitions, respectively, where the Tank type is derived from the Vehicle type. The automatically generated example implementation code can be used to run a TENA execution with a Tank publisher application (that is built against the Example-Tank object model) and a Vehicle subscriber application (that is built against the Example-Vehicle object model). This execution scenario illustrates how an existing subscribing application could be built against an older object model that only knows about the type Vehicle, and it will still operate properly when there is a newer application that uses an extension of the Vehicle type (i.e., Tank) that is not known to the subscribing application.

Observer Mechanism

Observer Mechanism

The TENA Middleware uses the concept of "**Observers**" to allow application developers to package software that is used to process various subscription events. When an application subscribes to a particular SDO or message type, one or more Observers can be attached to the subscription that control the application behavior in response to different events associated with the subscription. For example, type-based SDO subscriptions may produce *Discovery*, *State Change*, and *Destruction* events that correspond to the application discovering a new SDO , receiving a state update for an existing SDO, and receiving a destruction notification for an existing SDO, respectively. The Observer Mechanism enables application developers to effectively package application specific software that is executed when one of these subscription events occur. The structure of the Observer Mechanism enables developers to create behavior related to subscription activities that is common to many applications and share this Observer software among various applications.

Description

When a TENA application subscribes to Stateful Distributed Objects (SDOs, or objects) or Messages within an execution, the middleware requires a mechanism to notify the user application code about events related to the subscription. For example, if the application indicates that it is interested in subscribing to SDOs of type *Vehicle*, then it may be necessary to notify the application whenever a new *Vehicle* is discovered. In this situation the middleware knows when a particular subscription event occurs, but it doesn't know exactly what the application should do in response to the event. The middleware provides a framework in which application developers can write their application specific code to handle these events and register this code with the middleware.

Common software design patterns for handling this situation are the [command \(or callback\) pattern](#) and the [observer pattern](#). The middleware uses both patterns in order to support the needs of the user community. Prior to Release 6 of the middleware, only the callback structure was available to application developers, but based on user experience the current release of the middleware promotes the use of the observer pattern. The observer pattern is actually implemented on top of the callback framework and offers several advantages:

- All of the application event handling code for a particular SDO model type are contained within a single C++ class, whereas the callback framework required multiple callback and callback factory classes for each event type per SDO model type. Combining all of the event handling code in a single class greatly simplifies understanding and code maintenance.
- Multiple, independent observers can be associated with a particular subscription, whereas the callback framework requires all event handling code associated with a particular event type and SDO model type to be combined in the same method. Using independent observers simplifies development and reusability of the event handling code.

The observer code structure is based on an abstract class for each SDO model type that is auto-generated by the SDO Model Compiler. The interface class is named "AbstractObserver" to indicate that application developers should inherit from that class to add their application specific event handling code. The `AbstractObserver` interface for SDO (Stateful Distributed Object) types includes the following methods:

- `discoveryEvent`
- `stateChangeEvent`
- `destructionEvent`
- `enteredScopeEvent`
- `leftScopeEvent`

and the interface for Message types includes only:

- `messageEvent`

Application developers define their own class that inherits from the appropriate `AbstractObserver` class, and implement the event methods that require application specific behavior. Application developers can ignore providing implementation code for particular event types that are not needed. For example, if the observer just keeps track of the SDOs currently in existence, then the `stateChangeEvent`, `enteredScope`, and `leftScope` methods can be ignored in the user defined observer class.

If the event handling code requires access to application data, then that access should be provided through the constructor of the derived observer class. For example, if the observer provided a capability to log values to a database, the constructor for the data logging observer class would require arguments that enabled the event methods to perform database operations.

After an application invokes the `subscribe` method to indicate interest in receiving certain SDO or Message types, a `Subscription` object is returned which represents the particular subscription interest. The application can then instantiate the appropriate observers that are needed for the application and attach them to the `Subscription` object. Observers can be attached and detached from the `Subscription` as necessary.

i — Note that in the case of SDO Subscriptions, when Observers are attached or detached from the `Subscription` object those changes will only affect future SDOs that are discovered. Existing SDOs will continue to utilize the Observers that were attached to the `Subscription` at the time the discovery occurred. If applications need to modify the Observers on a per SDO basis, the SDO Proxy class has a `setSubscription` method that will replace the current Observers attached to the SDO Proxy with the new set of Observers associated with the `Subscription` passed in to the `setSubscription` method.

Although an application has performed the necessary `subscribe` calls and attached the appropriate observers, the actual event methods will not be invoked until the application calls the `evokeMultipleCallbacks` method on the `Session` object. As discussed in the [callback framework documentation page](#), the middleware requires the application to provide a programming thread for processing the subscription events. This is to make the application responsible for thread safety issues.

If the application uses multiple programming threads, then the application developer needs to consider thread safety with respect to the observer event methods. For example, if the application makes multiple simultaneous calls to `evokeMultipleCallbacks` it is important that the observer event methods support reentrancy. Even with only a single thread in `evokeMultipleCallbacks` and other application threads, the developer needs to ensure that there are not code sections in the observer event methods that can be concurrently accessed by other application threads.

Usage Considerations

In addition to the thread safety design considerations mentioned above, the developers of observers should note several other considerations related to order of processing and excessive processing time. Multiple observers attached to a particular subscription are processed in an indeterminate order, so developers should not assume that observers will be executed in a specific order.

Developers should also consider the processing requirements for executing the observer event code. If the observer event code encounters an excessive processing delay, it could cause timing related issues with the processing of other observers or the application in general. If there are multiple programming threads used to call `evokeMultipleCallbacks` simultaneously, then only the observers associated with the particular SDO or Message instance will be blocked waiting for an observer encountering an excessive processing delay. If there is only a single thread active in `evokeMultipleCallbacks`, then all observer processing will be blocked by the observer encountering an excessive processing delay. Generally speaking, if the observer event handling code is not able to perform its operation in a bounded and timely manner (relative to responsiveness needed by the application), then the observer should attempt to offload the processing intensive operations to a mechanism that is run asynchronously from the observer code.

A key benefit of the observer pattern is the ability for developers to design observer code that can be shared with other applications. For example, a developer could generate common data logging code for a particular SDO model in the form of observer classes. These observer classes can be packaged in a software library that is shared with other application developers with simple instructions to construct the appropriate observer and attach to the subscription. This Logging observer code could then provide a consistent logging mechanism that is easily shared by multiple applications.

C++ API Reference and Code Examples

As mentioned above, part of the generated API for subscribing to a type defined in an object model is an `AbstractObserver` class for each SDO and Message type in the object model. Application developers that are implementing an observer write a class that extends the generated `AbstractObserver` class.

Example observers are automatically generated as part of the example source code included with OM C++ distributions. Two different observer classes are generated, a `CountingObserver` (keeps counts different SDO events such as discovery) and a `PrintingObserver` (prints the attribute values when updated).

The important sections of the `CountingObserver` class declaration and definition for the `Example::Vehicle` type is shown in the code below.

```

// file MyApplication/Example_Vehicle/CountingObserver.h
...
namespace MyApplication
{
    namespace Example_Vehicle
    {
        class CountingObserver
            : public Example::Vehicle::AbstractObserver
        {
        public:
            CountingObserver();

            TENA::uint32
            getDiscoveredCount() const;

            virtual
            void
            discoveryEvent(
                Example::Vehicle::ProxyPtr const & pProxy,
                Example::Vehicle::PublicationStatePtr const & pPubState );
            ...
        private:
            //! An MT-safe counter to count invocations of discoveryEvent()
            std::atomic< ACE_SYNCH_MUTEX, TENA::uint32 > discoveredCount_;
            ...
        };
    } // End of namespace Example_Vehicle
} // End of namespace MyApplication

// file MyApplication/Example_Vehicle/CountingObserver.cpp
...
MyApplication::Example_Vehicle::
CountingObserver::
CountingObserver()
: discoveredCount_( 0 ),
  changedCount_( 0 ),
  destructedCount_( 0 )
{
}

TENA::uint32
MyApplication::Example_Vehicle::
CountingObserver::
getDiscoveredCount() const
{
    return discoveredCount_.value();
}
...
void
MyApplication::Example_Vehicle::
CountingObserver::
discoveryEvent( Example::Vehicle::ProxyPtr const &,
    Example::Vehicle::PublicationStatePtr const & )
{
    ++discoveredCount_;
}
...

```

Release 6.0.5 Update! – Prior to middleware version 6.0.5, the type `TENA::Middleware::Utils::AtomicOp` was used instead of the C++11 standard `std::atomic` type.

In the code example above, the application developer designs the constructor of the derived observer class. If the observer implementation needs access to application information, the constructor would have those arguments. In this example, no application information is needed and the constructor merely initializes the counts.

Accessor methods (e.g., `getDiscoveredCount`) are provided to allow the application to obtain the count values. Developers can add whatever methods are appropriate for their observer. Note that the count variables used by the observer are multi-thread safe counters. This implementation enables an application to use multiple programming threads to call `evokeMultipleCallbacks` concurrently.

Note that the inherited `Event` methods, such as the `discoveryEvent` method shown, provide arguments for the proxy (specifically the `ProxyPtr`) and the publication state of the proxy at the time the event was generated. Note that since callbacks are queued before they are processed it is possible that the copy of the state provided to the `Event` method may not be the current state of the proxy. This is important for applications such as data loggers that need to operate on the state values corresponding to the time of the event, not the latest values corresponding to the proxy when the event was processed.

Java API Reference and Code Examples

As mentioned above, part of the generated API for subscribing to a type defined in an object model is an `AbstractObserver` class for each SDO and Message type in the object model. Application developers that are implementing an observer write a class that extends the generated `AbstractObserver` class.

Example observers are automatically generated as part of the example source code included with OM Java distributions. Two different observer classes are generated, a `CountingObserver` (keeps counts different SDO events such as discovery) and a `PrintingObserver` (prints the attribute values when updated).

The example code for the `CountingObserver` class for the `Example.Vehicle` type is shown below. Note that this simple example does not particular handle issues of multithreaded access, since the methods are not `synchronized`, and the `++` operator is not atomic. If multiple threads are used to evoke callbacks, synchronization or use of pther techniques such as classes from `java.util.concurrent.atomic` would be necessary.

```
package org.Example.Vehicle_Subscriber;

public class ExampleVehicleCountingObserver extends Example.Vehicle.AbstractObserver {
    private int discoveredCount_ = 0;

    public int getDiscoveredCount() { return discoveredCount_; }

    public void discoveryEvent(
        Example.Vehicle.Proxy proxy, Example.Vehicle.PublicationState pubState ) {
        ++discoveredCount_;
        // release proxy is recommended
        proxy.releaseReference();
    }

    public void stateChangeEvent(
        Example.Vehicle.Proxy proxy, Example.Vehicle.PublicationState pubState ) {
        ++changedCount_;
        // release proxy is recommended
        proxy.releaseReference();
    }

    public void destructionEvent(
        Example.Vehicle.Proxy proxy, Example.Vehicle.PublicationState pubState ) {
        ++destructedCount_;
        // release proxy is recommended
        proxy.releaseReference();
    }
}
```

★ Releasing proxy references is recommended in Observer methods, see [Java-264](#).

Reusable Observers

Reusable Observers

Observers are used to provide specific implementation behavior related to subscription events (e.g., SDO discovery, SDO update, Message receipt). This documentation page has been created to collect and document Observer implementations that can be reused across the user community. Users interested in submitting Observer implementations can contact the [helpdesk](#)

Observer Implementations

- [UniqueIDmapper](#) – Observer implementation that maintains a map of discovered SDO Proxies using a UniqueID as the map key.

UniqueIDMapper

UniqueIDMapper

The UniqueIDMapper is a simple [Observer](#) that can be used for SDO subscriptions to maintain a collection of discovered SDO proxies that can be efficiently accessed by the SDO's corresponding UniqueID value. The Observer implementation uses a standard C++ `std::map` with the UniqueID value as the key. Applications using the UniqueIDMapper are required to define a function that returns the appropriate UniqueID from a corresponding PublicationState.

 The TENA SDA project is interested in feedback regarding the use of this class. Please submit a [helpdesk case](#) with any suggested improvements.

Description

When an application subscribes to [SDOs](#) that are associated with a `TENA::UniqueID`, it is often useful to maintain a `std::map` (standard C++ collection class) that holds `ProxyPtrs` for discovered SDOs using the UniqueID as the key to efficiently access the specific `ProxyPtr`. The UniqueIDMapper class provides this capability for arbitrary SDO types.

Users can attach a UniqueIDMapper Observer to an SDO Subscription and then use methods like `find()` and `erase()` on the UniqueIDMapper to access or remove particular proxies within the container.

The automatically generated [Subscription](#) class already maintains a discovered SDO Proxy list (accessible through the `getDiscoveredSDOList` method), but that `std::map` is keyed on the SDO::ID value, and not the UniqueID value. The existing [Subscription](#) list is still available, but the access performance will not be as efficient as using a `std::map` that is keyed on the UniqueID value.

UniqueIDMapper is a parameterized class and will work for any particular SDO type. The key elements of this class declaration are shown below.

```

template<typename T>
class UniqueIDmapper
    : public T::AbstractObserver
{
public:
    typedef boost::function<
        TENA::ImmutableUniqueIDPtr (typename T::PublicationStatePtr const &) >
    UniqueIDaccessor;

    typedef std::map<
        TENA::ImmutableUniqueIDPtr,
        typename T::ProxyPtr,
        UniqueIDless >
    Map_t;

    typedef typename TENA::Middleware::Utils::SmartPtr< UniqueIDmapper<T> > Ptr_t;

    UniqueIDmapper( UniqueIDaccessor accessor );

    void
    discoveryEvent(
        typename T::ProxyPtr const & pProxy,
        typename T::PublicationStatePtr const & pPubState );

    void
    destructionEvent(
        typename T::ProxyPtr const & pProxy,
        typename T::PublicationStatePtr const & pPubState );

    typename T::ProxyPtr
    find( TENA::ImmutableUniqueIDPtr const & id )
        const;

    size_t
    erase( TENA::ImmutableUniqueIDPtr const & id );

    Map_t
    getMap()
        const;
};


```

The operation of this Observer class is such that an application needs to define an "accessor" function that is used to obtain the particular UniqueID value from a PublicationStatePtr. For example, if the SDO type is TENA::AMO, then the UniqueID accessor function can be defined as shown in the code below, where the UniqueID value of the "AMOid" attribute is used as the std::map key for the discovered TENA::AMO proxies.

```

static
TENA::ImmutableUniqueIDPtr
getAMOid( TENA::AMO::PublicationStatePtr const & pState )
{
    return pState->get_AMOid();
}

```

The constructor for the UniqueIDmapper requires a function pointer, UniqueIDaccessor, as shown in the code fragment below. A typedef is used to simplify the coding.

```

typedef UniqueIDmapper<TENA::AMO_SDO> AMOmapper;

// construct the UniqueIDmapper<TENA::AMO>, passing in the
// getAMOid function so that the mapper knows how to get the
// UniqueID you want to use as the key to the map
AMOmapper::Ptr_t pAMOmapper( new AMOmapper(getAMOid) );

```

The UniqueIDMapper for the SDO type TENA::AMO, i.e., AMOmapper, can be used as an Observer that is attached to a TENA::AMO::subscription, as shown in the code below. A modified TENA-AMO-v2-AllPublishSubscribe-v2.cpp (main for auto-generated "allPublishSubscribe" ExampleApplication for type TENA::AMO) application is used to illustrate the operation of the UniqueIDMapper class. In this example, the AMOmapper is attached to a TENA::AMO::Subscription so that the AMOmapper object will maintain a std::map of discovered TENA::AMO SDO proxies using their TENA::AMO AMOid attribute value (which is a TENA::UniqueID). Once the subscribe invocation occurs, the subscribing application will begin (after some delay) to receive DiscoveryEvents for any TENA::AMO SDOs that have been published in the TENA Execution. The AMOmapper implementation will automatically insert the discovered ProxyPtrs in a std::map and delete the appropriate ProxyPtr upon a DestructionEvent.

The user application using the AMOmapper is only required to **evoke callbacks** to cause the Observer events to occur. The application is free to invoke UniqueIDMapper methods whenever necessary, regardless of the state of the callback processing. The UniqueIDMapper is designed to be thread-safe, meaning that multiple callback and application programming threads can use the AMOmapper as all of the methods are protected with common mutex.

The code fragment below illustrates how the UniqueIDMapper is attached (as an Observer) to a TENA::AMO::Subscription.

```
bool selfReflection(
    appConfig["enableSelfReflection"].getValue< bool >() );

bool wantPruning(
    appConfig["pruneExpiredStateChange"].getValue< bool >() );

TENA::AMO::SubscriptionPtr pTENAAMOSubscription(
    new TENA::AMO::Subscription( wantPruning ) );

pTENAAMOSubscription->addObserver( pAMOmapper );

// Declare the application's interest in AMO objects.
TENA::AMO::subscribe(
    pSession,
    pTENAAMOSubscription,
    selfReflection );
```

Release 6.0.2 Update — An alternative subscribe method was added with release 6.0.2 of the middleware in which the SessionPtr argument is used, versus the Session reference (i.e., "*pSession") that was used in the previous releases. Users are encouraged to use the SessionPtr interface. See [M W-4286](#) for more details.

In the modified TENA-AMO-v2-AllPublishSubscribe-v2.cpp example application, an Observer is created that uses the AMOmapper that was attached to the TENA::AMO Subscription. This Observer is for a different object model type, TENA::Alert (which is also defined in the TENA::AMO object model. The Alert Message Observer obtains the AMOmapper through its constructor and then consults the AMOmapper when an Alert Message is received to attempt to find a TENA::AMO SDO proxy with the same UniqueID value as found as an attribute in the TENA::Alert Message.

```

// simplistic alert observer as an example of how the UniqueIDmapper
// might be used
class MyAlertObserver
    : public TENA::Alert::AbstractObserver
{
public:
    MyAlertObserver( AMOmapper::Ptr_t pAMOmapper )
        : pAMOmapper_( pAMOmapper )
    {
        // better not have passed in a null AMO UniqueIDmapper
        assert( pAMOmapper.isValid() );
    }

    void
    messageEvent( TENA::Alert::ReceivedMessagePtr const & pReceivedMessage )
    {
        std::cout << "Received Message" << std::endl;

        // look to see if we've discovered the AMO in the source
        TENA::AMO::ProxyPtr pSource(
            pAMOmapper_->find( pReceivedMessage->get_sourceAMOid() ) );
        if ( pSource.isValid() ) {
            std::cout << "found source AMO proxy: "
                << pSource->getObjectID()
                << std::endl;
        } else {
            std::cout << "source AMO "
                << pReceivedMessage->get_sourceAMOid()
                << " not found" << std::endl;
        }
    }

    // look to see if we've discovered the AMO in the destination
    TENA::AMO::ProxyPtr pDestination(
        pAMOmapper_->find( pReceivedMessage->get_destinationAMOid() ) );
    if ( pDestination.isValid() ) {
        std::cout << "found destination AMO proxy: "
            << pDestination->getObjectID()
            << std::endl;
    } else {
        std::cout << "destination AMO "
            << pReceivedMessage->get_destinationAMOid()
            << " not found" << std::endl;
    }
}

private:
    AMOmapper::Ptr_t pAMOmapper_;
};


```

The complete source code for the `UniqueIDMapper` class and the modified `TENA-AMO-v2-AllPublishSubscribe-v2.cpp` file can be found in the API Reference and Code Example section below.

Usage Considerations

Non-Unique UniqueIDs — When using the `UniqueIDMapper` observer, it is possible that publishing applications will not create `UniqueID` values that are unique. When attempting to insert a `ProxyPtr` into the map using a `UniqueID` value that already exists, the `UniqueID` class will throw a runtime exception, specifically a `std::runtime_error`, if there exists a `ProxyPtr` with the same `UniqueID` value, but a different `SDO::ID`. This indicates that `UniqueID` values with the execution are not unique. Since this runtime exception will occur in conjunction with a user defined callback, the middleware will catch the exception, report an `Alert`, and then proceed with the callback processing. In this situation, no Observer events will be lost, but the `UniqueIDmap` per will not hold a `ProxyPtr` for that SDO whose `UniqueID` value is not unique.

Held ProxyPtrs — Since the `UniqueIDMapper` class will hold a `ProxyPtr` for each discovered object, the application will continue to receive updates, scope changes (if [Advanced Filtering](#) is employed), and destructions for these proxies. Additionally, the `Subscription` object will also hold a `ProxyPtr` for each discovered object. If the application is no longer interested in a particular `Proxy`, then the corresponding `ProxyPtr` in both `UniqueIDMapper` and `Subscription` need to be deleted. This is done using the `UniqueIDMapper::erase` and `Subscription::removeSDO` methods.

C++ API Reference and Code Examples

UniqueIDMapper Source Code

- **UniqueIDMapper.h** — Contains implementation code for the UniqueIDMapper class.
- **TENA-AMO-v2-AllPublishSubscribe-v2.cpp** — Contains implementation code for the modified Example Application that is used to demonstrate the use of the UniqueIDMapper class. This file must be used in conjunction with a TENA Middleware installation that includes the default TENA-AMO-v2-AllPublishSubscribe-v2 application installed in the `src/TENA-AMO-v2/TENA-AMO-v2-AllPublishSubscribe-v2` location (at least for the 6.0.1 release).

Subscription Slicing

Subscription Slicing

The TENA Meta-Model (which defines the constructs and rules for TENA object models) supports inheritance for Stateful Distributed Objects (SDOs) and Messages. Subscribing applications can subscribe to SDOs or Messages that have been subsequently derived for more specific SDOs and Messages for other applications. When a subscribing application receives a derived SDO or Message where the application only knows of a less derived type, the middleware is required to remove the attributes and methods that are in the more derived class that the subscribing application does not know. This operation is referred to as SDO or message slicing.

Description

TENA applications are only required to be compiled and linked against the object models that they use for publication and subscription activities. This permits TENA executions to use a collection of various object models that have been developed and evolved independently. The only issue that matters is that applications that are attempting to share object model information and services are required to have a consistent definition for the items that they are sharing.

In the case of inheritance of SDOs or messages, it is possible that one application was written without knowledge of a derived class that extended that original SDO with additional attributes and/or methods. The middleware needs to ensure that the application written with the original less-derived SDO type will still operate properly with another application using the more-derived SDO type. Of course, the original application is unable to provide or process any attributes or methods added to the derived class, but the two applications should still be able to collaborate with the less-derived type.

Lets consider an object model example. The code fragment below shows the object model definition, in the TENA Definition Language (TDL) format, for SDO types `Vehicle` and `Tank`. The `Tank` class has added several methods and attributes to the `Vehicle` definition.

```
class Vehicle
{
    string name;
    Team team;
    Location location;
};

...

class Tank : extends Vehicle
{
    void loadSoldier( in Soldier * pSoldier ) raises ( CannotLoadSoldier );
    void unloadSoldier( in Soldier * pSoldier );
    void driveTo( in vector < Location > wayPoints );
    void fireGun( in Location targetLocation );

    optional float32 damageInPercent;
    vector < Soldier * > passengers;
};
```

An application that was written to subscribe to the type `Vehicle` will not know about the methods and attributes added to the derived type `Tank`. If there is a `Tank` publisher, the middleware associated with the `Vehicle` subscriber will know how to take the `Tank` information and present it to the `Vehicle` subscriber as if the `Tank` was really a `Vehicle`. In other words, the `Vehicle` subscriber would be provided an SDO Proxy that corresponds to the `Tank` SDO Servant created by the publisher, but the proxy will be a `Vehicle` proxy and the subscriber would not be able to read the `damageInPercent` or `passengers` attributes, as well as invoke the `loadSoldier`, `unloadSoldier`, `driveTo`, and `fireGun` methods.

The process in which the middleware receives information related to an SDO or message and removes unknown derived methods and attributes is referred to as *subscription slicing*. This mechanism allows object models to evolve through inheritance to provide additional methods and/or attributes to provide new capabilities or test new techniques without breaking existing applications that already work with the base SDO types.

Object Models

Object Models

TENA Object Models are used to define the information and services that are shared between collaborating applications participating in a TENA execution. The TENA Meta-Model defines the permissible constructs that are supported by the Object Model Compiler and the TENA Middleware. An object model forms a *binding contract* between the applications attempting to interoperate effectively. The middleware and object model attempt to enforce this contract and therefore avoid run-time problems that hinder interoperability between the distributed systems.

TENA Object Models define, in a formal manner from a computer software perspective, the data structures and services that can be shared among collaborating applications that publish and subscribe to the object model elements. Users can design and develop object models for their particular needs using the constructs supported by the TENA Meta-Model (e.g., objects, messages, inheritance, enumerations). The formal definition of the data elements and services enable the use of code generation technology to create software interfaces that ensure that applications use the object model definitions correctly and thereby improve interoperability and reduce integration costs.

Data structures within an object model can be a simple collection of fundamental data types (e.g., integers, floating point values, strings) that are sent once to interested subscribing applications in the form of TENA Messages, or the data structures can be used to represent an item whose complex attribute values change over time and the item provides both remote and local services that can be used by a subscribing application.

Fundamentally, TENA object models are intended to properly capture the syntactic (and some semantic) definition of the information and services that are shared by applications that need to collaborate in a real-time manner. Using formal definitions and code generation technology promotes properly engineered solutions to the collaboration requirements related to testing and training events.

Through the use of the composition construct (supported by the meta-model `import` mechanism), object models can be built from smaller, reusable building block object models. Continued development and maturation of reusable object models offers significant interoperability and development productivity improvements across the TENA user community.

The TENA project continually looks to cultivate reusable building block objects based on real-world requirements and experience from the user community. Object models that are expected to be highly reusable are reviewed through the TENA Architecture Management Team (AMT) Meetings and defined within the TENA package prefix.

Examples of several reusable object models defined within the TENA namespace and maintained by the TENA project are listed below.

Model Name	Brief Description
TENA::Time	Provides common implementation code using a standard interface to convert between various time representations (e.g., Unix, GPS, UTC).
TENA::TSPI	Provides a reusable structure for packaging time, space, position information with appropriate conversion software.
TENA::Platform	Many test and training applications need to model an individual platform acting in a range environment. A platform, also called an entity, is defined as a vehicle (aircraft, ship, ground vehicle, etc.), installation (building, radar installation, sensor system), or human being acting as a single indivisible item on a range.

In addition to syntactical data information associated with the data and services provided by these object models, object models can support common reusable implementation code. For example, the TENA::TSPI (Time Space Position Information) object model provides an implementation that encapsulates the SEDRIS Spatial Reference Frame (SRF) software to perform various coordinate conversions. Through the packaging of common implementation code, the user community benefits from the ability to rapidly develop software applications using real world tested object model components that will provide interoperability.

- [Meta-Model](#) — The TENA Meta-Model defines the constructs used to define the information and services used by collaborating TENA applications.
- [TENA Definition Language](#) — The TENA Definition Language is an unambiguous text representation of the object model definitions using TENA Meta-Model constructs.
- [Object Model Compiler Overview](#) — TENA object models are processed by the Object Model Compiler to automatically generate the necessary software which connects an application to the middleware.
- [Object Model Definition](#) — The TENA Object Model Definition represents the software required to be used by applications to ensure type safety and adherence to the object model definition.
- [Object Model Implementation](#) — A TENA Object Model Implementation is the software used to provide implementation behavior associated with Local Class and Message constructors and operations.
- [Example Object Models](#) — The TENA project uses several Example object models to illustrate the various constructs associated with the TENA Meta-Model and Middleware.
- [TENA Standard Object Models](#) — TENA Standard Object Models provide common definitions and capabilities for basic elements found in the test and training range environment.
- [TENA Candidate OM Process](#) — Describes the process used to create or update the TENA Standard OMs (a.k.a Candidate OM Process)

Meta-Model

Meta-Model

The TENA Meta-Model defines the constructs used to define the information and services used by collaborating TENA applications.

The TENA Meta-Model is a formal definition of the modeling constructs that can be used to define the information and services that can be shared among TENA applications. TENA applications use Object Models that are based on the rules of the TENA Meta-Model to define objects and messages, as well as remote and local services. Automatic code generation technology is used to create software that ensures each TENA application adheres to the object model definitions to promote interoperability and reduce integration costs for test and training events.

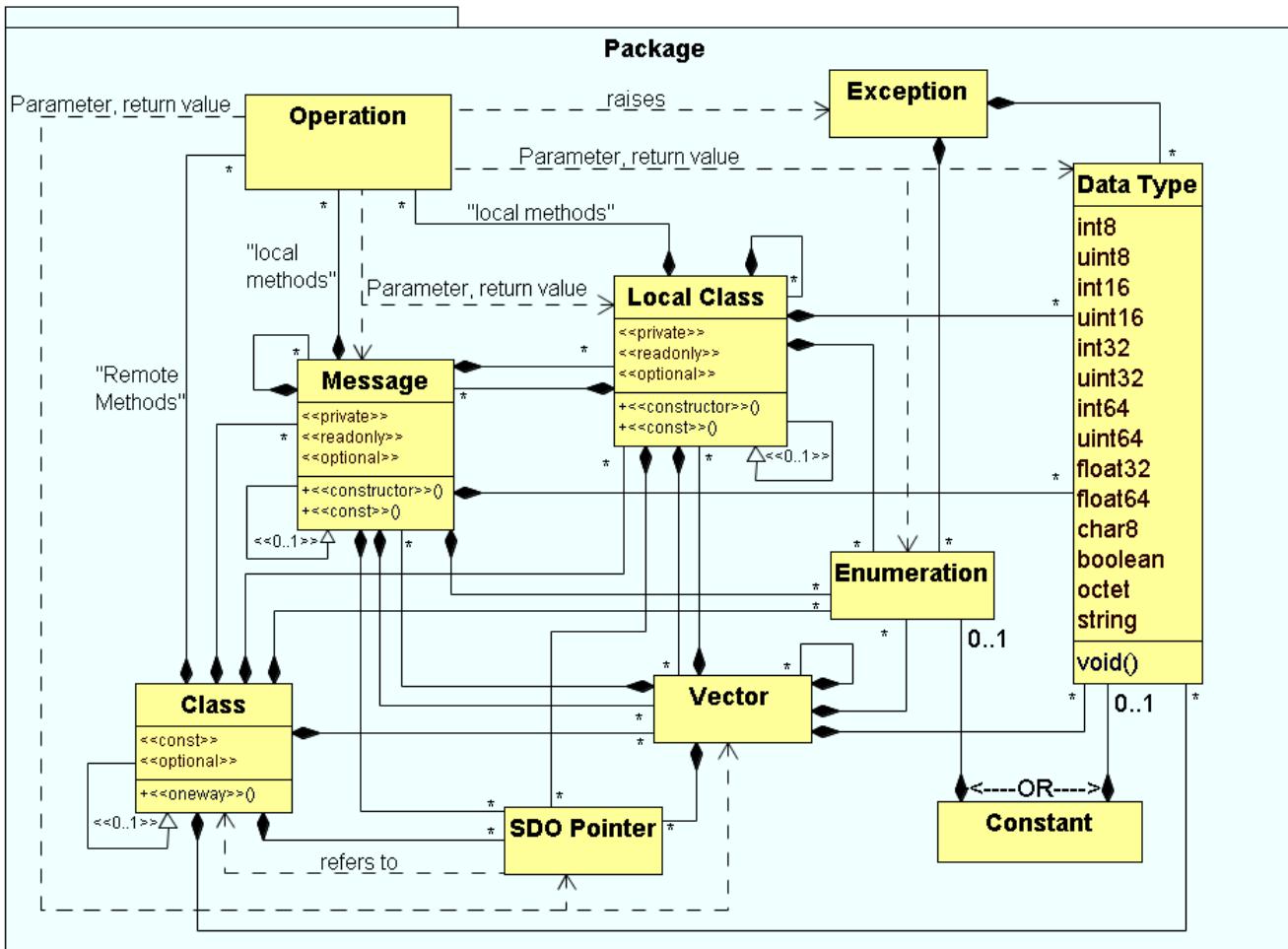
Description

An understanding of the TENA Meta-Model is important for developers of TENA applications since the meta-model describes the basic capabilities that are supported by the TENA Middleware. The meta-model defines the rules imposed on the specification of models (i.e., TENA Object Models) that are used by TENA Applications to rigorously specify the information and services that are shared among the applications.

A TENA object model consists of those object/data definitions, derived from whatever source, that are used in a given execution to meet the immediate needs and requirements of a specific user for a specific range event. The object model is shared by all TENA resource applications in an execution. It may contain elements of the standard TENA Object Model, but it is not required to do so. Each execution is semantically bound together by its object model. Defining an object model for a particular execution is therefore the most important task to be performed to integrate the separate range resource applications into a single event.

In order to support the formal definition of TENA object models, a standard meta-model has been developed to specify the modeling constructs that are supported by TENA. This meta-model is specified by the XMI (XML Metadata Interchange) standard and can be represented by UML (Universal Markup Language). Standards for representing meta-models are being developed under the OMG (Object Management Group) Model Driven Architecture activities.

A graphical depiction of the TENA Meta-Model is shown in the figure below. This picture is a compact, yet powerful, representation of the capabilities of the TENA Meta-Model.



TENA Meta-Model

The TENA Object Model Compiler is based on the formal representation of this meta-model. User submitted object models are verified against the meta-model.

The table below describes the primary constructs supported by the TENA Meta-Model. These constructs enable developers to create sophisticated object models that are used to represent the information and services associated with their range applications. Proper representation is required to enable interoperability between these applications that share the information or utilize the provided services.

- **Fundamental Meta-Model Types** — The TENA Meta-Model and application programming interface uses fundamental type aliases to deal with cross platform portability issues.
- **Class Meta-Model Type** — The class construct in the meta-model is used to define Stateful Distributed Objects (SDOs) that publish state updates and support remote methods.
- **Local Class Meta-Model Type** — Local Classes are complex data structures, defined by attributes and local operations, that can be used as SDO and Message attributes.
- **Message Meta-Model Type** — Messages are complex data types with attributes and operations used to disseminate information about the occurrence of a particular event.
- **Attribute Meta-Model Construct** — Attributes are used to characterize the state of the particular Class, Local Class, or Message.
- **Operation Meta-Model Construct** — The TENA Meta-Model allows object model designers to define operations (aka methods) for Classes, Local Classes, and Messages that provide remote and local services to users.
- **Inheritance Meta-Model Support** — The TENA Meta-Model allows object model developers to use inheritance to extend/evolve existing constructs and/or support polymorphic behavior.
- **Class Pointer Meta-Model Construct** — A class (i.e., Stateful Distributed Object, SDO) pointer represents a distributed "pointer" to an SDO instance that can be shared with other applications.
- **Remote Method Exception Meta-Model Construct** — Remote Method Exceptions allow operations to throw an exception representing an unusual circumstance that could be caught and handled appropriately by the "invoker".
- **Enumeration Meta-Model Construct** — An enumeration represents a user-defined type that can take one of several pre-defined values.
- **Vector Meta-Model Construct** — A vector is a container that holds an arbitrary number of elements of a particular type.
- **Package Meta-Model Construct** — The TENA Meta-Model package construct provides a "namespace" to separate object model elements that may have similar names from different object models.
- **Import Meta-Model Construct** — The import statement allows the definitions of an object model to be used in the context of another object model.
- **Package Scoped Constant Meta-Model Construct** — Object models can define constants that can be used by applications when consistent constant values are needed.

It is important to recognize the difference between the TENA Meta-Model and a particular TENA object model. The object model captures the formal definition of the particular object/data elements that are shared between TENA applications participating in a particular execution. The object model is constrained by the features supported by the meta-model.

Fundamental Meta-Model Types

Fundamental Meta-Model Types

TDL includes built-in data types for boolean variables, integers, characters, and strings. There are corresponding `typedefs` in the TENA C++ namespace that are used by (and for use with) the Middleware runtime.

Description

The following table lists the fundamental built-in types in TDL along with their mappings in C++:

TENA Fundamental Types

TDL Type Name	TENA C++ Type	TENA Java Type	Description
int8 	TENA::int8	byte	An 8-bit integer.
uint8 	TENA::uint8	TENA.UnsignedByte	An 8-bit unsigned integer.
int16	TENA::int16	short	A 16-bit integer.
uint16 	TENA::uint16	TENA.UnsignedShort	A 16-bit unsigned integer.
int32	TENA::int32	int	A 32-bit integer.
uint32 	TENA::uint32	TENA.UnsignedInt	A 32-bit unsigned integer.
int64	TENA::int64	long	A 64-bit integer number.
uint64 	TENA::uint64	TENA.UnsignedLong	A 64-bit unsigned integer.
float32	TENA::float32	float	A 32-bit floating point number.
float64	TENA::float64	double	A 64-bit floating point number.
char8	char	char 	An 8-bit character.
octet	TENA::octet	byte	An octet (8 bits).
boolean	bool	boolean	A boolean variable.
string	std::string	java.lang.String	A string.

 – Java characters are stored as 16 bit values. Because the underlying middleware uses c-style 8 bit *chars*, java string representations will only be valid when using the ASCII character set.

 – Use of unsigned types is being considered for deprecation, and their use is generally discouraged. A more comprehensive concept allowing range checking is being considered for a future version of the middleware, and unsigned types may impose significant inefficiencies when using other languages such as Java because of the lack of support in the language.

 – "int8" is set to type "char" in the c++ implementation. This can cause unexpected behavior such as printing of strange ascii characters instead of a number when a print function is passed a variable of type int8

Usage Considerations

When writing application code, developers are encouraged to use the `typedefs` provided for numeric types in C++. These `typedefs` denote the size of the instance; and this can avoid both superficial ambiguity in the code as well as problems arising from differences in the size of fundamental C++ types on different platforms. For example, `long` is 32 bits on some platforms and 64 bits on others.

Class Meta-Model Type

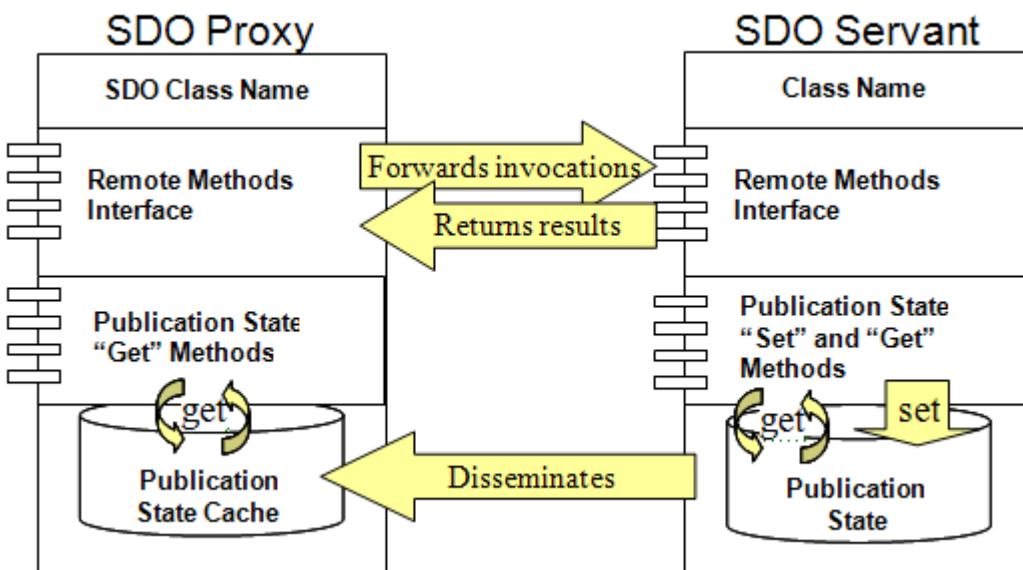
Class Meta-Model Type

The "class" construct is used to define the attributes that form the state of an SDO and to define operations (methods) that can be invoked remotely on a servant via a proxy. SDO classes may inherit from a single other SDO class using the "extends" keyword. In this case the derived SDO class has all of the methods and all of the publication state attributes of its superclass as well as any additional methods and attributes that it may define.

Description

Stateful Distributed Object (SDOs) are defined using the "class" keyword of the TENA Meta-model. An SDO is analogous to a software "class," in that it has both methods and state information. An SDO has a non-zero lifetime during the logical range execution. It is created by a TENA application, it exists for a time, responds to invocations of its methods and updates of its state information, then it may be destroyed.

An application that instantiates a given SDO is called that SDO's owner. Only a single application may own a given SDO at any one time. An application that creates and modifies an SDO is called the "server" of that SDO, while the served SDO is called a "servant." An application that wishes to read or use an SDO that is created by another application is called a "client" of that SDO. Clients do not interact directly with servants; they interact through intermediary objects called "proxies" that reside in the client's process space and represent the servant SDO. The natures of SDO servants and proxies are illustrated in the Figure below.



Structure and Function of SDO Servant and Proxy

Each SDO class contains a "publication state," a list of attributes that the SDO publishes to its proxies held by any client applications. When a client accesses the proxy's publication state it is fetched immediately from the local cache inside the proxy. A client application can only read a proxy's publication state – it cannot write it. Only the owner application, with the servant object in its process space, can update the servant's publication state.

An SDO may contain many other different types of information, including fundamental data types, local classes, and others. Although an SDO is not able to contain another SDO, object model designers are able to have an SDO contain a local class or an SDO Pointer.

SDOs have the following properties:

- An SDO may inherit from at most one other SDO.
- SDOs may be referred to by SDO Pointers, including pointers to the SDOs own type (may need to "forward declare" an SDO type if used before it is defined).
- SDOs may contain:
 - fundamental types,
 - SDO Pointers,
 - vectors, but SDOs may not be contained in vectors,
 - enumerations,
 - local classes,
 - messages,
 - operations.

Additional information on the use of SDOs can be found in the [SDO documentation page](#).

TDL Examples

A simple example of an SDO class defined in the `Example-Vehicle.td1` file is shown below. The `Vehicle` class has three attributes (`name`, `team`, and `location`) and no operations.

```
class Vehicle
{
    string name;
    Team team;
    Location location;
};
```

A derived class, `Tank`, is shown below:

```
class Tank : extends Vehicle
{
    void loadSoldier( in Soldier * pSoldier ) raises ( CannotLoadSoldier );
    void unloadSoldier( in Soldier * pSoldier );
    void driveTo( in vector < Location > wayPoints );
    void fireGun( in Location targetLocation );

    optional float32 damageInPercent;
    vector < Soldier * > passengers;
};
```

In this example, the `Tank` class inherits from the `Vehicle` class. `Tank` has all the publication state attributes that `Vehicle` does plus several additional attributes. The `Tank` class also adds several operations. An SDO class may only inherit from one other class – multiple inheritance is disallowed. Note that the syntax of inheritance is different from C++, IDL, and Java, in that it contains both a colon (like C++ and IDL) and the keyword "extends" (as in Java).

A **forward declaration** of an SDO type can be used when there is a circular dependency between two or more SDO types. For example, if SDO type A contains a pointer to SDO type B, and type B contains a method that uses type A as an argument, there is a circular dependency in which a forward declaration is required. A TDL illustration of a forward declaration is shown below.

```
// Forward declare type B
class B;

class A
{
    B * ptrB;
};

class B
{
...
};
```

Usage Considerations

All SDO classes are *concrete* classes, as opposed to abstract classes that are common in programming languages. Object model designers need to ensure that SDO classes are meant to be instantiated and should be implementable by all publishing applications. SDO implementation refers to the user defined code needed for all of the operations (remote methods) defined by the class.

Use inheritance when it is necessary to create a concrete object that is almost exactly like one already in use, but needs to add operations or attributes to the publication state. Be sure to obey the rules of *substitutability* when creating a derived class – it should be able to be used and operate properly in any situation in which a base class object is expected. The derived class object must be *substitutable* for the base class.

Local Class Meta-Model Type

Local Class Meta-Model Type

A Local Class is a complex data type containing both state information and local operations. The Local Class, as its name implies, exists locally, typically disseminated as an attribute of a SDO (Stateful Distributed Object) or Message. The Local Class construct is useful to group certain state information atomically, and provide local operations that can use or manage that state information in providing a service to the user (similar to a C++ or Java object).

Description

The TENA Meta-Model supports the definition of a construct named Local Class that contains attributes and supports operations. A Local Class is similar to the Class construct, although there is no notion of a "servant" and "proxy" – the Local Class is always local to the current process (i.e., operations /methods are not remote), although a copy of the Local Class can be disseminated to other applications through SDO updates, SDO methods, or Messages.

A Local Class is a useful modeling construct when it is necessary to group attributes together that need to be used atomically. The Local Class may have constructors and methods that are used to provide specific behavior associated with the attributes.

Local Class Properties

Local Classes have the following properties:

- Local Classes may be contained in:
 - SDOs,
 - Messages,
 - Vectors.
- Local Classes may contain:
 - Fundamental Types,
 - Enumerations,
 - Vectors,
 - SDO Pointers,
 - Messages, and
 - other Local Classes.
- Local Classes may contain constructors and methods that require user-defined implementations.
- Local Classes may inherit from at most one other Local Class type.
- Local Classes may be used as method parameters and/or return values.

Additional information on the use of Local Classes can be found in the [Local Class documentation page](#).

Usage Considerations

Common use cases for local Classes include:

Units of Measure — In situations in which applications may utilize different units of measurement (e.g., Fahrenheit versus Celsius), a Local Class could encapsulate the internal representation and require users to use local set/get methods that indicate the appropriate units (e.g., `getTemperatureInCelsius`).

