Refinitiv Management Classes Version 2.2.0

DEVELOPERS GUIDE

Document Version: 2.2.0
Date of issue: December 2020
Document ID: RMC220UM.200

REFINITIV™

Contents

1 Overview 1
1.1 What are the Refinitiv Management Classes? 1
1.2 Audience 1
1.3 Additional Resources: C++ Programming 1
1.4 Additional Resources: Object-Oriented Concepts 1
1.5 Organization of Refinitiv Management Classes Documents 1
1.6 Chapter Overview 2
1.7 Conventions 2
1.7.1 Typographic 2
1.7.2 Programming 2
1.7.3 Notation for Class Diagrams 3
1.8 Refinitiv Management Classes Development Process and Change Management Policies 3
1.9 Availability 3

2 Design Philosophy 4
2.1 Refinitiv Management Classes Modelling Principles 4
2.2 Object-Oriented Principles 4
2.3 Design by Contract 5
2.4 Component Design 5
2.5 Application Design 5

3 Refinitiv Management Classes Framework 6
3.1 Introduction 6
3.2 Application Roles 6
3.2.1 Managed Application 6
3.2.2 Managing Application 6
3.3 Refinitiv Management Classes Infrastructure 7
3.3.1 Refinitiv Management Classes API 7
3.4 Programming Constructs 7
3.4.1 How Publishers Publish Information 7
3.4.2 How Consumers Use Proxies and Handles 8
3.4.3 Browser-Based Views 9
3.4.4 Directory-Based Views 9
3.5 Application Architectures 10
3.5.1 Managed vs. Managing Applications 10
3.5.2 Browser vs. Directory-based Access 13
3.6 Pseudo Code Example 14

4 Managing Applications 15
4.1 Overview 15
4.2 Common Concepts 16
4.2.1 Accessing Handles 16
4.2.2 Client Management 16
4.3 Proxy Managed Variables 17
4.3.1 Identity 18
4.3.2 Attributes 19
4.3.3 State 24
4.3.4 Proxy Managed Variable Client Management 25
4.3.5 Proxy Managed Variable Clients 26
4.3.6 Class Diagrams (Simplified) 28

4.4 Proxy Managed Object ... 31
4.4.1 Identity ... 31
4.4.2 State ... 33
4.4.3 Access to Child Proxy Managed Objects ... 34
4.4.4 Access to Contained Managed Variables ... 36
4.4.5 Proxy Managed Object Client Management ... 37
4.4.6 Proxy Managed Object Clients ... 37
4.4.7 Class Diagram (Simplified) ... 40
4.5 Proxy Managed Object Server ... 41
4.5.1 State ... 42
4.5.2 Sequential Access to Root Proxy Managed Object Handles ... 43
4.5.3 Access to Proxy Managed Objects ... 43
4.5.4 Proxy Managed Object Server Client Management ... 43
4.5.5 Proxy Managed Object Server Clients ... 44
4.5.6 Class Diagram (Simplified) ... 45
4.6 Proxy Managed Object Server Pool ... 46
4.6.1 Access to Proxy Managed Object Servers ... 47
4.6.2 Proxy Managed Object Server Pool Client Management ... 47
4.6.3 Proxy Managed Object Server Pool Clients ... 48
4.7 Shared Memory Proxy Managed Object Server Pool ... 49
4.7.1 Constructor ... 49
4.7.2 Operations ... 49
4.8 Proxy Managed Object Class Directory ... 50
4.8.1 Attributes ... 50
4.8.2 Access to Proxy Managed Object Handles ... 50
4.8.3 Proxy Managed Object Class Directory Client Management ... 52
4.8.4 Proxy Managed Object Class Directory Clients ... 53
4.8.5 Class Diagram (Simplified) ... 54
4.9 Proxy Managed Object Class Directory Factory ... 54
4.10 Shared Memory Proxy Managed Object Class Directory Factory ... 55
4.11 Proxy Managed Object Pool ... 56
4.11.1 Access to Proxy Managed Objects ... 57
4.11.2 Proxy Managed Object Pool Client Management ... 58
4.11.3 Proxy Managed Object Pool Clients ... 58
4.11.4 Class Diagram (Simplified) ... 59
4.12 Managing Application Example ... 60
4.12.1 Managing Application Example ... 60

5 Managed Applications ... 73
5.1 Overview ... 73
5.2 Common Concepts ... 73
5.2.1 Client Management ... 73
5.3 Managed Variables ... 74
5.3.1 Identity ... 75
5.3.2 Attributes ... 75
5.3.3 Class Diagrams (Simplified) ... 81
5.4 Public Variable Types ... 83
5.4.1 Operations ... 84
5.5 Managed Object ... 86
5.5.1 Identity ... 86
5.5.2 State ... 87
5.5.3 State Attributes ... 88
5.5.4 Access to Child Managed Objects ... 88
5.5.5 Access to Contained Managed Variables ... 89
5.5.6 Managed Object Client Management ... 90

5.5.7 Managed Object Clients ... 90
5.5.8 Operations ... 91
5.5.9 Event Processing ... 91
5.5.10 Class Diagram (Simplified) ... 91
5.6 Public Object ... 92
5.7 Managed Process ... 92
5.8 Server Shared Memory Root ... 93
5.8.1 State ... 93
5.8.2 Example ... 94
5.9 Shared Memory Managed Object Server Memory Pool ... 94
5.9.1 State ... 94
5.9.2 Attributes ... 94
5.9.3 Operation ... 95
5.9.4 Example ... 95
5.9.5 Class Diagram (Simplified) ... 96
5.10 Shared Memory Managed Object Server ... 96
5.10.1 Constructors ... 96
5.10.2 States ... 97
5.10.3 Identity ... 97
5.10.4 Attributes ... 97
5.10.5 Operations ... 97
5.11 Shared Memory Server ... 97
5.11.1 Constructors ... 97
5.11.2 States ... 98
5.11.3 Identity ... 98
5.11.4 Attributes ... 98
5.11.5 Operations ... 98

6 Additional Topics ... 99
6.1 Thread Safety of Refinitiv Management Classes 2.X ... 99
6.1.1 Thread Safe Classes ... 99
6.1.2 Guidelines for Multi-threaded Applications to Use Refinitiv Management Classes 2.X Safely ... 100
6.2 Event Loops ... 100
6.3 I/O Events ... 100
6.4 Timer Events ... 101
6.5 Example - Event Loops, I/O and Timing ... 101
6.6 Smart Pointers ... 103
6.7 Performance Constraints ... 103

Appendix A Glossary ... 104

1 Overview

1.1 What are the Refinitiv Management Classes?

Refinitiv Management Classes are a collection of C++ classes which model the problem domain of application management. It is a shared memory implementation for management applications. At the highest level, the Refinitiv Management Classes defines a methodology for publishing and subscribing to management information efficiently. It provides a publisher application (also called the **managed application**) a means to publish information into shared memory. It also enables a consumer application (also called the **managing application**) to monitor/manage events published by the publisher application (also called the **managed application**) of the Refinitiv Real-Time Distribution System Infrastructure within the same node.

This release of the Refinitiv Management Classes is for use by clients, third parties, and Refintiv groups. Based on this API, managing applications can be created to view (and update if allowed) managed variables and states from publishing applications (e.g., from a Refinitiv Real-Time Advanced Distribution Server or Refinitiv Real-Time Advanced Data Hub).

The purpose of the Refinitiv Management Classes is to provide a suite of clearly defined, easy to use conceptual models, or abstractions, which runs efficiently. These abstractions represent the fundamental data types of interest to applications and also the infrastructure necessary to create both libraries and applications.

1.2 Audience

Users of the Refinitiv Management Classes should have working knowledge of C++ and a general understanding of the concepts of object-oriented analysis and design.

1.3 Additional Resources: C++ Programming

Some useful books for learning C++ are:

1. "*Object-Oriented Programming using C++*" by Ira Pohl, ISBN #:0-8053-5382-8
2. "*Effective C++*" by Scott Meyers, ISBN #: 0-201-56364-9
3. "*Advanced C++ - Programming Styles and Idioms*" by James O. Coplien, ISBN# 0-204-54855-0

1.4 Additional Resources: Object-Oriented Concepts

Some useful books for learning object-oriented concepts are:

1. "*Object-Oriented Software Construction*" by Bertrand Meyer, ISBN #: 0-13-629049-3
2. "*Object-Oriented Analysis and Design*" by Grady Booch, ISBN #:0-8053-5340-2

1.5 Organization of Refinitiv Management Classes Documents

Refinitiv Management Classes documentation is divided into four documents:

- *Software Installation Guide*
- *Developer's Guide*
- *Tutorial Guide*
- An HTML version of the *Class Reference Manual*. Given an understanding of the Refinitiv Management Classes model, the Class Reference Manual provides the necessary level of detail required to learn the other parts of the Refinitiv Management Classes.

1.6 Chapter Overview

The purpose of this *Developer's Guide* is to provide an introduction to Refinitiv Management Classes, explain the product's philosophy, and explore the nature of a typical Refinitiv Management Classes-based application.