Alternative Representations — Similar to units of measure, there are often alternative representations of particular information that can be encapsulated within a single Local Class or a collection of Local Classes. The TENA::TSPPI object model uses a collection of similar Local Classes for different coordinate system representations of items such as position, acceleration, orientation. A "Holder" Local Class is used disseminate the information generically between applications and users construct specific Local Class types as necessary.

State Consistency — When dealing with a collection of related attributes, it may be necessary to evaluate the particular values of all of the attributes in combination. For example, there may be rules that if the value of attribute A is negative, then the value of attribute B must also be negative. Enforcing this type of arbitrary state consistency with a group of attribute values can be accomplished with a Local Class. The constructor and set methods of the local Class would implement the particular rules.

Local Helper Functions — Local Classes can also be used to develop common helper functions that are used by many applications. For example, a Distance Local Class could be created that used to calculate the three dimensional distance between two points (where one of the points could be stored within the Local Class). Using a Local Class for these helper functions allows the user community to develop and test the code once, and then share among the community.

Custom Data Marshaling — In some circumstances, it may be appropriate to use custom data marshaling of information (typically a bit stream) when it is disseminated to other applications. A Local Class can be used to encapsulate the raw unmarshaled data and local methods can be used to allow the end user to obtain the necessary information in a semantically rich manner.

When using a Local Class with either a constructor or method, there needs to be implementation code provided by each application attempting to use that Local Class type. Typically, all applications in a particular event using a Local Class will use a common implementation of the Local Class methods and the implementation code is packaged as an object library that is linked with the applications. Object model designers need to take the implementation and dissemination requirements into consideration when defining Local Classes, e.g., who is going to implement the behavior needed for this Local Class? will it be implemented across all of the computer platforms that are being used for this event and future events? do we need to ensure everyone in the event is using the same implementation?

C++ API Reference and Code Examples

An simple Local Class (from the `Example-Vehicle` object model) is defined below. This Local Class only has two attributes and there are no constructors or methods, so there is no user defined Local Class implementation that is required.

```
local class Location
{
    float64 xInMeters;
    float64 yInMeters;
};
```

Another Local Class definition (from the `Example-Tank` object model) is shown below. In this example, the Local Class has a constructor and several methods which requires implementation registration for applications using this Local Class type (see the [Local Class Method Factory Registration documentation page](#) for additional information on this registration process).

```
local class HomeStation
{
    // Ensure name is not the empty string
    HomeStation( string name, Location location );

    void set_name( in string name ); // Ensure name is not the empty string
    float64 distanceFrom( in Location location ) const;

    readonly string name;
    Location location;
};
```

Note that the code fragments referenced are for particular versions of the Example object models, specifically `Example-Vehicle-v1` and `Example-Tank-v3`. These object models may be updated/refined as necessary.

Message Meta-Model Type

Message Meta-Model Type

A message is a complex data type containing both operations/methods and state information, exactly like a local class, except that messages can be sent using the TENA Middleware's messaging service as bursts of information. For example, a user might define "Fire," "Detonation," or "Missile Away" messages to carry the appropriate information to the rest of the logical range participants when a fire event, detonation event, or missile away event occurs.

Description

A Message is used to represent the occurrence of a particular event. The Message is characterized by a collection of attributes, and there are local methods to support Message operations in a similar manner to Local Classes. A Message is ephemeral and within the distributed logical range exercise only exists for the instant in which it is transmitted. The publishing application prepares the particular Message and then transmits the message to interested subscribers. Subscribers effectively have a copy of the transmitted Message as there is no connection to the publishing application like there is with a Class (or Stateful Distributed Object, SDO).

Message Properties

Messages have the following properties:

- Messages may be contained in:
 - SDOs,
 - Local Classes,
 - Vectors.
- Messages may contain:
 - Fundamental Types,
 - Enumerations,
 - Vectors,
 - SDO Pointers,
 - Local Classes, and
 - other Messages.
- Messages may contain constructors and methods that require user-defined implementations.
- Messages may inherit from at most one other Message type.
- Messages may be used as method arguments and return values.

Additional information on the use of Messages can be found in the [Message documentation page](#).

Usage Considerations

Similar to Local Classes, when Messages have a constructor or method, there needs to be implementation code provided by each application attempting to use that Message type. Typically, all applications in a particular event using a Message will use a common implementation of the Message methods and the implementation code is packaged as an object library that is linked with the applications. Object model designers need to take the implementation and dissemination requirements into consideration when defining Messages, e.g., who is going to implement the behavior needed for this Message? will it be implemented across all of the computer platforms that are being used for this event and future events? do we need to ensure everyone in the event is using the same implementation?

C++ API Reference and Code Examples

The TDL syntax for a simple Message (from the `ExampleVehicle` object model) is shown below. In this example the Message only contains a single attribute which is a character string. Since there is no constructor or method, no user-defined implementation code is required for this Message.

```
message Notification
{
    string text;
};
```

A slightly more complicated Message example from the `Example-Tank` object model is shown below. In this example, the `Explosion` Message inherits from the `Notification` Messages and adds a constructor, method, and attribute. Applications using the Message type are required to register the implementation (see the [Message Implementation Registration documentation page](#) for additional information on this registration process).

```
message Explosion : extends Notification
{
    Explosion( Location reportedlocation );
    float64 distanceFromCenterOfExplosionInMeters( in Location here ) const;
    AreaOfEffect affectedArea;
};
```

Attribute Meta-Model Construct

Attribute Meta-Model Construct

The basic elements for the TENA modeling constructs are attributes. Attributes are used to characterize the state of a particular Class, Local Class, or Message. Attributes can be fundamental data types, enumerations, Vectors, SDO Pointers, or Local Classes. Applications set the appropriate values for the attributes to represent the item the item that they are sharing with other applications.

Description

The key TENA Meta-Model constructs are the Class, Local Class, and Message. Each of these constructs use attributes whose values are used by the application to properly represent the particular item. For example, the Class `Soldier` could have attributes for `name`, `rank`, and `location`. The values for these attributes can be used to characterize the particular soldier being represented in the TENA event.

Attribute types can be fundamental data types (e.g., integers, floating point numbers, character strings), enumerations, SDO Pointers, Local Classes, or Vectors of any of these types. Publishing applications that create a Class, Local Class, or Message are permitted to initialize and set the attribute values associated with these model types.

When defining attributes in an object model, designers can use several qualifiers for each attribute described below. Each qualifier listed has a linked documentation page with additional information.

Qualifier	Description
Const	Object attributes and package scoped attributes can be qualified in the object model as "const" to indicate that their values will not change.
Optional	Object and Message attributes can be marked with the "optional" qualifier to indicate that the attribute values may not be set by the publisher.
 Readonly	Object model attributes for Local Classes and Messages can be qualified as "readonly" to indicate that users are not able to directly set these values.

C++ API Reference and Code Examples

A simple illustration of the specification of attributes within a Class (`Soldier` from Example-Tank object model) is shown below. The attributes `name` and `serialNumber` are marked `const` because those values will not change. The attributes `team` and `homeStation` are marked as `optional` because it is expected that some publishing applications may not be able to specify these attributes.

```
class Soldier
{
    void moveTo( in Location destination );

    const string name;
    string rank;
    const uint32 serialNumber;
    optional const Team team;
    Location location;
    optional HomeStation homeStation;
};
```

Const Attribute Qualifier

Const Attribute Qualifier

In some circumstances, SDO (Stateful Distributed Object) attributes do not change value during the existence of the object. These attributes can be qualified as "const" in the object model to indicate to application developers that the values do not need to be checked and re-processed on every update. An object model can also define package scoped attributes that can be used as constants (e.g., PI = 3.1415). Additionally, Local Class methods can be qualified as const if the method does not change the state of the local class.

Description

The TENA Meta-Model supports the use of a `const` qualifier to identify when certain attributes will have constant values (i.e., value will not change during the execution), as well as identify local class methods that will not change the value of the local class attribute values. The `const` qualifier can be applied to SDO attributes and must be used for package scoped attributes.

One major benefit of using the `const` qualifier is for the TENA Middleware to optimize the communication requirements for attributes whose values will not change. For example, consider an object that contains a vector of fixed locations. The size of the vector contained in a network packet due to an object update may be several hundred bytes, but if this attribute was marked as `const` in the object model the value will only be transmitted once to each subscriber during discovery.

Marking local class methods as `const` is useful when a subscribing application needs to operate on the local class through the discovered SDO Proxy or received Message. Normally, without the `const` qualifier, the local class methods do not exist on the local class interface obtained through the SDO proxy or received message. This is because a subscribing application is not able to modify the state of the SDO proxy or received message.

Additionally, a `const` package scope attribute can be used to define constants that developers can use in their applications, and be ensured that all applications use the same constant. For example, an approximation for the constant "pi" could be defined as "`const pi = 3.1415`".

Usage Considerations

When designing a TENA object model, there are three potential uses of the `const` qualifier that need to be reviewed by the object model designers:

- Package Scope Attributes
- SDO Class Attributes
- Local Class Methods

Package scope attributes must always be defined as `const` since there is no mechanism to change the value of these attributes. These constants are useful programming constructs that can be shared by application developers to ensure that the same constant value is used by all applications. The OM Compiler will indicate an error if a package scope attribute is not defined with the `const` qualifier.

Each SDO class defined within an object model should be reviewed to determine whether any attributes should have constant values and, therefore, use the `const` qualifier. As indicated above, the use of `const` will minimize network bandwidth utilization for these objects. When an attribute is defined as `const`, the value will need to be provided by the publishing application when an object of that SDO type is created (using the SDO initialization mechanism).

Additionally, any local class methods defined in the object model should be reviewed to determine if it is expected that any implementation of the method would not change any of the local class attribute values. Defining these methods as `const` will simplify the usage of these methods by subscribers by allowing direct invocation of the methods from the proxy's publication state.

C++ API Reference and Code Examples

Excerpts from an example object model that illustrates the use of the `const` qualifier for package scope constants, SDO attribute constants, and local class const methods are shown below.

```
// Package scope constant
package Example
{
    const float64 minimumRadiusOfEffect = 0.0;
    ...
    class Soldier
    {
        const string name;
    ...
    };
    ...
    local class HomeStation
    {
        float64 distanceFrom( in Location location ) const;
        ...
    };
}
```

Note that in the case of const local methods, the correct syntax is to place the `const` qualifier at the end of the method signature. It is also necessary to specify the `optional` qualifier before the `const` qualifier if both are necessary in the case of an attribute.

Optional Attribute Qualifier

Optional Attribute Qualifier

When designing object models to capture all of the information about a particular object or message, there may be certain attributes in which the publisher may not have a valid value to use. Rather than attempting to use special values to convey that an attribute has not been set, object model designers can use the "optional" qualifier to indicate to application developers that this attribute may not be set by the publisher. Subscribing applications are required to check (using the `is_attribute_set` method) whether the attribute is set before attempting to obtain its value.

Description

When designing an object model, each attribute associated with an SDO, Message, or Local Class can be marked as an "optional" qualifier. An optional attribute indicates that a publishing application is not required to provide a value for the attribute. This may be useful in circumstances where information may not be available to every application, or when the application has intermittent access to the attribute value.

The optional attribute mechanism avoids the need for publishing applications to develop some additional mechanism to inform subscribers that an attribute value is not available. Using the TENA Middleware, a subscribing application can check if an optional attribute has a value before attempting to read the value. This check is necessary to avoid a runtime exception that may occur if the attribute value was not provided by the publishing application.

If an attribute is not marked as `optional`, then it is therefore required, and the publishing application must provide a value when the object is updated or the message is sent. Any attempt to update an object or send a message without provided a required attribute will result in a runtime exception.

Usage Considerations

When an application developer attempts to utilize the particular object or message types in an object model, an important programming consideration is whether the attributes are specified as `optional`. From a publishing application perspective, any non-optional (i.e., required) attributes will need to be at least initialized in each object (although not necessarily set for each update) and must be set in each message that is sent. If a developer is unsure what to use for attribute values, they should refer to the object model documentation and/or the event coordinators.

An optional attribute is considered to be "set" as a side-effect of providing any value; but, just as optional attributes can be "set", they can also be "unset". A publisher should take advantage of this feature in the event that the value for an optional attribute has ceased to be available or pertinent — for instance, if a hardware sensor has failed to report any value. "Unsetting" is accomplished using the `unset_attributeName` method on the publication state. For example, since the `Example::Tank` SDO has an optional attribute named "damageInPercent", the following method exists in the C++: `Example::Tank::PublicationState->unset_damageInPercent()`.

Subscribing applications will need to check each optional attribute before attempting to read the attribute value. The mechanism to check is the method on the SDO proxy named `is_attributeName_set()`, where `attributeName` is the particular name used in the object model. This method returns a boolean value (`bool`), where a value of true value indicates that the attribute value has been set.

In the generated C++ code, local classes and messages are referenced through a reference-counted smart pointer. In release 5 of the Middleware, this smart pointer could be set to a null value; and it was not uncommon for object models to use the null value to indicate that an attribute had not been set. In release 6 of the Middleware, the availability of optional attributes renders this pattern unnecessary. Consequently, **the LocalClassPtr cannot be null**. Attempts to initialize or reset the `LocalClassPtr` to a null value (i.e., zero) will result in a `std::invalid_argument` exception. This is a simplification of semantics that yields a subtle-but-useful benefit for client code: `LocalClassPtr` does not need to be checked for validity. If you have a `LocalClassPt`, you can simply assume it's valid and use the local class instance.

C++ API Reference and Code Examples

An example object model class using the `optional` qualifier is shown in the object model definition fragment below.

```
package Example
{
    class Tank : extends Vehicle
    {
        ...
        optional float32 damageInPercent;
        ...
    };
}
```

In the object model example for class `Tank`, the attribute `damageInPercent` is optional. Subscribing applications are required to check if the `damageInPercent` attribute value has been set before attempting to read the value. A code fragment for performing this check is shown below.

```
Example::Tank::PublicationStatePtr pTankPubState( pTank->getPublicationState() );  
  
if ( ! pTankPubState->is_damageInPercent_set() )  
{  
    os << "<uninitialized>" << ::std::endl;  
}  
else  
{  
    os << pTankPubState->get_damageInPercent() << std::endl;  
}
```

Readonly Attribute Qualifier

Readonly Attribute Qualifier

When an object model defines attributes for a Local Class or Message, the Object Model Compiler will automatically generate methods for applications to set and get the value of those attributes. In some circumstances, the object model designer does not want set methods to be generated which provide an implementation for the user to perform a simple set of the attribute value. In these cases, the object model designer can qualify the attribute with the keyword "readonly" to indicate that the set methods should not be generated. This behavior is helpful when the Local Class or Message wants to use constructors and/or methods to control the values of these readonly attributes.

Description

The TENA Meta-Model supports the attribute qualifier "readonly" for Local Classes and Messages, which object model designers can use to indicate that the `set` methods that allow users to set the attribute's value should not be automatically generated. The `readonly` qualifier is used when the object model designer plans to use other mechanisms, such as constructors and/or methods to internally set the values of these readonly attributes. Users will then only be able to use the `get` methods to access the attribute's values.

When a Local Class or Message has one or more readonly attributes, it is necessary that the Local Class or Message has defined a constructor or method – otherwise there would be no mechanism for the attribute value to ever be set. Even when a constructor or a method is defined, it is still possible that the implementation code does not set the readonly attribute value. In this situation, the generated code detects that the attribute value has not been set and a run-time exception will occur when a user invokes the `get` method. Therefore, object model designers and implementers need to ensure that readonly attributes are always set before a user attempts to get the value.

Usage Considerations

There are several common use cases for "readonly" attributes that are defined below.

Constructor Initialization — When designing a Local Class or Message, the object model designer may want to force the user to utilize a constructor to properly initialize the Local Class or Message at creation time. The alternative is to create the Local Class or Message with invalid attribute values and hope that the user sets the values before the local Class or Message is used. When forcing constructor initialization, the attribute values can be marked as `readonly` indicating to the user that the values must be provided in the constructor (and in fact there is no way to create the Local Class or Message without providing the necessary attribute values).

Attribute Value Checking — If the object model implementation needs to perform checking of attribute values before being set, such as reject negative values for the speed, then the attribute can be defined as `readonly` preventing the generation of the default `set` method. Instead the object model would require the speed value to be defined in user defined constructors and/or methods and the implementation for the constructor and/or methods would be responsible to check that the speed value is non-negative before it is set internally.

Dependent Attributes — In some cases, the values of a collection of attributes need to be evaluated for correctness together, versus individually. If these attributes are not defined as `readonly`, the automatically generated `set` methods would allow a user to set the values and the implementation would permit the collection of values to be in an inconsistent state. In this situation the dependent attributes can be marked as `readonly` and constructors and/or methods can be used to control the atomic setting of all of the attributes.

Union Semantics — Currently, the TENA Meta-Model does not support the "union" programming construct in which there are a collection of attributes in which only one of the attributes can be set at a time. The union construct can be supported by defining the attributes as `readonly` and then define `set` methods that either toggle the attribute values to ensure only one is set at a time, or throw a run-time exception if the user does not unset the other attributes before setting a different one.

C++ API Reference and Code Examples

A simple Local Class with readonly attributes is shown in the object model fragment below (for the `Example-Tank` object model). This Local Class requires users to use the constructor to set the attribute values.

```
local class AreaOfEffect
{
    AreaOfEffect( Location centerOfArea, float64 radius );

    boolean isLocationAffected( in Location here );

    readonly Location centerOfArea;
    readonly float64 radius;
};
```

Private Attribute Qualifier

Obsolete

The "private" attribute qualifier is no longer supported and should not be used.

Private Attribute Qualifier

When an object model defines attributes for a Local Class or Message, the Object Model Compiler will automatically generate methods for applications to set and get the value of those attributes. In some circumstances, the object model designer may not want set and get methods to be generated, as these attributes are meant to only be used internally by the Local Class or Message implementation. In these cases, the object model designer can qualify the attribute with the keyword "private" to indicate that the set and get methods should not be generated. This behavior is helpful when the Local Class or Message needs to use some form of custom marshaling with the data transmitted across the network and will use methods to present the user with access to the relevant information.

Description

The TENA Meta-Model supports the attribute qualifier "private" for Local Classes and Messages, which object model designers can use to indicate that the set and get methods that allow users to set or get the attribute's value should not be automatically generated. The private qualifier is used when the object model designer needs to store internal data for the Local Class or Message implementation, but users are not required to set or access this data.

When a Local Class or Message has one or more private attributes, it is necessary that the Local Class or Message has defined a constructor or method – otherwise there would be no mechanism for the attribute value to ever be set.

Usage Considerations

The private keyword is similar to the readonly qualifier (see [Readonly Attribute Qualifier documentation page](#)) with the addition that the private qualifier does not allow the user to obtain the attribute value that was actually transmitted by the publishing application.

The private keyword is used when the object model designer does not want the user to access the attributes that were actually transmitted by the publishing application. This may be appropriate in situations in which there is custom marshaling of the data transmitted over the network and it would be too complicated for a user to attempt to perform this marshaling – instead the Local Class or Message encapsulates the marshaling and demarshaling for the user and presents intuitive methods to set and get the necessary information.

As noted, the private mechanism prevents users from accessing the actual data that was transmitted by the publisher. In some cases this may interfere with the requirements of data collection systems. Since object model designers may not know the future data collection requirements it is generally recommended to use the readonly qualifier instead of the private qualifier.

Operation Meta-Model Construct

Operation Meta-Model Construct

When developing object models, designers can define Operations (aka methods) associated with Classes, Local Classes, and Messages to provide users with services that can be used by applications. In the case of Classes, the Operations represent remote method invocations that a subscribing application can invoke on the servant object in the publishing application. Operations associated with Local Classes and Messages are performed locally with respect to the process space in which the Local Class or Message exists.

Description

The TENA Meta-Model supports the definition of **Operations** for Classes, Local Classes, and Messages. These Operations (aka methods) enable object model designers to provide behavior to the users, versus just exchanging data only structures between publishing and subscribing applications. Typically these operations will operate on the state information of the Class, Local Class, or Message, but it is possible to define constructs that don't require state and merely provide a service to users.

Operations are defined by their name, arguments, and return values. Arguments and return values can be fundamental types (e.g., integers, floating point values, character strings), Local Classes, Messages, SDO Pointers, or Vectors of any of those types. In the case of inheritance, derived operations with the same name must have the same definition in terms of arguments and return values.

Operation arguments are qualified with either `in`, `out`, or `inout` to indicate whether the argument is passed in by the caller, returned to the caller, or passed in and returned. These semantics are necessary to control whether the argument is passed by reference or by value. For example, if the argument is passed into the operation, a `const` reference can be used. Users only need to consider if the operation is returning an argument value to the invoker. Typically, arguments are passed in to the operations. Operations are also permitted to raise exceptions, which are defined in the object model.

Operations can be qualified as either "oneway" or "const" within the object model. When a oneway method is used, it indicates that the invoker does not need to be informed if there is an error in invoking the operation. A normal operation will return a run-time exception if the implementation raises an exception or if there is a communication fault in invoking the method. A const operation is used for Local Classes and Messages to indicate that the operation will not modify the state of the Local Class or Message. Additional information can be found on the [Oneway Operation Qualifier](#) and [Const Operation Qualifier](#) documentation pages.

C++ API Reference and Code Examples

An example of a Class with operations is shown below, for the `Example-Tank` object model. The `Tank` class defines several operations. A subscribing application that discovers a particular `Tank` is able to invoke any of these methods to have the method implementation in the publishing application perform the appropriate activity.

Note that all of these operations do not have a return value, so `void` is used for the definition. Additionally, the `loadSoldier` operation can raise a `CannotLoadSoldier` exception that would be returned to the invoker.

```
exception CannotLoadSoldier { string reason; };

class Tank : extends Vehicle
{
    void loadSoldier( in Soldier * pSoldier ) raises ( CannotLoadSoldier );
    void unloadSoldier( in Soldier * pSoldier );
    void driveTo( in vector < Location > wayPoints );
    void fireGun( in Location targetLocation );

    optional float32 damageInPercent;
    vector < Soldier * > passengers;
};
```



As of May 2013, the use of `out` and `inout` parameters in methods is deprecated and is prevented for new object model submissions. This is motivated by a lack of a clear requirement for the capability and issues of compatibility with programming languages other than C++.

Const Operation Qualifier

Const Operation Qualifier

Operations associated with Local Classes or Messages can be qualified as "const" if those operations do not modify the state of the Local Class or Message. This is necessary because received Local Classes or Messages are treated as immutable from the middleware perspective to prevent applications from accidentally modifying the state of a received Local Class or Message. Only operations marked as const can be invoked on immutable Local Classes or Messages.

Description

The TENA Middleware prevents applications from directly modifying the state of received Local Classes or Messages. The received Local Classes or Messages are contained within classes name `Immutable` to indicate that the state can not be changed. If a receiving application needs to modify the state of a received Local Class or Message, then a new `Mutable` Local Class or Message needs to be created by copying the received version.

Local Classes and Messages can be defined to have operations (aka methods) and the TENA Meta-Model supports a "const" qualifier that can be applied to the operation to indicate that the operation does not modify the state. In this situation, an application that has an `Immutable` Local Class or Message can only invoke `const` operations that will not modify the state.

As an example, there could be a `Velocity` Local Class that contains a velocity vector. This Local Class could define a `getSpeed` method that calculates and returns the magnitude of the velocity vector. Since this operation does not modify the state of the Local Class, the method can be defined as `const`.

Usage Considerations

As part of object model development, all operations associated with Local Classes or Messages should be reviewed to determine if implementations of the operation would need to modify the state. If it is expected that the operation will not modify the state, then the operation should be defined as `const`. Note that multiple implementations of an object model that are possible, so object model designers need to consider alternative or future implementations when considering whether an operation should be defined as `const`.

Oneway Operation Qualifier

Oneway Operation Qualifier

The TENA Meta-Model supports the use of the qualifier "oneway" for Class operations (aka methods). The oneway qualifier indicates that the operation does not block waiting for the operation to complete. The result of this behavior is that the invoker will not be informed if the operation fails either due to a communication fault, invocation error, or implementation problem.

Description

TENA object model classes can define operations that are used for remote method invocations between a subscribing application and the application that is publishing the SDO (Stateful Distributed Object). These operations allow the application to provide arbitrary distributed services to the subscribing applications. In a distributed event, there are network delays in invoking remote methods that will block the application thread invoking the operation.

When the operation does not return any value (or have any "out" arguments) or raise any exceptions, the object model designer can define the operation to be "oneway". This qualifier changes the behavior to immediately return from the invocation without waiting for confirmation that the operation succeeded. Therefore, oneway operations avoid the blocking behavior associated with the network communication and implementation processing, but they are unreliable since the invoking application is unable to determine if the invocation was successful.

Usage Considerations

Object model designers should only utilize oneway operations when the blocking behavior is intolerable and applications can tolerate the unreliable nature of these invocations. In some circumstances, the application may need to develop other means to detect and compensate for oneway operations that fail. For example, a oneway operation could be used to instruct another application to perform some work and respond with a separate oneway response. If the application does not receive the response after some timeout period, it may be necessary to perform the invocation again.

Inheritance Meta-Model Support

Inheritance Meta-Model Support

The TENA Meta-Model supports the concept of inheritance for Classes, Local Classes, and Messages. Inheritance is used to create a "derived class" that inherits the attributes and operations of the "base class". From a TENA object model perspective, inheritance may be useful to extend previously defined constructs where interoperability is required between fielded applications using the original base class definition and new applications using the derived class. Additionally, the primary purpose of inheritance in software programming, polymorphism, also applies in the design of TENA object models using inheritance.

Description

The primary modeling constructs supported by the TENA Meta-Model are Classes, Local Classes, and Messages. All of these meta-model constructs support the property of inheritance in which a derived class can inherit the attributes and operations of a base class. This allows the derived class to add attributes and operations allowing some subscribing applications to use the item as a base type and others applications to use the item as a derived type.

Inheritance should be used when the derived type **"is a"** base type and it is necessary to support both types (as opposed to just using a single type). The primary motivation for using inheritance in TENA object models is to support **polymorphism**, but inheritance is also useful for object model evolution.

When an object model type has been established and existing applications have been developed for that type, it can be difficult to quickly introduce extensions to that object model type because of the fielded applications. In this case, inheriting from the existing type to add attributes and/or operations would allow existing applications using the base type to interoperate with applications using the derived type. Obviously, the existing applications would not be able to use or provide the extended attributes and operations, but that may be an acceptable transition strategy.

Inheritance within the TENA Meta-Model is limited to only inheriting from a single base class — multiple inheritance is not supported. The derived class is not permitted to add attributes or operations with the same name/signature. The implementation of the operations for the derived class must provide implementation code for all operations, including the inherited operations. If the derived class operation wants to delegate to the base class operation, then the derived class operation must perform that operation as the generated object model code does not perform that delegation automatically.

When using derived types, whether SDOs (Stateful Distributed Objects), Local Classes, or Messages, the TENA Middleware supports the property of substitutability. Substitutability is the property where a derived type can be used in the place where a base type is expected. The Middleware utilizes smart pointers in the API for the object model types, and these pointers enable the substitutability property providing automatic conversion when necessary.

Additionally, the TENA Middleware supports dynamically casting a base type to a derived type using the `TENA::Middleware::Utils::DynamicCast` mechanism. This downcasting mechanism works with the middleware smart pointers (`TENA::Middleware::Utils::SmartPtr`) that are used within the Middleware API for holding an SDO, Local Class, or Message. The downcasting of a smart pointer will only succeed if the base type is actually a derived type. This function will throw a `std::bad_cast` exception if the cast was not successful.

Note that in addition to the `SmartPtr`, some middleware API elements are held in a `ManagedPtr`. Although users are not expected to downcast managed pointers, their behavior with respect to an unsuccessful downcast would be to return a null pointer.

Additional information related to inheritance support for the specific object model constructs can be found on their specific documentation pages:

- [SDO Inheritance](#)
- [Local Class Inheritance](#)
- [Message Inheritance](#)

C++ API Reference and Code Examples

An example of Class inheritance is shown in the object model fragment below. The class `Tank` is shown to inherit (specified through the keyword `extends`) from the `Vehicle` class. A number of operations and attributes are added through this inheritance.

```
class Tank : extends Vehicle
{
    void loadSoldier( in Soldier * pSoldier ) raises ( CannotLoadSoldier );
    void unloadSoldier( in Soldier * pSoldier );
    void driveTo( in vector < Location > wayPoints );
    void fireGun( in Location targetLocation );

    optional float32 damageInPercent;
    vector < Soldier * > passengers;
};
```

Class Pointer Meta-Model Construct

Class Pointer Meta-Model Construct

In some modeling situations it may be useful to provide other applications a reference to a particular Class instance. The TENA Meta-Model uses Class Pointers to support this distributed pointer mechanism. Object models can use Class Pointers as attributes or operation arguments/return values. An application that obtains an SDO Pointer can use the middleware to perform an "instance-based" subscription to the particular SDO instance so that an SDO Proxy is returned so that update, scope, and destruction notifications are delivered to the application (similar to an SDO Proxy obtained through a normal type-based subscription).

Description

TENA object models will often need to define an association with a particular Class (i.e., Stateful Distributed Object, SDO) instance. This association can be represented in the TENA Meta-Model as a Class Pointer, which results in an SDO Pointer in application code. Applications can use the SDO Pointer to obtain a proxy for the SDO in which attribute updates can be obtained and remote methods can be invoked. Within an object model, Class Pointers can be used as attributes (of Classes, Local Classes, or Messages), and as operation arguments or return values.

As an example, a `Car` class may need to define an association with a `Person` instance that is defined as the `driver` attribute for the car. In this situation, a subscribing application could discover a `Car` SDO and use the `driver` attribute value to obtain a proxy to the current driver for that car. It should be noted that in this model the `driver` pointer can be updated during the execution (i.e., car driver changes) and the `driver` pointer can also be (effectively) set to "null" to indicate that there is no driver for the particular car.

Class Pointer Properties

Class Pointers have the following properties:

- A given Class Pointer refers to a specific type of Class.
- Class Pointers may be used as arguments and return values in operation signatures.
- SDO Pointers may be contained in:
 - Classes,
 - Local Classes,
 - Messages, and
 - Vectors.

Additional information on the use of SDO Pointers can be found in the [SDO Pointer documentation page](#).

C++ API Reference and Code Examples

An example of the use of Class Pointers is shown in the object model fragment below (from the `Example-Tank` object model). In this example, the `Tank` class has a vector of Class Pointers to `Soldiers` representing the `passengers` attribute. Additionally, the `loadSoldier` and `unloadSoldier` operations use a `Soldier` Class Pointer as arguments.

```
class Tank : extends Vehicle
{
    void loadSoldier( in Soldier * pSoldier ) raises ( CannotLoadSoldier );
    void unloadSoldier( in Soldier * pSoldier );
    void driveTo( in vector < Location > wayPoints );
    void fireGun( in Location targetLocation );

    optional float32 damageInPercent;
    vector < Soldier * > passengers;
};
```

Remote Method Exception Meta-Model Construct

Remote Method Exception Meta-Model Construct

Under some circumstances it may be useful or necessary to allow SDO remote methods to throw an exception that could be caught and handled by the "invoker". Nominally, exceptions should be used when an exceptional situation occurs leaving the method implementation unable to proceed. The exception allows the invoker to be informed of the condition and take appropriate programmatic action.

Description

A user defined exception is declared within the object model as follows:

```
exception MyException
{
    string explanation;
};
```

Exceptions can have any fundamental types, such as strings and enums, as attributes. Other meta-model constructs such as Classes, Class Pointers, Local Classes, Messages, and Vectors cannot be attributes of Remote Method Exceptions. Remote Method Exceptions do not support inheritance.

Only SDO methods (i.e., remote methods) can be declared to raise exceptions in the object model using the "raises" keyword. Multiple possible exceptions are separated with a comma, e.g.,

```
void myMethod() raises (MyException, MyOtherException);
```

Enumeration Meta-Model Construct

Enumeration Meta-Model Construct

Enumerations are used as a mechanism to define a modeling type with a fixed set of pre-defined values that can be used where Fundamental Types are supported (e.g., attributes, operation arguments). Enumerations are a type-safe alternative to creating attributes as unsigned longs with a semantic mapping that is maintained outside of the model (e.g., value of 1 means January, 2 means February, etc.)

Description

Similar to many programming languages, the TENA Meta-Model supports an enumeration construct. An enumeration is a model type that can be used to represent one value from a pre-defined fixed set of possible values. For example, the `Example-Vehicle` object model defines the following enumeration:

```
enum Team
{
    Team_Red,
    Team_Blue,
    Team_Green
};
```

This enumeration can then be used within an object model as an attribute of a Class, Local Class, Message, or Remote Method Exception. Enumerations can also be used as arguments or return values to operations.

Enumeration Properties

Enumerations within the TENA Meta-Model use the "enum" keyword and have the following properties:

- Enumerations may be used as parameters and return values from operations.
- Enumerations may be contained in:
 - Classes,
 - Local Classes
 - Messages
 - Exceptions, and
 - Vectors.

Usage Considerations

The Object Model Compiler generates C++ code for object model enumerations using type-safe techniques that prevent the accidental automatic conversion to or from integers. Therefore, applications must use the generated C++ enumeration values for creating an enumeration (unable to use an integer) and if an application needs to convert a TENA enumeration to an integer representation, the `value` method needs to be used which will return a `TENA::unit32` value. There is also an associated `ostream <<` operator that will provide a string representation of the enumeration value.

Vector Meta-Model Construct

Vector Meta-Model Construct

Within the TENA Meta-Model, a Vector represents a container that can hold an arbitrary and dynamic number of elements of the same type. The contained type can be fundamental data types (e.g., integers, floating point values, strings) as well as other meta-model constructs (e.g., Local Classes, Class Pointers).

Description

Often, real-world systems can be effectively modeled using a collection of elements of the same type in which the number of elements is dynamic. For example, a real-world vehicle does not always have the same or a fixed number of passengers. For these types of situation, the TENA Meta-Model offers a construct called "Vector".

The TENA Meta-Model provides Vectors to model the concept of varying cardinality of attributes. The TENA Meta-Model supports vectors of Fundamental Data Types, Local Classes, Messages, and SDO Pointers.

Vectors are defined within an object model using the `vector` keyword and the brackets "< >" to designate the contained type. Some examples are shown in the object model fragments below.

```
vector < TENA::int32 > numbers;
vector < Soldier * > passengers;
```

Vector Properties

Within the TENA Meta-Model, Vectors have the following properties:

- A vector may be contained in:
 - Classes
 - Local Classes
 - Messages
- A vector may contain:
 - Fundamental Data Types
 - Enumerations
 - Local Classes
 - Messages
 - Class Pointers (that all refer to the same Class type)
- A vector may be used as an argument or a return type in an operation.



So called *multidimensional* vectors (e.g. `vector < vector < TENA::int32 > > myArray`) have been deprecated and new object models using them will not be accepted. Mapping an n-dimensional array into 1-dimensional vector can be used instead, and wrapped within a Local Class if necessary.

Usage Considerations

The automatically generated code associated with an object model Vector uses the C++ standard `vector` class. Application developers unfamiliar with the standard C++ container classes are urged to read documentation on proper use of these standard classes (such as the [wikipedia C++ vector documentation page](#)). In other languages, a generated type-specific Vector class is generated.

Package Meta-Model Construct

Package Meta-Model Construct

All elements of an object model are required to be declared within a named package to prevent naming collision with elements of the same name, but from different object models. Object model Packages are translated into C++ "namespaces" during automatic code generation to ensure unambiguous object model code. Packages can be nested (i.e., one package defined within another package), but the outer package name needs to match the TENA User Group that owns the object model (see [TENA User Groups](#) for more information on User Groups).

Description

In the TENA Meta-Model, the keyword "package" was chosen to denote a namespace. The syntax for a Package is shown in the object model fragment below for the package named "Example".

```
package Example
{
    // all of the material in this namespace
};
```

Fully qualified names are written using the ":" operator. For example, the Vehicle Class in package Example is referenced as Example::Vehicle.

Packages can be nested, like shown in the example below.

```
package TENA
{
    package Embedded
    {
        local class ModeSdata
        ...
    };
}
```

In the example above, the fully qualified name for ModeSdata is TENA::Embedded::ModeSdata.

Import Meta-Model Construct

Import Meta-Model Construct

The ability to "import" an existing object model is a critical feature of TENA. It allows an object model to be easily used in the context of another object model. The ability to import small building block object models promotes reuse and interoperability by allowing object models to focus on specific model elements, versus having a single large object model for a specific event.

Description

The TENA Meta-Model supports the ability for one object model to "import" the definitions of another object model, thus allowing the importing object model to build from the imported object model by adding new elements or inheriting from imported elements. This import capability allows object models to focus on specific modeling elements, rather than be focused on large monolithic models that are written for a specific event. In some situations an event specific object model can be created that merely imports many other functionally specific object models.

Importing an object model file is not a preprocessor text inclusion such as a C++ include. The imported object model file must be a complete and syntactically valid. All the object model definitions in the imported object model file are accessible for use by the importing object model.

The syntax of importing a TDL file is fairly simple, as shown below. All object models need to be versioned, and the import statement needs to indicate the specific version that is being imported.

```
import < Example-Vehicle-v1.tdl >
```

Package Scoped Constant Meta-Model Construct

Package Scoped Constant Meta-Model Construct

The TENA Meta-Model supports the definition of a constant using a Fundamental Data Type and corresponding value within an object model package. This constant can then be used consistently with all of the applications using the particular object model.

Description

A Packaged Scoped Constant is a Fundamental Data Type (e.g., integer, floating point value, string) and corresponding value that is defined within an object model package. The constant's value is fixed and permits application developers to use the constant consistently. For example, an object model could define the constant "PI" to be "3.1415" and any applications using that object model could use the PI constant within their application code.

An example of a Package Scoped Constant from the Example-Tank object model is shown below. The constant, minimumRadiusOfEffect, is defined in the automatically generated header file for the Example-Tank object model definition.

```
package Example
{
    const float64 minimumRadiusOfEffect = 0.0;
    ...
}
```

TENA Definition Language

TENA Definition Language

TDL (TENA Definition Language) is a text-based representation used to define TENA object models. TDL grammar is based on the Object Management Group's (OMG's) Interface Definition Language (IDL), which is used to define CORBA interfaces. TDL extends IDL to support the specific needs of TENA in defining Classes, Local Classes, Messages, and other TENA Meta-Model constructs.

Description

TENA uses a text-based representation for defining object models using a grammar that supports the TENA Meta-Model. The TDL syntax follows the Object Management Group's IDL (Interface Definition Language) standard, with extensions added to support TENA concepts such as SDOs (Stateful Distributed Objects), Local Classes, and Messages.

 — Support for UML representations of TENA object models exists and is described in the [UML Support documentation page](#).

The TDL syntax for each of the TENA Meta-Model constructs is described in the various meta-model documentation pages (see [TENA Meta-Model](#)). A summary of the key TDL constructs are illustrated in the [Example-Vehicle](#) object model shown below.

```
package Example
{
    enum Team
    {
        Team_Red,
        Team_Blue,
        Team_Green
    };

    local class Location
    {
        float64 xInMeters;
        float64 yInMeters;
    };

    class Vehicle
    {
        string name;
        Team team;
        Location location;
    };

    message Notification
    {
        string text;
    };
}
```

Object model definitions are contained within a package scope, which, like most of the TDL structures, uses braces for marking purposes — "{ ... }". The basic object model constructs for an enumeration, Local Class, Class (i.e., SDO), and Message are illustrated in the object model example.

Note that in TDL, as in IDL, upper and lowercase versions of letters refer to the same letter, i.e., "foo," "Foo," and "FOO" are identical variable names in TDL. Keywords in TDL, however, must be spelled in exactly the case defined, i.e., message not Message or MESSAGE.

A table summarizing the key TDL constructs using an example are shown below. Each item listed is a link to a documentation page that provides additional details on the construct.

TDL Construct Examples

TDL Construct	Example Syntax
Fundamental Types	<pre>uint32 float64 string</pre>

Class	<pre>class Vehicle { string name; Team team; Location location; };</pre>
Local Class	<pre>local class Location { float64 xInMeters; float64 yInMeters; };</pre>
Message	<pre>message Notification { string text; };</pre>
Attribute	<pre>float64 xInMeters; const uint32 serialNumber; optional const Team team; readonly float64 radius;</pre>
Operation	<pre>void setNameAndLocation(in CommandPost * commandPost); float64 distanceFrom(in Location location) const; Tank * requestTank(in Soldier * pSoldier, in Location pickupLocation);</pre>
Inheritance	<pre>class Tank : extends Vehicle { ... };</pre>
Class Pointer	<pre>Soldier * pSoldier</pre>
Remote Method Exception	<pre>exception CannotLoadSoldier { string reason; }; void loadSoldier(in Soldier * pSoldier) raises (CannotLoadSoldier);</pre>
Enumeration	<pre>enum Team { Team_Red, Team_Blue, Team_Green };</pre>
Vector	<pre>vector < Soldier * > passengers;</pre>

Package	package Example { ... };
Import	import <Example-Vehicle-v1.tdl>
Package Scoped Constant	const float64 minimumRadiusOfEffect = 0.0;

TDL Keywords

As mentioned previously, TDL extends the IDL standard and there are a number of reserved keywords associated with the TDL grammar (even if many of them are not used directly by TENA). These reserved cannot be used as variable or method names in an object model definition. These reserved keywords are captured in the following table.

TDL Reserved Keywords

abstract	and	and_eq	any	asm	attribute	auto	bitand
bitor	bool	boolean	break	case	catch	char	class
compl	component	const	const_cast	consumes	context	continue	custom
default	delete	do	double	dynamic_cast	else	emits	enum
eventtype	exception	explicit	export	extends	extern	factory	FALSE
false	finder	fixed	float	float32	float64	for	friend
getraises	goto	home	if	implements	import	in	inline
inout	int	int8	int16	int32	int64	interface	local
long	message	module	multiple	mutable	namespace	native	new
not	not_eq	object	octet	oneway	operator	optional	or
out	package	primarykey	private	protected	provides	public	publishes
or_eq	raises	readonly	register	reinterpret_cast	return	sequence	set.raises
short	signed	sizeof	static	static_cast	string	struct	supports
switch	template	this	throw	TRUE	true	truncatable	try
typedef	typeid	typename	typeprefix	uint8	uint16	uint32	uint64
union	unsigned	uses	using	ValueBase	valuetype	vector	virtual
void	volatile	wchar	wchar_t	while	wstring	xor	xor_eq

 Unknown macro: 'xsIt'

UML Support

UML Support

TENA object models can be documented using the Unified Modeling Language (UML). UML is a general, open standard capable of representing TENA Meta-Model constructs.

Description

Although the TENA infrastructure for supporting object models is based on the text-based TDL (TENA Definition Language) representation, there is support for using the Unified Modeling Language (UML) representation. The TENA project plans to expand this support to work with multiple UML tools.

Object Model Compiler Overview

Object Model Compiler Overview

The TENA Object Model Compiler uses code generation technology to parse an arbitrary object model definition and automatically generate software that connects an application using that object model to the middleware. The generated OM definition software provides an API (application programming interface) that ensures that applications adhere to the specification requirements associated with the particular object model. In addition to the generated OM definition software, the Object Model Compiler also (by default) generates complete working applications based on the particular user object model that can be used as the foundation to develop specific applications. Since the Object Model Compiler is designed to support multiple "back-end" plugins for specific code generation requirements, users are permitted to develop custom plugins to support their particular code generation requirements.

Description

The TENA Object Model Compiler can be viewed as being composed of several parts:

- Front-end Parser
- Grammar Validator
- Code Generation Engine
- Back-end Plugin Framework
- Code Generation Plugins

The front-end parser accepts text-based TDL (TENA Definition Language) files that contain a definition of a particular object model. The parser uses the grammar validator to ensure that the TDL file follows correct grammar and the object model definition is legal. The code generation engine is then required to process the "parse tree" to examine the requirements specified by the code generation plugin being used to determine what code needs to be generated.

Users access the Object Model Compiler through the website repository when uploading and building object models. Information on uploading and installing object models can be found in the [Uploading Object Models](#) and [Installing Object Models](#) documentation pages.

Additional details on the Object Model Compiler can be found in the [Object Model Compiler space](#).

Uploading Object Models

Uploading Object Models

The TENA Repository is used to browse and build TENA object models. Users are required to upload object models to the repository in order to build the necessary object model definition code to be used by TENA applications. The open nature of the repository promotes reuse of existing object models and refinements to further interoperability. Additionally, the centralized deployment of the Object Model Compiler and associated plugins enables rapid refinements and corrections to the automatic code generation capabilities.

Description

The TRMC Repository (<https://www.trmc.osd.mil/repository>) has been developed to facilitate the sharing of TENA products across the user community. The primary products accessible from the repository are object models, and their related computer platform software distributions. The TENA project promotes an open repository of object models to encourage reuse and refinement of object model definitions (and implementations). In some circumstances, object models can be view restricted due to the incomplete or sensitive nature of the object model.

Object models, like most website components, are managed through User Groups. Each User Group has a set of related permission groups (named with suffix -admin, -team, and -user) that control privileges to add, update, and view objects models. Every object model in the repository is "owned" by a particular User Group, including the TENA standard object models that are owned by the "TENA" User Group.

Users can browse the repository as shown in the figure below. The search capability allows a user to select the particular middleware version of interest and other search criteria.

The screenshot shows the TENA Test and Training Enabling Architecture (TENA) repository search results page. The top navigation bar includes links for 'TENA Website', 'Repository Home', 'Components', and 'Help'. The main search interface features a 'Browse | Search' button, a search input field, and filter options for 'Middleware Version', 'Platform', 'Component Type', 'Group', and 'Category'. The search results table displays 30 items found, with columns for 'Name', 'Description', 'Type', 'Last Updated', and 'Attributes'. The table rows list various TENA object models, such as 'TENA-AMO-v2', 'TENA-Embedded-v3', and 'TENA-Exercise-v1', along with their descriptions, types (ObjectModel), last update times, and release status (Released or Draft). A note at the bottom indicates that the 'TENA-Platform-v4' entry is a placeholder.

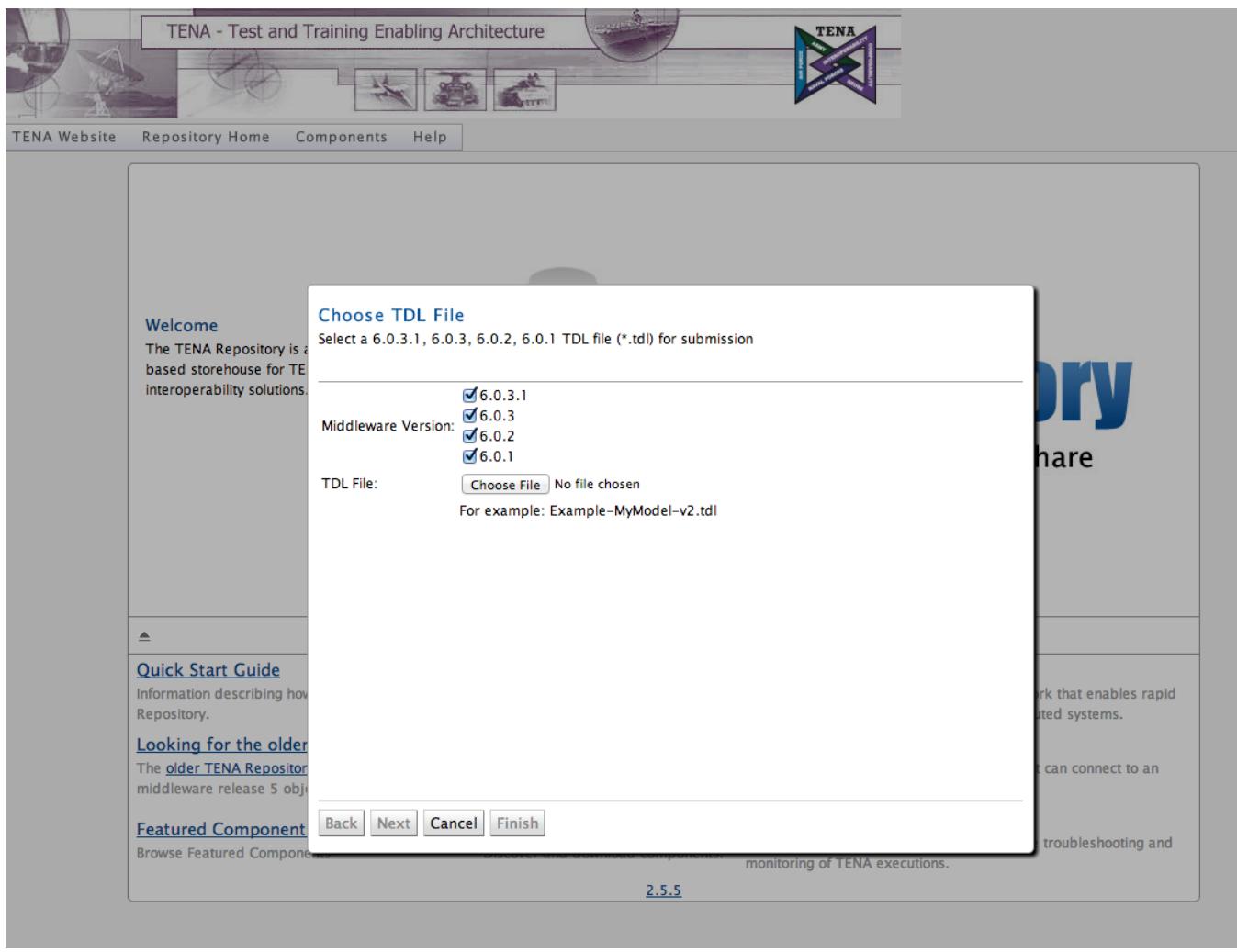
Name	Description	Type	Last Updated	Attributes
TENA-AMO-v2	The TENA AMO is used for publishing the health and status information of TENA applications	ObjectModel		Released Public
TENA-Embedded-v3	Defines SDOs for use with systems that are embedded in/on platforms	ObjectModel	6/4/13 2:30 PM	Released Public
TENA-Engagement-v4	Defines various messages representing fire engagements between platforms	ObjectModel		Released Public
TENA-Entity-v1.OM1135.2	Defines a common base class for items that have identity and spatial information.	ObjectModel		Released Public
TENA-Exercise-v1	Defines information about military exercises including force and organization structures.	ObjectModel		Released Public
TENA-GPS-v3	Defines objects related to Global Positioning System receivers and tracks	ObjectModel		Released Public
TENA-Identifier-v1.OM1155.1	Defines an Identifier local class that supports the concept of a single ID in the TENA execution being comprised of three numbers (siteID, applicationID, and TENA::Identifier::id) to help identify particular elements within a TENA execution.	ObjectModel		Draft Private
TENA-MeasuredTSPI-v1.OM1135.3	Exploring concepts related to extending TSPI to support uncertainty.	ObjectModel		Draft Public
TENA-Munition-v3	Defines a munitions SDO for complex autonomous or guided munitions that have their own TSPIs	ObjectModel		Released Public
TENA-NamedTSPI-v1.OM1135.3	Defines a common base class for items that have identity and spatial information.	ObjectModel	2/13/14 5:20 PM	Released Public
TENA-Platform-v4	Defines the TENA::Platform and TENA::Track SDOs.	ObjectModel		Released Draft

Repository Search Results Example

When a user needs to upload a new (or updated) object model, the name of the TDL (TENA Definition Language) file needs to be named according to the standard naming conventions. The conventions are described in detail on the [TENA Naming Conventions documentation page](#), but the summary is that the TDL file name should be <UserGroup>-<objectModelName>-v<version>, where UserGroup is the name of the User Group that owns the object model, objectModelName is the user defined name for the object model, and version is the particular version number/string.

The user uploading the new or updated object model must be in the -team permission group for the User Group that will own the object model. Additional information concerning User Groups can be found in the [User Groups documentation page](#).

New object models are uploaded to the repository by clicking the "Submit Object Model" menu item of the repository web page. When selected, the user is presented with a dialog that will walk the user through the process of uploading and validating an object model. A screenshot of the dialog used to upload an object model is shown in the figure below.



Repository Add Model Example

When adding an object model to the repository, the submitter can indicate whether the object model should initially be marked as "private" to indicate that only those with User Group -team permission can view the particular object model. This may be appropriate if the object model is incomplete or in draft form. The private versus public setting, as well as a "US Only" marking can be applied to the object model by anyone with User Group team permission at any time by changing the "Restrictions" setting when viewing the object model in the repository.

Users with -team permission can modify the summary and description of the object model after it has been uploaded. Users are encouraged to provide documentation with their object models to help with proper usage. Additionally, UML diagrams of the object model can be attached to the details page.

After an object model has been uploaded, any user with -user access to that object model can request a download of the object model. When viewing the object model details page, the "Download" link is in the upper left below the object model name. If the object model has not already been built for the selected object model and computer platform, then the download process will initiate a build of the object model using the Object Model Compiler. When a build is required, the user will be notified by email when the build has been completed (typically within an hour for simple object models).

Additional documentation related to the use of the Repository may be found in the [Quick Start Guide](#).

Installing Object Models

Installing Object Models

Object Model Packages obtained from the TENA Repository contain all of the object model definitions associated with the object model (including imported object models) and the example application code that can be used to build simple applications that publish and subscribe to the types defined by the object model.

Description

After an Object Model Package has been downloaded from the TENA Repository, a user can install the contents by executing the package (double-clicking under Windows, or invoking the executable name under Unix). The Object Model Package is created using the [TENA Installer](#) which supports a GUI (graphical user interface) to support the installation of the software. For Object Model Packages, the GUI will list all of the individual components with the package (e.g., definitions, implementations, example applications). Users can deselect components already installed, but it is recommended to install everything in the package. If the Installer detects that a file will be overwritten that is not identical, the user will be prompted with an overwrite warning that can stop the installation.

Object Model Packages will select the default installation location to be based on the value of the `TENA_HOME` environment variable, if defined. Through the GUI (or command line specification), the user can specify the installation location. Object Model Packages are designed to be installed in the [TENA Software Directory Structure](#) so that the build files for the example applications work properly.

Existing user applications that need to incorporate a newly installed object model package are required to adjust their build files (e.g., Microsoft Visual Studio project files or Unix Makefiles) to reference the include file location and object model libraries for the new object model. When using the TENA Software Directory Structure, the location of the include files will already exist, but the linking of the new object model libraries will need to be added. When the new object model has Local Method implementations for Local Classes or Messages (either constructors or operations), the existing application will also need to add an include line for the new object model to register the implementation. Information on this registration process can be found in the [Local Class Implementation Registration](#) and [Message Implementation Registration](#) documentation pages.

Object Model Definition

Object Model Definition

A TENA Object Model is composed of several parts: Object Model Declaration, Object Model Definition, and optionally, Object Model Implementation. The declaration is used to specify the object model and needs to be defined according to the TENA Definition Language grammar rules. The object model definition is the software generated by the Object Model Compiler to ensure type safety and adherence to the object model definition by applications using the object model. The object model implementation contains the software used to implement the Local Class and Message constructors and operations that may exist in an object model.

Description

A principle of the Test and Training Enabling Architecture (TENA) is to formally define object model contracts that specify the information and services that are shared among collaborating applications. These object model contracts are enforced through code generation technology in which applications are required to use the generated API (application programming interface) software that is based on the object model specification. This generated API and related software is used to ensure type safety and adherence to the object model specification.

For example, if the object model indicates that the type `Location` requires three 32 bit floating point values for the specification, then the application will be prevented from using an integer value, or only providing two of the three values. The generated API is designed for the application developer's compiler to detect these errors at compile-time, rather than detecting at run-time when the cost of a software defect is much higher. Where possible, the API is designed to make it hard to be used wrong.

The OM specific API and related software automatically generated by the Object Model Compiler represents the Object Model Definition. It contains the OM specific code to publish, subscribe, and use the various constructs that are defined in the particular object model. Applications are required to use the appropriate object model definitions, in conjunction with the TENA Middleware, to participate in a TENA execution.

As mentioned, object model definitions are generated by the Object Model Compiler. This can be done through the TENA Repository (<https://www.tena-sda.org/repository>) as discussed in the [Uploading Object Models documentation page](#). Once an object model definition is installed on a computer, the appropriate header files are located in the `TENA_HOME/TENA_VERSION/include/OMs` directory with a top-level directory name that corresponds to the name of the object model (e.g., `TENA/6.0.0/include/OMs/Example-Vehicle-v1`). The object code libraries associated with the object model definition are installed in the `TENA_HOME/TENA_VERSION/bin/PLATFORM` and/or `TENA_HOME/lib` directories (`bin` is used for Windows-based platforms) and named for the object model (e.g., `TENA/6.0.0/bin/libExample-Vehicle-v1-xp-vc80-v6.0.0.dll`).

Applications are required to use the object model definitions for all object models that are used by that particular applications. If the TENA execution is using a particular object model that is not used by an application, that application is not required to install and build against that object model definition.

Object Model Implementation

Object Model Implementation

A TENA Object Model is composed of several parts: Object Model Declaration, Object Model Definition, and optionally, Object Model Implementation. The declaration is used to specify the object model and needs to be defined according to the TENA Definition Language grammar rules. The object model definition is the software generated by the Object Model Compiler to ensure type safety and adherence to the object model definition by applications using the object model. The object model implementation contains the software used to implement the Local Class and Message constructors and operations that may exist in an object model.

Description

Associated with the object model definition is an object model implementation. An object model implementation represents the implementation software associated with the Local Class and Message constructors and operations (aka methods) that may be defined within the object model. If an object model does not define any Local Class or Message constructors or operations, then there will not be an object model implementation.

When an object model contains Local Classes or Messages that have a constructor or operation, an application is required to have all of the implementation code for those constructors and operations. The implementation code provides the behavior needed when the constructor or operation is invoked. Even if the application does not directly use the Local Class or Message, the implementation code is necessary because the middleware is unable to determine whether the type will be used or not.

Although multiple implementations are possible for the Local Class or Message implementations, it is typical that a common implementation is created and shared among the applications participating in a TENA execution. Implementations use a `getDescription` method that returns a string that is used to name the particular implementation. The Execution Manager can be configured to require that all applications joining the execution are using the same implementation code for all Local Classes and Messages (see [OM Consistency Checking documentation page](#) for more information).

Applications are required to register their Local Class and Message implementation code with the middleware. This ensures that implementation code is linked into the application and the middleware can access the implementation code when necessary. Details on implementation registration can be found in the [Local Class Implementation Registration](#) and [Message Implementation Registration](#) documentation pages.

Since it is common for object model implementations to be shared among many applications, users need to be able to package up their particular implementations for reuse. Information on this sharing can be found on the [Sharing OM Implementations documentation page](#).

Sharing Object Model Implementations

Sharing Object Model Implementations

Object Model Implementations provide shared code used by applications to initialize and use Local Classes or Messages defined by an object model (when Local Class or Message constructors or methods are defined). SDOs (Stateful Distributed Objects) with remote methods also require implementation code to implement remote methods, although the SDO implementation code is maintained with the application and not incorporated with shared Object Model Implementations. In the case of Local Classes and Messages with constructors or methods, all applications within an event will commonly need to share this common object model implementation code. As a result, a mechanism is needed to package and distribute the shared implementation code.

Description

Making an Object Model Implementation distribution

Many of the same steps to share and deploy TENA application code are appropriate for distributing implementation library code. See the discussion in [Distributing TENA Applications](#). This discussion will focus on distributing an implementation library for Local Class and Message constructors and methods, since these implementations are typically shared for applications participating in a TENA execution.

Creating a Shared Implementation Library for an Object Model

When an object model definition is downloaded from the TENA Repository, the distribution will include a variety of example source code. If the object model contains Local Classes or Messages with **constructors** or **methods**, an implementation is required. In this case, in addition to any other example code, a makefile or Visual Studio project defining a `SharedLocalImpl` will be included. This is automatically generated "stub" code that provides implementations for every constructor and/or method (related to a Local Class or Message) in the object model.

The `SharedLocalImpl` will show up in the name of the shared library and the directory used for installation, as well as special symbols used to ensure linkage. The generated `SharedLocalImpl` is provided as an example of the minimum code necessary to be able to run an application using the `SharedLocalImpl` "out-of-the-box."

Rename the Implementation

The TENA-SDA recommends renaming the the auto-generated `SharedLocalImpl` to a different, more descriptive name. The reason for this is that the generated example code provided by the TENA Repository always names the local methods implementation `SharedLocalImpl` and generally assigns a "1" as the version number. Unfortunately, if different users download an OM that needs a local methods implementation and each user writes their own implementation code, their applications will not necessarily interoperate. If the different implementations are all named `SharedLocalImpl`, it becomes nearly impossible to determine which one is being used be which applications.

Changing the name of the implementation requires finding all occurrences of `SharedLocalImpl` in the generated source example code, and replacing it with the desired name. This includes renaming any directories named `SharedLocalImpl`. Once done, the resulting source code and modified build files will generate an implementation library with a different name.

Modify the generated Local Methods Impl library

The generated `SharedLocalImpl` code can be edited to add the desired functionality, typically in the `LocalMethodsImpl.cpp` files. Using the Makefile or Visual Studio .sln file in the top directory of the example code will then build and install the library and required header files. Installation of the library and header files is necessary to copy these files to a directory location that is accessible to the other automatically generated example applications.

Change the description string in all the LocalMethodFactoryImpls.cpp

When a local method implementation is developed, the `getDescription` method on the `LocalMethodsFactoryImpl` needs to be modified to provide a descriptive string which should uniquely name the local method implementation. If the implementation library contains multiple local methods implementation for different object model types, the `getDescription` string should be changed to use the same descriptive name. This string is used by the Middleware to support the local class implementation consistency checking feature.

 Changing the return value from the `getDescription` methods is critical to the local method implementation consistency checking feature.

Remove all references to Arbitrary Values

The [Arbitrary Values](#) feature of the auto-generated code provides the capability for the code to compile and run immediately after installation. In order to properly implement a shared local methods implementation library, all Arbitrary Value references must be removed from the auto-generated example code.

The simplest method for finding all of these references is to remove the `TENA_MIDDLEWARE_ALLOW_ARBITRARY_VALUE` compiler directive from the the Visual Studio vcproj files and the Unix Makefiles.

 Removing the Arbitrary Value references from the implementation is critical to the proper operation of the local methods in an operational environment.

Making the distribution

Two scripts in the `myBuildFiles` directory define all the files that are installed for a `SharedLocalImpl`: `installLocalMethodsImpl.sh` and `installLocalMethodsImpl.bat`.

 The header files do not provide an interface to the implementation, since that is defined by the OM definition. A shared local implementation is registered simply by linking it into an application. The header files are needed on certain operating systems (e.g., Windows, OSX) to ensure that the linker does not optimize away the included libraries. See [Local Class](#) or [Message](#) documentation pages for additional information.

The important difference between distributing a library and an application is that when sharing libraries any necessary header files must be packaged in a defined structure to allow applications to use the libraries for development.

After building, the implementation code will have been installed into the directories defined by the installation scripts. These directories follow the standard directory structure associated with the TENA Middleware. Below, the files for the `Example-Tank` object model shared local implementation are shown:

```
6.0.1/bin/xp-vc90/libExample-Tank-v4-SharedLocalImpl-v3-xp-vc90-v6.0.1.dll  
lib/libExample-Tank-v4-SharedLocalImpl-v3-xp-vc90-v6.0.1.lib  
6.0.1/include/impls/Example-Tank-v4-SharedLocalImpl-v3/Example/AreaOfEffect/Impl.h  
6.0.1/include/impls/Example-Tank-v4-SharedLocalImpl-v3/Example/ComplexAreaOfEffect/Impl.h  
6.0.1/include/impls/Example-Tank-v4-SharedLocalImpl-v3/Example/Explosion/Impl.h  
6.0.1/include/impls/Example-Tank-v4-SharedLocalImpl-v3/Example/HomeStation/Impl.h  
6.0.1/include/impls/Example-Tank-v4-SharedLocalImpl-v3/libExample-Tank-v4-SharedLocalImpl-v3-Export.h
```

Making a `.zip` file of these files, and then extracting it under `%TENA_HOME%` on each target machine will allow development and application execution of applications using the `Example-Tank-v4` object model. On non-windows platforms, the library in `lib/` is used at both link time and run time, so there is no file under `bin/`.

Providing Distributions for Multiple Platforms.

Because the resulting libraries are specific to the operating systems and compilers used, the shared object model implementation code must be built on all the desired build environments, just as object model definitions are built. The steps to create any individual platform-specific distribution is the same.

These local method implementations can be uploaded to the TENA Repository and attached to the corresponding object model. If assistance is required in sharing local class implementations from the TENA Repository, submit a helpdesk case.

 **TENA-SDA will build user implemenaiton libraries across all supported platforms**

User Impls are a very good way to share common code/algorithms across the TENA community; however, very few TENA users have access to all of the computer and operating systems that the TENA Middleware is supported on. Therefore, the TENA-SDA will build user implementations across all TENA MW R6.0 computer platforms.

The basic process is

1. User submits a helpdesk case requesting cross-platform build support.
2. User is encouraged to make the OM private until the `UserImpl` is available.
3. User writes the initial local methods implementation by starting from the `SharedLocalImpl` auto-generated example code.
4. User provides source code (including build files and instructions) to the TENA-SDA Development Team.
5. TENA SDA Dev Team builds the `UserImpl` on all supported computer platforms.
6. TENA SDA Dev Team uploads all the built `UserImpl` packages to the TENA Repository.
7. User downloads the OM package with the `UserImpl` from the TENA Repository for verification and validation.
8. User makes the OM and `UserImpl` Public at their discretion.

Benefits when TENA SDA assists the User

- Provides a mechanism to build and share User Impls across all MW supported platforms.
- Ensures that other TENA Users who download the User's OM get a working implementation.
- Ensures that when the User's OM is imported into another OM, the resulting Package will contain the `UserImpl`.

Sharing Remote Method Implementation code

When an object model contains SDOs (Stateful Distributed Objects, referred to as Classes in the TENA Meta-Model) which have remote methods, an implementation is required for these remote methods. Unlike constructors and methods for Local Classes and Messages, the implementation for remote methods is only required by *an application publishing the particular type of SDO with the remote method*. As a result, in most cases, this implementation will be specific to a particular publishing application. The example code generated (for example, in Example-Tank-v4/Example-Soldier-Publisher-v3 will include the remote methods implementation as part of the application implementation.

If for some reason, the remote methods implementation needs to be shared, this implementation code and header files should be collected into an appropriately structured project, with build files to generate a library and install the header files. The files should be installed in a fashion similar to that for the shared local method implementation code. The exact details for projects that intend to use the resulting shared library will be dependent on the details of the remote methods implementation and the installation structure used.

Related Topics and Links

- [Distributing TENA Applications](#)

Example Object Models

Example Object Models

The TENA project uses several object models to illustrate the various constructs associated with the TENA Meta-Model and Middleware. These Example object models are used for documentation and training programs.

Description

The most basic Example object model is named `Example-Vehicle` and contains a simple Enumeration, Local Class, Class (i.e., Stateful Distributed Object, SDO), and Message. This object model is useful to illustrate the basic operations associated with the middleware.

An `Example-Tank` object model extends the basic `Example-Vehicle` object model with more complex model constructs. Some of the constructs in the `Example-Tank` object model inherit from the basic elements in the `Example-Vehicle` object model.

Users can utilize these example object models when it is needed to explore certain capabilities associated with the TENA Middleware. The object model definitions and example applications can be obtained by downloading the Object Model Packages from the repository ([Example-Vehicle-v1](#), [Example-Tank-v4](#)).

The TDL (TENA Definition Language) definitions and UML diagrams of these object models are shown below.

Example-Vehicle-v1 TDL File

```
// Simple example object model illustrating basic
// TENA meta-model constructs.

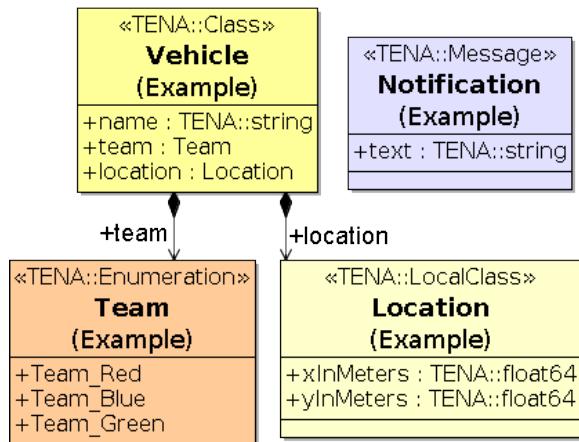
package Example
{
    enum Team
    {
        Team_Red,
        Team_Blue,
        Team_Green
    };

    local class Location
    {
        float64 xInMeters;
        float64 yInMeters;
    };

    class Vehicle
    {
        string name;
        Team team;
        Location location;
    };

    message Notification
    {
        string text;
    };
} // package Example
```

Example-Vehicle-v1 UML Diagram



Example-Tank-V4 TDL File

```
// This is an example object model illustrating
// TENA meta-model constructs.

import <Example-Vehicle-v1.tdl>

package Example
{
    local class HomeStation
    {
```

Example-Tank-v4 UML Diagram

```

// Ensure name is not the empty string
HomeStation( string name, Location location );

void set_name( in string name ); // Ensure
name is not the empty string
    float64 distanceFromLocationInMeters( in
Location location ) const;

    readonly string name;
    Location location;
};

class Soldier
{
    void moveTo( in Location destination );

    const string name;
    string rank;
    const uint32 serialNumber;
    optional const Team team;
    Location location;
    optional HomeStation homeStation;
};

class Tank;

class CommandPost
{
    Tank * requestTank( in Soldier * pSoldier, in
Location pickupLocation );

    const string name;
    const Team team;
    const Location location;
};

const float64 minimumRadiusOfEffectInMeters =
0.0;

local class AreaOfEffect
{
    AreaOfEffect( Location centerOfArea, float64
radiusInMeters );

    boolean isLocationAffected( in Location here )
const;

    readonly Location centerOfArea;
    readonly float64 radiusInMeters;
};

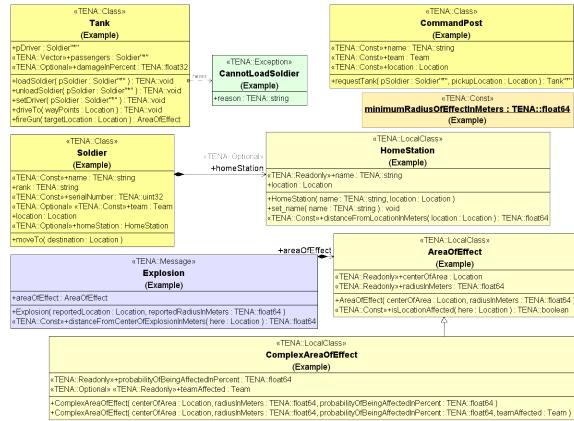
local class ComplexAreaOfEffect : extends
AreaOfEffect
{
    ComplexAreaOfEffect( Location centerOfArea,
        float64 radiusInMeters,
        float64
probabilityOfBeingAffectedInPercent );

    ComplexAreaOfEffect( Location centerOfArea,
        float64 radiusInMeters,
        float64
probabilityOfBeingAffectedInPercent,
        Team teamAffected );

    readonly float64
probabilityOfBeingAffectedInPercent;
    readonly optional Team teamAffected;
};

exception CannotLoadSoldier { string reason; };

```



```

class Tank : extends Vehicle
{
    void loadSoldier( in Soldier * pSoldier )
raises ( CannotLoadSoldier );
    void unloadSoldier( in Soldier * pSoldier );
    void setDriver( in Soldier * pSoldier );
    void driveTo( in vector < Location > wayPoints
);
    AreaOfEffect fireGun( in Location
targetLocation );

    Soldier * pDriver;
    vector < Soldier * > passengers;
    optional float32 damageInPercent;
};

message Explosion : extends Notification
{
    Explosion( string text, Location
reportedLocation, float64 reportedRadiusInMeters );

    float64 distanceFromCenterOfExplosionInMeters(
in Location here ) const;

    AreaOfEffect areaOfEffect;
};

}; // package Example

```

TENA Standard Object Models

TENA Standard Object Models

The TENA Standard Object Models (OM) provide a basic set of building blocks that have been found to be common across many DoD test and training ranges. The TENA Standard OM's have been successfully used in numerous events at many ranges, have been vetted through various TENA Community User Groups, and have been approved by the former TENA Architecture Management Team (AMT) or the current JMTC Configuration Review Board (JCRB). TENA applications are encouraged to maximize their use of the TENA Standard OM's in order to continually improve inter-range interoperability.

Description

The TENA Standard Object Models (OM) define common interfaces to systems and shared information that is used across the DoD ranges. Using auto-code generation, the interface specification is packaged into reusable software libraries that can be easily integrated into range applications. These OM's provide a solid set of building blocks that can be used to support various types of applications and user-defined object models. The TENA Standard OM's are listed in the table below and are the culmination of many years of work by people across the TENA user community to encode information that is commonly required across DoD test and training ranges. These OM's provide a common interface for TENA applications to communicate in an interoperable manor and play an integral part in TRMC's mission to improve interoperability across the ranges as well as reduce development and integration costs.

The TENA Standard OM's (like all OM's) are stored in the [TENA Repository](#), which provides access to the object model distributions, the TDL file, UML diagrams, and the usage documentation. In order to support the ability to export this entire user guide as a PDF document, the documentation portion of the repository information is contained in separate pages listed below. It is recommended that when viewing this documentation to use the repository links shown at the top of each page.

TENA Standard Object Models

- [TENA-AMO-v2](#)
- [TENA-Embedded-v3](#)
- [TENA-Engagement-v4](#)
- [TENA-Exercise-v1](#)
- [TENA-GPS-v3](#)
- [TENA-Munition-v3](#)
- [TENA-Platform-v4](#)
- [TENA-PlatformDetails-v4](#)
- [TENA-PlatformType-v2](#)
- [TENA-Radar-v3.1](#)
- [TENA-SRFserver-v2](#)
- [TENA-SyncController-v1](#)
- [TENA-Time-v2](#)
- [TENA-TSPI-v5](#)
- [TENA-UniqueID-v3](#)
- [TENA-Instrumentation-IFF-v1.0.0](#)
- [TENA-Instrumentation-LiftoffIndicator-v1.0.0](#)
- [TENA-Instrumentation-Optics-System-v1.0.0](#)
- [TENA-Instrumentation-PointableSystem-v1.0.0](#)
- [TENA-Instrumentation-Radar-SurveillanceSystem-v1.0.0](#)
- [TENA-Instrumentation-Radar-TrackingSystem-v1.0.0](#)
- [TENA-Instrumentation-SpectrumAnalyzer-v1.1.0](#)
- [TENA-Instrumentation-Telemetry-AntennaControlSystem-v1.0.0](#)
- [TENA-Instrumentation-Telemetry-Decommutator-v1.0.0](#)
- [TENA-Instrumentation-Telemetry-Receiver-v1.0.0](#)
- [TENA-Instrumentation-Track-Azimuth-v1.0.0](#)
- [TENA-Instrumentation-Track-AzimuthElevation-v1.0.0](#)
- [TENA-Instrumentation-Track-RadialVelocity-v1.0.0](#)
- [TENA-Instrumentation-Track-Range-v1.0.0](#)
- [TENA-Instrumentation-Track-RangeAzimuth-v1.0.0](#)
- [TENA-Instrumentation-Track-State3D-v1.0.0](#)
- [TENA-Instrumentation-Weather-Station-v1.0.0](#)
- [TENA-Hardware-System-v1.0.0](#)
- [TENA-exceptions-v1.0.0](#)
- [TENA-AudioVideo-StreamDescriptor-v1.0.0](#)
- [TENA-Geospatial-PointOfInterest-v1.0.0](#)
- [TENA-Geospatial-TSPcovariance-v1.0.0](#)
- [TENA-Hardware-System-v1.0.0](#)
- [TENA-WeatherServices-Parameters-v1.0.0](#)

Benefits of using TENA Standard OM's

OM designers and application developers are encouraged to maximize their use of the TENA Standard OMs. For designers, these OMs provide a large amount of pre-defined data that is ready-made and easy to articulate. Take for example an OM designer who has to describe and document how moving objects on the range need to report their position, velocity, and orientation at specific times. Prior to having a standard TENA-TSPI (Time-Space-Position-Information) OM, the designer had to define the characteristics of position, then define the characteristics of velocity and orientation in relation to that position, and finally describe what is meant by "time." Instead the OM designer can simply document that each moving range object must produce its TENA-TSPI value at specific times.

Application developers also benefit from using the TENA Standard OMs. Again take the TENA-TSPI and TENA-Time standard OMs as examples. Because the TENA-TSPI and TENA-Time OMs define various time, space, position information representations used across the ranges combined with the fact that both come with an already coded and tested standard implementations, developers are free to use any of the different representations for position, time, etc in the standard OMs that they prefer without having to worry about interoperability with other TENA applications using a different representation. As long as all TENA range applications, both on the developer's local range and any connected remote range, are TENA-TSPI and TENA-Time aware, the TENA Standard OM and standard implementations provide creation and conversions for the majority of the TSPI information used on ranges.

Improving Interoperability

Interoperability across ranges is a major driving force behind the efforts of the TENA project. The TENA Standard OMs provide common, reusable definitions for time, space, and position information; platform, entity, and track representations; and for associating sensors and weapons with platforms; as well as other range and simulation related objects. The TENA Standard [Object Models](#) form a *binding contract* between the producers of the information (publishers) and the consumers of the information (subscribers). These contracts are used to produce a set of software libraries called [Object Model Definitions](#) that the TENA Middleware uses to enforce the contract at program execution. Additionally, many of the TENA Standard OMs come with [Object Model Implementations](#) libraries that contain tested code to validate information and to perform common conversions between data formats.

Reducing costs

The code reuse aspect of the TENA Standard OMs is very important, especially in OMs that require [Object Model Implementations](#), such as the TENA-TSPI and the TENA-Time Standard OMs. The TENA-TSPI and TENA-Time Standard OMs each come with their own standard implementations that take care of enforcing that all of the time, position, and space local classes are correctly created. The standard implementation also provides the functionality to convert between the different types of time as well as between the various position and space types. This is code that has been written, maintained, and tested by the TENA-SDA and provided to the TENA community at no cost.

Orientation Conventions

Orientation Conventions

The top of the ball represents the North Pole. The "plus" mark on the ball, represents (Latitude = 0, Longitude = 0).

All axes use the same convention for showing which is X, Y, and Z axis.

- The X axis, has a red arrow at its tip.
- The Y axis has an orange rectangle at its tip.
- And the Z axis does not have any indicator.

The Yellow Frame is the airplane's Front-Right-Down (FRD) Body Frame. The Red Frame is the LTPenu Frame at the given location. The Silver Frames is the Geocentric Frame (which is shown to the side of the ball, since it can't be physically place it in the center of the grey ball).

For the airplane's Front-Right-Down (FRD) Body Frame:

- The X axis comes out of the nose
- The Y axis comes out of the right wing
- The Z axis is pointing down.

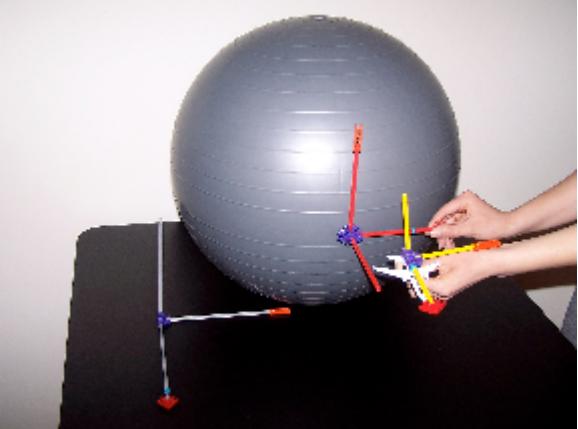
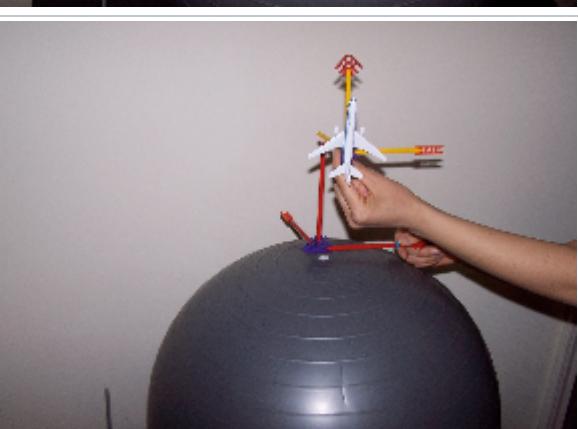
For the LTPenu Frames:

- The X axis points "Eastward"
- The Y axis points "Northward"
- The Z axis is perpendicular to the ground (technically to the geode) at that point (i.e., "up")

For the Geocentric frame, it's the well-known Geocentric convention

- The X axis comes out of the equator (at that point marked with a "plus" sign, which (Lat = 0, Long = 0)
- The Z axis goes up through the North Pole.

						TENA:: FRDwrtGeocentricBodyFixedZYX
Case	Picture (Click on the picture to enlarge it.)	Description	Lat, Long, Height	(Z,Y',X") Orientation of Red (LTPenu) Frame wrt Silver (Geocentric) Frame	(Z,Y',X") Orientation of Silver (Geocentric) Frame wrt Red (LTPenu) Frame	(Z,Y',X") Orientation of Yellow (Fi Frame wrt Silver (Geocentric) Fra
1		Aircraft on the equator with belly on the ground and nose pointing northward.	(0, 0, 0)	(Z=90, Y'=0, X'=90)	(Z=-90, Y'=-90, X'=0)	(Z=0, Y'=-90, X'=0)

2		Aircraft on the equator with belly northward nose pointing straight up (lifting off at the equator).	(0, 0, 0)	Same as Case 1	Same as Case 1	(0, 0, 0)
3		Aircraft on the North Pole with belly on the ground and nose pointing "northward" (along the travel direction on the prime meridian as you go from equator to North Pole).	(90, 0, 0)	(90, 0, 0)	(-90, 0, 0)	a) (0, 180, 0) or b) (180, 0, 180) or c) (-180, 0, 180)
4		Aircraft on the North Pole with belly "northward" (along the travel direction on the prime meridian as you go from equator to North Pole) and nose pointing straight up (lifting off at the North Pole).	(90, 0, 0)	Same as Case 3	Same as Case 3	(0, -90, 0)
5	No picture for this, but it is similar to Case 1, except the airplane is heading in the opposite direction of Case 1.	Aircraft on the equator with belly on the ground and nose pointing southward.	(0, 0, 0)	Same as Case 1	Same as Case 1	a) (180, 90, 0) or b) (0, 90, 180) or c) (0, 90,-180)

TENA Candidate OM Process

TENA Candidate OM Process

The process to update existing or create new TENA standard object models (OMs), also known as the Candidate OM Process, provides a mechanism for users to submit requests and justification for modifying existing TENA standard OMs or to propose new TENA standard OMs.

Description

The TENA community has the desire to minimize the frequency of changes to the TENA Standard OMs and also provide a controlled method for requesting, reviewing, and validating changes to the TENA Standard OMs. In order to support these objectives, the TENA-SDA has defined a Candidate Object Model process.

Objectives for Candidate OM Process

- Support the controlled modification of TENA Standard OMs for evaluating proposed changes prior to the AMT ([Architecture Management Team](#)) standardization process.
- Enable the testing and evaluation of OMs or OM changes proposed for standardization in a manner that will minimize the transition to the OMs if accepted as a standard.
- Provide an efficient mechanism for bundling reviewed and tested OMs or OM changes with their accompanying documentation for AMT approval.

Notifications of Proposed Changes

The TENA-SDA will notify the members of the TENA User Community about proposed changes to TENA Standard Object Models via the [TENA Object Model Helpdesk](#). Each proposed change will be tracked on an individual helpdesk case. Each Candidate OM will have an aggregating helpdesk case associated with it that has the individual proposed changes listed as sub-tasks. All members of the **OM-watchers** Group are added to the individual Candidate OM helpdesk cases. Watchers can remove themselves from individual cases to avoid email notification if they are not interested in following a specific OM case.



Join the OM-watchers Group

Users can request access to the OM-watchers group on the [OM Membership page](#). Follow the instructions and request membership

OM Candidate Control Concepts that have been Established

- A Candidate OM and its Standard OM will not be allowed to be used in the same TENA execution. This is necessary because of the interoperability problems that would occur with attempting to simultaneously use conflicting object model definitions.
- Candidate OMs are defined in the same namespace as their Standard OMs.
 - Using the same namespace ensures that [OM Type Consistency Checking](#) is enforced at run-time and eases community transition from the candidate to the final OM by only requiring testers to update their build files.
- Requested changes will be grouped into an aggregating OM Helpdesk case. Each change to a Standard OM will have a separate helpdesk case, and related changes will be defined as sub-tasks to the aggregating case.
 - Using separate helpdesk cases will provide the ability to track comments about each individual change separately and provide traceability from individual OM changes to the appropriate documentation.
- The aggregating helpdesk case number will be included in the Candidate OM's TDL file name.

Candidate OM Process Flow

Overview of Candidate OM Process

1. User opens a new OM Helpdesk case
2. TENA-SDA notifies TENA Community of a requested change
3. Community reviews and comments on the change via the helpdesk case
4. TENA-SDA and Requester consolidate comments
5. TENA-SDA produces a Candidate OM for community evaluation
6. Community evaluates the Candidate OM in applications
7. TENA-SDA collects Candidate OMs into a presentation for review at an AMT
8. New TENA Standard OMs are released to the Community

Details of Candidate OM Process

1. User opens a new OM Helpdesk case

A TENA user identifies a need to modify a TENA Standard object model and creates a new [OM helpdesk case](#). The User enters all of the required information in to the helpdesk case making special effort to explicitly identify the requested change. The user provides a clear description of the problem the requested change is intended to solve and what the expected benefit to the community would be. The user should also provide any additional information that they feel would be helpful to reviewers and testers of the requested change. This user will be referred to as the "Requester" in the subsequent text.

The TENA-SDA reviews the newly entered helpdesk case and ensures that the requested change is clearly articulated and that the motivation and background of the change is clearly expressed. If additional information or clarification is necessary, the TENA-SDA will work with the Requester to finalize the description of the requested change. If the requested change is extensive, the TENA-SDA may create a separate website wiki page to articulate the requested change.

Additionally, the TENA-SDA will review all open OM helpdesk changes related to the new helpdesk case to ensure that different user groups are not proposing similar or mutually exclusive changes. The TENA-SDA will begin identifying other related helpdesk cases to aggregate with the new request into a Candidate OM. The TENA-SDA may choose to create the aggregating helpdesk case at this point if previously submitted helpdesk cases relate to the new request.

2. TENA-SDA notifies TENA Community of a requested change

Working with the Requester, the TENA-SDA determines a suitable initial review period for the requested change. The TENA-SDA will attempt to ensure that the user community is provided an adequate review period based on the magnitude of the proposed change. If necessary, meetings will be scheduled to discuss the proposed change and any testing/evaluation performed with the Candidate OM.

In order to notify the TENA community of the requested change, the TENA-SDA will add all members of the OM-watchers group as watchers to the helpdesk case. The helpdesk case watchers will automatically receive email notifications whenever the helpdesk case is modified or a comment has been added to the case.



TENA-SDA should validate the TDL

Visible to TENA-team only

The TENA-SDA should create a private version of the TENA Standard OM that includes the requested changes (and any changes that are being considered for aggregation). This private copy of the OM should be validated prior to notifying the community of the requested change. The private version of the OM should not be made public via the TENA Repository, but could be attached to the helpdesk case, if appropriate.

3. Community reviews and comments on the change via the helpdesk case

The initial review period begins with the sending of the notification emails from the helpdesk system. All members of the OM-watchers group are encouraged to review the proposed changes and comment direct in the helpdesk case on the proposed changes (or new OM). The review comments should include statements about how the change would affect their user OMs and their applications. Reviewers can respond with a simply comment indicating that they agree with the proposal if appropriate. Reviewers may also submit questions to clarify either the change or their understanding of the change.

The TENA-SDA requests that the Reviewers not attempt to add their own changes to the Requester's case via comments to the case. If this situation arises, Reviewers should submit their own helpdesk case and request to be included in the aggregated case.



Silence is agreement!

OM-watchers group members who do not comment on the newly submitted helpdesk case during the initial review period will be assumed to agree with the proposed change.

The TENA-SDA may host teleconferences or face-to-face meetings in order to streamline and coordinate the changes. The announcement and agendas for these teleconferences/meetings will be published via a helpdesk comment from a TENA-SDA team member.



Visible to TENA-team only

During the initial review period, the TENA-SDA can upload a modified TDL file to the TENA Repository. This will ensure that the initially proposed changes pass validation.

- The OM is initially marked **Private** and **Candidate**
- The Candidate OM TDL file name has the helpdesk case number without the '-' embedded in it. The filename will be <group>-<name>-v<case number>-#.t#l (e.g., TENA-Radar-vOM813-3.1.tdl)

4. TENA-SDA and Requester consolidate comments

At the end of the initial review period, the TENA-SDA will work with the Requester to review the comments received from the community and determine the next course of action.

Post-Initial Review Courses of Action

- **Move forward** – Add the helpdesk case to an aggregating case. Continue with [TENA-SDA produces a Candidate OM](#).
- **Rework** – Modify the details of the proposed change and repeat the community review period. Go back to [TENA-SDA notifies TENA Community of a requested change](#).
- **Stop** – Determine that the proposed change will not be acceptable for the user community and resolve the case as Won't Fix.

The TENA-SDA or the Requester will post the decision about which course is being followed to the helpdesk case.

5. TENA-SDA produces a Candidate OM for community evaluation

The TENA-SDA reviews the current Candidate OM cases that have not yet been finalized to determine if the new requested change can be added to it. Otherwise, a new aggregating helpdesk case is created. The new requested change and any additional changes on other helpdesk cases that have been identified as related are made sub-tasks of the aggregating helpdesk case. All of the members of the OM-watchers group will be made watchers on the aggregating helpdesk case.