CHAPTER	DESCRIPTION
Chapter 1, Overview	The remaining sections in this chapter discuss the conventions used in the manual, the Refinitiv Management Classes development process, and the current scope of the product.
Chapter 2, Design Philosophy	Gives an overview of the object-oriented principles which have a bearing on library design and application structure.
Chapter 3, Refinitiv Management Classes Framework	Describes how these principles are applied in the Refinitiv Management Classes and introduces the Refinitiv Management Classes application framework.
Chapter 4, Managing Applications	Presents an in depth discussion of the analysis and design of a managing application.
Chapter 5, Managed Applications	Presents an in depth discussion of the analysis and design of a managed application.
Chapter 6, Additional Topics	Discusses thread safety and introduces some supporting models for things such as event loops, I/O, and timers. Also includes recommendations for avoiding any performance impact.
Appendix A, Glossary	Defines key terms.

Table 1: Guide Contents

1.7 Conventions

1.7.1 Typographic

This manual uses the following typographic conventions:

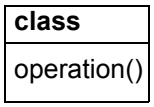
- Commands, classes, objects, in-line code snippets, log file text, methods, events, variables, and text to be typed "as is" are shown in **`Lucida Console`** font.
- Parameters, filenames, tools, utilities, items from the Graphic User Interface, and directories are shown in **Bold** font.
- Document titles and variable values are shown in *italics*.
- When initially introduced, concepts are shown in ***Bold, Italics***.

1.7.2 Programming

- All class names start with the letters "RTR" (i.e., Refinitiv classes; e.g., **`RTRString`**).
- All function names start with lower case letters, but subsequent words are capitalized (e.g., **`append()`** and **`lastString()`**).
- In order to make function names easy to understand, abbreviations are used sparingly. Some class names may seem rather long, but these names should be easier to understand than abbreviated names. On the technical side, it is important that class names do not clash with other class libraries or client defined classes. The combination of the RTR" prefix and the long name format should prevent name collisions with classes from other libraries.
- To provide consistent, easy to learn class interfaces, function names are assigned such that functions are given the same name in different classes if they have the same use. Application developers will be able to understand new classes more quickly when the naming is consistent. For instance, the **`addClient()`** function is used throughout the Refinitiv Management Classes as the mechanism that allows components to register for events.
- Refinitiv Management Classes use a common boolean type **`RTRBOOL`**, which can have either **RTRTRUE** or **RTRFALSE** as its value.

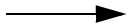
- Where class methods return a pointer, the return value may be null. In circumstances where returned values will never be null, an object reference will be returned.
- Refinitiv Management Classes use the following memory management guideline: “If you allocated the object, you delete it. If you didn’t allocate the object, don’t delete it.” Any exceptions to the guideline will be explicitly highlighted in the appropriate section(s) of the *Class Reference Manual*.

1.7.3 Notation for Class Diagrams

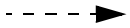


A class is defined by a box with the class name in bold at the top. The key operations of the class appear below the class name.

The notation for class inheritance is a triangle connecting a subclass to its parent class.



An arrow-headed line denotes acquaintance.



A dashed arrow-headed line points to the class that is instantiated.

A filled circle means “more than one.” When it appears at the head of a reference, it means multiple objects are being referenced.

1.8 Refinitiv Management Classes Development Process and Change Management Policies

Refinitiv Management Classes models and classes are developed under the assumption that they are not correctly defined until proven useful in practical applications. Development is based on an iterative life-cycle. Validation of a new model requires both multiple uses of the library and multiple implementations. The latter ensures that the abstract interfaces are sufficiently independent of implementation details. The former ensures that the library is useful as well as abstract. Not all implementations which result from this process are part of the delivered product.

Often, much of the required iteration is the result of internal usage. However, it is usually the case that external clients provide the final validation of a new library component prior to release of the product.

Users of the Refinitiv Management Classes should be prepared for some degree of change especially when using the newer models. Every effort is made to ensure that, where possible, changes are backwards compatible with existing software, but such compatibility is not always possible. In general, conceptual consistency takes precedence, especially in the early stages of model evolution. Users of the Refinitiv Management Classes are strongly encouraged to provide feedback on both the conceptual models and the implementation.

1.9 Availability

The Refinitiv Management Classes is currently available on the Linux and Windows platforms. Because of the incompatibility between the binaries produced by different C++ compilers, the Refinitiv Management Classes is supported for only one compiler on most platforms. Additional platforms will be supported on the basis of demand. Contact your Refintiv representative for exact details on current availability.