OM-813 Initial Candidate OM Aggregating Helpdesk Case

As an example, the TENA-Radar-OM813-v3.1 Candidate OM was created via helpdesk case \$paramkey, which aggregates the following two requests TENA-Radar-v3 OM changes

- \$paramkey – Target Data trackID needs to be optional
- \$paramkey – Add enumeration to TENA::Radar's Beam local class

Since the TENA-Radar-v3 object model required a local methods implementation, TENA-Radar-v3-checkValues, the TENA-Radar-OM813-v3.1 Candidate OM required an OM implementation as well, so TENA-Radar-OM813-v3.1-checkValues standard implementation was created.

The amount of time that an aggregating Helpdesk case remains opened but not finalized varies depending on several factors, including but not limited to:

- the period of time that the Requester needs the Candidate OM,
- the number of related cases that have been identified but are still being reviewed by the community,
- the date of the cut-off for the next annual submission to the AMT for approval.

Once the TENA-SDA has decided to finalize a set of changes on an aggregating helpdesk case, the TENA-SDA will create the Candidate OM and upload it to the TENA Repository. The Candidate OM will have a comment block at the top of the file stating that this is not a current TENA Standard OM, but is in fact a Candidate OM. The comment block will also make reference to all of the related helpdesk cases that are included. An example comment header is shown below:



Example comment block at the top of a Candidate OM

```
// TENA-Radar-OM813-v3.1                                     -*- IDL -*-  
  
/*!!!!!!!!!!!!!! -- WARNING -- !!!!!!!!!!!!!!!*/  
*  
* This is NOT the current TENA-Radar Standard Object Model. This is a  
* candidate OM that is being evaluated for possible inclusion in a future  
* release of the TENA-Radar Standard OM. The current TENA-Radar Standard  
* OM for use with Release 6.0 is TENA-Radar-v3.  
*  
* Details and Background on this candidate OM can be found on helpdesk case  
* OM-813 at https://www.tena-sda.org/helpdesk/browse/OM-813, which is an  
* aggregating case for the user-submitted helpdesk cases shown below.  
*  
* Changes from the TENA-Radar-v3 Standard OM:  
* OM-804 -- Target Data trackID needs to be optional  
*     * Removed the TargetData( ... ) constructor  
*     * Removed the set_trackID( ... ) local method  
*     * Removed the readonly keyword from the trackID attribute  
*  
* OM-810 -- Add enumeration to TENA::Radar's Beam local class  
*     * Added new TransitionBeamFunction enumeration  
*     * Added a uint16 beamID to the Beam local class  
*     * Added a transmitBeamFunction attribute to the TransmitBeam local class  
*     * Added an optional beamID attribute to the TargetData local class  
*  
*!!!!!!!!!!!!!! -- WARNING -- !!!!!!!!!!!!!!!*/  
* . . .
```

The Candidate will remain available in the TENA Repository until it is archived.



Visible to TENA-team only

Once the modified TDL file is uploaded to the TENA Repository and successfully built for several platforms the OM should be changed from **Private** to **Public**, but it should remain a **Candidate OM**.

6. Community evaluates the Candidate OM in applications

Interested community users are free to download the Candidate OM package or to import it into their user OMs to evaluate the changes in their applications. TENA community members import the Candidate OM using the `import <OM-Name-v#.tdl>` statement in updated copies of their OMs. The import statement must include the exact name and version of the Candidate OM.



Example import statement using a Candidate OM

```
...  
import <TENA-Radar-OM813-v3.1.tdl>  
import <TENA-SRFserver-v2.tdl>  
...
```

TENA-SDA requests that users of the Candidate OM post comments to the relevant helpdesk cases detailing their experience using the Candidate OM

7. TENA-SDA collects Candidate OMs into a presentation for review at an AMT

Annually, nominally in the Spring, the TENA-SDA will review all Candidate OMs that have been available to the community for at least two months to evaluate them for incorporation into their respective Standard OM. The TENA-SDA will provide a recommendation for each Candidate OM by posting a comment on the aggregating cases. The recommendation will be one of the following:

Annual Review Recommendation Options

- **Standardize** – the Candidate OM is recommended by the TENA-SDA for inclusion in the upcoming TENA Standard OM release.
- **Continue** – the Candidate OM is recommended by the TENA-SDA to continue in the candidate process for additional community evaluation.
- **Terminate** – the Candidate OM is recommended by the TENA-SDA to be terminated and perform no further investigation because it does not properly support the needs of the user community.

The TENA-SDA will include rational for the recommendation on each Candidate OM. The OM-watchers are encouraged to also post comments to these cases stating whether they agree or disagree with the TENA-SDA recommendations for each Candidate OM. If an OM_watcher disagrees, they need to provide rational for their position as part of their comment on the Candidate OM helpdesk case. The TENA-SDA and the OM-watchers will strive for a consensus recommendation.



Silence is agreement!

OM-watchers group members who do not provide a recommendation on the cases at this point in the process are assumed to agree with the TENA-SDA recommendation.

The TENA-SDA presents the status of all Candidate OMs at the the AMT for consideration. Candidate OMs that are being recommended for incorporation into the TENA Standard OMs will be presented with either the consensus recommendation or with the competing recommendations. Additionally, the TENA-SDA will provide a list of Candidate OMs that are being recommended for termination and for continuation into the following year.

8. New TENA Standard OMs are released to the Community

After the Candidate OMs have been presented at the AMT, the TENA-SDA will carry out the decision made by the AMT for each Candidate OM and nominally in the Summer timeframe, the updated TENA Standard OMs are released for use by the TENA user community. The TENA-SDA will upload the new revisions of the TENA Standard OMs to the TENA Repository and make them public. Only Standard OMs that have AMT approved changes will be uploaded to the TENA Repository. The user community is encouraged to update their applications to use the latest TENA Standard OMs as soon as is possible.

Programmers Guide 6.0.1 Addendum

TENA Middleware Programmer's Guide 6.0.1 Addendum

 A common version of the TENA Middleware Programmer's Guide is used for all of the minor 6.0.x middleware releases. The changes to the usage and operation of the 6.0.1 version of the TENA Middleware from the previous version are shown below (and also highlighted in the guide pages as well).

- [Programmers Guide 6.0](#)

There are no Programmer's Guide changes for the 6.0.1 release

Programmers Guide 6.0.2 Addendum

TENA Middleware Programmer's Guide 6.0.2 Addendum

i A common version of the TENA Middleware Programmer's Guide is used for all of the minor 6.0.x middleware releases. The changes to the usage and operation of the 6.0.2 version of the TENA Middleware from the previous version are shown below (and also highlighted in the guide pages as well).

- [Programmers Guide 6.0](#)

Middleware 6.0.2 Changes Relevant to the Programmer's Guide

Alternative Subscription Related Methods

As referenced in [\\$paramkey](#), several alternative API functions were introduced to reduce the risk of a Session object being invalidated (by another thread) while being used within these methods. The alternative functions use a `SessionPtr` as an argument, versus using a reference to a `Session` object. The `SessionPtr` is a smart pointer that has a mechanism for protecting the underlying object from being invalidated while it is being used.

Existing applications should be modified to use the alternative API calls to reduce the risk of this Session invalidation problem. The API changes are identified through code examples below, where the `pSession` variable is a `TENA::Middleware::SessionPtr`.

Method Type	Old Method Usage	Alternative Method Usage
SDO Type Subscribe	<code>Example::Vehicle::subscribe(*pSession,</code> <code>pExampleVehicleSubscription, false);</code>	<code>Example::Vehicle::subscribe(pSession,</code> <code>pExampleVehicleSubscription, false);</code>
SDO Type Unsubscribe	<code>Example::Vehicle::unsubscribe(*pSession);</code>	<code>Example::Vehicle::unsubscribe(*pSession);</code>
SDO Subscription Change	<code>Example::Vehicle::changeSubscription(*pSession,</code> <code>pExampleVehicleSubscription, false, TENA::Middleware::Filter(1),</code> <code>TENA::Middleware::Filter(2));</code>	<code>Example::Vehicle::changeSubscription(pSession,</code> <code>pExampleVehicleSubscription, false, TENA::Middleware::Filter(1),</code> <code>TENA::Middleware::Filter(2));</code>
Message Type Subscribe	<code>Example::Notification::subscribe(*pSession,</code> <code>pExampleNotificationSubscription, false);</code>	<code>Example::Notification::subscribe(pSession,</code> <code>pExampleNotificationSubscription, false);</code>
Message Type Unsubscribe	<code>Example::Notification::unsubscribe(*pSession);</code>	<code>Example::Notification::unsubscribe(pSession);</code>
SDO Instance Subscription	<code>Example::Soldier::ProxyPtr pSoldierProxy</code> <code>= pSoldierSDOp pointer->subscribe(*pSession,</code> <code>pExampleSoldierSubscription);</code>	<code>Example::Soldier::ProxyPtr pSoldierProxy</code> <code>= pSoldierSDOp pointer->subscribe(pSession,</code> <code>pExampleSoldierSubscription);</code>
SDO Pointer Pull PublicationState	<code>Example::Soldier::PublicationStatePtr pState</code> <code>= pSoldierSDOp pointer->pullPublicationState(*pSession);</code>	<code>Example::Soldier::PublicationStatePtr pState</code> <code>= pSoldierSDOp pointer->pullPublicationState(pSession);</code>
Create ServantFactory	<code>Example::Vehicle::ServantFactory::create(*pSession);</code>	<code>Example::Vehicle::ServantFactory::create(pSession);</code>
Create MessageSender	<code>Example::Notification::MessageSender::create(*pSession);</code>	<code>Example::Notification::MessageSender::create(pSession);</code>

Configuration Parameter `requestPublisherAck` Default Value

The default value for the configuration parameter `requestPublisherAck` was changed to "0 (false)" for the 6.0.2 release of the middleware. This change is described in [MW-4199](#).

Programmers Guide 6.0.3 Addendum

TENA Middleware Programmer's Guide 6.0.3 Addendum

i A common version of the TENA Middleware Programmer's Guide is used for all of the minor 6.0.x middleware releases. The changes to the usage and operation of the 6.0.3 version of the TENA Middleware from the previous version are shown below (and also highlighted in the guide pages as well).

- [Programmers Guide 6.0](#)

Middleware 6.0.3 Changes Relevant to the Programmer's Guide

"Best Match" Subscriptions

Release 6.0.3 of the middleware provided the ability to subscribe to multiple types in a class hierarchy, but only discover a single proxy corresponding to the most derived type (or receive only the most derived Message). Without the "Best Match" subscription option, a subscribing application would receive multiple proxies (or messages) which can cause processing complexity. This capability is fully defined in the [Best Match](#) section of the Programmer's Guide.

User-Defined Time Source

Release 6.0.3 of the middleware enables application developers to provide a user-defined time source to be used by the middleware. By default, the middleware uses the system clock associated with the operating system for timestamps associated with diagnostic logging and meta-data information. If applications have access to an external clock source (e.g., IRIG, GPS), they can register that time source with the middleware to be used instead of the system clock. This mechanism is fully defined in the [Application-Specific Clock](#) section of the Programmer's Guide.

API Additions

Several minor additions were made to the release 6.0.3 middleware API (in addition to the items identified above) that do not affect existing applications, but can be used if necessary. These additions are captured in the table below.

API Addition	Description	Helpdesk Case
TENA::Middleware::MulticastProperty.h	Header file provides access to algorithm used to map (a reverse map) an object model type, and optional advanced filtering tag, to the particular multicast address used for "best effort" communication of that data type.	MW-4503
TENA:::Middleware:::Execution:::Metadata::: getExecutionManagerEndpointsV ector()	Provides ability to obtain the vector of endpoints used by the Execution Manager.	MW-4075
TENA:::Middleware:::SDO::: MetadataBase::: getPublisherEndpoint()	Provides ability to obtain the endpoint of the publisher associated with a particular SDO Proxy.	MW-4075
Configuration parameter <code>popupWindow sDisabled</code>	Configuration parameter disables the Widows error popup mechanism which can interfere with the operation of a distributed system. By default, Windows error popups are disabled for "release" builds and enabled for "debug" builds.	MW-4438

Deprecated Names

In order to maintain consistent and understandable naming of various middleware API elements, it is sometimes necessary to rename certain API elements. Release 6.0.3 of the middleware has made the name changes listed in the table below. The old API names are still supported, but their use will result in a deprecation warning message at compile time. The effect of this deprecation macro can be disabled by defining `TENA_MIDDLEWARE_DISABLE_WARN_DEPRECATED` in the compiler settings.

API Deprecation	Helpdesk Case
<code>TENA:::Middleware:::dynamicCast</code> deprecated, use <code>TENA:::Middleware:::Utils:::dynamicCast</code>	MW-4468
<code>corbaTransientRetries</code> deprecated, use <code>transientCommunicationAttempts</code>	MW-4435

announceTransientRetries deprecated, use announceTransientAttempts	MW-4435
TENA::Middleware::SDO::PointerImplBase deprecated, use TENA::Middleware::SDO::SDOpointerBase	MW-4376
TENA::Middleware::SDO::SDOpointer deprecated, use TENA::Middleware::SDO::SDOpointerPtr	MW-4376
TENA::Middleware::SDO::MetadataBase::isTerminated() deprecated, use TENA::Middleware::SDO::MetadataBase::isRemoved()	MW-4265
TENA::Middleware::SDO::ProxyBase::terminated() deprecated, use TENA::Middleware::SDO::ProxyBase::isRemoved()	MW-3712

Programmers Guide 6.0.4 Addendum

TENA Middleware Programmer's Guide 6.0.4 Addendum

i A common version of the TENA Middleware Programmer's Guide is used for all of the minor 6.0.x middleware releases. The changes to the usage and operation of the 6.0.4 version of the TENA Middleware from the previous version are shown below (and also highlighted in the Programmer's Guide pages as well).

- [Programmers Guide 6.0](#)

Middleware 6.0.4 Changes Relevant to the Programmer's Guide

Versioned Namespaces ([MW-5076](#), [MW-4776](#), [MW-4478](#), [MW-4883](#))

In order to support multiple versions of the middleware, object models, and third-party software libraries used by the middleware, release 6.0.4 encapsulated all of this code using versioned C++ namespaces. For example, the fully qualified name for the Runtime class is `TENA::Middleware::MW604::Runtime`. All of the header files associated with the middleware have been designed to utilize the C++ "using" mechanism to make the introduction of these versioned namespaces transparent with respect to the middleware API, except in the situation where an application is required to utilize multiple versions.

However, there are a couple of use cases in which this versioned namespace change may cause compiler errors with existing application code. These issues are related to the use of the ostream inserter operator, `operator<<(std::ostream & os, ...)`, such as those used with the auto-generated example applications to print out attribute values. These error conditions are captured in the following helpdesk cases:

- [MW-5213](#) – Compile error for static ostream operator.
- [MW-5231](#) – Ambiguous overload for ostream inserter operator << (UNIX).
- [MW-5229](#) – Ambiguous overload for ostream inserter operator << (Windows)

The multiple version support provided with release 6.0.4 of the middleware has several limitations that are expected to be addressed in the next middleware release. Support for multiple versions of the middleware and third-party products must be done with separate compilation units. Additional documentation related to supporting multiple versions within the same application can be found in the [Multiple Version Support](#) documentation page.

Join Options ([MW-4776](#), [MW-5256](#))

An alternative `TENA::Middleware::Runtime::joinExecution()` method was added to the 6.0.4 middleware API that added an argument of type `TE_NA::Middleware::ExecutionJoinOptions`. The existing `joinExecution()` method will continue to function normally although the new method will need to be used if the application is attempting to use multiple versions of the same object model in the same process space. In this situation, the application must explicitly identify which object models will be used by which execution. Additional information on this capability is provided in the Multiple Version Support for Object Models section of the [Multiple Version Support](#) documentation page.

In addition to providing support for multiple object models, the new `joinExecution` method allows the application to specify user-defined siteID and applicationID values that will be integrated into the middleware meta-data and available to [TENA Console](#) operators. The concept of user-defined site and application identifiers is embedded within the [TENA-UniqueID](#) object model, but there has not been sufficient management support to assist with ensuring uniqueness of these identifiers. Adding these identifiers to the `joinExecution` method is only the first step in having the middleware assist with the management of this concept that is likely to affect future versions of the [TENA-UniqueID](#) object model. Information related to these user-defined site and application identifiers (i.e., `userSiteID`, `userApplicationID`) are discussed in the Join Execution Options section of the [Execution Management Services](#) documentation page.

API Additions

Several minor additions were made to the release 6.0.4 middleware API (in addition to the items identified above) that do not affect existing applications, but can be used if necessary. These additions are captured in the table below.

API Addition	Description	Helpdesk Case
<code>OM_SDO_TYPE::Subscription::removeAllSDOs()</code>	Convenience method to allow application to remove all discovered SDOs associated with a Subscription object used as part of a subscribe call. This API addition exists for all SDO types.	MW-5202
<code>OM_SDO_TYPE::PublicationStateInitializer::unset_OPTIONAL_ATTRIBUTE</code>	Provides ability to unset a previously set optional attribute on the SDO Initializer class. This API addition exists for all optional attributes belonging to an SDO type.	MW-4763
<code>Middleware ID classes (e.g., RuntimeID, ExecutionID, SessionID) hash_value() function</code>	All of the middleware ID classes added a related <code>hash_value()</code> function to support their use in unordered containers.	MW-4710

TENA::Middleware::Configuration and TENA::Middleware::ApplicationConfiguration constructor and parse method argument for configFileRequired with default value	Constructors and parse methods added boolean "configFileRequired" parameter, which defaults to true. The addition of this parameter is to support the ability to define a configuration file name, but allow the file to be optional (i.e., may not exist).	MW-4444
TENA::::Middleware::ExecutionAlreadyJoined Exception addition for ExecutionPtr member	The ExecutionAlreadyJoined exception that may be thrown during the joinExecution() invocation now includes the pointer to the execution.	MW-4676
TENA::::Middleware::ObjectModelTypeInconsistencyError Exception added	Added an exception for when an application attempts to use an OM type that was not explicitly defined to be used during the joinExecution operation.	MW-3443
TENA::::Middleware::Session::getActiveSubscriptionTypes()	Added a method on the Session class to provide a list of all active type-based subscriptions associated with the Session.	MW-3709
TENA::::Middleware::TypeRegistry::TypeInfo	Added std::string description attribute to TypeInfo struct.	MW-4702
TENA::::Middleware::BasicConfiguration::resetSettings(), TENA::::Middleware::Setting::reset()	Added methods to reset configuration setting for "flag" type settings.	MW-4775
TENA::::Middleware::Tag, and other aliases for FilteringContext	Added an alias for FilteringContext, called Tag in several places to assist with understanding the Advanced Filtering concepts.	MW-4809

API Deprecated Items

In order to maintain consistent and understandable naming of various middleware API elements, it is sometimes necessary to rename certain API elements. Release 6.0.4 of the middleware has made the name changes listed in the table below. The old API names are still supported, but their use will result in a deprecation warning message at compile time. The effect of this deprecation macro can be disabled by defining `TENA_MIDDLEWARE_DISABLE_WARN_DEPRECATED` in the compiler settings.

API Deprecation	Helpdesk Case
<code>CreateServant(..., SmartPtr<RemoteMethodsImpl>, ...)</code> is deprecated, use <code>CreateServant(..., std::ptr<RemoteMethodsImpl>, ...)</code>	MW-5295
<code>TENA::::Middleware::ProxyView::ProxyBase::getSession()</code> was removed, use <code>TENA::::Middleware::ProxyView::ProxyBase::getSessionPtr()</code>	MW-2096

Programmers Guide 6.0.5 Addendum

TENA Middleware Programmer's Guide 6.0.5 Addendum

i A common version of the TENA Middleware Programmer's Guide is used for all of the minor 6.0.x middleware releases. The changes to the usage and operation of the 6.0.5 version of the TENA Middleware from the previous version are shown below (and also highlighted in the Programmer's Guide pages as well).

- [Programmers Guide 6.0](#)

Middleware 6.0.5 Changes Relevant to the Programmer's Guide

Versioned Namespaces ([MW-5328 - Multiple Version Support Limitations](#) RESOLVED)

In order to support multiple versions of the middleware, object models, and third-party software libraries used by the middleware, release 6.0.5 encapsulated all of this code using versioned C++ namespaces. For example, the fully qualified name for the Runtime class is `TENA::Middleware::MW605::Runtime`. All of the header files associated with the middleware have been designed to utilize the C++ "using" mechanism to make the introduction of these versioned namespaces transparent with respect to the middleware API, except in the situation where an application is required to utilize multiple versions. Users should not use the middleware versioned namespace in their application code unless they are attempting to support multiple middleware versions.

API Changes

API Change	Description	Helpdesk Case
<code>std::string const TENA::Middleware::Runtime::getMiddlewareVersion() const</code>	Return value for <code>TENA::Middleware::Runtime::getMiddlewareVersion</code> changed from <code>"std::string"</code> to <code>"std::string const"</code> .	 MW-3706 - Inconsistent return values on methods of <code>TENA::Middleware::Runtime::Metadata</code> RESOLVED
<code>std::string const TENA::Middleware::Runtime::getApplicationEndpointsString() const</code>	Return value for <code>TENA::Middleware::Runtime::getApplicationEndpointsString</code> changed from <code>"std::string &"</code> to <code>"std::string const"</code> .	 MW-3706 - Inconsistent return values on methods of <code>TENA::Middleware::Runtime::Metadata</code> RESOLVED
<code>size_t evokeCallback(TENA::uint32 waitTimeInMicroseconds);</code>	The parameter name to <code>evokeCallback</code> was renamed to be called <code>waitTimeInMicroseconds</code> .	 MW-5553 - parameter to <code>evokeCallback</code> and <code>evokeMultipleCallbacks</code> should be called <code>waitTimeInMicroseconds</code> RESOLVED
<code>size_t evokeMultipleCallback(TENA::uint32 waitTimeInMicroseconds);</code>	The parameter name to <code>evokeMultipleCallbacks</code> was renamed to be called <code>waitTimeInMicroseconds</code> .	 MW-5553 - parameter to <code>evokeCallback</code> and <code>evokeMultipleCallbacks</code> should be called <code>waitTimeInMicroseconds</code> RESOLVED

API Additions

Several minor additions were made to the release 6.0.5 middleware API (in addition to the items identified above) that do not affect existing applications, but can be used if necessary. These additions are captured in the table below.

API Addition	Description	Helpdesk Case

<pre>void TENA:: : Middleware :: Execution: : isEMjoined() TENA: : Middl eware :: Utils :: Retur nValu eInfo & re turnV alueI nfo) const</pre>	<p>Provides ability to check if the Execution Manager (EM) indicates if the application is joined to the execution. The method takes a reference to a ReturnValueInfo object that indicates three possible responses: Yes, No, or Maybe. The Maybe response is necessary when the application's middleware encounters a transient communication fault with the EM. This method is related to the TENA::Middleware::Runtime::isEMreachable() method.</p> <p>⚠ – These methods should be considered experimental and are subject to change in a future version of the middleware. It is anticipated that a callback mechanism will replace these polling methods to inform an application when an EM is reachable or when an application has lost communication with the EM (or EMs) associated with the joined execution.</p>	<p>+ MW-5258 - Add API method for application to check if connected to execution RESOLVED</p>
<p>Numerous methods were added to the TENA::Middleware::Configuration class, such as:</p> <pre>void setLi stenE ndpoi nts(s td::v ector <TENA ::Mid dlewa re::U tills:: MW605 ::End point >); std:: string g get Multi castI nterf ace();</pre>	<p>Provides explicit methods to get/set Middleware configuration settings on the TENA::Middleware::Configuration class.</p>	<p>+ MW-5336 - Provide Explicit Methods to get/set Middleware Configuration Setting on TENA::Middleware::Configuration RESOLVED</p>

<pre>#define TENA_EX_NO_TE(EXECUTION_ID) #define TENA_EX_WARING(EXECUTION_ID) #define TENA_EX_EROR(EXECUTION_ID)</pre>	<p>TENA Alert macros are not scoped to a particular Execution when the application joins more than one Execution. Additional macros were defined to limit the Alert messages to only a particular Execution.</p>	<p> MW-5396 - Alerts need to be scoped to particular execution RESOLVED</p>
--	--	---

API Deprecated Items

In order to maintain consistent and understandable naming of various middleware API elements, it is sometimes necessary to rename certain API elements. Release 6.0.4 of the middleware has made the name changes listed in the table below. The old API names are still supported, but their use will result in a deprecation warning message at compile time. The effect of this deprecation macro can be disabled by defining `TENA_MIDDLEWARE_DISABLE_WARN_DEPRECATED` in the compiler settings. Note that some of these deprecated items were actually deprecated in previous 6.0.x releases of the middleware, but for version 6.0.5 the compiler will now issue a warning.

API Deprecations	Description	Helpdesk Case
<code>boost::lexical_cast</code> , <code>boost::noncopyable</code>	<p>The use of the third-party Boost Library constructs was eliminated from the Middleware API to allow users to utilize a different version than is being used by the Middleware.</p> <p>Alternative to <code>boost::lexical_cast</code>:</p> <p><code>boost::lexical_cast</code> has a lot of capability, but it is frequently overkill for simple cases. An alternative approach is provided by <code>std::to_string()</code>. Compilers that support c++11 provide <code>std::to_string()</code>. For those compilers that do not support this part of the c++11 standard, TENA provides implementations in <code>TENA/types.h</code>.</p> <p>Alternative to <code>boost::noncopyable</code>:</p> <p>TENA provides a simple alternative to <code>boost::noncopyable</code> with <code>TENA::Middleware::Utils::noncopyable</code> defined in <code>TENA/Middleware/Utils/noncopyable.h</code>.</p>	<p> MW-5490 - Eliminate boost from our (public) API RESOLVED</p>
<code>TENA::Middleware::Utils::AtomicOp</code>	<p>The <code>AtomicOp</code> class was removed from the Middleware API. Suggested alternatives to using <code>TENA::Middleware::Utils::AtomicOp</code> are:</p> <ol style="list-style-type: none"> 1. For compilers that support the c++11 standard, atomic functionality is provided by <code>std::atomic</code>. 2. gcc44 does not fully support c++11, but it does provide <code>std::atomic</code> by including the header file <code>stdatomic.h</code>. 3. Visual Studio 2010 does not support <code>std::atomic</code>. When using VS 2010 the recommended approach is to use Windows Interlocked API described at https://msdn.microsoft.com/en-us/library/windows/desktop/ms684122(v=vs.85).aspx 	<p> MW-5615 - TENA /Middleware/Utils /AtomicOp.h removed from MW API RESOLVED</p>
<code>File TENA/Middleware/SDO/SDOpointer.h</code> is deprecated, <code>File TENA/Middleware/SDO/SDOpointerPtr.h</code> <code>use h</code>	<p>The header file <code>SDOpointer.h</code> was renamed to <code>SDOpointerPtr.h</code> to be consistent with similar files.</p>	<p> MW-5949 - Mark Deprecated API Functions as Deprecated with <code>TENA_Middleware_DEPRECATED()</code> macro RESOLVED</p>

<pre>TENA::Middleware::getTENAmiddlewareVersion() is deprecated, use TENA::Middleware:: getTENAmiddlewareVersionStruct()</pre>	<p>The getMiddlewareVersion method was renamed to support returning a struct.</p>	+ MW-5949 - Mark Deprecated API Functions as Deprecated with TENA_Middleware_D EPRECATED() macro RESOLVED
<pre>TENA::Middleware::ProxyView::ProxyBase:: terminated() is deprecated, use TENA::Middleware::ProxyView::ProxyBase:: isRemoved()</pre>	<p>The method named terminated was renamed to isRemoved.</p>	+ MW-5949 - Mark Deprecated API Functions as Deprecated with TENA_Middleware_D EPRECATED() macro RESOLVED
<pre>TENA::MiddlewareSDO::PointerImplBase is deprecated, use TENA::Middleware::SDOpointerBase</pre>	<p>The type PointerImplBase was renamed to SDOpointerBase.</p>	+ MW-5949 - Mark Deprecated API Functions as Deprecated with TENA_Middleware_D EPRECATED() macro RESOLVED
<pre>TENA::Middleware::SDO::SDOpointer< POINTERIMPL > i s deprecated, use TENA::Middleware::SDO:: SDOpointerPtr</pre>	<p>The SDOpointer template was changed to SDOpointerPtr.</p>	+ MW-5949 - Mark Deprecated API Functions as Deprecated with TENA_Middleware_D EPRECATED() macro RESOLVED
<pre>TENA::Middleware::Utils::toTimestamp(TENA:: int64 const &timeInNanosecondsSince1970, TENA:: uint8 displayedResolution, bool useLocalTimeZone) is deprecated, use TENA::Middleware::Utils::toTimestamp(TENA:: int64 const &timeInNanosecondsSince1970, TENA:: uint8 displayedResolution)</pre>	<p>The toTimestamp function that takes three arguments was removed, as it is supported by the two argument function.</p>	+ MW-5949 - Mark Deprecated API Functions as Deprecated with TENA_Middleware_D EPRECATED() macro RESOLVED
<pre>OM_TYPE::MessageSender::create(... Session & ...), OM_TYPE::subscribe(... Session & ...), OM_TYPE::unsubscribe(... Session & ...), OM_TYPE::changeSubscription(... Session & ...), is deprecated, use OM_TYPE::MessageSender::create(... SessionPtr ...), OM_TYPE::subscribe(... SessionPtr ...), OM_TYPE::unsubscribe(... SessionPtr ...), OM_TYPE::changeSubscription(... SessionPtr ...),</pre>	<p>Several methods that used a Session reference were changed to use the SessionPtr type.</p>	+ MW-5949 - Mark Deprecated API Functions as Deprecated with TENA_Middleware_D EPRECATED() macro RESOLVED
<pre>OM_TYPE::ImmutablePublicationState, OM_TYPE::ImmutablePublicationStatePtr is deprecated, OM_Type::PublicationState, use OM_Type::PublicationStatePtr</pre>	<p>The ImmutablePublicationState is not used and the PublicationState type can be used instead.</p>	+ MW-5949 - Mark Deprecated API Functions as Deprecated with TENA_Middleware_D EPRECATED() macro RESOLVED

Programmers Guide 6.0.6 Addendum

TENA Middleware Programmer's Guide 6.0.6 Addendum

 A common version of the TENA Middleware Programmer's Guide is used for all of the minor 6.0.x middleware releases. The changes to the usage and operation of the 6.0.6 version of the TENA Middleware from the previous version are shown below (and also highlighted in the Programmer's Guide pages as well).

- [Programmers Guide 6.0](#)

Middleware 6.0.6 Changes Relevant to the Programmer's Guide

None.

API Changes

None.

API Additions

None.

API Deprecated Items

None.

Programmers Guide 6.0.7 Addendum

TENA Middleware Programmer's Guide 6.0.7 Addendum

i A common version of the TENA Middleware Programmer's Guide is used for all of the minor 6.0.x middleware releases. The changes to the usage and operation of the 6.0.7 version of the TENA Middleware from the previous version are shown below (and also highlighted in the Programmer's Guide pages as well).

- [Programmers Guide 6.0](#)

Middleware 6.0.7 Changes Relevant to the Programmer's Guide

Summary

1. Replace `std::auto_ptr` with `std::unique_ptr`.
2. When passing a variable that was declared as a `std::unique_ptr` as an argument in a method, use `std::move` around the variable.

Replace `std::auto_ptr` with `std::unique_ptr`

The evolution of the C++ language has motivated a small change to the TENA Middleware API for 6.0.7. The use of `std::auto_ptr` has been deprecated since C++11. In C++17, `std::auto_ptr` has been completely removed from the language. Furthermore, C++17 forbids adding `auto_ptr` back to the `std` namespace, so that is not an acceptable option for backward compatibility. The replacement for `std::auto_ptr` is `std::unique_ptr`, but converting to use `std::unique_ptr` is slightly more complicated than a simple find/replace operation. Taking advantage of C++ language improvements, `std::unique_ptr` disallows the copy operation that was the weakness of `std::auto_ptr`. With `std::unique_ptr`, transferring ownership of the pointer being managed is done using move semantics using `std::move()`.

User code that was developed using TENA Middleware versions prior to 6.0.7 will not compile without modification with 6.0.7, but the necessary modifications are limited and straightforward (even though the error message emitted by various compilers are not especially helpful). For TENA-related code, there are three changes that need to happen. If user code makes other use of `std::auto_ptr`, there will be similar issues and solutions. The discussion below makes use of the Example-Tank-Publisher-v7.2.7 example application from the TENA Middleware 6.0.6 release.

The short description of the necessary changes is summarized below.

Example Changes

Example from Middleware Versions Prior To 6.0.7

```
std::auto_ptr< Example::Tank::RemoteMethodsInterface > pRemoteMethods(
    new MyApplication::Example_Tank::RemoteMethodsImpl );
.....
.....
Example::Tank::ServantPtr pServant(
    pServantFactory->createServant(
        pRemoteMethods, // Note: std::auto_ptr<> will be NULL after this call
        *initializer,
        communicationProperties ) );
```

Example from Middleware Version 6.0.7

```
std::unique_ptr< Example::Tank::RemoteMethodsInterface > pRemoteMethods( // Note: auto_ptr replaced by
unique_ptr
    new MyApplication::Example_Tank::RemoteMethodsImpl );
.....
.....
Example::Tank::ServantPtr pServant(
    pServantFactory->createServant(
        std::move( pRemoteMethods ), // Note: std::unique_ptr<> will be NULL after this call. Note: std::move
        *initializer,
        communicationProperties ) );
```

Sample Compilation Errors and Messages

The rest of this article provides a much more detailed discussion based on the Example-Tank-Publisher-v7.2.7 example application. This describes what happens when the application is compiled against the 6.0.7 version of the TENA Middleware. There will be compilation errors in createServant.cpp and updateServant.cpp relating to createInitializer(), createServant(), and commitUpdater(). Sample error messages from Visual Studio 2017, g++, and clang are shown below.

createInitializer()

The lines

```
std::auto_ptr< Example::Tank::PublicationStateInitializer >
initializer( pServantFactory->createInitializer() );
```

result in a compilation error similar to the following with g++

```
myImpl/MyApplication/Example_Tank/createServant.cpp:37:55: error: no matching function for call to 'std::auto_ptr<Example::Example_Tank_v4_MW607::Tank::PublicationStateInitializer>::auto_ptr(std::unique_ptr<Example::Example_Tank_v4_MW607::Tank::PublicationStateInitializer>)'
initializer( pServantFactory->createInitializer() );
```

On macOS with clang, the error is

```
myImpl/MyApplication/Example_Tank/createServant.cpp:36:8: error: no member named
'auto_ptr' in namespace 'std'
std::auto_ptr< Example::Tank::PublicationStateInitializer >
~~~~~^
myImpl/MyApplication/Example_Tank/createServant.cpp:36:61: error: expected
'(' for function-style cast or type construction
std::auto_ptr< Example::Tank::PublicationStateInitializer >
~~~~~^
myImpl/MyApplication/Example_Tank/createServant.cpp:37:5: error: use of
undeclared identifier 'initializer'
initializer( pServantFactory->createInitializer() );
```

On Windows 10 with Visual Studio 2017, the error is

```
1> c:\users\tenabuild\mborick\example-tank-publisher-v7.2.7\myimpl\myapplication\example_tank\createservant.cpp(37): error
C2664: 'std::auto_ptr<Example::Example_Tank_v4_MW607::Tank::PublicationStateInitializer>::auto_ptr(std::auto_ptr<Example::Example_Tank_v4_MW607::Tank::PublicationStateInitializer> &) noexcept': cannot convert argument 1 from 'std::unique_ptr<Example::Example_Tank_v4_MW607::Tank::PublicationStateInitializer, std::default_delete<_Ty>>' to '_Ty *'
1> with
1> [
1> _Ty=Example::Example_Tank_v4_MW607::Tank::PublicationStateInitializer
1> ]
1> c:\users\tenabuild\mborick\example-tank-publisher-v7.2.7\myimpl\myapplication\example_tank\createservant.cpp(37): note: No
user-defined-conversion operator available that can perform this conversion, or the operator cannot be called
1> c:\users\tenabuild\mborick\example-tank-publisher-v7.2.7\myimpl\myapplication\example_tank\createservant.cpp(67): error
C2664: 'const Example::Example_Tank_v4_MW607::Tank::ServantPtr Example::Example_Tank_v4_MW607::Tank::ServantFactory::createServant(std::unique_ptr<Example::Example_Tank_v4_MW607::Tank::RemoteMethodsInterface, std::default_delete<_Ty>>, const Example::Example_Tank_v4_MW607::Tank::PublicationStateInitializer &, const TENA::Middleware::MW607::CommunicationProperties &, const TENA::Middleware::MW607::ConcurrencyProperties &)' cannot convert argument 1
from 'std::auto_ptr<Example::Example_Tank_v4_MW607::Tank::RemoteMethodsInterface, std::default_delete<_Ty>>'
1> with
1> [
1> _Ty=Example::Example_Tank_v4_MW607::Tank::RemoteMethodsInterface
1> ]
1> c:\users\tenabuild\mborick\example-tank-publisher-v7.2.7\myimpl\myapplication\example_tank\createservant.cpp(70): note: No
user-defined-conversion operator available that can perform this conversion, or the operator cannot be called
1>Done building target "CICompile" in project "Example-Tank-Publisher-v7.2.7-2017.vcxproj" -- FAILED.
```

To correct this replace std::auto_ptr with std::unique_ptr. Also, change the other use of std::auto_ptr to std::unique_ptr in the line

```
std::auto_ptr< Example::Tank::RemoteMethodsInterface > pRemoteMethods(
new MyApplication::Example_Tank::RemoteMethodsImpl );
```

Change this to

```
std::unique_ptr< Example::Tank::RemoteMethodsInterface > pRemoteMethods( new MyApplication::Example_Tank::RemoteMethodsImpl );
```

createServant()

Compile createServant.cpp with the above changes. The code now looks like the following

```
Example::Tank::ServantPtr pServant( pServantFactory->createServant( pRemoteMethods, // Note: std::unique_ptr<> will be NULL after this call *initializer, communicationProperties ) );
```

With g++ this will result in an error when compiling

```
myImpl/MyApplication/Example_Tank/createServant.cpp: In function 'const ServantPtr createExampleTankServant(const ServantFactoryPtr&, TENA::Middleware::MW607::CommunicationProperties, TENA::uint32, std::ostream&)'  
myImpl/MyApplication/Example_Tank/createServant.cpp:69:31: error: use of deleted function 'std::unique_ptr<_Tp, _Dp>::unique_ptr(const std::unique_ptr<_Tp, _Dp>&) [with _Tp = Example::Example_Tank_v4_MW607::Tank::RemoteMethodsInterface; _Dp = std::default_delete<Example::Example_Tank_v4_MW607::Tank::RemoteMethodsInterface>]'  
    communicationProperties ) );  
          ^  
[ some lines omitted ]  
/opt/rh/devtoolset-8/root/usr/include/c++/8/bits/unique_ptr.h:394:7: note: declared here  
    unique_ptr(const unique_ptr&) = delete;  
    ^~~~~~  
In file included from myImpl/MyApplication/Example_Tank/createServant.h:20,  
     from myImpl/MyApplication/Example_Tank/createServant.cpp:16:  
/home/tenabuild/mboriadk/install/TENA/6.0.7/include/OMs/Example-Tank-vtrunk-4/Example/Tank.h:1398:9: note: initializing argument 1 of 'const ServantPtr Example::Example_Tank_v4_MW607::Tank::ServantFactory::createServant(std::unique_ptr<Example::Example_Tank_v4_MW607::Tank::RemoteMethodsInterface>, const Example::Example_Tank_v4_MW607::Tank::PublicationStateInitializer&, const TENA::Middleware::MW607::CommunicationProperties&, const TENA::Middleware::MW607::ConcurrencyProperties&)'  
    createServant(  
    ^~~~~~
```

On macOS the error is

```
myImpl/MyApplication/Example_Tank/createServant.cpp:67:7: error: call to implicitly-deleted copy constructor of '::std::unique_ptr<RemoteMethodsInterface>'  
    pRemoteMethods, // Note: std::auto_ptr<> will be NULL after this call  
    ^~~~~~  
/Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr/include/c++/v1/memory:2440:3: note: copy constructor is implicitly deleted because 'unique_ptr<Example::Example_Tank_v4_MW607::Tank::RemoteMethodsInterface, std::default_delete<Example::Example_Tank_v4_MW607::Tank::RemoteMethodsInterface> >' has a user-declared move constructor  
    unique_ptr(unique_ptr& __u) noexcept  
    ^  
/Users/mboriadk/NTM/GOMER/TENA/6.0.7/include/OMs/Example-Tank-vtrunk-4/Example/Tank.h:1399:55: note: passing argument to parameter 'pRemoteMethods' here  
    ::std::unique_ptr< RemoteMethodsInterface > pRemoteMethods,
```

1 error generated.

On Windows 10 with Visual Studio 2017, the error is

```
1> createServant.cpp
1> c:\users\tenabuild\mboriack\example-tank-publisher-v7.2.7\myimpl\myapplication\example_tank\createservant.cpp(66): error
C2280: 'std::unique_ptr<Example::Example_Tank_v4_MW607::Tank::RemoteMethodsInterface, std::default_delete<_Ty>>::
unique_ptr(const std::unique_ptr<_Ty, std::default_delete<_Ty>> &)' attempting to reference a deleted function
1> with
1> [
1> _Ty=Example::Example_Tank_v4_MW607::Tank::RemoteMethodsInterface
1> ]
1> c:\program files (x86)\microsoft visual studio\2017\enterprise\vc\tools\msvc\14.16.27023\include\memory(2337): note: see
declaration of 'std::unique_ptr<Example::Example_Tank_v4_MW607::Tank::RemoteMethodsInterface, std::default_delete<_Ty>>::
unique_ptr'
1> with
1> [
1> _Ty=Example::Example_Tank_v4_MW607::Tank::RemoteMethodsInterface
1> ]
1> c:\program files (x86)\microsoft visual studio\2017\enterprise\vc\tools\msvc\14.16.27023\include\memory(2337): note: 'std::
unique_ptr<Example::Example_Tank_v4_MW607::Tank::RemoteMethodsInterface, std::default_delete<_Ty>>::unique_ptr(const std::
unique_ptr<_Ty, std::default_delete<_Ty>> &)' function was explicitly deleted
1> with
1> [
1> _Ty=Example::Example_Tank_v4_MW607::Tank::RemoteMethodsInterface
1> ]
1>Done building target "CICompile" in project "Example-Tank-Publisher-v7.2.7-2017.vcxproj" -- FAILED.
1>
1>Done building project "Example-Tank-Publisher-v7.2.7-2017.vcxproj" -- FAILED.
1>
1>Build FAILED.
1>
1>c:\users\tenabuild\mboriack\example-tank-publisher-v7.2.7\myimpl\myapplication\example_tank\createservant.cpp(66): error
C2280: 'std::unique_ptr<Example::Example_Tank_v4_MW607::Tank::RemoteMethodsInterface, std::default_delete<_Ty>>::
unique_ptr(const std::unique_ptr<_Ty, std::default_delete<_Ty>> &)' attempting to reference a deleted function
1> 0 Warning(s)
1> 1 Error(s)
1>
```

The problem that these error messages refer to is that the default delete method for std::unique_ptr was explicitly deleted to prevent unintended transfer of the pointer that it is managing. The proper way to use std::unique_ptr in an argument to a method is to use the std::move method. Therefore, in the code above pRemoteMethods becomes std::move(pRemoteMethods). The snippet of code now looks like

```
Example::Tank::ServantPtr pServant(
pServantFactory->createServant(
std::move( pRemoteMethods ), // Note: std::unique_ptr<_Ty> will be NULL after this call
*initializer,
communicationProperties ) );
```

The other place where std::auto_ptr was used in the TENA Middleware API prior to 6.0.7 occurs in the Example-Tank-Publisher-v7.2.7 example application in updateServant.cpp. If the updateServant.cpp is compiled without changing auto_ptr to unique_ptr, the following error results with g++

```
myImpl/MyApplication/Example_Tank/updateServant.cpp: In function 'void updateExampleTankServant(Example::
Example_Tank_v4_MW607::Tank::ServantPtr, TENA::uint32, std::ostream&)':

myImpl/MyApplication/Example_Tank/updateServant.cpp:42:38: error: no matching function for call to 'std::auto_ptr<Example::
Example_Tank_v4_MW607::Tank::PublicationStateUpdater>::auto_ptr(std::unique_ptr<Example::Example_Tank_v4_MW607::
Tank::PublicationStateUpdater>)'
    p_TankServant->createUpdater() ;

In file included from /opt/rh/devtoolset-8/root/usr/include/c++/8/memory:84,
                 from /home/tenabuild/mboriack/install/TENA/6.0.7/include/TENA/Middleware/Utils/Managee.h:17,
                 from /home/tenabuild/mboriack/install/TENA/6.0.7/include/TENA/Middleware/Utils/ManagedPtr.h:24,
                 from /home/tenabuild/mboriack/install/TENA/6.0.7/include/TENA/Middleware/ExecutionPtr.h:20,
```

```

from /home/tenabuild/mboriack/install/TENA/6.0.7/include/TENA/Middleware/exception.h:21,
from /home/tenabuild/mboriack/install/TENA/6.0.7/include/TENA/Middleware/Configuration.h:21,
from /home/tenabuild/mboriack/install/TENA/6.0.7/include/TENA/Middleware/config.h:21,
from myImpl/MyApplication/Example_Tank/updateServant.h:19,
from myImpl/MyApplication/Example_Tank/updateServant.cpp:16:

/opt/rh/devtoolset-8/root/usr/include/c++/8/backward/auto_ptr.h:266:7: note: candidate: 'std::auto_ptr< <template-parameter-1-1> >::auto_ptr(std::auto_ptr_ref<_Tp>) [with _Tp = Example::Example_Tank_v4_MW607::Tank::PublicationStateUpdater]'

auto_ptr(auto_ptr_ref<element_type> __ref) throw()
^~~~~~

/opt/rh/devtoolset-8/root/usr/include/c++/8/backward/auto_ptr.h:266:7: note: no known conversion for argument 1 from 'std::unique_ptr<Example::Example_Tank_v4_MW607::Tank::PublicationStateUpdater>' to 'std::auto_ptr_ref<Example::Example_Tank_v4_MW607::Tank::PublicationStateUpdater>'

/opt/rh/devtoolset-8/root/usr/include/c++/8/backward/auto_ptr.h:127:9: note: candidate: 'template<class _Tp1> std::auto_ptr<<template-parameter-1-1>>::auto_ptr(std::auto_ptr<_Up>&)' 

auto_ptr(auto_ptr<_Tp1>& __a) throw() : _M_ptr(__a.release()) {}

^~~~~~

/opt/rh/devtoolset-8/root/usr/include/c++/8/backward/auto_ptr.h:127:9: note: template argument deduction/substitution failed:

myImpl/MyApplication/Example_Tank/updateServant.cpp:42:38: note: 'std::unique_ptr<Example::Example_Tank_v4_MW607::Tank::PublicationStateUpdater>' is not derived from 'std::auto_ptr<_Up>'

p_TankServant->createUpdater() ;

^

[ some lines deleted ]

/opt/rh/devtoolset-8/root/usr/include/c++/8/backward/auto_ptr.h:114:7: note: candidate: 'std::auto_ptr< <template-parameter-1-1> >::auto_ptr(std::auto_ptr<<template-parameter-1-1>>&) [with _Tp = Example::Example_Tank_v4_MW607::Tank::PublicationStateUpdater]'

auto_ptr(auto_ptr& __a) throw() : _M_ptr(__a.release()) {}

^~~~~~

/opt/rh/devtoolset-8/root/usr/include/c++/8/backward/auto_ptr.h:114:7: note: no known conversion for argument 1 from 'std::unique_ptr<Example::Example_Tank_v4_MW607::Tank::PublicationStateUpdater>' to 'std::auto_ptr<Example::Example_Tank_v4_MW607::Tank::PublicationStateUpdater>&'

/opt/rh/devtoolset-8/root/usr/include/c++/8/backward/auto_ptr.h:105:7: note: candidate: 'std::auto_ptr< <template-parameter-1-1> >::element_type* [with _Tp = Example::Example_Tank_v4_MW607::Tank::PublicationStateUpdater; std::auto_ptr<<template-parameter-1-1>>::element_type = Example::Example_Tank_v4_MW607::Tank::PublicationStateUpdater]'

auto_ptr(element_type* __p = 0) throw() : _M_ptr(__p) {}

^~~~~~

/opt/rh/devtoolset-8/root/usr/include/c++/8/backward/auto_ptr.h:105:7: note: no known conversion for argument 1 from 'std::unique_ptr<Example::Example_Tank_v4_MW607::Tank::PublicationStateUpdater>' to 'std::auto_ptr<Example::Example_Tank_v4_MW607::Tank::PublicationStateUpdater>::element_type*' {aka 'Example::Example_Tank_v4_MW607::Tank::PublicationStateUpdater*'}

myImpl/MyApplication/Example_Tank/updateServant.cpp:83:19: error: no matching function for call to 'Example::Example_Tank_v4_MW607::Tank::Servant::commitUpdater(std::auto_ptr<Example::Example_Tank_v4_MW607::Tank::PublicationStateUpdater>&)' 

p_TankUpdater );

^

In file included from myImpl/MyApplication/Example_Tank/updateServant.h:20,
from myImpl/MyApplication/Example_Tank/updateServant.cpp:16:

/home/tenabuild/mboriack/install/TENA/6.0.7/include/OMs/Example-Tank-vtrunk-4/Example/Tank.h:989:9: note: candidate: 'TENA::uint32 Example::Example_Tank_v4_MW607::Tank::Servant::commitUpdater(std::unique_ptr<Example::Example_Tank_v4_MW607::Tank::PublicationStateUpdater>)'
```

```

commitUpdater( ::std::unique_ptr< PublicationStateUpdater > pUpdater );

^~~~~~
/home/tenabuild/mboriack/install/TENA/6.0.7/include/OMs/Example-Tank-vtrunk-4/Example/Tank.h:989:9: note: no known
conversion for argument 1 from 'std::auto_ptr<Example::Example_Tank_v4_MW607::Tank::PublicationStateUpdater>' to 'std::
unique_ptr<Example::Example_Tank_v4_MW607::Tank::PublicationStateUpdater>'

make: *** [Makefile:101: obj/centos7-gcc82-64/myImpl/MyApplication/Example_Tank/updateServant.o] Error 1

```

On macOS the error is

```

myImpl/MyApplication/Example_Tank/updateServant.cpp:40:8: error: no member named 'auto_ptr' in namespace 'std'
    std::auto_ptr< Example::Tank::PublicationStateUpdater >
    ~~~~~^
myImpl/MyApplication/Example_Tank/updateServant.cpp:40:57: error: expected '(' for function-style cast or type construction
    std::auto_ptr< Example::Tank::PublicationStateUpdater >
    ~~~~~~^
myImpl/MyApplication/Example_Tank/updateServant.cpp:41:5: error: use of undeclared identifier 'p_TankUpdater'
    p_TankUpdater(
    [ some lines omitted ]

```

On Windows 10 with Visual Studio the error is

```

1> c:\users\tenabuild\mboriack\example-tank-publisher-v7.2.7\myimpl\myapplication\example_tank\updateservant.cpp(41): error
C2664: 'std::auto_ptr<Example::Example_Tank_v4_MW607::Tank::PublicationStateUpdater>::auto_ptr(std::auto_ptr<Example::Example_Tank_v4_MW607::Tank::PublicationStateUpdater> &) noexcept': cannot convert argument 1 from 'std::unique_ptr<Example::Example_Tank_v4_MW607::Tank::PublicationStateUpdater, std::default_delete<_Ty>>' to '_Ty *'
1> with
1> [
1> _Ty=Example::Example_Tank_v4_MW607::Tank::PublicationStateUpdater
1> ]
1> c:\users\tenabuild\mboriack\example-tank-publisher-v7.2.7\myimpl\myapplication\example_tank\updateservant.cpp(42): note: No
user-defined-conversion operator available that can perform this conversion, or the operator cannot be called
1> c:\users\tenabuild\mboriack\example-tank-publisher-v7.2.7\myimpl\myapplication\example_tank\updateservant.cpp(82): error
C2664: 'TENA::uint32 Example::Example_Tank_v4_MW607::Tank::Servant::commitUpdater(std::unique_ptr<Example::Example_Tank_v4_MW607::Tank::PublicationStateUpdater, std::default_delete<_Ty>>)': cannot convert argument 1 from 'std::auto_ptr<Example::Example_Tank_v4_MW607::Tank::PublicationStateUpdater>' to 'std::unique_ptr<Example::Example_Tank_v4_MW607::Tank::PublicationStateUpdater, std::default_delete<_Ty>>'
1> with
1> [
1> _Ty=Example::Example_Tank_v4_MW607::Tank::PublicationStateUpdater
1> ]
1> c:\users\tenabuild\mboriack\example-tank-publisher-v7.2.7\myimpl\myapplication\example_tank\updateservant.cpp(83): note: No
user-defined-conversion operator available that can perform this conversion, or the operator cannot be called
1>Done building target "CICCompile" in project "Example-Tank-Publisher-v7.2.7-2017.vcxproj" -- FAILED.
1>
1>Done building project "Example-Tank-Publisher-v7.2.7-2017.vcxproj" -- FAILED.
1>
1>Build FAILED.
1>
1>c:\users\tenabuild\mboriack\example-tank-publisher-v7.2.7\myimpl\myapplication\example_tank\updateservant.cpp(41): error
C2664: 'std::auto_ptr<Example::Example_Tank_v4_MW607::Tank::PublicationStateUpdater>::auto_ptr(std::auto_ptr<Example::Example_Tank_v4_MW607::Tank::PublicationStateUpdater> &) noexcept': cannot convert argument 1 from 'std::unique_ptr<Example::Example_Tank_v4_MW607::Tank::PublicationStateUpdater, std::default_delete<_Ty>>' to '_Ty *'
1>c:\users\tenabuild\mboriack\example-tank-publisher-v7.2.7\myimpl\myapplication\example_tank\updateservant.cpp(82): error
C2664: 'TENA::uint32 Example::Example_Tank_v4_MW607::Tank::Servant::commitUpdater(std::unique_ptr<Example::Example_Tank_v4_MW607::Tank::PublicationStateUpdater, std::default_delete<_Ty>>)': cannot convert argument 1 from 'std::auto_ptr<Example::Example_Tank_v4_MW607::Tank::PublicationStateUpdater>' to 'std::unique_ptr<Example::Example_Tank_v4_MW607::Tank::PublicationStateUpdater, std::default_delete<_Ty>>'
1> 0 Warning(s)
1> 2 Error(s)
1>
```

To correct this problem, change std::auto_ptr to std::unique_ptr.

commitUpdater()

Compiling updateServant.cpp with no further changes will have problems with the following lines

```
p_TankServant->commitUpdater(  
    p_TankUpdater );
```

With g++ the error is

```
myImpl/MyApplication/Example_Tank/updateServant.cpp: In function 'void updateExampleTankServant(Example::  
Example_Tank_v4_MW607::Tank::ServantPtr, TENA::uint32, std::ostream&)'  
  
myImpl/MyApplication/Example_Tank/updateServant.cpp:83:19: error: use of deleted function 'std::unique_ptr<_Tp, _Dp>::  
unique_ptr(const std::unique_ptr<_Tp, _Dp>&) [with _Tp = Example::Example_Tank_v4_MW607::Tank::PublicationStateUpdater;  
_Dp = std::default_delete<Example::Example_Tank_v4_MW607::Tank::PublicationStateUpdater>]'  
  
    p_TankUpdater );  
          ^  
  
[ some lines deleted ]  
  
/opt/rh/devtoolset-8/root/usr/include/c++/8/bits/unique_ptr.h:394:7: note: declared here  
  
    unique_ptr(const unique_ptr&) = delete;  
    ^~~~~~  
  
In file included from myImpl/MyApplication/Example_Tank/updateServant.h:20,  
from myImpl/MyApplication/Example_Tank/updateServant.cpp:16:  
  
/home/tenabuild/mboriack/install/TENA/6.0.7/include/OMs/Example-Tank-vtrunk-4/Example/Tank.h:989:9: note: initializing  
argument 1 of 'TENA::uint32 Example::Example_Tank_v4_MW607::Tank::Servant::commitUpdater(std::unique_ptr<Example::  
Example_Tank_v4_MW607::Tank::PublicationStateUpdater>)'  
  
    commitUpdater( ::std::unique_ptr< PublicationStateUpdater > pUpdater );  
    ^~~~~~  
  
make: *** [Makefile:101: obj/centos7-gcc82-64/myImpl/MyApplication/Example_Tank/updateServant.o] Error 1
```

On macOS with clang the error is

```
myImpl/MyApplication/Example_Tank/updateServant.cpp:83:5: error: call to implicitly-deleted copy constructor of '::std::  
unique_ptr<PublicationStateUpdater>'  
  
    p_TankUpdater );  
    ^~~~~~  
  
/Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr/include/c++/v1/memory:2440:3: note: copy  
constructor is implicitly deleted because 'unique_ptr<Example::Example_Tank_v4_MW607::Tank::PublicationStateUpdater, std::  
__1::default_delete<Example::Example_Tank_v4_MW607::Tank::PublicationStateUpdater>>' has a user-declared move constructor  
  
    unique_ptr(unique_ptr& __u) noexcept  
    ^  
  
/Users/mboriack/NTM/GOMER/TENA/6.0.7/include/OMs/Example-Tank-vtrunk-4/Example/Tank.h:989:69: note: passing argument  
to parameter 'pUpdater' here  
  
    commitUpdater( ::std::unique_ptr< PublicationStateUpdater > pUpdater );  
    ^  
  
1 error generated.
```

On Windows 10 with Visual Studio 2017, the error is

```
1> updateServant.cpp
1> c:\users\tenabuild\mboriack\example-tank-publisher-v7.2.7\myimpl\myapplication\example_tank\updateservant.cpp(82): error
C2280: 'std::unique_ptr<Example::Example_Tank_v4_MW607::Tank::PublicationStateUpdater, std::default_delete<_Ty>>::
unique_ptr(const std::unique_ptr<_Ty, std::default_delete<_Ty>> &)': attempting to reference a deleted function
1> with
1> [
1> _Ty=Example::Example_Tank_v4_MW607::Tank::PublicationStateUpdater
1> ]
1> c:\program files (x86)\microsoft visual studio\2017\enterprise\vc\tools\msvc\14.16.27023\include\memory(2337): note: see
declaration of 'std::unique_ptr<Example::Example_Tank_v4_MW607::Tank::PublicationStateUpdater, std::default_delete<_Ty>>::
unique_ptr'
1> with
1> [
1> _Ty=Example::Example_Tank_v4_MW607::Tank::PublicationStateUpdater
1> ]
1> c:\program files (x86)\microsoft visual studio\2017\enterprise\vc\tools\msvc\14.16.27023\include\memory(2337): note: 'std::
unique_ptr<Example::Example_Tank_v4_MW607::Tank::PublicationStateUpdater, std::default_delete<_Ty>>::unique_ptr(const std::
unique_ptr<_Ty, std::default_delete<_Ty>> &)': function was explicitly deleted
1> with
1> [
1> _Ty=Example::Example_Tank_v4_MW607::Tank::PublicationStateUpdater
1> ]
1>Done building target "C|Compile" in project "Example-Tank-Publisher-v7.2.7-2017.vcxproj" -- FAILED.
1>
1>Done building project "Example-Tank-Publisher-v7.2.7-2017.vcxproj" -- FAILED.
1>
1>Build FAILED.
1>
1>c:\users\tenabuild\mboriack\example-tank-publisher-v7.2.7\myimpl\myapplication\example_tank\updateservant.cpp(82): error
C2280: 'std::unique_ptr<Example::Example_Tank_v4_MW607::Tank::PublicationStateUpdater, std::default_delete<_Ty>>::
unique_ptr(const std::unique_ptr<_Ty, std::default_delete<_Ty>> &)': attempting to reference a deleted function
1> 0 Warning(s)
1> 1 Error(s)
```

Similar to the createServant() case, just replace p_TankUpdater with std::move(p_TankUpdater).

API Changes

Replace std::auto_ptr with std::unique_ptr. See the discussion above.

API Additions

None.

API Deprecated Items

None.

Programmers Guide 6.0.8 Addendum

TENA Middleware Programmer's Guide 6.0.8 Addendum

 A common version of the TENA Middleware Programmer's Guide is used for all of the minor 6.0.x middleware releases. The changes to the usage and operation of the 6.0.8 version of the TENA Middleware from the previous version are shown below (and also highlighted in the Programmer's Guide pages as well).

- [Programmers Guide 6.0](#)

Middleware 6.0.8 Changes Relevant to the Programmer's Guide

Summary

Release 6.0.8 uses [C++17](#) which provides significant programming advancements!

This implies that applications built using the TENA Middleware SDK will need to support C++17 as well.

Enable C++ 17 Support in Compiler

For GCC and Clang compilers, this means passing the `-std=c++17` option to the compiler.

For Visual Studio compilers, this means passing `/std:c++17` to the compiler. If using Visual Studio project files, this can be accomplished by ensuring this line appears in the project file:

```
<LanguageStandard>stdcpp17</LanguageStandard>
```

Note that, by default, Visual Studio always returns the value "199711L" for the `_cplusplus` preprocessor macro. This seems an odd choice, and while has no effect on TENA Middleware code, can cause problems with a myriad of 3rd parts software.

Consequently, you may wish to pass the `/Zc:_cplusplus` compiler option, which enables the `_cplusplus` preprocessor macro to report an updated value for recent C++ language standards support. (See: <https://docs.microsoft.com/en-us/cpp/build/reference/zc-cplusplus?view=msvc-160>)

C++ 17 Removed std::auto_ptr

Notably, C++17 eliminates `std::auto_ptr`. As a result, the previous Release (6.0.7) stopped using `std::auto_ptr` are now using `std::unique_ptr`. Applications using 6.0.7 have already been updated. (More information about this change and what C++ programmer's can expect can be found in the [Programmers Guide 6.0.7 Addendum](#).)

All of the improvements and fixes are listed in the helpdesk tables below.

API Changes

Replace `std::auto_ptr` with `std::unique_ptr`. See the discussion above.

API Additions

1.  [MW-6112](#) - Add get_() methods to PublicationStateInitializer RESOLVED
2.  [MW-6170](#) - Add ability to create ExecutionJoinOptions with specified OMs and without userSiteID or userApplicationID RESOLVED
 - `TENA::Middleware::ExecutionJoinOptions(std::vector< std::string > const & specifiedOMs);`
3.  [MW-6245](#) - TENA::Middleware::Utils::ExitDetector lacks ability to reset its counters RESOLVED
 - `void TENA::Middleware::Utils::ExitDetector::resetControlCdetected();`
`void TENA::Middleware::Utils::ExitDetector::resetControlBreakDetected();`
`void TENA::Middleware::Utils::ExitDetector::resetLogoffDetected();`
`void TENA::Middleware::Utils::ExitDetector::resetShutdownDetected();`
`void TENA::Middleware::Utils::ExitDetector::resetAll();`

API Deprecated Items

-  [MW-7106](#) - Deprecate `TENA::Middleware::Runtime::isEMreachable` method RESOLVED

- bool TENA::Middleware::Runtime::isEMreachable(TENA::Middleware::MW608::Endpoint const & endpoint)
 - Attempting the join the execution provides better information regarding the state of an EM at a given endpoint.

Stateful Distributed Object

Stateful Distributed Object

The Stateful Distributed Object (SDO) is a network-aware object that publishes updates to its data members to subscribed applications. SDOs combine the concept of a distributed shared memory construct using publish-subscribe semantics and support for client applications to invoke remote methods. SDOs comprise the primary building blocks used by TENA Middleware applications for distributed collaboration.

Description

A Stateful Distributed Object (SDO) is a concept formed by the application of publish-subscribe semantics to the data members of a distributed object. The object's data—the state—is disseminated via anonymous publish-subscribe and cached locally at each subscriber. SDOs also incorporate the notion of remote methods: function calls that share the conventional syntax of a local function call, but are actually implemented and processed by a remote application on the network. The TENA Middleware extends this notion of locational transparency to an SDO's data attributes: these are readily accessible to the application as locally available data.

TENA applications use SDOs to represent physical or informational items that provide remote services to other applications, and/or have state information that changes over time and needs to be disseminated to interested applications. A publishing application creates an SDO (referred to as an **SDO Servant**) and subscribing applications that have expressed an interest in that object will discover an **SDO Proxy** used to represent that particular object instance. Therefore, for a particular object instance, there is only a single SDO Servant owned by the publishing application and potentially many SDO Proxies corresponding to that servant in various subscribing applications.

When the publishing application updates the state information of the SDO Servant, the middleware disseminates those state changes to the subscribing applications holding a corresponding SDO Proxy. The subscribing applications can then read the state values through their proxy. This publish-subscribe paradigm implemented with SDO servants and proxies provides a distributed shared memory mechanism for that particular SDO instance. Nominally, the subscribing application will only need access to the most current state information, although an observer notification mechanism can be used to access every state update performed by the servant if necessary.

In addition to the distributed shared memory behavior with the SDO proxies held by subscribing applications, the SDO can define remote methods that can be accessed through the SDO proxies. When a subscribing application invokes a method on the local proxy, the middleware will handle the invocation of that method on the corresponding servant that will typically exist in an application running on a different computer system across the network.

When the publishing application no longer needs to maintain the SDO Servant, it can be destroyed. All subscribing applications that are holding an SDO Proxy corresponding to the destroyed servant are then notified of the destruction.

SDO Object Model Definition

SDOs are declared in TDL (TENA Definition Language) using the `class` keyword. This definition indicates the syntactical information associated with an SDO's state and remote methods. The TENA Object Model Compiler (OMC) uses these definitions to automatically generate code that is used by publishing and subscribing applications to ensure that the definitions are used properly from a type safety perspective.

A simple SDO definition from the `Example-Vehicle` object model is shown below. The SDO type `Vehicle` contains three data members (or, attributes): `name`, `team`, and `location`. SDO data members can be any [fundamental type](#), [enums](#), [local classes](#), or [messages](#).

```
package Example
{
    enum Team
    {
        Team_Red,
        Team_Blue,
        Team_Green
    };

    local class Location
    {
        float64 xInMeters;
        float64 yInMeters;
    };

    class Vehicle
    {
        string name;
        Team team;
        Location location;
    };
}
```

Additional information related to defining SDO types within an object model can be found in the [Class meta-model documentation](#).

SDO Creation

The following code fragment illustrates how a publishing application creates an SDO servant. Nominally, there are three steps associated with creating an SDO servant:

1. Create an `Initializer` that is used to set the initial attribute values.
2. Create the appropriate `ServantFactory`.
3. Create the SDO Servant using the factory and the initializer.

```
// Create the initializer and set attribute values
std::unique_ptr< Example::Vehicle::PublicationStateInitializer >
initializer( pServantFactory->createInitializer() );

initializer->set_name( "vehicle #1" );

initializer->set_team( Example::Team_Red );

Example::Location::LocalClassPtr location(
    Example::Location::create(3.0, 4.0) );
initializer->set_location( location );

// Create a ServantFactory to create Example::Vehicle objects
Example::Vehicle::ServantFactoryPtr p_ExampleVehicleServantFactory(
    Example::Vehicle::ServantFactory::create( pSession ) );

// Create a Example::Vehicle::Servant, returned in a smart pointer
// The caller must hold a reference or the Servant will be destroyed
Example::Vehicle::ServantPtr pServant(
    pServantFactory->createServant(
        *initializer,
        communicationProperties ) );
```

Release 6.0.2 Update — An alternative `create` method was added with release 6.0.2 of the middleware in which the `SessionPtr` argument is used, versus the `Session` reference (i.e., `*pSession`) that was used in the previous releases. Users are encouraged to use the `SessionPtr` interface. See [MW-4286](#) for more details.

In this example, the SDO type does not have any remote methods. If the SDO did define remote methods, then the application would be required to provide access to the code that implements the remote methods for the SDO servant in the `createServant` call. Additional details can be found in the [remote methods documentation page](#).

SDO Update

After the SDO servant has been created, the publishing application can update the state attributes as necessary. SDO attribute values are updated through a separate `Updater` object. This enables applications to update multiple attribute values atomically and permits multiple programming threads to perform updates.

The code example below illustrates how this updating is performed. The attribute `location`, which is a Local Class composed of two floating point values, is updated in this example.

```

// Get the updater, that is held in an auto_ptr
std::unique_ptr< Example::Vehicle::PublicationStateUpdater >
p_VehicleUpdater(
    p_VehicleServant->createUpdater() );

// create a LocalClass instance for attribute location
Example::Location::LocalClassPtr p_VehicleUpdater_location_LocalClass(
    Example::Location::create(4.0, 5.0) );

// Change the value of the p_VehicleUpdater's
// LocalClass attribute location
p_VehicleUpdater->
    set_location(
        p_VehicleUpdater_location_LocalClass );

// Pass the updater in to the servant to modify the state atomically
p_VehicleServant->commitUpdater(
    std::move( p_VehicleUpdater ) );

```

SDO Subscription

Applications interested in discovering all SDO instances of a particular type can "subscribe" to that SDO type. The subscription operation notifies the middleware that the application wants to be informed of any matching SDO instances that have been created, or will be created, in the execution. When a subscribing application discovers an SDO, the middleware provides the application an SDO Proxy that corresponds to the particular SDO Servant. Applications can use the SDO Proxy to read the state (i.e., attribute values) of the SDO or to invoke remote methods on the SDO.

When the publishing application updates the SDO Servant's attributes, the value are disseminated to all of the subscribing applications that are holding an SDO Proxy corresponding to that servant. If the publishing application destroys the SDO Servant, then the subscribing applications holding a corresponding SDO Proxy are notified of the destruction.

If an application determines that it is not interested in a particular SDO instance that has been discovered, the application can "let go" of the SDO Proxy either by letting the `ProxyPtr` go out of scope or by invoking the `reset` method on the `ProxyPtr`. When an application is no longer holding a reference to a particular `ProxyPtr`, the middleware will no longer deliver state updates or destruction notification for that SDO instance.

i — Note that dropping proxies results in "receive-side" filtering within the middleware, which means that the state updates from the publishing application will continue to be delivered to the subscribing application.

Notifications corresponding to SDO discoveries, updates and destructions, as well as scope changes associated with Advanced Filtering, can be delivered to the subscribing application through an Observer mechanism. A subscribing application defines an `Observer` class with various methods written by the application developer to provide the specific behavior associated with the various SDO subscription events. The specific SDO subscription events that are supported by the Observer mechanism are:

- `discoveryEvent` — Indicates a new SDO instance has been discovered.
- `stateChangeEvent` — Indicates that one or more attribute values of a discovered SDO has changed.
- `destructionEvent` — Indicates that a discovered SDO has been destroyed.
- `enteredScopeEvent` — Indicates that a discovered SDO has entered the interest scope (used with Advanced Filtering).
- `leftScopeEvent` — Indicates that a discovered SDO has left the interest scope (used with Advanced Filtering).

The information related to these subscription events are packaged internally to the middleware as "callback" objects that are placed on a queue until the application is prepared to have these events processed. This allows the middleware to support different application threading models. Subscribing applications need to invoke the middleware `evokeMultipleCallbacks` (or related) method for the subscription event to be processed, which triggers the appropriate `Observer` method. Additional information on callback processing can be found in the [callback framework documentation page](#).

A simple SDO subscription example is shown in the code fragment below. In the example, the application is subscribing to the `Example::Vehicle` SDO type using an observer defined in the class `MyApplication::Example_Vehicle::CountingObserver`. There are two configuration parameters used in the code fragment, `selfReflection` is used to control whether to discover SDOs created by this application (specifically this applications Session), and `wantPruning` is used to control if only the latest state change event is necessary (versus the need to process every state change). Additional details on this pruning operation can be found in the [received state processing documentation page](#).

```

bool selfReflection(
    appConfig["enableSelfReflection"].getValue< bool >() );

bool wantPruning(
    appConfig["pruneExpiredStateChange"].getValue< bool >() );

Example::Vehicle::SubscriptionPtr pExampleVehicleSubscription(
    new Example::Vehicle::Subscription( wantPruning ) );

MyApplication::Example_Vehicle::CountingObserverPtr
pExampleVehicleCountingObserver(
    new MyApplication::Example_Vehicle::CountingObserver() );

pExampleVehicleSubscription->addObserver( pExampleVehicleCountingObserver );

// Declare the application's interest in Vehicle objects.
Example::Vehicle::subscribe(
    pSession,
    pExampleVehicleSubscription,
    selfReflection );

```

Release 6.0.2 Update — An alternative `subscribe` method was added with release 6.0.2 of the middleware in which the `SessionPtr` argument is used, versus the `Session` reference (i.e., `*pSession`) that was used in the previous releases. Users are encouraged to use the `SessionPtr` interface. See [MW-4286](#) for more details.

Additional SDO Information

The description of SDOs provided in this documentation page intended to provide an overview of the SDO concepts. Additional details related SDOs are found in the following documentation pages.

- [SDO Creation](#) — SDOs are created by publishing applications and discovered by subscribing applications.
- [SDO Update](#) — Changes to the SDO state are disseminated to subscribers via updates.
- [SDO Remote Method](#) — SDO remote methods provide distributed services that can be used by subscribing applications.
- [SDO Destruction](#) — When a publisher destroys an SDO Servant, subscribers holding a corresponding SDO Proxy receive a destruction event via their observers.
- [SDO Subscription](#) — SDO subscription registers interest in discovering SDOs matching the subscribed type and optional Filter.
- [SDO Pointer](#) — SDO pointers allow applications to refer to a particular SDO instance to other applications through an attribute, method argument, or method return value.
- [SDO Inheritance and Polymorphism](#) — The TENA Meta-Model allows SDO types to inherit from other SDO types to extend/evolve existing constructs.
- [SDO Unsubscribe](#) — Provides ability for application to unsubscribe to a particular SDO type that was used with a previous type-based subscription.
- [SDO ID](#) — SDOs can be uniquely identified through the SDO ID generated by the middleware.
- [SDO Reactivation](#) — SDO reactivation allows an application to reactivate a previously created object to recover from an application or computer fault.
- [Adding SDO Support to Existing Application](#) — Description of the steps involved to add SDO publication and subscription to an existing application.

Adding SDO Support to Existing Application

Adding SDO Support to Existing Application

There are several steps involved in adding the capability to publish and subscribe to a SDO (Stateful Distributed Object) in an existing application. This page describes some of the steps that are needed to add SDO publication or subscription support to an existing application. The specific details will vary depending on the exact nature of the existing application and details of the requirements for the integration. Additional details regarding SDO publication and subscription can be found in the [SDO documentation section](#).

Description

Integrating publishing or subscribing SDOs using the TENA Middleware to an existing application

An existing application that needs to add support for publishing or subscribing to SDOs of a particular type will need to modify the application source code and build files. The source code is modified to include the appropriate SDO header files for necessary declarations. For SDO publishing, a definition for remote methods is required if the SDO has remote methods defined. Application source code is also modified to publish and update SDOs. For SDO subscribing, Observer classes must be written to handle events, unless a polling approach using the list of discovered SDOs is desired.

The general requirements are to:

1. Include header files or definitions that define the desired SDOs
2. Add appropriate code to create a Runtime, connect to an Execution, and create a Session.
3. For publishing, add code to create desired Servants, and initialize and update created servants.
4. For subscribing, define code to define actions to respond to events (Observers), connect them to a Subscription and subscribe to desired types.
5. For subscribing, decide on appropriate way to integrate calls to evoke callbacks.
6. Ensure that appropriate changes to build files are made: appropriate paths and include libraries, as required.

The specific details will vary based on requirements. For example, when adding publication or subscription to an existing TENA Middleware application, the existing Execution and Session objects already being used in the code may be sufficient.

C++ API Reference and Code Examples

Adding ability to publish SDOs in a C++ Application

An example is shown, starting with the `Example-Notification-Publisher` C++ application and adding support to publish Vehicle SDOs. The following changes are needed to the `Example-Notification-Publisher.cpp` file. In this case, because the starting point is already a TENA middleware-based application, the code to get a Runtime, Execution and Session is already present. Only the code to create the Servant and update its state need be added.

Add the SDO type header file:

```
#include <Example/Vehicle.h>
```

Then instantiate the `ServantFactory` used to create SDO Servants.

```
Example::Vehicle::ServantFactoryPtr p_ExampleVehicleServantFactory(
    Example::Vehicle::ServantFactory::create( pSession ) );
```

An `Initializer` is required which is used to set the initial attribute values for the servant. The `Initializer` provides `set_` methods for all of the SDO's attributes. Certain set methods may be present for `const` attributes, which can only be set during servant creation. All required attributes must be set on the `Initializer`, otherwise a runtime exception will occur when the servant is attempted to be created. An attribute is required if it is not marked as optional in the TDL definition. Optional attributes will also be indicated by `is_<attr>.set()` methods on the `Initializer`.

```
std::unique_ptr< Example::Vehicle::PublicationStateInitializer >
    initializer( p_ExampleVehicleServantFactory->createInitializer() );

initializer->set_name( "someName" );
initializer->set_team( "someTeam" );

Example::Location::LocalClassPtr location
    Example::Location::create( 1.0, 2.0 );
initializer->set_location( location );
```

The `Initializer` object, after the attributed values have been set, is used to create a servant.

```
Example::Vehicle::ServantPtr p_ExampleVehicleServant(
    p_ExampleVehicleServantFactory->createServant(
        *initializer,
        communicationProperties ) );
```

During the lifetime of the SDO servant, the application may need to update the attribute values (the servant's publication state). This is accomplished by obtaining an `Updater` object from the servant, setting the attribute values that need to be changed, and then committing the `Updater`.

```
Example::Vehicle::PublicationStateUpdaterPtr
p_ExampleVehicleUpdater( p_ExampleVehicleServant->createUpdater() );

Example::Location::LocalClassPtr
p_ExampleVehicleUpdater_location_LocalClass(
    Example::Location::create( 3.0, 4.0 ) );

p_ExampleVehicleUpdater->set_location(
    p_ExampleVehicleUpdater_location_LocalClass );

p_ExampleVehicleServant->commitUpdater( std::move( p_ExampleVehicleUpdater ) );
```

When the `ServantPtr` goes out of scope (or is `reset()`), the servant will be destroyed and any subscribing applications will be notified of the destruction.

The build files (Makefile on UNIX and Project file for Microsoft Visual Studio) need to be modified to link the object model definition library. In this example, both the Vehicle SDO and the Notification Message are defined in the same OMdefinition library. So there are no modifications required to add the object model definition library. If an object model definition library needed to be added, the build file would need to add the definition library to list of libraries used by the linker (e.g., `libExample-Vehicle-v1-$ (TENA_PLATFORM)-v$ (TENA_VERSION).lib` gets added to `Linker:Input:Additional Dependencies` configuration option in the Visual Studio project). An additional include path would also be required for the various headers.

Adding ability to subscribe to SDOs in an existing C++ application.

An existing application that needs to add support for subscribing to SDOs of a particular type will need to modify the application source code and (possibly) the build files. The source code is modified to include the appropriate SDO header file to include the necessary declarations. Application source code is also modified to perform the subscription operations to subscribe to the particular SDO type. In this case, because the starting point is already a TENA middleware-based application, the code to get a Runtime, Execution and Session is already present.

The illustration of the source code modifications are shown below, starting with the `Example-Notification-Subscriber` application and adding support to subscribe to Vehicle SDOs.

For SDO subscriptions, an application needs to define one or more `Observers` with code to define the behavior when an SDO event (e.g., discovery, state change, destruction) occurs. Additional information on the `Observer` pattern and subscription can be found on the [subscription services documentation page](#).

In this example, an existing `PrintingObserver` class is used. This is automatically generated for the example code generated for SDO subscribing applications. The implementation code for this `Observer` is not shown below, but can be copied from an example subscribing application.

```

class PrintingObserver
    : public virtual Example::Vehicle::AbstractObserver
{
public:
    PrintingObserver( TENA::uint32 verbosity, std::ostream & os );

    virtual
    void
    discoveryEvent(
        Example::Vehicle::ProxyPtr const & pProxy,
        Example::Vehicle::PublicationStatePtr const & pPubState );

    virtual
    void
    stateChangeEvent(
        Example::Vehicle::ProxyPtr const & pProxy,
        Example::Vehicle::PublicationStatePtr const & pPubState );

    virtual
    void
    destructionEvent(
        Example::Vehicle::ProxyPtr const & pProxy,
        Example::Vehicle::PublicationStatePtr const & pPubState );

private:
    std::ostream & os_;
    TENA::uint32 verbosity_;
};

```

Now, the application source code must be modified. The following changes are made to the `Example-Notification-Subscriber.cpp` file. First, add the SDO type header and the observer header files. Note that the Observer class files are placed in the `myImpl/MyApplication/Example_Vehicle` directory, which follows the suggested convention of the automatically generated Example Application code, but users are free to locate their Observer class files in a different location if desired.

```
#include <Example/Vehicle.h>
#include "myImpl/MyApplication/Example_Vehicle/PrintingObserver.h"
```

The main application is then changed to instantiate the `Subscription` object. The `Subscription` object is used to hold subscription configuration information and attach Observers. In this example, only default subscription behavior is necessary.

```
Example::Vehicle::SubscriptionPtr
pExampleVehicleSubscription( new Example::Vehicle::Subscription );
```

Then the application source code is modified to instantiate the `PrintingObserver` and added to the `Subscription` object.

```
Example::Vehicle::AbstractObserverPtr pPrintingObserver(
    new MyApplication::Example_Vehicle::PrintingObserver(
        verbosity, std::cout ) );

pExampleVehicleSubscription->addObserver( pPrintingObserver );
```

Finally, the application subscribes to the `Vehicle` SDO type.

the session to the message type.

```
Example::Vehicle::subscribe(
    *pSession,
    pExampleVehicleSubscription,
    false /* no self-reflection */ );
```

As with adding publication support, the build files (Makefile on UNIX and Project file for Microsoft Visual Studio) need to be modified to link the object model definition library. Since the Vehicle SDO is defined in the Example-Vehicle object model whose definition library is already linked into the existing application, there are no modifications required to add the object model definition library. If the object model definition library needed to be added, the build file would need to add the definition library to list of libraries used by the linker (e.g., libExample-Vehicle-v1-\$(TENA_PLATFORM)-v\$(TENA_VERSION).lib gets added to Linker:Input:Additional Dependencies configuration option in the Visual Studio project).

The header and build files for the PrintingObserver also needs to be added to the application's build file. A complication with using the example PrintingObserver is that this code relies on a common helper class printFunctions. If the existing application already defined printFunctions code (such as in this example), the existing printFunctions code needs to merged with the printFunctions code copied into this application. For experimentation purposes, the copied MyApplication::Example_Vehicle::PrintingObserver::print method could be modified to just eliminate the "os_ << pPubState" code lines.

Java API Reference and Code Examples

For Java applications, the Java Binding libraries will have to be on your application classpath at build-time and run-time. Looking at the Ant build.xml and associated dependencies.xml for one of the generated example programs should give some idea of the required additions to the classpath. For example, in the example Vehicle publisher application in \$TENA_HOME/examples/Example-Vehicle-v1/Vehicle-Publisher_1.1.1_JavaApplication/, the dependencies.xml lists two jars – Bindings_Middleware_XXX_JavaBinding_XXX_MW6.0.XXX.jar and {{Example_Vehicle_1_JavaBinding_XXX_MW6.0.XXX.jar}}. (The specific versions for these dependencies may change over time.)

The structure of the generated example code is intended to be somewhat reusable as-is. For each example program, an Application class provides the code to initialize Runtime, Execution and Session, to publish and update, or subscribe, as appropriate, and perform shutdown and cleanup. Once the appropriate jars are added to the classpath, the Application class code should be usable from within an arbitrary application. Using the Java Binding for the TENA Middleware generally requires the use of fully scoped classes, because Java does not provide the equivalent of C++ using statements. Since, for example, the servant classes of Tanks and Vehicles are both named Servant, only fully scoped names Example.Tank.Servant and Example.Vehicle.Servant can be used.

 The discussion below is based on the java example code delivered for the Example,Vehicle,1 object model. This can be downloaded using the JavaBinding download link for the [Example,Vehicle,1 Object Model](#).

Using example publisher code to add SDO publication to an existing application.

The org.Example.Vehicle_Publisher.Application class, from the example code included with the JavaBinding distribution for the Example,Vehicle,1 object model, provides a simple API:

```
class Application {
    public Application();
    public int microsecondsPerIteration;
    public int numberIterations;
    public TENA.Middleware.EndpointVector endpoints;

    public int run(TENA.Middleware.Configuration config);
    public void shutdown();
}
```

Using this from some existing application could be done following the example of the Main class used to start an Application.

```
import org.Example.Vehicle_Publisher.Application;
...
String[] mwConfigArgs = ... // Arguments to configure middleware. Alternatively set configuration parameters directly on the Configuration
TENA.Middleware.EndpointVector endpoints = new TENA.Middleware.EndpointVector();
endpoints.add(new TENA.Middleware.Endpoint("myhostname:50000"));

Application app = new Application();
app.endpoints = endpoints;
app.numberIterations = 50;
app.microsecondsPerIteration = 1000000;

TENA.Middleware.Configuration cfg = new TENA.Middleware.Configuration( mwConfigArgs );

app.run(cfg);
```

This would use the generated publisher Application class, to publish create a single Vehicle publisher, and publish state updates 50 times. The endpoint vector is a required parameter to define where to connect to the execution manager. In example code provided, these are passed in to applications using an -emEndpoints argument. See [Execution Manager](#) for more information.

Looking into the example publisher Application class in a little more detail:

```

public synchronized int run(TENA.Middleware.Configuration configuration)
    throws NetworkError, Timeout, Exception
{
    // Load any implementation libraries required
    // None required for object model Example-Vehicle-v1

    // Initialize the runtime, execution, and session
    runtime = TENA.Middleware.Runtime.init( configuration );
    execution = runtime.joinExecution( endpoints ); // Join the execution
    session = execution.createSession( "Example_Vehicle-Publisher_v1.1.1_Session" );

    // Initialize the SDO
    exampleVehicleServantFactory = Example.Vehicle.ServantFactory.create(session);
    exampleVehicleServant = ExampleVehicleUtility.createServant( exampleVehicleServantFactory, TENA.Middleware.
CommunicationProperties.Reliable);

    // Loop to update
    for ( int i = 1; i <= numberIterations; ++i ) {
        session.evokeMultipleCallbacks( microsecondsPerIteration ); // In this case, just being used to delay
        ExampleVehicleUtility.updateServant( exampleVehicleServant );
    }
}

```

Using the classes generated in the example code directly would simply publish arbitrary changes to the SDO state, as defined in the `ExampleVehicleUtility` class.. To change the code to publish data from some external source, it would be necessary to modify `org.Example.Vehicle_Publisher.ExampleVehicleUtility`. Modifying the existing `ExampleVehicleUtility` `createServant` and `updateServant` methods, would allow an application to create and update one or more servants.

```

public static Example.Vehicle.Servant createServant(
    Example.Vehicle.ServantFactory servantFactory, CommunicationProperties commProps)
    throws ExecutionNotConfiguredForBestEffort, Timeout, NetworkError, CannotContactExecutionManager
{
    System.out.println("Creating Example.Vehicle.Servant");

    Example.Vehicle.PublicationStateInitializer initializer = servantFactory.createInitializer();
    // Initialize required (and any desired optional) attributes
    initializer.set_name( "" );
    initializer.set_team( Example.Team.Team_Red );
    initializer.set_location( Example.Location.LocalClass.create( /* xInMeters */ 0.0d, /* yInMeters */ 0.0d ) );
    ...
    // Instantiate class to handle any remote method invocations (if needed, as defined by the object model)
    // None required for Example.Vehicle Servant (no remote methods)

    // Return a newly created servant
    return servantFactory.createServant( initializer, commProps );
}

public static void updateServant(
    Example.Vehicle.Servant exampleVehicleServant )
    throws Timeout, NetworkError
{
    // Get an Updater
    Example.Vehicle.PublicationStateUpdater exampleVehicleUpdater = exampleVehicleServant.createUpdater();

    // Change the value of the whatever desired state
    exampleVehicleUpdater.set_location( Example.Location.LocalClass.create( random.nextDouble(), random.
nextDouble() ) );
    exampleVehicleUpdater.set_team( Example.Team.Team_Blue );
    ...
    // Pass the updater in to the servant to modify the state atomically
    exampleVehicleServant.commitUpdater( exampleVehicleUpdater );
}

```

This shows how an `Initializer` and an `Updater` are used to create and update SDO Servants. ⚠ Note that it is important for the application to hold a reference to each created servant. When all references are dropped and the underlying instance is garbage collected, a notification of the SDO's destruction will be sent. Because Java underspecifies when an instance is garbage collected, there is also a method `Servant.releaseReference` to force destruction events to occur at a predictable time. When a servant is destroyed any subscribing applications will be notified of the destruction.

This generally shows how code to publish an SDO could be added to an existing application. Note that this is only example code showing the steps required. Some key things to note:

1. This implementation of run() blocks until all the updates have been sent. A more realistic application will need to define appropriate architecture to allow creation and updates to happen concurrently with other application activities.
2. It is important to hold a reference to the Runtime, Execution, and Session. If these are garbage collected (or if releaseReference is explicitly called) then any created Servants are also destroyed. The example Application holds these as member variables.
3. The Runtime instance is a singleton for an application. Multiple Executions or Sessions may be created depending on application requirements.

Using example subscriber code to add SDO subscription to an existing application.

This is very similar to the discussion above, which described adding SDO publication to an existing application. The `org.Example.Vehicle_Subscriber.Application` class, from the example code included with the JavaBinding distribution for the `Example,Vehicle,1` object model, provides a simple API:

```
class Application {  
    public Application();  
    public int microsecondsPerIteration;  
    public int numberofIterations;  
    public TENA.Middleware.EndpointVector endpoints;  
  
    public int run(TENA.Middleware.Configuration config);  
    public void shutdown();  
}
```

Using this from some existing application could be done following the example of the Main class used to start an Application.

```
import org.Example.Vehicle_Subscriber.Application;  
...  
String[] mwConfigArgs = ... // Arguments to configure middleware. Alternatively set configuration parameters  
directly on the Configuration  
TENA.Middleware.EndpointVector endpoints = new TENA.Middleware.EndpointVector();  
endpoints.add(new TENA.Middleware.Endpoint("myhostname:50000"));  
  
Application app = new Application();  
app.endpoints = endpoints;  
app.numberofIterations = 50;  
app.microsecondsPerIteration = 1000000;  
  
TENA.Middleware.Configuration cfg = new TENA.Middleware.Configuration( mwConfigArgs );  
  
app.run(cfg);
```

This would use the generated subscriber Application class, to subscribe to Vehicle SDOs, processing updates for a total of 50 seconds. The endpoint vector is a required parameter to define where to connect to the execution manager. In example code provided, these are passed in to applications using an `-emEndpoints` argument. See [Execution Manager](#) for more information.