2 Design Philosophy

2.1 Refinitiv Management Classes Modelling Principles

The Refinitiv Management Classes is based on a conceptual model of the problem domain which comprises application management.

The Refinitiv Management Classes provides the functionality where information (data, configuration, etc.) that needs to be managed is grouped into **managed objects**, which contain **managed variables** that have data values. Processes that wish to retrieve these values or receive events on any changes, register interest in the managed variables by becoming clients of them.

2.2 Object-Oriented Principles

The purpose of developing an abstract model is to ensure that software components are properly isolated from non-essential characteristics of their suppliers. The effect of this approach is to decouple the various parts of an application or system and provide simpler, more intuitive programmatic interfaces. In practice, it also means that components can be more easily maintained (they can be re-worked with no impact on other parts of the system), and they can also be readily exchanged with "like" components making systems more adaptable to changing requirements.

Effective models provide abstraction and encapsulation, and exhibit modularity. Booch[1] defines these terms as follows:

- **Abstraction** - *"An abstraction denotes the essential characteristics of an object that distinguish it from all other kinds of object and thus provide crisply defined conceptual boundaries, relative to the perspective of the viewer."*
- **Encapsulation** - *"Encapsulation is the process of compartmentalizing the elements of an abstraction that constitute its structure and behavior; encapsulation serves to separate the contractual interface of an abstraction and its implementation."*
- **Modularity** - *"Modularity is the property of a system that has been decomposed into a set of cohesive and loosely coupled modules."*

How is a model developed? Development of a model starts with an analysis of the problem domain. If expertise is not available, knowledge must be gained through experimentation. Regardless of the technique used, analysis yields a complete theoretical model of the components to be found in the problem domain and is represented by a set of abstract data types.

Conceptual models are rarely correct the first time. Through trial and error a model evolves over time, gradually becoming stable. Stability is achieved through practical application of the model in building systems, both through using the interface defined by the model and by implementing the data types required by the model.

A model can be defined by English text (or some other natural language mechanism), by a formal modeling language, by a programmatic interface, or by some combination of these techniques. All of these techniques have advantages and disadvantages. What is important is to have a comprehensive description of the model which can then be used to provide programmatic implementations.

Ultimately, application programming interfaces (APIs) are required. When using an object-oriented programming environment, the result is a set of class libraries. A given model may have more than one implementation. The models which comprise a class library must be modular in their design and implementation. That is, it must be possible to add and remove both models and implementations thereof. This allows the scope of the library to grow and change over time in a manageable fashion.

Why go to all the trouble of developing abstract data types, iterating through prototypes, and building components? Clearly, the production of a reusable component requires high standards of design and implementation, and, perhaps, imposes restrictions on the way in which software can be implemented.

Perhaps the most important benefit of building components is the flexibility it affords. Components can be swapped out, enhanced, maintained, and interchanged. Monolithic software systems inhibit maintenance and enhancement.

Additionally, a well conceived conceptual model affords a much friendlier, more intuitive programming environment, letting programmers concentrate on problems specific to their particular domain rather than having to master the details of many different domains and implementations.

1. *Object-Oriented Analysis and Design* by Grady Booch

2.3 Design by Contract

In many of the class function descriptions (in the *Class Reference Manual*), you will notice the terms “REQUIRE” and/or “ENSURE” along with some conditional code. These statements, or **assertions**, suggest a contract between the client (caller) and supplier such that if the client fulfills the required preconditions, then the supplier will ensure that the postconditions will be true upon return from the function.

This method of programming is more commonly known as “design by contract” and is used not only in the abstract interfaces described in the manuals but is also used extensively throughout the Refinitiv Management Classes implementation.

Assertions are programmed into the library code using macros that are based on the standard “assert” mechanism. The Refinitiv Management Classes distribution provides two versions of each library—one with assertions, one without. Client programmers have the option to link their applications with either version. There is a performance penalty for using the version with assertions. Users of the library should link with assertions while developing and debugging their code, then use the non-assertion version for their finished product.

If an assertion statement fails during execution, the program will exit and print out the condition that failed, the name of the file the assertion failed in and the line number of the assertion in the file. This typically provides a good starting point for tracing back to determine where the problem originated.

The following is an example of the use of assertions:

> **RTRBOOL hasClient(RTRManagedVariableClient&) const;**
> Is the given client registered to receive events?
>
> **void addClient(RTRManagedVariableClient& newClient);**
> Register the given client to receive events.
> `REQUIRE: !hasClient(newClient)`
> `ENSURE : hasClient(newClient)`

In this example, the **`addClient()`** is used to register a new client. The assertions explain that a given client may register only once (**`REQUIRE: !hasClient`**) and that following the call to **`addClient()`**, the given client will be registered (**`ENSURE: hasClient`**).