Looking into the example subscriber Application class in a little more detail:

```

public synchronized int run(TENA.Middleware.Configuration configuration)
    throws NetworkError, Timeout, Exception
{
    // Load any implementation libraries required
    // None required for object model Example-Vehicle-v1

    // Initialize the runtime, execution, and session
    runtime = TENA.Middleware.Runtime.init( configuration );
    execution = runtime.joinExecution( endpoints ); // Join the execution
    session = execution.createSession( "Example_Vehicle-Publisher_v1.1.1_Session" );

    // Set up Subscription, adding observers, and then subscribe
    // It may be desirable to hold the Subscription as a member variable to allow use of Subscription.
    unsubscribeAndRelease.
    Example.Vehicle.Subscription exampleVehicleSubscription = new Example.Vehicle.Subscription();
    // Add any number of Observers
    exampleVehicleSubscription.addObserver( new ExampleVehiclePrintingObserver() );
    Example.Vehicle.Subscription.subscribe(session, exampleVehicleSubscription);

    // Loop to handle any subscription events
    for ( int i = 1; i <= numberIterations; ++i ) {
        session.evokeMultipleCallbacks( microsecondsPerIteration ); // Process events for some amount of time
        ExampleVehicleUtility.updateServant( exampleVehicleServant );
    }
}

```

This code uses the example code class `ExampleVehiclePrintingObserver` to handle any discovery, update or destruction events for Vehicle SDOs.

```

public class ExampleVehiclePrintingObserver extends Example.Vehicle.AbstractObserver {

    public ExampleVehiclePrintingObserver() {
    }

    public void discoveryEvent( Example.Vehicle.Proxy proxy, Example.Vehicle.PublicationState pubState )
    {
        System.out.println( "Discovery : Example.Vehicle " + pubState.get_name());
        proxy.releaseReference(); // release proxy to allow unsubscribe to be independent of garbage collection
    }

    public void stateChangeEvent( Example.Vehicle.Proxy proxy, Example.Vehicle.PublicationState pubState )
    {
        System.out.println( "StateChange : Example.Vehicle " + pubState.get_name() + " " +
                           pubState.get_location().get_xInMeters() + pubState.get_location().get_yInMeters());
        proxy.releaseReference(); // release proxy to allow unsubscribe to be independent of garbage collection
    }

    public void destructionEvent(
        Example.Vehicle.Proxy proxy, Example.Vehicle.PublicationState pubState )
    {
        System.out.println( "Destruction : Example.Vehicle " + pubState.get_name());
        proxy.releaseReference(); // release proxy to allow unsubscribe to be independent of garbage collection
    }
}

```

By passing resources to a suitable Observer constructor, the event methods could be modified to update displays, send data out through other APIs, log into databases, etc. Multiple Observers may be added to a Subscription, and they may be added or removed while the Subscription is in use. There is an issue with how the TENA Middleware handles subscriptions – as long as a proxy is held, events may still be delivered even if `Subscription.unsubscribe` is invoked. As a result, it is recommended that each created Proxy is explicitly released within the event methods. The exception to this rule would be a case where an application desires to interact with the discovered proxies directly, say by holding them in a Map. In this case, the Proxy instances would be entered into the Map in the `DiscoveryEvent`, and `releaseReference()` would not be called. The other event methods WOULD still call `releaseReference()`, however – each passed in Proxy is a new proxy instance even if it refers to the same underlying SDO.

SDO Creation

SDO Creation

A publishing application creates SDO Servants using a `ServantFactory` that requires an `Initializer` holding initial attribute values and a `RemoteMethodsInterface` implementation, if remote methods exist. Subscribing application discover an SDO Proxy that corresponds to the created servant.

Description

A TENA Middleware application acting as a publisher creates **SDO Servants** according to their application needs. The publishing application is required to use an `Initializer` object to provide the initial attribute values for the created SDO. Publishing applications can then update the SDO state as necessary. Additionally, any remote methods defined for the SDO type must be implemented by the publishing application. Details on the SDO creation process can be found in the [SDO Initializer](#) and [SDO Servant](#) documentation pages.

When an application creates an SDO servant, it can be configured to support different communication properties for the SDO updates. Currently, SDO updates can be sent using either a "Reliable" or "Best Effort" communication protocol. Note that the SDO discovery and destruction (as well as the Advanced Filtering scope updates) are always sent using the Reliable transport. Additional information can be found in the [SDO update communication properties documentation page](#).

Subscribing application that have indicated an interest in the SDO type that was created will receive a discovery notification that contains an `SDO Proxy`. The SDO proxy corresponds to the particular SDO servant, and the subscribing application can read the state and invoke `remote methods`. The discovery event includes both the normal SDO state and the values of any const attributes; subsequent updates will include data only for the non-const attributes.

SDO servants are destroyed by the publishing application when all of the smart pointers that hold a reference to the particular servant are released. This is done by letting the smart pointer go out of scope, or through the explicit `reset` method on the smart pointer. When the SDO servant is destroyed by the middleware, a notification is sent to all subscribing applications that are holding a corresponding SDO proxy. This process is explained in the [SDO destruction documentation page](#).

C++ API Reference and Code Examples

Creating an SDO Servant is a four-step process:

1. Create the `ServantFactory`.
2. Create the `PublicationStateInitializer`.
3. Create the remote methods implementation object.
4. Create the Servant.

ServantFactory

The object model definition library provides a `Servant` class for each SDO; this `Servant` is created using a provided a factory class called `ServantFactory`:

```

class ServantFactory
    : TENA::Middleware::Utils::noncopyable
{
...
public:
    static
    ServantFactoryPtr const
    create( ::TENA::Middleware::SessionPtr session );

    virtual
    ~ServantFactory();

    ::std::unique_ptr< PublicationStateInitializer >
    createInitializer();

...
    ServantPtr const
    createServant(
        ::std::unique_ptr< RemoteMethodsInterface > pRemoteMethods, // only required for SDO's that define remote
methods.
        PublicationStateInitializer const & initializer,
        ::TENA::Middleware::CommunicationProperties const & commProperties,
        ::TENA::Middleware::ConcurrencyProperties const & concProperties
        = ::TENA::Middleware::Blocking );
...
};


```

Release 6.0.5 Update — Prior to middleware version 6.0.5, the type `boost::noncopyable` was used instead of the `TENA::Middleware::Utils::noncopyable` type.

Release 6.0.2 Update — An alternative `create` method was added with release 6.0.2 of the middleware in which the `SessionPtr` argument is used, versus the `Session` reference (i.e., `*pSession`) that was used in the previous releases. Users are encouraged to use the `SessionPtr` interface. See [MW-4286](#) for more details.

A `ServantFactory` instance is created with a call to the static `ServantFactory::create` function, which takes a reference to the `TENA::Middleware::Session`:

```

Example::Tank::ServantFactoryPtr tankServantFactory =
    Example::Tank::ServantFactory::create( pSession );

```

Release 6.0.2 Update — An alternative `create` method was added with release 6.0.2 of the middleware in which the `SessionPtr` argument is used, versus the `Session` reference (i.e., `*pSession`) that was used in the previous releases. Users are encouraged to use the `SessionPtr` interface. See [MW-4286](#) for more details.

Next, the SDO's `PublicationStateInitializer` is created using the `ServantFactory::createInitializer` function:

```

std::unique_ptr< Example::Tank::PublicationStateInitializer > initializer =
    tankServantFactory->createInitializer();

```

The initializer's mutator functions are used to set the initial SDO state:

```

initializer->set_name( "MyTankName" );
initializer->set_team( Example::Team_Red );
initializer->set_location( Example::Location::create( 0.0, 0.0 ) );
initializer->set_pDriver( Example::Soldier::SDOpointerPtr() );
initializer->set_passengers( std::vector< Example::Soldier::SDOpointerPtr >() );

```

For more information on initializer usage, see the [SDO Initializer documentation page](#).

Next, if the SDO has defined remote methods, the concrete remote methods implementation object is created. This is an instance of a class derived from the SDO type-specific `RemoteMethodsInterface` abstract class in the object model definition library. Application developers are required to create this implementation class to provide the appropriate behavior for the SDO methods. The publishing application provides the implementation of the [SDO's remote methods](#) using this derived class as an argument in the `createServant` invocation.

```
std::unique_ptr< Example::Tank::RemoteMethodsInterface > tankRemoteMethods(
    new MyApplication::Example_Tank::RemoteMethodsImpl );
```

Finally, `ServantFactory::createServant` creates the servant:

```
Example::Tank::ServantPtr tankServant =
    tankServantFactory->createServant(
        std::move( tankRemoteMethods ),
        *initializer,
        TENA::Middleware::Reliable );
```

Note that ownership of `tankRemoteMethods` is transferred to the `createServant` function via the use of `unique_ptr` with `std::move()`. After `createServant` returns, `tankRemoteMethods` will be null.

The third parameter of `createServant` is the "communication properties"; its use is detailed in the [SDO communication properties documentation page](#).

The fourth parameter of `createServant` is the "concurrency properties"; its use is detailed in the [SDO concurrency properties documentation page](#).

Java API Reference and Code Examples

Creating an SDO Servant is a multi-step process:

1. Create the `ServantFactory`.
2. Create the `PublicationStateInitializer`.
3. Create the remote methods implementation object (if required).
4. Create the Servant.

ServantFactory

The object model java binding for an object model provides a Servant class for each SDO; this Servant is created using a provided a factory class called `ServantFactory`:

```
class ServantFactory
{
    ...
    public static ServantFactory create( TENA.Middleware.Session session );
    public PublicationStateInitializer createInitializer();
    public Servant createServant(
        AbstractRemoteMethods remoteMethods, // only required for SDO's that define remote methods.
        PublicationStateInitializer initializer,
        TENA.Middleware.CommunicationProperties commProperties);
    ...
};
```

See [example API](#) from the Example-Vehicle object model.

A `ServantFactory` instance is created with a call to the static `ServantFactory.create` function, which takes a `TENA.Middleware.Session`:

```
Example.Vehicle.ServantFactory vehicleServantFactory = Example.Tank.ServantFactory.create( session );
```

Next, an SDO `PublicationStateInitializer` is created using the `ServantFactory.createInitializer` function:

```
Example.Vehicle.PublicationStateInitializer initializer = vehicleServantFactory.createInitializer();
```

The initializer's mutator methods are used to set the initial SDO state:

```
initializer->set_name( "MyName" );
initializer->set_team( Example.Team.Team_Red );
...
```

For more information on initializer usage, see the [SDO Initializer documentation page](#).

Next, if the SDO has defined remote methods, a concrete remote methods implementation object is created. This is an instance of a class derived from the SDO type-specific `AbstractRemoteMethods` abstract class in the object model java binding library. Application developers are required to extend this class to provide an implementation with the appropriate behavior for the SDO methods. The publishing application provides the implementation of the SDO's **remote methods** using this derived class as an argument in the `createServant` invocation.

 **Only SDOs with remote methods will require this.**

Finally, `ServantFactory::createServant` creates the servant, passing an `AbstractRemoteMethods` implementation, if appropriate.

```
tankServant = tankServantFactory.createServant(
    new org.Example.Tank_Publisher.ExampleTankRemoteMethodsImp(),
    *initializer,
    TENA::Middleware::Reliable );
```

Note that the `tankRemoteMethods` should not be reused or shared among multiple servants. The underlying `AbstractRemoteMethods` class will be invalidated to cause a `NullPointerException` if reuse is attempted. This is to ensure the underlying threading requirements of the TENA Middleware are satisfied.

The other parameters of `createServant` are the "communication properties" and "concurrency properties" for the servant. For more information, see [SDO communication properties documentation page](#) and [SDO concurrency properties documentation page](#).

Related Topics

Additional topics related to SDO creation can be found in the following pages:

- [SDO Servant](#) — The SDO Servant encapsulates the publisher-side implementation of the Stateful Distributed Object.
- [SDO Initializer](#) — SDO initializers are used to set the initial attribute values for the object when it is created.
- [SDO Communication Properties](#) — SDO servants can publish updates using either "reliable" or "best effort" distribution strategies.
- [SDO Concurrency Properties](#) — SDO servants can create Publication State Updaters using either "blocking" or "non-blocking" updater strategies.

SDO Communication Properties

SDO Communication Properties

SDO servants can publish updates using either "reliable" or "best effort" distribution strategies for state updates.

Description

SDO servants can publish updates using either a "[reliable](#)" (TCP) or a "[best effort](#)" (UDP multicast) distribution strategy. Reliable transport uses a separate TCP socket between the publisher and every subscriber interested in the particular SDO instance. The TCP mechanism provides a level of reliability in the communication, detecting and reporting communication errors if they occur. Best effort transport uses UDP multicast which utilizes the network devices to replicate the network packets to the interested subscribers. The UDP multicast mechanism is unreliable and there is no notification if the state update was not delivered to the subscriber.

The publishing application decides the communication transport at servant creation time using the third argument to `ServantFactory::createServant`. Therefore, an SDO servant instance is created to use a particular communication transport that can't be changed after creation.

Note that the communication transport used for the SDO discovery and destruction notifications are always sent using the reliable transport. The communication properties used in the servant creation only controls the transport of the state updates for that particular SDO servant.

Usage Considerations

Whether to use reliable or best effort transport for SDO updates involves a performance trade-off with guaranteed delivery (or at least notification of delivery failure). Using best effort across wide area networks in which there are many subscribing applications will result in less network traffic than the reliable transport. There is also less processing code for the publisher with best effort because there is only a single network write operation, as compared to a single network write operation for every subscriber in the case of reliable transport.

The downside with best effort transport is that the publisher has no guarantee that the updates are being received by all subscribers. If there is a communication problem, that fault will (most likely) be undetected by the publishing and subscribing application.

Users are recommended to design their applications to have configuration control of whether reliable or best effort transport are used for their SDOs (and Messages). Reliable transport should be used unless there is empirical evidence that the equipment will not support this form of communication.

When using best effort, it may be prudent for publishing applications to periodically update the SDO state when there has not been a recent update (e.g., every several minutes). This "heartbeating" may help in situations where a subscriber has missed a previous best effort update.

C++ API Reference and Code Examples

`TENA::Middleware::CommunicationProperties` is a C++ enum defining the enumerants `TENA::Middleware::Reliable` and `TENA::Middleware::BestEffort`. As seen in the description of the [SDO creation process](#), one of these values is used as the third argument to `ServantFactory::createServant`:

```
Example::Tank::ServantPtr tankServant =
    tankServantFactory->createServant(
        std::move( tankRemoteMethods ),
        *initializer,
        TENA::Middleware::Reliable );
```

Java API Reference and Code Examples

`TENA.Middleware.CommunicationProperties` is a Java enumeration defining the enumerants `TENA.Middleware.CommunicationProperties.Reliable` and `TENA.Middleware.CommunicationProperties.BestEffort`. As seen in the description of the [SDO creation process](#), one of these values is used as the final argument to `ServantFactory::createServant`:

```
Example.Tank.Servant tankServant =  
    tankServantFactory.createServant(  
        tankRemoteMethods,  
        initializer,  
        TENA.Middleware.CommunicationProperties.Reliable );
```

SDO Concurrency Properties

SDO Concurrency Properties

SDO servants can create Publication State Updaters using either "blocking" or "non-blocking" updater creation strategies for state updates.

Description

SDO servants can be [updated](#) using a publication state updater. For a given SDO instance, the application programmer can choose whether or not to allow more than one instance of a publication state updater to be in existence at a time.

By default, attempting to create a publication state updater for an SDO will block if another updater for the given SDO instance already exists.

Usage Considerations

In most cases, creating an SDO that only supports a single publication state updater at a time is desirable to mitigate confusion. That is why blocking on an attempt to create more than one instance of a publication updater at a time is the default.

However, creating more than one instance of a publication updater at a time for a single instance of an SDO can be used to good advantage in certain circumstances. For example, it would be possible to make ready a sequence of publication state updates in advance and then easily commit them at a future time.

C++ API Reference and Code Examples

`TENA::Middleware::ConcurrencyProperties` is a C++ enum defining the enumerants `TENA::Middleware::Blocking` and `TENA::Middleware::Nonblocking`. As seen in the description of the [SDO creation process](#), one of these values can be used as the forth argument to `ServantFactory::createServant`:

```
Example::Tank::ServantPtr tankServant =
    tankServantFactory->createServant(
        std::move( tankRemoteMethods ),
        *initializer,
        TENA::Middleware::Reliable,
        TENA::Middleware::Nonblocking );
```

Java API Reference and Code Examples

`TENA.Middleware.ConcurrencyProperties` is a Java enum defining the enumerants `TENA.Middleware.ConcurrencyProperties.Blocking` and `TENA.Middleware.ConcurrencyProperties.Nonblocking`. As seen in the description of the [SDO creation process](#), one of these values can be used as the final argument in the alternate signature to `ServantFactory::createServant`:

```
Example.Tank.Servant tankServant =
    tankServantFactory.createServant(
        tankRemoteMethods,
        initializer,
        TENA.Middleware.CommunicationProperties.Reliable,
        TENA.Middleware.ConcurrencyProperties.Nonblocking );
```

SDO Initializer

SDO Initializer

When a publishing application attempts to create an SDO (Stateful Distributed Object), an Initializer is required for the `create` method. The Initializer holds all of the initial values for the object, allowing the developer to set the attribute values individually and ensuring that as soon as the object is created all of the attribute values are set. This mechanism avoids type-safety and combinatorial problems with attempting to set all of the attribute values in class constructors and avoids the inherent race conditions if the values were individually set after the object is constructed.

Description

When a Stateful Distributed Object (SDO) is created, that object is immediately available to subscribing applications. In order to ensure that publishing applications only create SDOs with appropriate attribute values, an "Initializer" mechanism enforces the creation of SDOs with any required attributes set to specific values provided by the application developer. Any attributes defined as "const" can only be set through the Initializer, ensuring that these values are only set once.

The generated `PublicationStateInitializer` class is defined within an object model definition and operates in a similar manner to the SDO's publication state updater. The publishing application creates an `PublicationStateInitializer` for the appropriate SDO type, then sets the attribute values using `set_` methods on the `PublicationStateInitializer`. All required attributes must be set to support the creation of an SDO from that `PublicationStateInitializer`. Optional attributes may be set on the `PublicationStateInitializer` if desired.

Once the `PublicationStateInitializer` has been used to set the attribute values, it can be provided to the SDO servant factory to create an SDO servant. The same `PublicationStateInitializer` can be used to create multiple SDOs (which will begin with the same attribute values).

Usage Considerations

Developers of publishing applications need to review the object models for SDO types that the application publishes to ensure that the required attribute values are available, and whether any optional values will be used. Attempting to create an SDO servant with a `PublicationStateInitializer` that has not set all of the required attributes will result in a runtime exception when the creation of an SDO is attempted.

All available initial attribute values should be set with the initializer, even if they are not required. This will ensure that subscribing applications will discover objects with all appropriate attribute values defined. If the publishing application was to create the object with unset attribute values and then performed updates for other values, a subscribing application would discover and process the object with an incomplete state.



Initializers Can Be Re-Used

The `PublicationStateInitializer` returned from servant factory is wrapped in a C++ `unique_ptr` and the `PublicationStateInitializer` is passed by reference to the `createServant` method. This allows the application to reuse the `PublicationStateInitializer` object if necessary. The effect of the `unique_ptr` is to pass ownership to the calling scope. Users can extend the lifetime of this resource by taking ownership from the `unique_ptr`; otherwise, the `PublicationStateInitializer` will be destroyed when the `unique_ptr` goes out of scope.

Release 6.0.4 corrected a deficiency in which an `unset` method did not exist on the `Initializer` case to support the use case where an application needs to unset an optional value (see [MW-4763](#)).

C++ API Reference and Code Examples

Similar to SDO `PublicationStateUpdaters`, the `PublicationStateInitializers` are type-specific and generated through the object model build process. A publishing application needs to include the `<TypeName>.h` header file for the particular SDO type (e.g., `#include <Example/Vehicle.h>`).

`PublicationStateInitializers` are wrapped in a C++ `unique_ptr` when created within a publishing application. The code fragment below illustrates the creation of an initializer for the `Example::Vehicle` SDO type.

```
// Initializer returned within unique_ptr and passed to createServant as
// a reference (using "*" syntax) to allow reuse of the initializer
// in other calls to createServant.
std::unique_ptr< Example::Vehicle::PublicationStateInitializer >
    initializer( pServantFactory->createInitializer() );
```

As noted in the code comment above, the initializer is passed into the `createServant` call by reference, using the "*" syntax. Passing this argument by reference is used for efficiency purposes because the middleware needs to create a copy of the initializer data and it permits the application code to reuse the initializer for creating multiple objects.

Once the initializer is created, the application code is required to set values for all of the required attributes and any "const" attributes that need to be set. Any optional attributes values that are going to be set should also be set through the initializer so that the initializer contains all of the attribute values that the object should have when it is first created. The code fragment below illustrates the setting of attribute values with the initializer.

```
initializer->set_name( TENA::Middleware::ArbitraryValue< std::string >() );
initializer->set_team( TENA::Middleware::ArbitraryValue< Example::Team >() );
/// \todo In a real application, instances of \c ArbitraryValue<T>
///      should be replaced with values that are sensible for the
///      particular application.
Example::Location::LocalClassPtr location(
    Example::Location::create(
        /* xInMeters */ TENA::Middleware::ArbitraryValue< TENA::float64 >(),
        /* yInMeters */ TENA::Middleware::ArbitraryValue< TENA::float64 >() ) );
initializer->set_location( location );
```

Once the initializer has been properly set with appropriate attribute values, the SDO servant can be created with the `createServant` method. The initializer is passed into this call as a reference by using the "*" notation as shown in the code fragment below.

```
// Create a Example::Vehicle::Servant, returned in a smart pointer.
// The caller must hold a reference or the Servant will be destroyed
Example::Vehicle::ServantPtr pServant(
    pServantFactory->createServant(
        *initializer,
        communicationProperties ) );
```

Java API Reference and Code Examples

Similar to SDO `PublicationStateUpdaters`, the `PublicationStateInitializers` are type-specific and generated through the object model build process.

The code fragment below illustrates the creation of an initializer for the `Example.Vehicle` SDO type.

```
Example.Vehicle.PublicationStateInitializer initializer = servantFactory.createInitializer();
```

Once the initializer is created, the application code is required to set values for all of the required attributes and any "const" attributes that need to be set. Any optional attributes values that are going to be set should also be set through the initializer so that the initializer contains all of the attribute values that the object should have when it is first created. The code fragment below illustrates the setting of attribute values with the initializer.

```
initializer.set_name( "MyName" );
initializer->set_team( Example::Team::Team_Red );

initializer->set_location( Example::Location::create( /* xInMeters */ 0.0d, /* yInMeters */ 0.0d ); );
```

Once the initializer has been properly set with appropriate attribute values, the SDO servant can be created with the `createServant` method.

```
// Create an Example.Vehicle.Servant.
// The caller must hold a reference or the Servant will be destroyed.
// Allowing the instance to be garbage collected, or explicitly calling
// releaseReference() will send a destruction event.

Example.Vehicle.Servant servant =
    servantFactory.createServant( initializer, communicationProperties );
```

SDO Servant

SDO Servant

An SDO (Stateful Distributed Object) Servant is created from a type-specific Servant Factory by the publishing application and initialized with an SDO Initializer. Changes to the SDO state (i.e., attribute values) are disseminated to the SDO Proxies held by subscribing applications. The Servant will also include the implementation for the SDO's remote methods (if any exist).

Description

The SDO Servant encapsulates the publisher-side implementation of the Stateful Distributed Object. The servant is a higher-level abstraction that couples the concepts of distributed object state and remote methods associated with that data. The servant is created from a type-specific Servant Factory by the publishing application. At creation-time, the servant is initialized using an [SDO Initializer](#).

The servant is, literally, a server of object state and a provider of services in the form of remote methods. Subscribing applications are made aware of it when it is created; they are informed of [changes to its state](#); they are able to invoke its [remote methods](#); and ultimately they are informed when it is no longer available (i.e., [destroyed](#)).

Usage Considerations

In previous versions of the Middleware, servant creation and activation were separate steps. In version 6 of the Middleware, activation happens automatically as part of the creation process. This means that once `ServantFactory::createServant` returns a `ServantPtr`, the initial (discovery) update has been sent to subscribers.

C++ API Reference and Code Examples

The following code fragment from the `Example::Vehicle` example application illustrates how an SDO Servant is created and updated. The example below shows the use of a configuration parameter to control whether the created servant uses Reliable or Best Effort communication. It is recommended that applications provide this type of configuration control since it is common to be able to need to change the communication properties.

```

// Messages and state updates can be disseminated either using Reliable
// (i.e., TCP/IP) or BestEffort (i.e., UDP/IP multicast).
TENA::Middleware::CommunicationProperties communicationProperties(
    appConfig["bestEffort"].isSet() ?
        TENA::Middleware::BestEffort :
        TENA::Middleware::Reliable );

// Create a ServantFactory to create Example::Vehicle objects.
Example::Vehicle::ServantFactoryPtr p_ExampleVehicleServantFactory(
    Example::Vehicle::ServantFactory::create( pSession ) );

// Initializer returned within unique_ptr and passed to createServant as
// a reference (using "*initializer") to allow reuse of the initializer
// in other calls to createServant.
std::unique_ptr< Example::Vehicle::PublicationStateInitializer >
    initializer( pServantFactory->createInitializer() );

initializer->set_name( TENA::Middleware::ArbitraryValue< std::string >() );
initializer->set_team( TENA::Middleware::ArbitraryValue< Example::Team >() );

/// \todo In a real application, instances of \c ArbitraryValue<T>
///       should be replaced with values that are sensible for the
///       particular application.
Example::Location::LocalClassPtr location(
    Example::Location::create(
        /* xInMeters */ TENA::Middleware::ArbitraryValue< TENA::float64 >(),
        /* yInMeters */ TENA::Middleware::ArbitraryValue< TENA::float64 >() ) );
initializer->set_location( location );

// Create a Example::Vehicle::Servant, returned in a smart pointer.
// The caller must hold a reference or the Servant will be destroyed
Example::Vehicle::ServantPtr pServant(
    pServantFactory->createServant(
        *initializer,
        communicationProperties ) );

```

Release 6.0.2 Update — An alternative `create` method was added with release 6.0.2 of the middleware in which the `SessionPtr` argument is used, versus the `Session` reference (i.e., `*pSession`) that was used in the previous releases. Users are encouraged to use the `SessionPtr` interface. See [MW-4286](#) for more details.

Note that the above code fragment uses the `ArbitraryValue` mechanism that generates values to support the building of example applications without modification. Users are required to replace the use of `ArbitraryValue` with appropriate behavior for non-example applications.

A code fragment that illustrates update servant attributes is shown below.

```

// Get the updater, that is held in an unique_ptr
std::unique_ptr< Example::Vehicle::PublicationStateUpdater >
p_VehicleUpdater(
    p_VehicleServant->createUpdater() );

// Change the value of the string attribute "name"
p_VehicleUpdater->
    set_name( updateLabel );

// Update the enumeration
p_VehicleUpdater->
    set_team(
        TENA::Middleware::ArbitraryValue< Example::Team >( Example::Team_Red ) );

// create a LocalClass instance for attribute location
Example::Location::LocalClassPtr p_VehicleUpdater_location_LocalClass(
    Example::Location::create(
        /* xInMeters */ TENA::Middleware::ArbitraryValue< TENA::float64 >(),
        /* yInMeters */ TENA::Middleware::ArbitraryValue< TENA::float64 >() ) );

// Change the value of the p_VehicleUpdater's
// LocalClass attribute location
p_VehicleUpdater->
    set_location(
        p_VehicleUpdater_location_LocalClass );

// Pass the updater in to the servant to modify the state atomically
p_VehicleServant->commitUpdater(
    std::move( p_VehicleUpdater ) );

```

Java API Reference and Code Examples

The following code fragment from the Example.Vehicle example application illustrates how an SDO Servant is created and updated.

```

// Messages and state updates can be disseminated either using Reliable
// (i.e., TCP/IP) or BestEffort (i.e., UDP/IP multicast).

// Create a ServantFactory to create Example.Vehicle objects.
Example.Vehicle.ServantFactory vehicleServantFactory = Example.Vehicle.ServantFactory.create( session );

Example.Vehicle.PublicationStateInitializer initializer = vehicleServantFactory->createInitializer();

initializer.set_name( "MyName" );
initializer.set_team( Example.Team.Team_Red );
initializer.set_location( Example.Location.create( 0.0d, 0.0d ) );

// Create an Example::Vehicle::Servant. A reference to the returned servant must be held.
// When the servant is garbage collected or explicitly destroyed using releaseReference, the
// SDO will notify subscribers of its destruction. The example below shows the use of
// reliable communications over TCP/IP.

vehicleServant = vehicleServantFactory.createServant(
    initializer,
    TENA.Middleware.CommunicationProperties.Reliable );

```

A code fragment that illustrates updating servant attributes is shown below.

```
// Get an updater
Example.Vehicle.PublicationStateUpdater vehicleUpdater = vehicleServant.createUpdater();

// Change the value of the string attribute "name"
vehicleUpdater.set_name( "updated name" );

// Update the enumeration
vehicleUpdater.set_team( Example.Team.Team_Blue ) ;

// Change the value of the vehicleUpdater's
// LocalClass attribute location
vehicleUpdater.set_location( Example::Location::create( 0.0d, 0.0d ) );

// Pass the updater in to the servant to modify the state atomically
vehicleServant.commitUpdater( vehicleUpdater );
```

SDO Destruction

SDO Destruction

When a publisher destroys an SDO Servant, subscribers holding a corresponding SDO Proxy receive a destruction event via their observers. Observers are associated with an SDO subscription to receive notification of subscription events (e.g., discovery, update, destruction). SDO proxies (actually the `ProxyPtrs`) are "invalidated" after notification of the destruction event.

Description

When a publishing application causes the last `SDO ServantPtr` (associated with a particular instance) to go out of scope or if the pointer is `reset`, the SDO servant is destroyed. Normally, there will only be a single `ServantPtr`, but it is possible that multiple pointers could exist that refer to the same servant.

As part of the servant destruction process, the middleware will send notifications to all subscribing applications that are holding a SDO proxy corresponding to the destroyed SDO servant. The middleware associated with the subscribing applications will create a callback object to deliver the destruction notification. When the destruction callback is executed, any `Observers` created by the subscribing application for the subscription for this SDO will have the `destructionEvent` method invoked allowing the application to take appropriate action in response to the SDO destruction.

During the `Observer destructionEvent` method, the subscribing application will be provided a copy of the final state of the SDO in case the application needs to perform processing based on the final state. Note that the subscribing application will need to use the [middleware evoke callback services](#) in order for the destruction notification to occur.

After the destruction callback has been processed, the `ProxyPtrs` associated with the destroyed SDO will be invalidated. Any attempt to use the `ProxyPtrs` after invalidation will result in a runtime exception. An application can test if a `ProxyPtr` has been invalidated with the `isValid` method on the pointer, but application developers are encouraged to let go of `ProxyPtrs` after they have received the destruction notification.

As described above, there will be some time between when the publishing application destroys the SDO servant and the subscribing application receives notification of the destruction. During this time, any SDO remote method invocations will result in a runtime exception. Due to this situation (along with general fault handling), subscribing applications are strongly encouraged to use appropriate exception handling around all SDO remote method invocations.

Usage Considerations

Subscribing applications may be required to perform specific processing in response to an SDO destruction notification. For example, a display application may need to remove an icon representing the SDO from the display. In other cases, the application may need to modify internal data structures that maintain application information related to discovered lists. The necessary application code that needs to be written to handle the SDO destruction processing should be put in an `Observer` class that inherits from the appropriate automatically generated `AbstractObserver` class, specifically with the application code put in the `destructionEvent` method implementation of this class.

C++ API Reference and Code Examples

The declaration of the example `AbstractObserver` class and `destructionEvent` method for the `Example::Tank` SDO type is shown below. A subscribing application would define their own `Observer` class that inherits from this class, and at a minimum, implement the `destructionEvent` method to contain the appropriate application code to handle the SDO destruction. Note that multiple `Observers` can be attached to a subscription, so that from a code structure perspective, if the application destruction processing requires separate and orthogonal behavior, it would be best to create separate `Observer` classes.

```

class Example::Tank::AbstractObserver
{
...
public:
    /*! \brief Invoked each time a Proxy object is destructed.
     *
     * Observer classes that derive from this class should implement this
     * method to perform whatever actions are desired each time a
     * Proxy object is destructed.
     *
     * Unless this method is overridden, the "do-nothing" implementation
     * is used.
     *
     * \param[in] pProxy A reference-counted "smart" pointer to the Proxy
     * that was discovered.
     * \param[in] pPubState A reference-counted "smart" pointer to the
     * PublicationState of the Proxy at the time it was destructed. Note
     * well that by the time this method is invoked, the current
     * PublicationState of the Proxy may well be different. That is, the
     * Servant may have performed an update in between when the
     * Middleware "noted" the destruction and when the application code
     * "finally got around to" calling this method.
     */
    virtual
    void
    destructionEvent(
        ProxyPtr const & pProxy,
        PublicationStatePtr const & pPubState );
...
};

```

Java API Reference and Code Examples

The declaration of the example `AbstractObserver` class and `destructionEvent` method for the `Example.Tank` SDO type is shown below. A subscribing application would define an `Observer` class extending this class, and at a minimum, implement the `destructionEvent` method to contain the appropriate application code to handle the SDO destruction. Note that multiple `Observers` can be attached to a subscription, so that from a code structure perspective, if the application destruction processing requires separate and orthogonal behavior, it would be best to create separate `Observer` classes.

```

package Example.Tank;

public class AbstractObserver {
...
/** Invoked each time a Proxy is destroyed */
public void destructionEvent( Proxy proxy, PublicationState pubstate );
...
};

```

⚠ Note: because of details of java garbage collection and how the underlying C++ middleware reference counts pointers, it is generally advisable to call `pProxy.releaseReference()` in observer event handling methods after the proxy is no longer needed. Without this step, an "instance subscription" may be maintained even after unsubscribing, because of the indeterminate nature of Java garbage collection.

SDO ID

SDO ID

SDOs (Stateful Distributed Objects) are provided an execution unique identifier (i.e., `ObjectID`) by the middleware that applications can use for identification purposes.

Description

SDOs (Stateful Distributed Objects) are provided an execution unique identifier (i.e., `ObjectID`) by the middleware that applications can use for identification purposes. Although the SDO identifier is an opaque object, methods exist to support obtaining a numerical or string representation if necessary.

The SDO identifier is similar to the other middleware generated identifiers. Refer to [middleware IDs documentation page](#) for details regarding the behavior and use of these identifiers.

SDO Inheritance and Polymorphism

SDO Inheritance and Polymorphism

The TENA Meta-Model supports the concept of inheritance for SDOs (Stateful Distributed Objects). An SDO type that inherits from a base type will contain the base type attributes and methods. Using SDO inheritance can be helpful in situations where a new derived type can be introduced (with additional attributes and/or methods) without affecting existing applications that are already deployed using the base type. SDO inheritance can also be used to support type-based filtering.

Description

A generalized discussion of inheritance within the TENA Meta-model can be found on the [Meta-model inheritance documentation page](#). This page describes some important limitations of inheritance as implemented for TENA. In summary:

- Multiple inheritance (inheriting from multiple classes) is not supported.
- Derived classes cannot add attributes with the same name or methods with the same signature as a base class.
- Automatic delegation of method implementation to base classes is not performed.

Another way of stating the last issue is that derived classes must supply implementation code for all methods, including inherited ones. The derived class implementation is free to delegate any operations to its base class.

Uses for SDO Inheritance

SDO inheritance is commonly used to support object model evolution when there are existing deployed applications that are built with a particular SDO type. If some additional users are interested in adding attributes and/or methods to the existing SDO type, then creating a new version of that SDO type will require those deployed attributes to re-compile/re-link their applications with the new object model version, regardless of whether they care about the new attributes and/or methods.

In this situation, the use of inheritance should be considered as a mechanism to add the new attributes and/or methods to a derived type that will not require any modification to existing applications using the base type. This is useful when only a subset of applications are interested in the new attributes and/or methods. In some cases, developing a new object model version to better capture the requirements of the user community may be necessary and therefore these types of changes should be incorporated in a new version of the object model type.

Another use for SDO inheritance is to create separate type to support type-based filtering. For example, the `Platform` SDO type could be used as base class for `AirPlatform`, `GroundPlatform`, `SeaPlatform` types. Subscribing applications can then only subscribe to the particular types that they are interested in. Unfortunately, creating SDO types solely based on anticipated filtering considerations may not prove to be appropriate for all events. Using [Advanced Filtering](#) is a better solution for filtering, but does require more complexity and all applications to use advanced filtering correctly.

Inheritance Considerations with Subscription

In the case of inheritance, an application subscribing to a inherited type of the SDO Servant will discover an SDO Proxy corresponding to the subscribed (inherited) type. Any attributes or methods that only exist in the derived SDO Servant type will not be available through the inherited type's SDO Proxy. For example, consider the `Example::Vehicle` and derived `Example::Tank` SDO Classes shown below.

```
class Vehicle
{
    string name;
    Team team;
    Location location;
};

...
class Tank : extends Vehicle
{
    void loadSoldier( in Soldier * pSoldier ) raises ( CannotLoadSoldier );
    void unloadSoldier( in Soldier * pSoldier );
    void driveTo( in vector < Location > wayPoints );
    void fireGun( in Location targetLocation );

    optional float32 damageInPercent;
    vector < Soldier * > passengers;
};
```

In the above object model example, a `Tank` publishing application with a `Vehicle` subscribing application will result in a `Vehicle` `ProxyPtr` discovered. The subscribing application will then not have access to the `Tank` methods and added attributes (i.e., `damageInPercent`, `passengers`). This subscription behavior is referred to as "slicing", since the derived attributes and methods are sliced from the discovered proxy. This slicing behavior is expected because the subscribing application may not be aware of the definition of the derived class.

If an application subscribes to multiple SDO types in the SDO Servant's inheritance type hierarchy, the subscribing application will discover an SDO Proxy for each matching subscription. Using the object model example above, if the subscribing application subscribed to both `Vehicle` and `Tank`, then two SDO Proxies will be discovered (one `Vehicle` and one `Tank`) that correspond to the same SDO servant. Applications are responsible for properly managing multiple proxies to the same servant if that is required, such as only keeping the most-derived matching SDO Proxy. This can be done using [Best Match subscriptions](#). For applications using middleware versions before release 6.0.3, the techniques described in [Discovery Order documentation page](#).

C++ API Reference and Code Examples

In the case of inheritance, a subscribing application can attempt to downcast a discovered SDO Proxy through the use of the `SDO Pointer` associated with the proxy. For example, if the application has an `Example::Vehicle` proxy and wants to attempt to downcast into a `Example::Tank` (which inherits from `Vehicle`) proxy, it is necessary to first obtain an SDO pointer from the proxy. The pointer can then be used in a `dynamicCast` template function in an attempt to downcast the SDO pointer to a more derived type and then perform an instance-based subscription to the downcasted SDO Pointer to obtain the proxy for the derived type. The code example below illustrates this operation.

```
// Application obtains Vehicle proxy
Example::Vehicle::ProxyPtr pVehicleProxy = ...

// Obtain the SDO Pointer from the proxy
Example::Vehicle::SDOpointerPtr pVehiclePointer = pVehicleProxy->getSDOpointer();

// Attempt to downcast to Example::Tank, and be prepared to catch std::bad_cast

try {
    Example::Tank::SDOpointerPtr pTankPointer(TENA::Middleware::dynamicCast< Example::Tank::SDOpointerPtr >
(pVehiclePointer));
    cout << "Cast Successful!" << std::endl;
}
catch (std::bad_cast) {
    cout << "Bad Cast!" << std::endl;
}
```

A derived proxy can be directly upcasted to an inherited type, as shown in the code fragment below.

```
// Application obtains Tank proxy
Example::Tank::ProxyPtr pTankProxy = ...

// Upcast to Vehicle proxy
Example::Vehicle::ProxyPtr
pVehicleProxy( pTankProxy );
```

Java/C# API Reference and Code Examples

Downcasting a *base type* Proxy into a *derived type* Proxy is accomplished by calling the `downCast` method on an `SDOpointer`.

```
// Application obtains Vehicle proxy
Example.Vehicle.Proxy vehicleProxy = ...

// Obtain the SDOpointer from the proxy
Example.Vehicle.SDOpointer vehiclePointer = vehicleProxy.getSDOpointer();

// Attempt to down cast to Example::Tank. This may throw an exception.
try
{
    Example.Tank.SDOpointer tankPointer = Example.Tank.SDOpointer.downCast( vehiclePointer );
}
catch (ClassCastException/InvalidCastException e) // The Java exception type is ClassCastException. The C#
exception type is InvalidCastException
{
    // Down cast failed
}
```

Because the inheritance relationship for Proxies follows the definition in the object model, upcasting is simply an assignment.

```
// Application obtains Tank proxy
Example.Tank.Proxy tankProxy = ...

// "Upcast" to Vehicle proxy
Example.Vehicle.Proxy vehicleProxy = tankProxy;
```

SDO Pointer

SDO Pointer

SDO pointers allow applications to refer to a particular SDO instance to other applications through an attribute, method argument, or method return value. Applications that obtain an SDO Pointer can subscribe to that corresponding SDO instance in order to receive state updates, invoke methods, and receive the destruction notification.

Description

In the TENA meta-model, a pointer to an SDO behaves in the same way as pointers to objects in the meta-models of other programming systems. An [SDO pointer](#) refers (or "points") to a particular SDO instance. Like conventional pointers, it can be null—in which case it points to no SDO instance. Naturally, the particular SDO instance pointed to can be changed (unless the SDO Pointer was declared to be [const](#)).

Because SDOs exist as remote objects, passing around copies of them is not a logical operation. Instead, SDOs, local classes, and messages in an object model can refer to SDOs using SDO pointers. Consider the Tank example:

```
exception CannotLoadSoldier { string reason; };

class Tank : extends Vehicle
{
    void loadSoldier( in Soldier * pSoldier ) raises ( CannotLoadSoldier );
    void unloadSoldier( in Soldier * pSoldier );
    void setDriver( in Soldier * pSoldier );
    ...
    Soldier * pDriver;
    vector < Soldier * > passengers;
    ...
};
```

The Tank uses SDO pointers to refer to the soldiers who may drive or occupy it.

Dereferencing SDO pointers

SDO pointers can be "dereferenced" in two different ways: creating an instance-based subscription (which provides the client with an SDO Proxy) and "pulling" the associated SDO's publication state.

Instance-based subscription

The most generally useful (and thus common) way to use an SDO pointer is to create an *instance-based subscription*. While a conventional subscription results in updates for all SDOs of a type (or matching a [filter](#)), the SDO pointer allows the client to subscribe to updates for only the pointed-to instance. This member function of the SDO pointer C++ class is called "subscribe":

```
class Soldier_SDOpointer
    : public ::TENA::Middleware::SDO::PointerBase
{
public:
    ...
    Soldier_SDO::ProxyPtr const
    subscribe(
        ::TENA::Middleware::SessionPtr pSession,
        Soldier_SDO::SubscriptionInterfacePtr const & subscription );
    ...
};
```

Release 6.0.5 Update  — Prior to middleware version 6.0.5, the type `TENA::Middleware::SDO::PointerImplBase` was used instead of the `TENA::Middleware::SDO::PointerBase` type.

Release 6.0.2 Update — An alternative `subscribe` method was added with release 6.0.2 of the middleware in which the `SessionPtr` argument is used, versus the `Session` reference (i.e., `"Session &"`) that was used in the previous releases. Users are encouraged to use the `SessionPtr` interface. See

[MW-4286 RESOLVED](#) for more details.

The terms "dereference" and "subscribe" will often be used interchangeably when discussing SDO pointers. Note that the result of `subscribe` is a pointer to an [SDO Proxy](#).

Unsubscribing from Instance-based Subscription

Conspicuously, there is no corresponding "unsubscribe" member function to the above SDO pointer class. The instance-based subscription created by calling "subscribe" ends when the returned Proxy is destroyed. The Proxy object is destroyed when all references to the Proxy have gone out of scope or the reset method is called on the pointers. If you use the `Subscription` class provided by the OM definition, in addition to releasing all Proxy references held by your application you will also need to call the `removeSDO` method of that `Subscription` class to remove the Proxy reference held by it.

Exception Handling

When attempting to obtain the proxy from an SDO Pointer, there can be several exceptions that may be thrown by the middleware. Proper exception handling is important when using SDO Pointer operations. The code below explains the potential exceptions and can be used as a template for exception handling for SDO Pointer operations.

```
catch (TENA::Middleware::NoSuchSDO) {
// SDO has been destroyed.
}
catch (TENA::Middleware::NetworkError) {
// SDO might have been destroyed, but it could be a network or firewall problem, or the publisher might be hung
or stopped in a debugger, etc.
}
catch (std::invalid_argument) {
// SDO is from a different Execution than the SessionPtr that was passed in to subscribe().
}
catch (std::exception)
{ // other random failures (e.g. the local Execution has been destroyed and some ManagedPtr threw an
InvalidAccess). }
```

Note that `NoSuchSDO` inherits from `NetworkError`, so order of the catch statements is important.

Pulling the publication state

Sometimes, subscribing to an SDO instance (i.e., committing to receive future state updates) can be unnecessary. The SDO pointer also provides a simpler way of getting the current SDO state in the form of the `SDOpointer::pullPublicationState` member function:

```
class Soldier_SDOpointer
: public ::TENA::Middleware::SDO::PointerBase
{
public:
...
    Soldier_SDO::PublicationStatePtr const
    pullPublicationState(
        ::TENA::Middleware::SessionPtr pSession )
    const;
...
};
```

Release 6.0.5 Update — Prior to middleware version 6.0.5, the type `TENA::Middleware::SDO::PointerImplBase` was used instead of the `STENA::Middleware::SDO::PointerBase` type.

Release 6.0.2 Update — An alternative `pullPublicationState` method was added with release 6.0.2 of the middleware in which the `SessionPtr` argument is used, versus the `Session` reference (i.e., `"Session &"`) that was used in the previous releases. Users are encouraged to use the `SessionPtr` interface. See [MW-4286](#) for more details.

Note that `pullPublicationState` makes an immediate (blocking) call to fetch the SDO's current publication state from the Servant. If an application plans to obtain the publication state from a servant multiple times, then it would likely be more efficient to obtain a proxy using the instance-based `subscribe` method. The proxy's state will be updated by the middleware in accordance with the servant updates.

Polymorphism

Like pointers in various programming languages, SDO pointers can facilitate polymorphism. That is, an SDO pointer to a Base SDO can be *downcast* to a pointer to a Derived SDO. Of course, this operation will only succeed if the pointer does, in fact, point to an SDO of the derived type.

Just as with conventional downcasting, applications should be designed to minimize the need to downcast SDO pointers. In particular, repeated tests of the type of a pointed-to SDO should be avoided. Such tests will not perform well and such code tends to be fragile. In these circumstances, that application is best served by subscribing to the derived types explicitly.

Comparing SDO pointers

Comparing SDO pointers directly using the typical comparison operators (<, ==, etc.) compares the values of memory locations on the local machine; as such, these comparisons are not appropriate for determining whether two SDO pointers refer to the same SDO. For that, one should compare the SDOs' ObjectIDs. These are accessible without dereferencing the SDO pointer:

```
if ( sdoptr1->getObjectType() == sdoptr2->getObjectType() )
{
    // sdoptr1 and sdoptr2 point to the same SDO.
}
```

Validity of SDO pointers

TENA applications refer to SDO pointers using the `SDOpointerPtr` class. `SDOpointerPtr` has an `isValid()` method that returns `false` if and only if:

- the pointed-to `SDOpointer` has been invalidated, or
- the `SDOpointerPtr` is `null` (by default construction or copying or assigning from a `null SDOpointerPtr`).

Note that destruction of the pointed-to SDO servant does *not* cause `SDOpointerPtr::isValid()` to return `false`.

The `SDOpointer` class also has an `isValid()` method, which behaves similarly.

If an application holding an SDO pointer wants to check whether the pointed-to SDO has been destroyed, the application must attempt to pull the publication state or make an instance subscription on the SDO pointer and handle `TENA::Middleware::NetworkError` in case the SDO has been destroyed.

⚠ – Application developers who find themselves checking whether an SDO pointer points to a live SDO before using that SDO pointer should re-think their approach, as it involves a race condition: After an application verifies that the pointed-to SDO is alive but before the application goes to use the SDO pointer, the SDO could be destroyed. The application needs to handle the possibility that the SDO has been destroyed when it dereferences an SDO pointer anyway, so a preliminary check for aliveness of the SDO is usually a waste of time and application complexity.

Usage Considerations

Distributed Blocking RMIs

When an application is provided an SDO Pointer to a particular SDO instance, the application will not necessarily have all of the state information (and internal middleware connectivity information) needed to create an "SDO Proxy" from the SDO Pointer. The operation of creating an SDO Proxy from an SDO Pointer is accomplished through a `subscribe` function that is similar to the type-based subscription mechanism.

In order for the middleware to construct an SDO Proxy, it is required to obtain the current state information of the SDO Servant. If the corresponding SDO Proxy is already active within the dereferencing application, the operation will be local to that application and the processing time associated with the dereferencing will be negligible. An SDO Proxy can be active within the application either through a subscription to the SDO type in which the particular SDO was already discovered, or through a previous dereferencing operation. In either case, the application must hold onto the SDO Proxy, typically by placing it into some sort of container (e.g., `std::list`).

When the application attempts to dereference an SDO Pointer to an object that does not currently exist within the application's process space (in the form of an existing SDO Proxy), the middleware is required to attempt to contact the application that is hosting the SDO Servant associated with the pointer. This operation is accomplished through a remote method invocation from the dereferencing application to the host application.

Several failure conditions may exist when performing the remote method invocation, including the host application no longer exists, the host application is not reachable (from a network communication perspective), the host application is in a defective state in which it is not servicing remote invocations, and the object corresponding to the pointer has been deleted. The application developer needs to be aware of these potential failure conditions, as they need to be handled within the application code. User code should wrap the instance-based subscribe invocation with `try/catch` blocks to handle these potential faults.

Repeated Instance-based Subscriptions

The middleware efficiently handles the situation where an application performs multiple instance-based subscriptions for the same SDO pointer. When the application has already obtained (and is maintaining) the corresponding SDO Proxy, repeated instance-based subscriptions will be performed locally and not result in middleware remote method invocations.

Caveats of Instance-based Subscription

As explained above, when an SDO pointer is dereferenced, a proxy for the SDO is obtained. Recall that a Proxy for an SDO is also obtained when the SDO is discovered when using conventional type-based subscription. If an application has many Proxies to SDOs of a given type without being subscribed to that type, performance can suffer:

- The subscriber's total interest, which must be sent across the network to each publisher upon each subscription change, has a size directly proportional to the number of such Proxies.
- When the subscriber resigns (or otherwise destroys its Session), the Middleware sends one interest change message to each publisher for each Proxy as it is destroyed. Each of those messages has a size directly proportional to the number of such Proxies. Consequently, the number of bytes sent across the wire in this case is proportional to the number of publishers times the square of the number of Proxies. (In the next release after 6.0.0, this is improved so that only one tiny message is sent to each publisher upon resignation.)

To avoid the large and numerous interest change messages described above, a subscribing application can simply subscribe to the type of the Proxies. The drawback to this approach, of course, is that the subscriber will receive discoveries for all Proxies of that type, even ones in which it might not have been interested.

If there are many SDOs of a given type and the subscriber application is interested in only a few of them, the subscriber application would probably be better off subscribing only to those SDOs and not to the type. If, on the other hand, there are many SDOs of a given type and the subscriber application is interested in most of them, the subscriber application would probably be better off subscribing to the type.

Reusing Observers

It is possible to reuse the same Observer instance in multiple instance-based subscriptions. However, be aware that this pattern exposes the Observer code to concurrent access by multiple threads. As such, note that data accessed/modified in event handler functions should be protected by muteness.

Caveats of "pulling" the Publication State

The pullPublicationState() method always performs a blocking two-way network operation (request/response). This is true even if the application has a ProxyPtr to the pointed to SDO. If an application plans to obtain the publication state from a servant multiple times, then it would likely be more efficient to obtain a proxy using the instance-based subscribe method. The proxy's state will be updated by the middleware in accordance with the servant updates.

C++ API Reference and Code Examples

Get an SDO pointer

Both the Proxy and the Servant include a member function to get an SDO pointer for an SDO:

```
SDOpointerPtr const getSDOpointer() const;
```

The name of the return value—"SDOpointerPtr"—might at first seem redundant; however, consider that the SDO pointer itself is *not* a conventional C++ pointer. Rather, it is a polymorphic C++ object, and as such, benefits from being passed around using a smart pointer.

Issue an Instance-Based Subscription with SDO pointer

As noted previously, "dereferencing" an SDO pointer creates an instance-based subscription. The SDO pointer's subscribe member is called with a reference to the Session and a SubscriptionInterface implementation as discussed in the [Subscription documentation page](#):

```
Example::Soldier::SubscriptionPtr subscription(new Example::Soldier::Subscription)
//
// Set up the Subscription; add any desired Observers.
//
...
Example::Soldier::ProxyPtr soldierProxy = soldierSDOpointer->subscribe(session, subscription);
```

"Pull" a Snapshot of the Pointed-to SDO's State

As noted previously, an SDO pointer can be used to perform one-shot "pull" of the current state of the corresponding SDO over the network. With the instance-based subscription described above, the user is provided an SDO ProxyPtr corresponding to the SDO to which the SDO pointer points. This ProxyPtr is automatically "pushed" any state changes for the corresponding SDO. Using this "pull" technique is more efficient in situations where a copy of the SDO's state is only needed once (or perhaps a few times).

```
Example::Soldier::PublicationStatePtr soldierPubState = soldierSDOpointer->pullPublicationState(session);
```

Null SDO pointers

A "null" SDO pointer doesn't point to any SDO. In general, a null SDO pointer is expressed in C++ as a null SDOpointerPtr. That is, it is sufficient to check the SDOpointerPtr because this smart pointer delegates to the SDOpointer it points to (if any) for the purpose of examining validity. Accordingly, the SDOpointerPtr default constructor creates a null SDO pointer:

```
Example::Soldier::SDOpointerPtr sdoPtr;
```

Validity is checked with the `SDOpointer::isValid` member function:

```
if (sdoPtr.isValid())
{
    // The pointer is valid and safe to use.
}
```

Downcasting

Downcasting of SDO pointers is performed using `TENA::Middleware::dynamicCast`:

```
DerivedSDOpointerPtr derivedPtr = TENA::Middleware::dynamicCast< DerivedSDOpointerPtr >(basePtr);
```

This behaves consistently with other uses of `TENA::Middleware::dynamicCast`; the function will throw a `std::bad_cast` exception if the cast was not successful.

Java API Reference and Code Examples

Get an SDO pointer

Both the Proxy and the Servant include a member function to get an SDO pointer for an SDO:

```
SDOpointer getSDOpointer();
```

Issue an Instance-Based Subscription with SDO pointer

As noted previously, "dereferencing" an SDO pointer creates an instance-based subscription. The SDO pointer's `subscribe` member is called with a reference to the Session and a `Subscription` instance as discussed in the [Subscription documentation page](#):

```
Example.Soldier.Subscription subscription = new Example.Soldier.Subscription();

// Set up the Subscription; add any desired Observers.
...
Example.Soldier.Proxy soldierProxy = soldierSDOpointer.subscribe(session, subscription);
```

"Pull" a Snapshot of the Pointed-to SDO's State

As noted previously, an SDO pointer can be used to perform a "pull" of the current state of the corresponding SDO over the network. With the instance-based subscription described above, the user is provided an SDO Proxy corresponding to the "pointed-at" SDO. This Proxy is automatically "pushed" any state changes for the corresponding SDO. Using this "pull" technique is more efficient in situations where a copy of the SDO's state is only needed occasionally.

```
Example.Soldier.PublicationState soldierPubState = soldierSDOpointer.pullPublicationState(session);
```

Null SDO pointers

A "null" SDO pointer doesn't point to any SDO. Creating an `SDOpointer` explicitly using its constructor always creates a null SDO pointer. A null SDO pointer may be used to initialize fields that require an SDO pointer.

```
Example.Soldier.SDOpointer nullSoldierPtr = new Example.Soldier.SDOpointer();
```

Calling `pullPublicationState` or `subscribe` on a null `SDOpointer` will throw an `TENA.Middleware.Utils.InvalidAccess` exception.

Downcasting

Downcasting a *base type* `SDOPointer` into a *derived type* `SDOPointer` is accomplished by calling the `downCast` method on an `SDOpointer`.

```
// Application obtains a SDO Pointer

Example.Vehicle.SDOpointer vehiclePointer = ...;

// Attempt to down cast to Example::Tank. This may throw an exception.

try
{
    Example.Tank.SDOpointer tankPointer = Example.Tank.SDOpointer.downCast( vehiclePointer );
}
catch (ClassCastException/InvalidCastException e) // The Java exception type is ClassCastException. The C#
exception type is InvalidCastException
{
    // Down cast failed
}
```

SDO Reactivation

SDO Reactivation

TENA applications can periodically save the state of an SDO (Stateful Distributed Object) to an output stream (i.e., ostream, such as a file). This mechanism can be used by applications to recover from an application or computer fault, in which re-starting the application or using an application on another computer can reactivate an object from the saved state. The application reactivating the object must be joined to the same execution as the saved state.

Description

When there is an abnormal termination of a TENA application it may be important for the objects that were created by the application to be *reactivated*, where the object can be created again with the same "appearance" to subscribing applications. The Object Reactivation capability permits a publishing application to periodically save object state information to a file (or any output stream) that allows another application to subsequently reactivate that object using the saved state information. The reactivated object will have the same publication state values from the last save, and it will have the same middleware object identifier (see [SDO ID](#) for more details).

A subscribing application will continue to receive state updates after a previously discovered object was reactivated. During the period of time between when the application abnormally terminates and the object is reactivated, applications that discovered the object (and are holding onto the proxy) will not receive updates and any attempt to invoke a remote method on the proxy will result in a runtime exception. After the object is reactivated, the subscribing applications will become transparently "re-connected" with the reactivated object and they will receive any state updates and be able to invoke remote methods.

The save mechanism used for potential reactivation requires an output stream to be used to write the state of the object that needs to be saved. Application designers are permitted to store and access this saved state information in any manner that is acceptable for their needs. For example, each object can have a separate file for every save on the file system, or the system can use a database that manages the saved state.

The saved state is intended for middleware purposes only and is written in a binary representation that is compatible on all middleware supported computer platforms. This format compatibility allows the object save information to be copied and used on a different computer, independent of the operating system/compiler vendors and versions.

Usage Considerations

TENA application developers will need to determine whether it is necessary to save the state of their published objects to support object reactivation. Typically, object reactivation is not required for most applications since if an abnormal application termination does occur, the application can usually be re-started and publish the necessary objects again without needing to have the same middleware object identifier or last attribute values. But in some situations, providing continuity of the middleware object id after an abnormal application termination is important for the event (or post-event) activities.

If an application needs to save the object state, a determination of how frequently the state should be saved needs to be evaluated. Each object save invocation will typically (can write to memory if appropriate) require an operating system write operation to a file on some disk drive. This write operation will require some computer processing resources that should be evaluated with respect to overall application processing requirements. For example, performing an object save operation after every update may not be required with respect to reactivation and it may negatively impact the application performance. In this situation, an update counter or timer can be maintained by the application that will only perform a save operation every so many updates or every so many minutes.

An application developer, in coordination with the event planners, will also need to consider where to store the object state information. For example, if the reactivation process needs to be done from a separate machine, in case the original machine has a failure or the network to that machine has a failure, it will be important to make the save files accessible to the machine (or machines) that would be responsible for performing the object reactivation. In some situations, a network drive on a separate device may need to be used to directly write the object save files. Alternatively, a manual or automated mechanism can be used to periodically copy the object save file to other computers on the network in support of a network failure.

For event planning purposes, it should be understood that there may be large update gaps and periods in which remote method invocations will fail for any objects associated with an abnormally terminated application. Subscribing applications will need to be designed to properly handle these conditions.

It is possible for an application to reactivate an object that already exists within the execution. When this occurs, the subscribing applications will report Alerts that are written to the application log files and reported to any connected TENA Consoles that indicate an object update was received from an application that is not recorded as the owner of the object. In this situation, the updates will be ignored and the event coordinator(s) should be notified of the problem.

Note that when an event operator detects that an application has abnormally terminated, that application should not be removed from the execution through the TENA Console if there is a need to reactivate the objects that belonged to that application. Object reactivation will not work if the application was removed, which causes a destruction callback for all objects belonging to the removed application. If the dead application was publishing some SDOs that need reactivation and other SDOs that need termination, the reactivations must be performed **before** the TENA Console is used to remove the dead application.

C++ API Reference and Code Examples

In order to reactivate an object, a save file can be used to hold the saved state information. Each saved object type needs to use a separate save file and application developers are encouraged to use a file naming convention to support their object reactivation requirements. For example, if an application may be required to permit the operator to select the type of objects to be reactivated, the name of the object save files could indicate the name of the object type.

The name of the object save file can be stored in a `std::string`. The string is used to create a `std::ofstream`, which is passed into the SDO servant's `save()` method. A code fragment showing this operation is shown below. Note that exception handling code is not shown in the code example.

```
std::string saveFileName = "object27.save";
short updateCount( 0 );
short saveInterval( 100 ); // Save every 100 updates
...
Example::Vehicle::ServantPtr pServant;
...
pServant->commitUpdater( pUpdater );
...
std::ios_base::openmode mode = std::ios_base::out
    | std::ios_base::binary
    | std::ios_base::trunc;
std::ofstream ofs( saveFileName.c_str(), mode );
...
if ( ( ( updateCount++ ) % saveInterval ) == 0 )
{
    pServant->save( ofs );
}
...
ofs.close();
```

In the event that an object needs to be reactivated, the reactivating application needs to know the filename of the object save file (and must be able to read from that file). The name of the object save file can be stored in a `std::string`. The string is used to create a `std::ifstream`, which is passed into the `ServantFactoryPtr`'s `createReactivationInitializer()` method. The code example below illustrates the operations associated with reactivating an object from a save file.

```
std::string saveFileName = "object27.save";
...
Example::Vehicle::ServantFactoryPtr pSF =
    Example::Vehicle::ServantFactory::create( *pSession );
...
std::ifstream ifstr( saveFileName.c_str(), std::ios_base::binary );

std::auto_ptr< Example::Vehicle::ReactivationInitializer > rinit(
    pSF->createReactivationInitializer( ifstr ) );
ifstr.close();
...
pServant = pSF->reactivateServant( rinit );
```

Note that the `createReactivationInitializer()` operation will result in a runtime exception, if the object save file is not accessible for reading or is not associated with the current logical range execution.

Java API Reference and Code Examples

In order to reactivate an object, a save file can be used to hold the saved state information. Each saved object needs to use a separate save file and application developers are encouraged to use a file naming convention to support their object reactivation requirements. For example, if an application may be required to permit the operator to select the type of objects to be reactivated, the name of the object save files could indicate the name of the object type.

In this simple example, the name of the object save file is stored in a string used to construct a `DataOutputStream` for use with the SDO servant `save` method. A code fragment showing this operation is shown below. Note that exception handling code is not shown in the code example. See a representative example of [Servant.save](#) for the API.

```

import java.io.DataOutputStream;
import java.io.FileOutputStream;

std::string saveFileName = "object27.save";
int updateCount = 0;
int saveInterval = 100; // Save every 100 updates
...
Example.Vehicle.Servant servant = ... ;

// Looping
while(updating) {
    ...
    servant.commitUpdater( updater );

    if (((updateCount++)%saveInterval) == 0) {
        DataOutputStream dos = new DataOutputStream( new FileOutputStream( saveFileName ) );
        servant.save(dos);
        dos.flush();
        dos.close();
    }
}

```

In the event that an object needs to be reactivated, the reactivating application needs to know the filename (or other saved data stream) for the objects saved state. The code example below illustrates the operations associated with reactivating an object from a save file. The API for getting a reactivation initializer can be in this example for [Example.Vehicle.ServantFactory.createReactivationInitializer](#).

```

String saveFileName = "object27.save";...Example.Vehicle.ReactivationInitializer rinit = servantFactory.
createReactivationInitializer(new DataInputStream(new FileInputStream(saveFileName) ) );
servant = servantFactory.reactivateServant( rinit );

```

Note that the `createReactivationInitializer()` operation will result in a `RuntimeException` if the object save file is not a valid saved servant state data file, or is not associated with the current logical range execution.

SDO Remote Method

SDO Remote Method

SDO remote methods provide distributed services that can be used by subscribing applications. These remote methods are associated with a particular SDO instance and can be used to manipulate the SDO state, or the methods can perform functions that are completely separate from the SDO state. These remote methods can take arguments, provide return values, and throw user-defined exceptions.

Description

SDO remote methods are a high-level abstraction of remote procedure calls (RPCs) that allow one application to invoke the distributed service in a location transparent manner (i.e., invocation of the service appears to be a local operation). Subscribing applications discover SDO instances as [SDO proxies](#) that have an interface that contains the remote methods defined by the SDO class. These remote methods support arguments, return variables, and exceptions—like normal methods and functions.

Object model designers can define arbitrary remote methods that can be used to manipulate the state (i.e., attribute values) of the particular SDO instance, or these methods can be used to perform functions independent of the SDO state. Examples of remote methods could be a function that provides the current status of a system when requested, or the remote method could be used to control the pointing direction of a camera. Users are able to define remote methods for any service that would be beneficial for other applications in the execution.

Remote methods are defined in TDL (TENA Definition Language) by defining [operations](#) on a [class](#). An example for the `Example::Tank` SDO class is shown below.

```
class Tank : extends Vehicle
{
    void loadSoldier( in Soldier * pSoldier ) raises ( CannotLoadSoldier );
    void unloadSoldier( in Soldier * pSoldier );
    void setDriver( in Soldier * pSoldier );
    void driveTo( in vector < Location > wayPoints );
    AreaOfEffect fireGun( in Location targetLocation );
    ...
};
```

SDO remote methods allow subscribers to invoke remotely-implemented functionality using the same syntax that is used for conventional local function calls and with semantics that are similar to local functions.

Method Arguments and Return Variables

Remote methods can define arbitrary arguments associated with the method invocation. Arguments are passed to the remote method by value (sometimes defined with the qualifier `in`), meaning that any changes to that argument made by the remote method implementation code will not be reflected in the variable used by the application invoking the method. Initially, the TENA meta-model supported `out` and `inout` argument qualifiers, but this became problematic when supporting other programming languages that did not have inherent support for these semantics. The alternative to using `out` and `inout` arguments is to use a return variable to provide value changes back to the invoking application. A Local Class can be used as a return variable if multiple arguments need to be returned. Permitted types for method arguments and return variables are [fundamental types](#), [enumerations](#), [vectors](#), [local classes](#), and [SDO pointers](#).

Exceptions

Remote methods can raise [exceptions](#) to indicate errors. Exceptions defined in TDL correspond to C++ exceptions that inherit from the `std::exception` class. Subscribing applications invoking remote methods should wrap the method invocation in a `try/catch` block to handle potential exceptions that may occur. In addition to the user-defined exceptions associated with the remote method, the middleware may also throw an exception due to network errors or if the SDO servant no longer exists.

When writing implementation code for a remote method, application developers should wrap their implementation code with a `try/catch` block that re-throws non-specific exceptions as `TENA::Middleware::RemoteMethodsImplError` exception. A code example related to this behavior is shown below.

oneway Remote Methods

Conventional remote methods "block" when called; that is, the current execution thread does not proceed until the remote method call returns. Usually these semantics are desirable. However, for situations where the caller cares neither about a remote method's return value nor whether the operation was successful, the middleware provides `oneway` qualifier that can be defined for remote methods.

As the previous comment suggests, `oneway` methods have a void return value and do not raise exceptions. That doesn't mean they cannot fail—it just means that there is no mechanism (within the middleware) for the application to determine whether there was a failure. In light of these caveats, `oneway` methods should be used very judiciously when designing an object model.

Usage Considerations

Concurrency

When implementing remote methods, it is important to keep in mind that the implementation can be entered concurrently by multiple middleware threads. Consider, for instance, concurrent invocations from multiple subscribers. Using [SDO Updaters](#) to modify the publication state protects the SDO itself from concurrency-related problems; but implementers should be mindful of how concurrent invocations of the remote method implementation can interact with the rest of their application. It is often appropriate to use some locking mechanism (e.g., `boost::lock_guard`, `ace_Guard`) to prevent multiple programming threads from concurrently executing a remote method.

 Remote method implementation code must be written in a thread-safe manner to prevent concurrency if necessary. The middleware uses multiple programming threads to process remote method invocations, and these invocations are permitted to occur concurrently.

Responsiveness of Remote Methods

Remote method implementers must consider that a subscriber is waiting on the method to return; consequently, operations that could take a nontrivial amount of time should be avoided. In some cases, it may be necessary for the implementation to defer work of the remote method to an asynchronous programming thread, but this must be done with care since return arguments and exceptions can't be properly supported when the work is deferred to a separate thread. This advice does not apply to `oneway` methods, which do not block their callers.

C++ API Reference and Code Examples

Creating Servants with Remote Methods

When a publishing application attempts to create an SDO servant that defines remote methods, the application developer is responsible for writing a class that implements the remote methods. This remote methods implementation class needs to inherit from the object model definition interface for the remote methods, as shown in the code example below. In this example, the `MyApplication::Example::Tank::RemoteMethodsImpl` class inherits from the `Example::Tank::RemoteMethodsInterface` class that is part of the object model definition.

```
namespace MyApplication
{
    namespace Example_Tank
    {
        class RemoteMethodsImpl
            : public virtual Example::Tank::RemoteMethodsInterface,
              TENA::Middleware::Utils::noncopyable // Disallow copying this class
        {
        ...
    }
}
```

Note that in the example generated code, this `RemoteMethodsImpl` class also inherits from the `TENA::Middleware::Utils::noncopyable` class because we do not want this class to be copied. The concept of operations is for a single remote methods implementation object to be associated with a particular SDO servant instance. The middleware API for creating a servant with remote methods requires the application to provide the remote method implementation object as part of the `createServant` invocation.

Release 6.0.5 Update! – Prior to middleware version 6.0.5, the type `boost::noncopyable` was used instead of the `TENA::Middleware::Utils::noncopyable` type.

RemoteMethodsImpl Lifetime Considerations

Example code associated with creating an SDO servant with remote methods is shown below for the `Example::Tank` class. In the example code, the `pRemoteMethods` argument uses a `std::auto_ptr` to pass the `RemoteMethodsImpl` instance into the `createServant` method. The use of `std::unique_ptr` correctly defines the semantics that the application is releasing ownership of the `RemoteMethodsImpl` instance to the middleware and the application should not attempt to use this instance since it may be deleted by the middleware asynchronously.

```
std::unique_ptr< Example::Soldier::RemoteMethodsInterface > pRemoteMethods(
    new MyApplication::Example_Soldier::RemoteMethodsImpl );

Example::Soldier::ServantPtr pServant(
    pServantFactory->createServant(
        std::move( pRemoteMethods ),
        *initializer,
        communicationProperties ) );
```

If an application has a need to access a `RemoteMethodsImpl` instance passed into the middleware, it will be necessary to utilize some shared object with a smart pointer within the `RemoteMethodsImpl` code. For example, assume that the `RemoteMethodsImpl` instance had state that was modified by the `RemoteMethodsImpl` code, but was also needed by the application. This state can be defined within a C++ object and then wrapped by a smart pointer (e.g., `TENA::Middleware::Utils::SmartPtr`) within the `RemoteMethodsImpl` class. In this situation, even though the middleware may destruct the `RemoteMethodsImpl` instance, the contained state object wrapped by the `SmartPtr` would still persist and be accessible to the application.

⚠ – In a previous release of the middleware (prior to 6.0.4), a version of the `createServant` method was added in which the `RemoteMethodsImpl` instance could be provided within a `TENA::Middleware::Utils::SmartPtr`. This method has been deprecated in favor of the `std::auto_ptr` version because the `SmartPtr` version could be misused by supplying the same `RemoteMethodsImpl` instance to multiple SDO Servants. The middleware will throw a `std::invalid_argument` exception if the application attempts to use the same `RemoteMethodsImpl` instance for multiple SDO Servants.

Remote Method Concurrency `std::invalid_argument`

Remote method implementation code should evaluate thread-safety concerns, as remote methods can be invoked concurrently by the multiple programming threads used by the middleware. When necessary, developers are encouraged to protect implementation code with a locking mechanism to prevent concurrency. For example the `boost::lock_guard` and `ace_Guard` mechanisms provide a simple mechanism where the guard, which is a C++ object, is obtained at the beginning of the method implementation and automatically released when the invocation completes. The guard prevents other programming threads from invoking the remote method code simultaneously.

Remote Method Exceptions

Additionally, it is appropriate for the remote method implementation to evaluate potential exceptions that may occur, especially for exceptions that are defined as part of the object model interface. Users are encouraged to follow the example application approach of wrapping the all of the code within a remote method implementation with a `try/catch` block for handling exceptions that occur within a remote method invocation. An example is shown below.

```
try
{
    ... // user implementation code
} catch ( std::exception const & ex ) {
    std::string msg( "Caught a std::exception at " );
    msg += '[';
    msg += ::TENA::Middleware::Utils::Debug::timestamp();
    msg += ',';
    msg += TENA_FILE_STRING;
    msg += ':';
    msg += __FUNCTION__;
    msg += ",TID=";
    msg += ::TENA::Middleware::Utils::Debug::threadID();
    msg += "]. The exception said: \"";
    msg += ex.what();
    msg += "\\n";
    throw TENA::Middleware::RemoteMethodImplError( msg.c_str() );
} catch ( ... ) {
    std::string msg( "Caught an unknown exception at " );
    msg += '[';
    msg += ::TENA::Middleware::Utils::Debug::timestamp();
    msg += ',';
    msg += TENA_FILE_STRING;
    msg += ':';
    msg += __FUNCTION__;
    msg += ",TID=";
    msg += ::TENA::Middleware::Utils::Debug::threadID();
    msg += "]. ";
    throw TENA::Middleware::RemoteMethodImplError( msg.c_str() );
}
```

Calling Remote Methods

Subscribers can invoke remote methods through an [SDO Proxy](#):

```

try
{
    tankProxy->loadSoldier( soldierSDOptr );
}
catch ( CannotLoadSoldier const & ex )
{
    std::cout << "could not load soldier: " << ex.what() << std::endl;
}
catch ( std::exception const & ex )
{
    std::cout << __FILE__ << ':' << __LINE__ << ": caught exception: " << ex.what() << std::endl;
    throw; // Rethrow the exception.
}

```

In the above example, the remote method `Example::Tank::loadSoldier` is invoked by the subscribing application. One notices pretty quickly that most of the code in this example is devoted to exception handling. The brevity of this example amplifies the proportion of exception handling code; however, it is important to be aware of the potential exceptions that can be raised from remote method calls.

If a `CannotLoadSoldier` exception is thrown, the code catches it and simply prints out a message. A real application would probably want to take further steps to recover from such an error. If some other exception (derived from `std::exception`) was thrown, the next catch block will catch it. It can be useful to catch exceptions that you aren't necessarily expecting near where they are thrown. This gives you the opportunity to print out diagnostic information that can aid in debugging; here, the example simply prints out the file and line number along with the exception's message. Finally, the example rethrows the caught `std::exception`. This is desirable when code further up the call stack cares that an exception has been thrown (and is prepared to deal with it).

Java API Reference and Code Examples

Implementing remote methods

As part of the object model definition library, the middleware generates an abstract class `AbstractRemoteMethods` for each SDO. For the Tank SDO we've been looking at, that class looks like this:

```

package Example.Tank;

public abstract class AbstractRemoteMethods {
    public abstract void          driveTo(ImmutableLocalClassVector wayPoints);
    public abstract ImmutableLocalClass fireGun(ImmutableLocalClass targetLocation);
    public abstract void          loadSoldier(SDOpointer pSoldier);
    public abstract void          setDriver(SDOpointer pSoldier);
    public abstract void          unloadSoldier(SDOpointer pSoldier);
}

```

`AbstractRemoteMethods` has an abstract method corresponding to each remote method defined on the SDO. The application that publishes an SDO must provide an implementation class that extends this class, providing implementations for these methods. An instance of this concrete class is used to create the [SDO Servant](#).

Calling remote methods

Subscribers can invoke remote methods through an [SDO Proxy](#):

```

try {
    tankProxy.loadSoldier( soldierSDOptr );
} catch ( CannotLoadSoldier clsex ) {
    System.out.println("Could not load soldier: " + clsex);
} catch ( Exception ex ) {
    System.out.println("Unexpected exception: Could not load soldier: " + ex);
    throw ex; // Rethrow the exception.
}

```

In the above example, the remote method `Example.Tank.loadSoldier` is called. One notices pretty quickly that most of the code in this example is devoted to exception handling. The brevity of this example amplifies the proportion of exception handling code; however, it is important to be aware of the potential exceptions that can be raised from remote method calls.

If a `CannotLoadSoldier` exception is thrown, the code catches it and simply prints out a message. A real application would probably want to take further steps to recover from such an error. If some other exception (derived from `Exception`) was thrown, the next catch block will catch it. It may sometimes be useful to catch exceptions that you aren't necessarily expecting near where they are thrown. Finally, the example rethrows the caught `Exception`. This is desirable when code further up the call stack cares that an exception has been thrown (and is prepared to deal with it).

Accessing Application Data from a Remote Method

Accessing Application Data from a Remote Method

SDO remote method implementation classes are derived from the automatically generated `RemoteMethodsInterface` class. Implementation developers can use the inherited `ServantPublicationStateView` method to access and update the state of the particular SDO instance.

Description

Most `remote method` implementations can be expected to access and possibly manipulate the SDO state attribute values. The abstract class provided by the middleware allows access to the SDO's state through a `ServantPublicationStateView` class. By accessing this class via the `getServantPublicationStateView` method, the current state may be viewed and modified.

The functions for accessing and updating the publication state mirror the methods available on the SDO servant itself. Although developers are responsible for any thread concurrency issues regarding the remote method implementation (since two threads may be dispatching remote method calls from two different clients), access and updating of the SDO via the `ServantPublicationStateView` is done by the middleware in a thread-safe manner.

C++ API Reference and Code Examples

```
class RemoteMethodsInterface
{
public:
    ...
    ServantPublicationStateViewPtr const &
    getServantPublicationStateView();
    ...
};
```

The remote methods implementation can call this function to access the `ServantPublicationStateView`, which allows application developers to access the `PublicationState` and create and commit state `updaters`. The interface to `ServantPublicationStateView` is shown below.

```
class ServantPublicationStateView
{
public:
    ...
    PublicationStatePtr const
    getPublicationState();

    PublicationStateUpdaterPtr
    createUpdater();

    TENA::uint32
    commitUpdater( PublicationStateUpdaterPtr pUpdater );
    ...
};
```

Java API Reference and Code Examples

```
class AbstractRemoteMethods {
    ...
    public ServantPublicationStateView getServantPublicationStateView();
    ...
};
```

The remote methods implementation can call this function to access the `ServantPublicationStateView`, which allows application developers to access the `PublicationState` and create and commit state `updaters`. The interface to `ServantPublicationStateView` is shown below.

```
class ServantPublicationStateView {  
...  
public PublicationState getPublicationState();  
public PublicationStateUpdater createUpdater();  
public TENA.UnsignedInteger commitUpdater( PublicationStateUpdaterPtr pUpdater );  
...  
};
```

SDO Subscription

SDO Subscription

An application must subscribe before it will discover SDOs (Stateful Distributed Objects) of a particular type. The subscription process includes invoking the type specific subscribe operation that permits attaching user defined Observers that provide the application specific processing behavior associated with the discovered SDOs. In addition to the type-based filtering, an optional Filter may be specified to define additional filtering criteria as part of the middleware [Advanced Filtering](#) capability.

Description

Subscribing to a SDO type communicates an application's interest in receiving SDO updates from all current and future applications that create SDOs matching that type. Since the TENA Middleware uses "send-side" filtering, an application's SDO interest must be communicated to all publishing applications to determine whether there is a match in what the publishing application is publishing and what the subscribing application is interested in receiving. Obviously, this interest communication is not instantaneous and there is a communication delay between when the application defines the subscription interest and the publisher begins sending matching SDOs.

Applications use the concept of a Session to internally organize publication and subscription actions, so SDO subscriptions are associated with a particular Session object. See the [Execution Management Services documentation page](#) for additional information on Sessions.

When subscribing to a particular SDO type, in C++ there is a `subscribe` function in the namespace of the SDO type. For other language bindings, `subscribe` is a static method on the `Subscription` class. `Subscribe` accepts a Session and a `SubscriptionInterface` derived object, and optionally a self-reflection flag and a Filter.

The `SubscriptionInterface` argument is typically provided in the form of a `Subscription` instance. The `Subscription` class is included as part of the standard OM definition. An application may use its own class that satisfies the `SubscriptionInterface` definition. For example, the `SubscriptionInterface` was designed to accept the `SubscriptionInfo` implementation that was used in release 5 of the TENA Middleware. Additional information regarding the `SubscriptionInterface` and attaching Observers to control application behavior with discovered SDOs is covered in the subsequent section.

 Note: the use of a user-provided `SubscriptionInterface` is primarily for backwards compatibility, and is only provided with C++. The newer language bindings only support using a concrete `Subscription` instance.

The `selfReflection` argument is used to control whether or not the subscribing application wants to receive matching Messages or SDOs published by the subscribing Session (in addition to all other matching ones). If `true` (the default behavior when the argument is not specified), the subscribing application receives all matching SDOs or Messages, including those published by the subscribing Session. If `false`, the subscribing application receives only matching SDOs or Messages published outside the subscribing Session. In most situations, an application is not interested in self reflection; however, for compatibility with previous releases of the TENA Middleware, the default behavior was set to enable self reflection.

The optional `Filter` argument allows for additional filtering criteria of SDOs. The default is a "no tag" Filter that matches publishers not using advanced filtering. This capability is described in detail by the [advanced filtering documentation page](#).

When a subscribing application is initially informed about an SDO instance that matches the subscribed type, the subscribing application is said to have "discovered" the SDO instance. As long as the subscribing application holds onto the discovered SDO (which is provided to the application as an `SDO Proxy`), the application will receive any state updates performed by the publishing application and will receive a destruction notification when the SDO is destroyed.

Using the `Subscription` class associated with the OM definition, holding discovered SDO proxies is handled automatically because each discovered SDO is added to an internal list. Applications can obtain a `copy` of this list by invoking the `Subscription::getDiscoveredSDOList` method. This list of SDO proxies represents the SDOs discovered by the application that have not been destroyed.

It is important to note that the returned list is a copy that was valid at the time the method was invoked, as it is possible that the list maintained internal to the `Subscription` object will change based on discoveries or destructions.

Applications that are not interested in receiving updates or destruction notification for certain SDO instances can remove the proxy maintained by the `Subscription` object by using the `Subscription::removeSDO` method. As long as the application is not holding any additional references to that particular SDO proxy, the middleware will stop delivering updates and any destruction notification.

Since the list returned from the `Subscription::getDiscoveredSDOList` contains copies of the SDO proxies, the application will continue to receive updates (and destruction notifications) for those SDOs even if the `Subscription::removeSDO` method was used subsequent to obtaining the list. Application should also remove the proxy from any list (or other data structure holding a copy of the SDO Proxy) to prevent further updates.

SDO Observers

An application that subscribes to SDOs will typically need to provide application specific behavior to be executed when an SDO is discovered, updated, and/or destroyed. This behavior may be to write information to a database, update a display, or perform some other custom logic needed by the application.

When a subscribing application is informed of an unknown SDO that matches the subscription interest, the middleware (associated with the subscribing application) will construct an SDO Proxy object. The middleware then needs to notify the application that there is a new SDO that has been discovered.

Internally to the middleware, SDO events (e.g., discovery, update, destruction) received by a subscribing application are dispatched via a callback mechanism. A `SubscriptionInterface` derived class provides the `CallbackFactory` that creates `Callback` objects that are placed in a callback queue. Automatically generated code associated with an object model definition will define a `Subscription` class (implementing `SubscriptionInterface`) that can be used by the subscribing application to attach `Observer` classes. These `Observer` instances provide the application specific behavior that is invoked when SDO events are received.

When the middleware performs the callback processing (see [callback framework documentation page](#) for additional details), any received SDO events will be dispatched to the appropriate event method (e.g., `discoveryEvent`, `stateChangeEvent`, `destructionEvent`) for all registered `Observers` for that particular subscription. The implementation code for these `Observer` event methods is written by the application developer to provide the appropriate behavior that the application should perform in response to receiving a particular SDO event.

An application developer can attach multiple observers to particular `Subscriptions`. This allows predefined classes with particular functionality to be used by multiple applications. For example, a predefined "logging" observer might be used by several different subscriber applications, each with their own specialized application-specific `Observers` – for example an `UpdateDisplayObserver`.

`Observers` are attached to the SDO type specific `Subscription` object through the `addObserver` method.

Details concerning the `Observer` mechanism can be found on the [Observer Mechanism documentation page](#). It should be noted that `callbacks` as implemented for TENA Middleware release 5 are still supported within C++. However, the `Observer` mechanism allows callback behavior to be packaged into separate components that can be dynamically and selectively enabled, and is believed to provide a superior user API.

As mentioned, application developers are responsible for developing the `Observer` classes that are needed for their particular needs. When an SDO event is received that corresponds to registered `Observer`, the middleware will invoke the corresponding event method on the `Observer` class. The arguments provided to the SDO event methods are the SDO `Proxy` and the SDO `PublicationState`. The `PublicationState` contains the state of the SDO at the time the event occurred and may be sufficient for many applications. Interacting with the SDO `Proxy` allows more elaborate interactions, for example calling a remote method. In addition, calling `Proxy.getPublicationState()` will return the most up-to-date SDO state.

⚠ — As previously mentioned, the subscribing application is required to use the middleware evoke callback services (e.g., `evokeMultipleCallbacks`) in order for the middleware to process the callback objects that have been placed on an internal queue. In turn, these callback implementations will invoke the appropriate event method associated with registered `Observers`. If an application uses multiple programming threads to evoke callbacks, application developers are responsible to ensure that the `Observer` event methods support concurrency.

Duplicate Subscriptions

Subscribing to the same SDO type (in the same Session) will replace the previous Subscription with the new Subscription and associated Observers. If two independent subscriptions are needed within an application, separate Sessions can be utilized, although attaching multiple `Observers` to the same Subscription may be more appropriate.

Instance-based Subscription

TENA object models can define SDO Pointers to refer to a particular SDO instance. Applications that obtain an SDO Pointer can dereference the pointer to obtain an SDO proxy. The returned proxy will behave in a fashion similar to a proxy obtained through the type-based subscription mechanism.

In order to dereference an SDO pointer, the application uses a similar `subscribe` function as the type-based subscription operation. The type-based subscription requires the appropriate `Session` and a `SubscriptionInterface` implementation that can have attached `Observers` for that particular instance-based subscription.

Additional information on SDO Pointers and instance-based subscriptions can be found on the [SDO pointer documentation page](#).

Changing Subscriptions

A subscribing application can associate a different `Subscription` object for an existing SDO `Proxy`, using the `setSubscription` method on the `Proxy` object. This may be necessary if the application wants to change the `Observers` that are attached to a particular SDO instance.

There are several considerations with a `Subscription` change:

- It is possible that after the subscribing application sets a new `Subscription`, that `Observers` from the old `Subscription` may still be active. This is because the underlying callback events processed by the middleware may have been already enqueued, but not processed yet by the application's [evoke callback operations](#). If an application needs deterministic control regarding changing `Observers`, then the `Observer` developers will need to build that required code logic into the `Observer` code to deal with whether the old or new `Observers` are used during the transition period from the old `Subscription` to the new `Subscription`.
- The old `Subscription` object will maintain the SDO `Proxy` within its `DiscoveredProxyList` because the `Proxy` is only added to the list based on a `DiscoveryEvent`. It may be necessary for the application to copy the discovered SDO Proxies from the old `Subscription`::`DiscoveredProxyList` to the new `Subscription`::`DiscoveredProxyList`, and then the old `Subscription` object can be safely deleted.

SDO Inheritance and Polymorphism Considerations

Within a TENA object model, SDOs support the object-oriented programming capability of inheritance. This meta-model concept is described in the [inheritance meta-model documentation page](#).

From an SDO subscription perspective, subscribing applications that subscribe to an inherited class belonging to a particular SDO instance, will discover that SDO as the inherited type. In other words, and attributes or methods that only exist in the SDO types that were derived from the subscribed type will not be accessible.

Additional information can be found on the [SDO inheritance and polymorphism documentation page](#).

SDO Unsubscribe

Subscribing applications may unsubscribe to a particular SDO type if necessary. This unsubscribe operation will cause the application to no longer receive discovery events of SDO's that match the original subscription. Existing SDO Proxies discovered through the corresponding subscribe operation will no longer receive updates (or scope changes if Advanced Filtering is being used).

Additional information can be found on the [SDO unsubscribe documentation page](#).

C++ API Reference and Code Examples

As described above, subscriptions are initiated using the `subscribe` function. An example `subscribe` declaration for `Example::Notification` Message type from the `Example-Vehicle` object model is shown below.

```
static
void
subscribe(
    ::TENA::Middleware::SessionPtr pSession,
    SubscriptionInterfacePtr const & subscription,
    bool enableSelfReflection = true,
    ::TENA::Middleware::Filter const & filter =
        ::TENA::Middleware::Filter() );
```

Release 6.0.2 Update — An alternative `subscribe` method was added with release 6.0.2 of the middleware in which the `SessionPtr` argument is used, versus the `Session` reference (i.e., "Session &") that was used in the previous releases. Users are encouraged to use the `SessionPtr` interface. See [MW-4286](#) for more details.

As described above, typically the `SubscriptionInterface` argument is provided by passing an instance of a `Subscription`, another class generated as part of an OM definition. Typically any `Observer` instances are added to the `Subscription` before passing it to `subscribe`. This ensures that the observers will be active if events happen immediately after subscribing. The class below shows the `Subscription` interface generated for an `Example::Vehicle` SDO.

```
namespace Example
{
    class Vehicle
    {
        class Subscription
            : public SubscriptionInterface
        {
            public:
                Subscription( bool isPruning = false,
                               bool enableStateChangeCallbacks = true );

                SDOlist const
                getDiscoveredSDOlist() const;

                void
                removeSDO( ProxyPtr const & pProxy );

                void
                addObserver( AbstractObserverPtr const & pAbstractObserver );

                void
                removeObserver( AbstractObserverPtr const & pAbstractObserver );

                ...
        };
    };
}
```

Like many of the middleware API elements, Subscription objects are pointed to by reference-counted "smart" pointers (called `SubscriptionPtr`). The `Subscription` instance is destroyed when the reference count reaches zero. The reference count is decremented when the `SubscriptionPtr` goes out of scope or is explicitly reset.

Applications that subscribe to an SDO type will typically define one or more Observer classes that provide application specific behavior associated with the SDO subscription events (e.g., discovery, update, destruction). The example applications include a simple printing and counting Observer. An abbreviation of the `Example::Vehicle::CountingObserver` class declaration is shown below.

```
class CountingObserver
: public Example::Vehicle::AbstractObserver
{
public:
    TENA::uint32
    getDiscoveredCount() const;

    virtual
    void
    discoveryEvent(
        Example::Vehicle::ProxyPtr const & pProxy,
        Example::Vehicle::PublicationStatePtr const & pPubState );

private:
    //! An MT-safe counter to count invocations of discoveryEvent()
    std::atomic< ACE_SYNCH_MUTEX, TENA::uint32 > discoveredCount_;
};
```

Release 6.0.5 Update  – Prior to middleware version 6.0.5, the type `TENA::Middleware::Utils::AtomicOp` was used instead of the C++11 standard `std::atomic` type.

The code declaration above illustrates how an Observer could be defined to count the number of `discoveryEvents` that occur for the `Example::Vehicle` SDO type. The class defines a member named `discoveredCount_` that is incremented in the `discoveryEvent` method that is called by the middleware each time a new SDO is discovered.

Note that the class member `discoveredCount_` is implemented with an atomic counter in case concurrent application programming threads are used for the evoke callback services (as discussed in the top section of this documentation).

After the application developer has developed (or reused) the necessary Observer classes, the application needs to instantiate these Observers, along with the type specific Subscription object. The Observers are attached to the Subscription object as shown in the code fragment below.

```
bool wantPruning(
    appConfig[ "pruneExpiredStateChange" ].getValue< bool >() );

Example::Vehicle::SubscriptionPtr pExampleVehicleSubscription(
    new Example::Vehicle::Subscription( wantPruning ) );

MyApplication::Example_Vehicle::CountingObserverPtr
pExampleVehicleCountingObserver(
    new MyApplication::Example_Vehicle::CountingObserver() );

pExampleVehicleSubscription->addObserver( pExampleVehicleCountingObserver );
```

The application configuration option `wantPruning` is an argument to the `Subscription` constructor to indicate if the application wants the middleware to eliminate redundant update notifications. This may be desired if the publishing application performs updates at a much higher rate than the subscribing application is able to process these updates, AND the subscriber only cares about the latest state values for the object. This topic is discussed in the [SDO received state processing documentation page](#).

After the `Subscription` object has been created and the Observers have been attached, the application can invoke the type-specific `subscribe` function. An illustration of this operation is shown in the code fragment below.

```
// Declare the application's interest in Vehicle objects.
Example::Vehicle::subscribe(
    pSession,
    pExampleVehicleSubscription,
    false /* no self-reflection */ );
```

Release 6.0.2 Update — An alternative subscribe method was added with release 6.0.2 of the middleware in which the SessionPtr argument is used, versus the Session reference (i.e., "*pSession") that was used in the previous releases. Users are encouraged to use the SessionPtr interface. See [MW-4286](#) for more details.

An example discoveryEvent method implementation is shown below for the Example::Vehicle SDO type from the Example-Vehicle object model. This example shows an observer that prints the SDO state each time it is updated.

```
void
MyApplication::Example_Vehicle::
PrintingObserver::
discoveryEvent( Example::Vehicle::ProxyPtr const &,
    Example::Vehicle::PublicationStatePtr const & pPubState )
{
    this->print( "Discovery      : Example::Vehicle", 'd', pPubState );
}

void
MyApplication::Example_Vehicle::
PrintingObserver::
print(
    std::string const & message,
    char letter,
    Example::Vehicle::PublicationStatePtr const & pPubState )
{
    if ( verbosity_ > 1 )
    {
        os_ << message << std::endl;
        if ( verbosity_ > 2 )
        {
            os_ << pPubState << std::endl;
            if ( verbosity_ > 3 )
            {
                os_ << pPubState->getMetadata() << std::endl;
            }
        }
    }
    else if ( verbosity_ == 1 )
    {
        os_ << letter << std::flush;
    }
}
```

In the above discoveryEvent example, the PrintingObserver calls a print methods that makes use of two application variables (verbosity, os) that were passed into the constructor (as shown below). Since the application developer designs the Observer classes, providing access to application specific information is easily handled.

```
namespace MyApplication
{
    namespace Example_Vehicle
    {
        class PrintingObserver
            : public virtual Example::Vehicle::AbstractObserver
        {
        public:
            PrintingObserver( TENA::uint32 verbosity, std::ostream & os );
```

Note that it is possible that the application does not need to define any Observers for a particular situation. In this case, the application will not receive notification of any of the subscription events, but instead can periodically obtain the discovered SDO list from the Subscription object and operate on the discovered SDO proxies in that manner. The code fragment shows an example of printing out the size of the discovered SDO list.

```

Example::Vehicle::SDOlist discoveredList(
    pExampleVehicleSubscription->getDiscoveredSDOlist() );

std::cout << "There are " << discoveredList.size()
    << " discovered Example::Vehicle SDOs on the list.\n"
    << std::endl;

```

As described in the documentation, subscribing applications **must** instruct the middleware to perform callback processing. The code fragment below shows an application that uses `evokeMultipleCallbacks` for an interval of time within a main processing loop.

```

for ( TENA::uint32 i = 1; i <= numberIterations; ++i )
{
    pSession->evokeMultipleCallbacks( microsecondsPerIteration );

    // Application loop processing
    ...
}

```

Java API Reference and Code Examples

As described above, subscriptions are initiated using the `subscribe` static method. In the Java Binding, alternate signatures are provided for the different ways of subscribing. An example `subscribe` declaration for `Example.Notification` Message type from the `Example-Vehicle` object model is shown below.

```

package Example.Vehicle;

class Subscription {
    ...
    // Static methods...
    static void subscribe(Session pSession, Subscription subscription);
    static void subscribe(Session pSession, Subscription subscription, boolean enableSelfReflection);
    static void subscribe(Session pSession, Subscription subscription, boolean enableSelfReflection, Filter
filter);

    static void unsubscribe(Session sess);
    static void unsubscribeAndRelease(Session sess);
    ...
}

```

Typically any `Observer` instances are added to the `Subscription` before passing it to `subscribe`. This ensures that the `Observers` will be active if events happen immediately after subscribing. The class below shows the `Subscription` class generated for an `Example.Vehicle` SDO.

```

package Example.Vehicle;

class Subscription {
    public Subscription();
    public Subscription(boolean isPruning);
    public Subscription(boolean isPruning, boolean enableStateChangeCallbacks); // Disable state change for
efficiency, when polling Proxy publication state

    public void addObserver(AbstractObserver pAddedAbstractObserver);
    public void removeObserver(AbstractObserver pRemovedAbstractObserver);
    public SDOlist getDiscoveredSDOlist();

    public void removeSDO( ProxyPtr const & pProxy );
}

```

Like many of the middleware API elements, `Subscription` objects are pointed to by reference-counted "smart" pointers (called `SubscriptionPtr`). The `Subscription` instance is destroyed when the reference count reaches zero. The reference count is decremented when the `SubscriptionPtr` goes out of scope or is explicitly reset.

Applications that subscribe to an SDO type will typically define one or more `Observer` classes that provide application specific behavior associated with the SDO subscription events (e.g., discovery, update, destruction). The example applications include a simple printing and counting `Observer`. An abbreviation of the `Example.Vehicle.CountingObserver` class declaration is shown below.

```

class CountingObserver extends Example.Vehicle.AbstractObserver {
    private int discoveredCount_ = 0;
    private int changedCount_ = 0;
    private int destructedCount_ = 0;

    public int getDiscoveredCount() { return discoveredCount_; }
    public int getChangedCount() { return changedCount_ ; }
    public int getDestructedCount() { return destructedCount_ ; }

    public void discoveryEvent( Example.Vehicle.Proxy proxy, Example.Vehicle.PublicationState pubState ) {
        ++discoveredCount_;
        // release proxy to allow unsubscribe to be independent of garbage collection
        proxy.releaseReference();
    }
    ...
};

}

```

The code declaration above illustrates how an Observer could be defined to count the number different events that occur for the `Example.Vehicle` SDO type. The class defines a member named `discoveredCount_` that is incremented in the `discoveryEvent` method that is called by the middleware each time a new SDO is discovered. This example does not provide any concurrency protection. If this were used in an application where `evokeCallback` might occur on multiple threads, it would be necessary to synchronize on access to the private state or use classes such as those provided by the `java.util.concurrent.atomic` package.

After the application developer has developed (or reused) the necessary `Observer` classes, the application needs to instantiate these `Observers`, along with the type specific `Subscription` object. The `Observers` are attached to the `Subscription` object as shown in the code fragment below.

```

Example.Vehicle.Subscription vehicleSubscription =
    new Example.Vehicle.Subscription( false /*wantPruning*/ );

MyApplication.Example_Vehicle.CountingObserver vehicleCountingObserver =
    new MyApplication.Example_Vehicle.CountingObserver();

vehicleSubscription.addObserver( pExampleVehicleCountingObserver );

```

The `wantPruning` option is an argument to the `Subscription` constructor to indicate whether the application wants the middleware to eliminate redundant update notifications. This may be desired if the publishing application performs updates at a higher rate than the subscribing application is able to process these updates, AND the subscriber only cares about the latest state values for the object. This topic is discussed in the [SDO received state processing documentation page](#).

After the `Subscription` object has been created and the `Observers` have been attached, the application can invoke the type-specific `subscribe` function. An illustration of this operation is shown in the code fragment below.

```

// Declare the application's interest in Vehicle objects.
Example.Vehicle.subscribe(
    session,
    vehicleSubscription,
    false /* no self-reflection */ );

```

Note that it is possible that the application does not need to define any `Observers` for a particular situation. In this case, the application will not receive notification of any of the subscription events, but instead can periodically obtain the discovered SDO list from the `Subscription` object and operate on the discovered SDO proxies in that manner. The code fragment shows an example of printing out the size of the discovered SDO list.

```

Example.Vehicle.SDOList discoveredList = vehicleSubscription.getDiscoveredSDOList();
System.out.println( "There are " + discoveredList.size() +
    " discovered Example.Vehicle SDOs on the list." );

```

As described in the documentation, subscribing applications **must** instruct the middleware to perform callback processing. The code fragment below shows an application that uses `evokeMultipleCallbacks` for an interval of time within a main processing loop.

```
for ( int i = 1; i <= numberOfIterations; ++i )
{
    pSession.evokeMultipleCallbacks( microsecondsPerIteration );
    // Application loop processing
    ...
}
```

SDO Proxy

SDO Proxy

A subscribing application obtains (or discovers) an SDO Proxy that corresponds to a (typically) remote SDO Servant. The SDO proxy contains the latest state (i.e., attribute values) of the servant, which is disseminated from the servant to all corresponding active proxies when the state is updated. If the SDO class type has remote methods defined, then the SDO proxy will have an interface to allow the subscribing application to invoke those remote methods.

Description

The SDO Proxy encapsulates the subscriber-side implementation of the Stateful Distributed Object corresponding to a particular publisher-side implementation called the [SDO Servant](#). A subscribing application obtains an SDO Proxy through a `subscribe` call that either specifies the desired SDO type (along with optional [Advanced Filtering](#) filter) or provides an [SDO Pointer](#) that refers to a particular SDO Servant. Once a subscribing application obtains an SDO Proxy, any state updates made to the corresponding SDO Servant will be disseminated to every "active" SDO Proxy in the execution. Note that the subscribing application is not required to perform any action to cause the active SDO Proxy's state to be updated.

An SDO Proxy is considered active when the application has not issued an `unsubscribe` for the SDO's type. Note that the `unsubscribe` operation does not affect SDO Proxies that were obtained through an SDO Pointer `subscribe` operation (referred to as "instance-based subscription" because the application is only subscribing to a particular SDO instance). After the `unsubscribe` invocation, the SDO Proxies that become inactive will no longer receive updates made to the corresponding SDO Servants. An SDO Proxy may also become inactive when Advanced Filtering is used and the Tag (referred to `FilteringContext` prior to 6.0.4 version) of the SDO Servant no longer matches the subscription `Filter`. In this situation, a `leftScopeEvent` will be invoked for all registered `Observers` associated with that SDO Proxy's subscription.

Whether the SDO Proxy is active or inactive (i.e., receiving or not receiving updates, respectively), if the SDO Servant is [destroyed](#) the subscribing application will be notified of the destruction through a `destructionEvent` through the `Observers` associated with that SDO Proxy's subscription. When the `DestructionEvent` invocations are completed, the SDO Proxy will be internally marked as invalid and any attempt to invoke methods on the SDO Proxy will result in a runtime exception. An SDO Proxy can be checked to determine if it has been invalidated through the `isValid` method on the proxy's interface. Note that checking if an SDO Proxy is invalid before invoking a method on the proxy will not guarantee that the proxy will be valid at the time of a subsequent proxy method invocation when the application uses a separate programming thread to [evoke callbacks](#) due to the inherent race condition.

As mentioned, the state of the proxy will be asynchronously updated when the middleware (associated with the subscribing application) receives an update from the corresponding servant. The middleware ensures that these updates are applied atomically, such that when the subscribing application invokes `getPublicationState` the returned `PublicationState` will represent attribute values associated with a particular update (also forced to be made atomically) on the servant. In this manner, a subscribing application can operate on SDO Proxies to read the state in a polling (or "pull") manner whenever the application needs to update the state. There is no need for the subscribing application to evoke callbacks to have the proxies state to be updated. However, the `evoke callbacks` operation is necessary to trigger the subscription `Observer` events (described in the [observer page](#)), which include the `discoveryEvents` that are necessary to discovery SDO Proxies based on a type-based subscription.

If a subscribing application prefers to be notified ("push" mechanism) when an SDO Proxy's state is modified (due to receiving an update from the servant), then the application can write an `Observer::stateChangeEvent` method that is attached to the subscription. The user code written in the `{(stateChangeEvent)}` will then be executed as part of the callback processing, requiring the subscribing application to call one of the `evoke callback` methods.

When the SDO type has defined [remote methods](#), a subscribing application can invoke those methods on the SDO Proxy. These method invocations will be transmitted by the middleware to the application that created the corresponding SDO Servant. Therefore, it should be recognized that the SDO remote method invocations may block for some indeterminate amount of time that includes the network transport delays and the servant method processing time.

A remote method without any return arguments (or out parameters or exceptions) can designate the method to be "oneway" to avoid this blocking behavior, but oneway methods are not recommended because the invoking application is unable to determine if the method invocation was successful. Instead, an application that is concerned with the remote method invocation delay should consider spawning a separate programming thread to perform the invocation.

If a subscribing application is not interested in a particular SDO Proxy, the application can "let go" of the proxy (or call `reset` on the `ProxyPtr` interface) to indicate to the middleware that the application is no longer interested in receiving state updates or other events associated with that proxy. If the proxy was obtained through an SDO Pointer (instance-based subscription), the middleware will notify the publishing application that created the servant that the application is no longer interested and the updates will no longer be delivered. If the proxy was obtained through a type-based subscription, letting go of the proxy will result in receive-side filtering of the updates (i.e., the updates for that proxy will still be transmitted to the subscribing application, but dropped by the middleware).

Note that SDO Proxies are wrapped in a smart pointer that performs reference counting. An application may make multiple copies of a particular proxy and only when all references are eliminated will the underlying proxy actually be destroyed.

Usage Considerations

SDO Proxy List — The type-based [SDO subscription mechanism](#) provides a `Subscription` object that maintains a list of discovered SDO Proxies associated with a particular subscription. Applications should avoid creating a separate container for discovered proxies unless that is required. The `SDOList` maintained within the `Subscription` object is written to be thread-safe using copy on write semantics to improve performance when obtained for use by an application.

Exception Safety — When invoking operations on an SDO Proxy, the application should consider exception safety. As mentioned above, if the application uses multiple programming threads in which one or more threads is dedicated to evoking callbacks, then the application should use `try, catch` blocks around proxy method invocations to protect against a proxy being invalidated due to be destructed. Additionally, remote method invocations should always be wrapped in `try, catch` blocks because communication faults can occur with the remote method invocations.

C++ API Reference and Code Examples

The key methods associated with an SDO Proxy are defined below. All proxies are wrapped within a managed pointer, so that the name of the proxy class follows the form `Example::Vehicle::Proxy`.

```
// Returns a unique identifier for the SDO Servant.  
ObjectID const & getObjectID() const;  
  
// Provides a count for the number of times the servant was reactivated.  
TENA::uint32 getReactivationEpoch() const;  
  
// Returns the Session associated with the proxy.  
Middleware::Session & getSession() const;  
  
// Indicates if the servant was terminated by a TENA Console operator.  
bool isRemoved() const; // Formerly, "terminated()"  
  
// Indicates if the proxy is "in scope" with respect to advanced filtering.  
bool isInScope() const;  
  
// Returns an SDO Pointer to the servant.  
SDOpointerPtr const getSDOpointer() const;  
  
// Returns a copy of the current publication state (which also contains the  
// metadata). Note that the method is non-const because the proxy lazily  
// unpacks the publication state which modifies the internal class members.  
PublicationStatePtr const getPublicationState();  
  
// Changes the Subscription associated with the proxy, which changes the  
// Observers that will be processed when proxy events occur.  
void setSubscription( SubscriptionInterfacePtr const & pNewSubscription )  
  
// Returns the inheritance vector of typeIDs, with most derived type first.  
::TENA::Middleware::TypeIDpath const & getTypeIDpath() const
```

Java API Reference and Code Examples

The key methods associated with an SDO Proxy are defined below.

```
ObjectID      getObjectID();           // Returns a unique identifier for the SDO Servant.  
UnsignedInteger    getReactivationEpoch(); // Provides a count for the number of times the servant was  
reactivated  
boolean   isInScope();                // Indicates if the proxy is "in scope" with respect to  
advanced filtering.  
boolean   isRemoved();                // Indicates if the servant was terminated by a TENA Console  
operator.  
  
PublicationState getPublicationState(); // Returns a copy of the current publication state  
SDOpointer getSDOpointer();           // Returns an SDO Pointer to the servant.  
  
// Changes the Subscription (and associated Observers) associated with the proxy  
void setSubscription(Subscription pNewSubscription);  
  
// Force the reset of the Proxy  
void releaseReference();
```

Because holding a Proxy instance keeps a subscription active, even after a call to `Subscription.unsubscribe()`, it is important to reset any Proxy references as soon as possible. Each API that provides a Proxy is returning a new wrapped reference. Generally developer code should release these Proxy instances after they are used. Relying on garbage collection to release them may result in inefficient behavior in applications that change their subscriptions over time. Note that some use cases hold Proxy references in list or other structures. The Proxy instances held by the application in some structure should not have `releaseReference()` invoked.

Discovery Order

Discovery Order

In the case of an SDO (Stateful Distributed Object) class that uses inheritance, it is possible for a subscribing application to subscribe to multiple levels of the inheritance hierarchy. In this situation, the subscribing application will discover the object at each matching inheritance level and have multiple discovered proxies for the same SDO servant. Some applications will need to recognize that multiple proxies to the same SDO servant exist and perhaps discard the less derived versions. Deterministic discovery order is required to allow applications to perform this type of processing and the middleware guarantees that the ordering of discoveries (or message receipts in the case of Messages) will occur from most-derived to least-derived.

Description

Release 6.0.3 Update — The preferred approach to this issue is to use a "best match" session, which ensures that SDOs are only discovered as the most-derived type for which a subscription exists. See [Best Match](#).

Some applications may be required to subscribe to multiple classes within an SDO (StatefulDistributedObject) or Message inheritance hierarchy. In this situation, the subscribing application may receive multiple discoveries or messages associated with each subscribed type, even though the multiple discovered proxies or messages correspond to the same object or message. The middleware will "slice" off attributes and methods associated with less derived classes.

For example, consider the `Tank` SDO class that inherits from the `Vehicle` SDO class. A subscribing application can subscribe to both `Vehicle` and `Tank` types, then an application that creates a `Tank` object will result in the subscribing application to discover that object as both a `Vehicle` and `Tank` proxy. In some circumstances, the subscribing application needs to determine the "best match" when multiple discoveries are possible. In Release 6 of the TENA Middleware, the inheritance discovery mechanism was refined to ensure guaranteed ordering of the discoveries (or message receipt) from most-derived to least-derived. In the previous example, the subscribing application will receive the `Tank` discovery before the `Vehicle` discovery.

The deterministic inheritance discovery order will allow subscribing applications to ignore (or process differently) discoveries or messages for more generalized types when an object or message has already been received as a more specific type.

Usage Considerations

Subscribing applications that need to process objects or messages differently with respect to the inheritance location will need to develop a mechanism to check the type of the received object/message, or check if the same object/message has already been processed at a different derived class level.

For example, if there was a `Derived` SDO type that inherited from a `Base` SDO type, the `discoveryEvent` method associated with the `Base` Observer class could check the discovered proxy type path (i.e., inheritance hierarchy) and remove that discovered proxy if it is actually a `Derived` type (because it will already be discovered by the `Derived` Observer code). A code fragment for doing test in the `Base` Observer is illustrated below.

```
// BaseObserver::discoveryEvent() - Detect if the discovered proxy is
// actually a Derived type that will be handled by DerivedObserver.
if (pProxy->getTypeIDpath() != Derived::toTypeIDpath())
{
    pSubscription->removeSDO( pProxy );
}
```

As an alternative mechanism, an application may need to maintain a list of active proxies that are being held by the application. In this case, the `discoverEvent` method could check the the ID of a newly discovered object to determine if the object has already been discovered by comparing against the list of IDs. A similar mechanism can be used for processed messages.

Received State Processing

Received State Processing

When an application discovers an SDO (Stateful Distributed Object), a "proxy" for that object is provided to the application. The proxy can be used to read the state values for the object or invoke remote methods. When the publishing application updates the state of the "servant" object, the middleware will automatically update the state of all the proxies throughout the execution. In this way, the servant and proxy objects operate in a form of distributed shared memory. A subscribing application can attach observers to a particular object (through the subscription mechanism) so that observer code is executed whenever an event, such as a state change, occurs. When an object is being updated more frequently than the subscribing application is able to process the state change events, it may be important for an application to be able to access both the object state corresponding to the particular state change event or the latest state of the object. The middleware supports processing control when the subscribing application is only interested in the latest state of an object, as well as the situation in which every state update must be processed.

Description

TENA applications that subscribe to SDOs (Stateful Distributed Objects) will discover a proxy for all objects that match the subscription criteria. As long as the subscribing application holds onto the proxy, state updates made by the publishing application for that object will be propagated to the subscribing application. A subscribing application may select to be notified through a state change observer mechanism whenever a state change occurs, or the subscribing application may not need to attach any observers and just read the object state when necessary.

The default behavior of the TENA Middleware, with respect to the proxy state updates, is to over-write the proxy state whenever a more recent update is received by the subscribing application. Under some circumstances, an application may need to access every state update received for a particular object, rather than just obtain the latest values. The observer mechanism of the generated code supports a `stateChangeEvent` method that provides access to the proxy and the publication state associated with that particular state change event. Application developers can implement their code in the `stateChangeEvent` method to use both the state information for the particular state change event, as well as accessing the current state of the proxy, in case it has changed since the `stateChangeEvent` was queued for processing.

For subscribing applications that are only interested in the latest object state values, the `Subscription` function used to subscribe to a particular object model type accepts an argument related to "pruning" redundant state change events. When the application subscribes to a particular object type with pruning enabled, then when the first state change event is processed, the middleware will discard any subsequent state change updates that are currently queued. This prevents the processing of redundant state change events when the application only needs to process the latest state values and not every state change.

Note that in the case of utilizing the pruning option, the application should obtain the latest state from the proxy object, and not use the copy of the `PublicationState` that is passed in the `stateChangeEvent` method. In summary, applications that are required to process every state change event, such as data loggers, need to use the publication state passed into the `stateChangeEvent` method of the observer. Whereas, applications that are using the pruning mechanism (option on `Subscription` object) should obtain the publication state from the proxy during the `stateChangeMethod`.

C++ API Reference and Code Examples

Auto-generated example applications utilize an application configuration option to control the "pruning" behavior of the application with respect to subscriptions. The code fragment below illustrates the use of this configuration option in the `Subscription` function for the object model type `Example::Vehicle`.

```
bool wantPruning(
appConfig["pruneExpiredStateChange"].getValue< bool >() );

Example::Vehicle::SubscriptionPtr pExampleVehicleSubscription(
    new Example::Vehicle::Subscription( wantPruning ) );
```

An example of the use of a `stateChangeEvent` method is shown in the code fragment below. In this `PrintingObserver`, the `print` function is printing the state values of the publication state that was passed in as an argument to the `stateChangeEvent` method. The publication state values passed in as an argument correspond to the values corresponding to the particular state change event, and may not be the current values of the proxy. To obtain the current values of the proxy (which is appropriate when pruning is enabled), the application can obtain the publication state from the `ProxyPtr` argument.

```
void
MyApplication::Example_Vehicle::
PrintingObserver::
stateChangeEvent( Example::Vehicle::ProxyPtr const & pProxy,
    Example::Vehicle::PublicationStatePtr const & pPubState )
{
    this->print( " State Change: Example::Vehicle queued PubState", 'c', pPubState );
    this->print( " State Change: Example::Vehicle current PubState", 'c', pProxy->getPublicationState() );
}
```

Java API Reference and Code Examples

The code fragment below illustrates the using a flag to create a subscription that is pruning events queued before a subscriber could process them, in the `Subscription` function for the object model type `Example::Vehicle`.

```
Example.Vehicle.Subscription vehicleSubscription = new Example.Vehicle.Subscription( true /*wantPruning*/ );
```

An example of the use of a `stateChangeEvent` method is shown in the code fragment below. In this `PrintingObserver`, the `print` function is printing the state values of the publication state that was passed in as an argument to the `stateChangeEvent` method. The publication state values passed in as an argument correspond to the values corresponding to the particular state change event, and may not be the current values of the proxy. To obtain the current values of the proxy (which is appropriate when pruning is enabled), the application can obtain the publication state from the `proxy` argument.

```
package org.MyApplication.Example_Vehicle;
class PrintingObserver extends Example.Vehicle.AbstractObserver {
    ...
    void stateChangeEvent( Example.Vehicle.Proxy proxy,
                           Example.Vehicle.PublicationState pubState ) {
        System.out.println(" State Change: Example::Vehicle queued PubState " + pubState.getName() );
        System.out.println(" State Change: Example::Vehicle latest PubState " + proxy.getPublicationState().
getNome() );
    }
    ...
}
```

SDO Unsubscribe

SDO Unsubscribe

An application can invoke the static `unsubscribe` method to indicate that the application is no longer interested in discovering SDOs (Stateful Distributed Objects) of a particular type. An application would have invoked the related static `subscribe` method for that SDO type to cause the application to discover SDOs. The static `subscribe` and `unsubscribe` methods are defined in the namespace corresponding to the particular SDO type (e.g., `Example::Vehicle::unsubscribe`). For language bindings other than C++, they are defined on the Subscription type: for example, [Example.Vehicle.Subscription.unsubscribe\(\)](#) in the Java Binding.

Description

A subscribing application interested in all SDOs of a particular type will invoke the static `subscribe` method associated with the particular SDO type of interest. This **type-based** subscription will instruct the TENA Middleware to notify the subscribing application of all SDOs that match the subscribed type (including derived types). These notifications are handled by **Observer** classes that the application developer creates to provide application specific behavior when a matching SDO is discovered, updated, destructed, and, in the case of [Advanced Filtering](#), when a discovered SDO goes in or out of scope.

In addition to the type-based SDO subscription, there are two other forms of subscription that are supported by the middleware. The first is **advanced filtering** subscription. Here an application augments a type-based subscription with a **Filter**, an integer tag that provides additional interest criteria (typically based on SDO attribute values) to identify matching SDOs. The second form of subscription is **instance-based** subscription, when an application de-references an **SDO Pointer** to obtain an SDO Proxy for that particular SDO instance. Additional information on SDO subscription can be found on the [SDO Subscription documentation page](#).

After a subscribing application has performed an SDO type-based subscription operation, the middleware will notify the application of matching discoveries and provide the application an SDO Proxy that represents the particular SDO Servant that exists within the execution and matches the subscription criteria. Applications can then hold onto the proxy to receive state updates (made to the servant by the publishing application), scope changes (in the case of advanced filtering), and destruction notifications (when the publishing application destroys the servant).

If the subscribing application wants to stop receiving discoveries and updates associated with the type-based subscription operation, the application can invoke the static `unsubscribe` method. This method requires the `Session` and optionally the `Filter` corresponding to the original subscription. See the API examples below for examples.

Resulting Behavior after `unsubscribe` Invocation

The unsubscribe operations associated with type-based and advanced filtering subscriptions are processed independently and only affect their corresponding subscription. For example, consider an application that issues a advanced filtering subscription to SDO type `Example::Vehicle` with `Filter = altitudeFilter` and then issues an advanced filtering subscription to SDO type `Example::Vehicle` with `Filter = geographicFilter`. These overlapping subscriptions are handled by the middleware separately, so an `Example::Vehicle::unsubscribe(pSession, altitudeFilter)` will not affect the `geographicFilter` subscription.

Additionally, the `unsubscribe` operation will have no effect on instance-based subscriptions of the same SDO type. Any SDO Proxy obtained through de-referencing an SDO Pointer will continue to behave normally after an `unsubscribe`. This is because an instance-based subscription is also treated independently of any type-based subscriptions and corresponding unsubscribe operations.

After an application unsubscribes, it will no longer be notified of any SDOs that match the corresponding subscription (i.e., no new discoveries will occur). Any existing SDO Proxies that were discovered due to the corresponding subscription will no longer receive state updates, or notifications to scope changes if advanced filtering is used. However, the subscribing application will be notified if the publishing application destroys the SDO Servant: a `destructionEvent` will be invoked on any registered **Observers**, and that Proxy will be invalidated. Additional information on SDO Destruction can be found on the [SDO Destruction documentation page](#).

Unsubscribing from Instance-based Subscription

There is no "unsubscribe" method for unsubscribing from an instance-based subscription. The instance-based subscription created by calling "subscribe" on an SDO pointer ends when the Proxy returned by that call is destroyed. The Proxy object is destroyed when all references to the Proxy have gone out of scope or the `reset` method is called on the pointers. If you use the `Subscription` class provided by the OM definition, in addition to releasing all Proxy references held by your application you will also need to call the `removeSDO` method of that `Subscription` class to remove the Proxy reference held by it.

Usage Considerations

Generally, an application will not need to perform SDO unsubscribe operations, as most applications will stay subscribed to a particular SDO type until the application resigns. If the application does perform unsubscribe operations, the application needs to consider how to identify (and perhaps destroy beforehand) any SDO Proxies that correspond to the unsubscribe operation. These proxies may have stale publication state providing misleading information to the application.

⚠ — Applications are responsible for properly managing SDO proxies that correspond to an unsubscribe operation. These proxies may become "stale" with respect to the current servant publication state since the middleware does not deliver updates for unsubscribed proxies.

Performance Considerations with `unsubscribe` Invocation

There are performance considerations associated with the `unsubscribe` operation when there are many SDO Proxies that have been discovered by the application. In order for the subscribing application to still be notified of destruction events, the unsubscribing application needs to inform the publishing applications corresponding to the existing SDO Proxies that the particular unsubscribing application is no longer interested in receiving update and state change events for certain SDO Servants, but the application does want to receive destruction notifications. This is because the unsubscribing application is still using those SDO Proxies and the middleware is compelled to provide notification of the destruction of the corresponding servants. There is network communication and processing required to establish this "destruction only notification" relationship between the publishers and the unsubscribing application. If the unsubscribing application is no longer interested in these proxies, then the proxies should be destroyed **before** the `unsubscribe` method is invoked.

The middleware `Subscription` object that was used for the type-based subscription contains a list of discovered SDO Proxies. If the unsubscribing application is no longer interested in the discovered proxies, the `Subscription` object can be destroyed prior to the `unsubscribe` invocation. Release 6.0.4 of the middleware added a `removeAllSDOs()` method to the `Subscription` class to remove all of the proxies if necessary. The existing `getSDOList()` method only returns a copy of the list and therefore can't be used to destroy the proxies contained within the `Subscription` object.

! — The `Subscription::removeAllSDOs()` method is available for TENA Middleware version 6.0.4 and beyond. See [MW-5202](#).

C++ API Reference and Code Examples

A code fragment of an application using `unsubscribe` is shown below for the `Example::Vehicle` SDO type.

```
// Declare the application's interest in Vehicle objects.  
Example::Vehicle::subscribe(  
    pSession,  
    pExampleVehicleSubscription,  
    false /* no self-reflection */ );  
  
...  
  
// Unsubscribe to Vehicle objects.  
Example::Vehicle::unsubscribe( pSession );
```

Release 6.0.2 Update — Alternative `subscribe` and `unsubscribe` methods were added with release 6.0.2 of the middleware in which the `SessionPtr` argument is used, versus the `Session` reference (i.e., `"*pSession"`) that was used in the previous releases. Users are encouraged to use the `SessionPtr` interface. See [MW-4286](#) for more details.

The static `unsubscribe` method is defined in the namespace of the particular SDO type, as shown in the definition for `Example::Vehicle` SDO type in the code below.

```
namespace Example {  
    class Vehicle {  
        ...  
        static  
        void  
        unsubscribe(  
            ::TENA::Middleware::SessionPtr session,  
            ::TENA::Middleware::Filter const & filter =  
                ::TENA::Middleware::Filter() );  
        ...  
    }  
}
```

! — Note that the actual definition code uses a `typedef` to the actual class (e.g., `Vehicle_SDO`) to support backwards compatibility with the previous versions of the middleware, but application developers should use the class names without the appending `_SDO` in their application code.

If the subscribing application is not using advanced filtering, the `unsubscribe` method will provide a default `Filter` argument that corresponds to a type-based subscription. If advanced filtering is employed, the application needs to provide a valid `Filter` that was previously used for an advanced filtering subscription.

Java API Reference and Code Examples

The static `unsubscribe` and `unsubscribeAndRelease` methods are defined on the `Subscription` class generated for each SDO, for example `Example.Vehicle.Subscription`. A code fragment of an application using `unsubscribeAndRelease` is shown below for the `Example.Vehicle` SDO type.

```
// Declare the application's interest in Vehicle objects.  
Example.Vehicle.Subscription.subscribe(session, vehicleSubscription);  
...  
  
// Unsubscribe to Vehicle objects.  
Example.Vehicle.unsubscribeAndRelease( session, vehicleSubscription );
```

The method `unsubscribeAndRelease` is added in the Java language binding, to attempt to ensure that all Proxies are released before unsubscribing by type. This is an issue because of Java garbage collection and how the middleware maintains subscriptions after a call to `unsubscribe`. This approach does not prevent problems if an application holds `Proxy` references itself, or fails to release `Proxy` references in Observer methods. See the Java example in [Observer Mechanism](#).

If advanced filtering is employed, the application needs to provide a valid `Filter` that was previously used for an advanced filtering subscription.

SDO Update

SDO Update

Changes to an SDO Servant's state (i.e., attribute values) are disseminated to subscribers via *updates*. An SDO Updater is created from the Servant and used to update the individual attribute values. When the values are properly set, the updater is then applied to the SDO Servant and updates are disseminated to subscribers of the particular SDO.

Description

Changes to the SDO state are disseminated to subscribers via *updates*. In order to provide applications with control over the frequency of updates and to demarcate SDO state changes clearly in the application, the Middleware requires that SDO state changes be marshaled in a `PublicationStateUpdater` object. To update the state of an SDO, the publisher will:

1. Acquire a new `PublicationStateUpdater` using `Servant::createUpdater` method.
2. Perform any state changes on the `PublicationStateUpdater` instance.
3. Commit the updater.

The reliance on an updater object ensures that applications can mutate the overall state of the SDO atomically. The act of committing the updater is what actually changes the SDO state and triggers the dissemination of state updates to subscribing applications. If no changes are made to the updater, no state update is sent.

Usage Considerations

Thread safety

The act of committing an updater is thread-safe; that is, multiple threads may call `Servant::commitUpdater` and the Middleware will ensure that each update happens atomically. However, modifications to the `PublicationStateUpdater` itself are **not** thread-safe. If application threads need to modify the same updater or access an updater while another thread may be modifying it, it is up to the application to provide and use thread synchronization primitives (e.g., mutexes) to ensure correct operation.

In general, application developers are encouraged to avoid designs that would require sharing an Updater between threads.

C++ API Reference and Code Examples

Returning to the Tank SDO declaration in TDL:

```
exception CannotLoadSoldier { string reason; };

class Tank : extends Vehicle
{
    ... // Method declarations elided for brevity.
    Soldier * pDriver;
    vector < Soldier * > passengers;
    optional float32 damageInPercent;
};
```

From this TDL, the Middleware generates this `PublicationStateUpdater` for the Tank SDO:

```

class PublicationStateUpdater
    : public Example::Vehicle::PublicationStateUpdater
{
public:
    Example::Soldier_SDOpointerPtr const
    get_pDriver()
        const;

    std::vector< Example::Soldier_SDOpointerPtr > const
    get_passengers()
        const;

    TENA::float32
    get_damageInPercent()
        const;

    bool
    is_damageInPercent_set()
        const;

    void
    set_pDriver( Example::Soldier_SDOpointerPtr const & );

    void
    set_passengers( std::vector< Example::Soldier_SDOpointerPtr > const & );

    void
    set_damageInPercent( TENA::float32 );

    void
    unset_damageInPercent();
};


```

Each of the SDO's attributes has a corresponding accessor and mutator function on the PublicationStateUpdater. Just as the Example::Tank SDO inherits the Example::Vehicle SDO in TDL, the Tank PublicationStateUpdater inherits the Vehicle PublicationStateUpdater. As such, all of the member functions to access and change the Vehicle state are also available on the Tank PublicationStateUpdater. Also, note the additional members associated with the [optional](#) attributes damageInPercent, is_damageInPercent_set and unset_damageInPercent.

Create an Updater

The PublicationStateUpdater is created with a call to Servant::createUpdater:

```

std::unique_ptr< Example::Tank::PublicationStateUpdater > tankUpdater =
    tankServant->createUpdater();

```

Update the Attributes

The PublicationStateUpdater's accessor functions can be used to query the current state of the SDO (or, more accurately, the state of the SDO at the time the PublicationStateUpdater was created). This is particularly convenient in cases where the new state should be influenced by the old state. Note, however, that once you mutate the state of the updater, the accessors will return the mutated state—even if the updater has not yet been committed. The actual SDO state does not change until the updater is committed.

```

tankUpdater->set_name( updateLabel );
tankUpdater->set_team( Example::Team_Blue );
tankUpdater->set_location( Example::Location::create( 10.0, 20.0 ) );
tankUpdater->set_damageInPercent( 20.0 );

```

optional attribute considerations

As far as attribute access is concerned, the `is_*_set` functions allows applications to query whether an [optional](#) attribute has been set:

```

if (tankUpdater->is_damageInPercent_set())
{
    // damageInPercent has been set for the SDO/updater; you can safely call get_damageInPercent.
}
else
{
    // damageInPercent has not been set for the SDO/updater; a call to get_damageInPercent would throw.
}

```

It is important to note that the `is_*``_set` function actually queries the updater instance rather than the SDO itself. That means that once you have set a value on the updater, the corresponding `is_*``_set` function will return `true` even though the updater has not yet been committed.

Note that optional attribute access follows the same rule that applies to the publication state: if your application's logic leaves **any** possibility that an optional attribute is unset, you **must** check this before attempting to access the attribute's value (or be prepared for the resulting exception).

Unless the optional attribute is also `const`, you can use the updater to "unset" the attribute:

```
tankUpdater->unset_damageInPercent();
```

Commit the updater

Once all desired changes to the SDO state have been made, it's time to commit the updater. Committing the updater is what actually changes the SDO's state and triggers the propagation of state updates to subscribers. To commit the updater, simply call `Servant::commitUpdater`:

```
tankServant->commitUpdater( std::move( tankUpdater ) );
```

Note that the call to `Servant::commitUpdater` takes the `tankUpdater` `auto_ptr` by value, transferring ownership to the `commitUpdater` call. You cannot use the updater after the call to `commitUpdater`; in fact, the `auto_ptr` will be null at this point.

Java API Reference and Code Examples

Returning to the Tank SDO declaration in TDL:

```

exception CannotLoadSoldier { string reason; };

class Tank : extends Vehicle
{
    ... // Method declarations elided for brevity.
    Soldier * pDriver;
    vector < Soldier * > passengers;
    optional float32 damageInPercent;
};

```

From this TDL, the Middleware generates this `PublicationStateUpdater` for the Tank SDO:

```

class PublicationStateUpdater
{

    // Updater methods from TDL base class elided

    public Example.Soldier.SDOpointer get_pDriver();
    public Example.Soldier.SDOpointerVector get_passengers();
    public float get_damageInPercent();
    public boolean is_damageInPercent_set();
    public void set_pDriver( Example.Soldier.SDOpointer );
    public void set_passengers( Example.Soldier.SDOpointerVector );
    public void set_damageInPercent( float );
    public void unset_damageInPercent();
};

```

⚠ Note: In the C++ implementation, there is an inheritance relationship between a derived class Updater and its base class Updater. This allows, for example, a Vehicle Servant to be updated by a (derived) Tank Updater. There is no use-case for this since the extra state will simply be ignored, and this relationship is removed in the Java Binding. The derived Updater simply mirrors the accessor and mutators from the base classes. Each of the SDO's attributes has a corresponding accessor and mutator function on the PublicationStateUpdater. Also, note the additional members associated with the `optional` attributes `damageInPercent`, `is_damageInPercent_set` and `unset_damageInPercent`.

Create an Updater

The PublicationStateUpdater is created with a call to `Servant.createUpdater`:

```
Example.Tank.PublicationStateUpdater tankUpdater = tankServant.createUpdater();
```

Update the attributes

The PublicationStateUpdater's accessor functions can be used to query the current state of the SDO (or, more accurately, the state of the SDO at the time the PublicationStateUpdater was created). This is particularly convenient in cases where the new state should be influenced by the old state. Note, however, that once you change the state of the updater, the accessors will return the changed state—even if the updater has not yet been committed. The actual SDO state does not change until the updater is committed.

```
tankUpdater.set_name( updateLabel );
tankUpdater.set_team( Example.Team.Team_Blue );
tankUpdater.set_location( Example::Location::create( 10.0, 20.0 ) );
tankUpdater.set_damageInPercent( 20.0 );
```

Optional attribute considerations

As far as attribute access is concerned, the `is_*_set` functions allows applications to query whether an `optional` attribute has been set:

```
if (tankUpdater.is_damageInPercent_set())
{
    // damageInPercent has been set for the SDO/updater; you can safely call get_damageInPercent.
}
else
{
    // damageInPercent has not been set for the SDO/updater; a call to get_damageInPercent would throw.
}
```

It is important to note that the `is_*_set` function actually queries the updater instance rather than the SDO itself. That means that once you have set a value on the updater, the corresponding `is_*_set` function will return `true` even though the updater has not yet been committed.

Note that optional attribute access follows the same rule that applies to the publication state: if your application's logic leaves **any** possibility that an optional attribute is unset, you **must** check this before attempting to access the attribute's value (or be prepared for the resulting exception).

Unless the optional attribute is also `const`, you can use the updater to "unset" the attribute:

```
tankUpdater.unset_damageInPercent();
```

Commit the updater

Once all desired changes to the SDO state have been made, it's time to commit the updater. Committing the updater is what actually changes the SDO's state and triggers the propagation of state updates to subscribers. To commit the updater, simply call `Servant::commitUpdater`:

```
tankServant.commitUpdater(tankUpdater);
```

Note that once an `Updater` has been passed to `Servant.commitUpdater`, it may not be reused. Attempts to reuse an `Updater` will raise a `NullPointerException`.