One more example:

> **RTRBOOL hasVariable(const RTRString& name) const;**
> Does this Managed Object contain a Managed Variable with the given name?
>
> **RTRProxyManagedvariablePtr variableByName(const RTRString& name) const;**
> The Managed Variable with the given name.
> `REQUIRE: result == 0 implies !hasVariable(name)`

In the postconditions, the term “result” means the value returned from the function. Therefore, the assertion explains that if the managed object contains the managed variable with the given name, **`variableByName()`** will return a smart pointer to that managed variable, otherwise it will return a null pointer.

2.4 Component Design

Components are pieces of software which are designed to be reusable. As such, they must be independent of context. In other words, components must be independent of application specific assumptions. In particular, components should rely on the abstract interfaces of other software suppliers and be as independent as possible of implementation details.

In general, components should be single purpose. This means that they tend to be simple and modular. The Refinitiv Management Classes abstractions and other models support this philosophy and are themselves implemented in accordance with it.

2.5 Application Design

Applications are collections of components which together provide the required functionality. Application designers decide which components are necessary and also, either at compile time or link time, which implementations of abstract models to use. In contrast to components, which are implementation independent, *applications tend to be implementation specific*.

3 Refinitiv Management Classes Framework

3.1 Introduction

The Refinitiv Management Classes provides a programmable interface for monitoring software components for the purpose of enhancing the manageability of a system.

The Refinitiv Management Classes software package contains the following:

- An object-oriented C++ class library for managed and managing applications.
- Supporting documentation.

The library can be used by Refinitiv groups and third party developers to create a variety of managed and managing applications. These managing applications would enable administrators to monitor a system on a node effectively and efficiently.

3.2 Application Roles

3.2.1 Managed Application

A managed application is one which publishes some data about itself, in order to be monitored or managed by another application. The managed application uses a publishing API to make its data available. An application that publishes data can also act as a managing application.

3.2.2 Managing Application

A managing application is one that consumes the published information of one or more managed applications for the purpose of monitoring or actively managing those applications. The managing application uses the Refinitiv Management Classes's client API.

As an example, a small system could be a datafeed server application, which has trading client applications connecting to it for data. A performance monitor may be running which monitors the health of the datafeed server and tracks various statistics about its clients and throughput. In this example the datafeed server is the managed application and the performance monitor is the managing application. Such a scenario is shown in Figure 1.

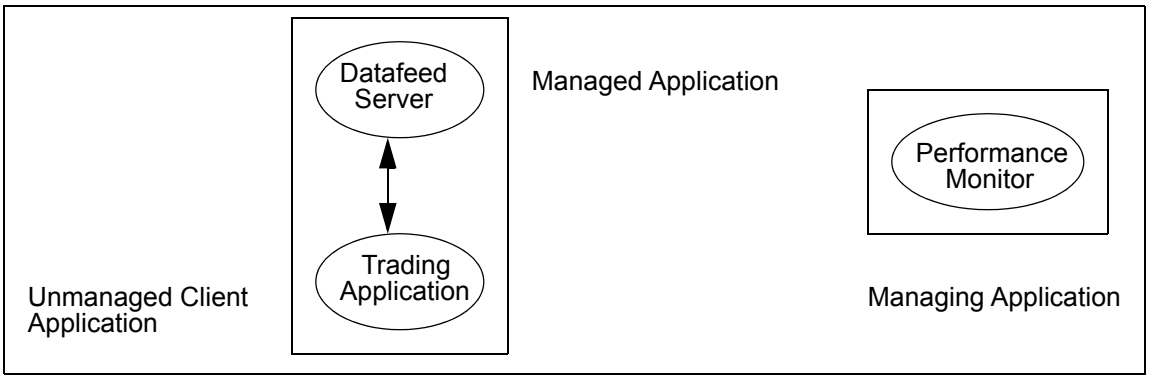


Figure 1. Simple System to be Managed

3.3 Refinitiv Management Classes Infrastructure

Managed applications that utilize the Publisher API may publish a variety of operational statistics to shared memory. Refinitiv Management Classes provide access to this published information in shared memory through the Refinitiv Management Classes client API.

3.3.1 Refinitiv Management Classes API

The Refinitiv Management Classes provides a Publisher API for managed applications to publish their statistics and a Consumer API for managing applications to retrieve the published statistics of all the managed applications in the shared memory on a particular node. See Figure 2.

In the following figure, RMC stands for Refinitiv Management Classes.

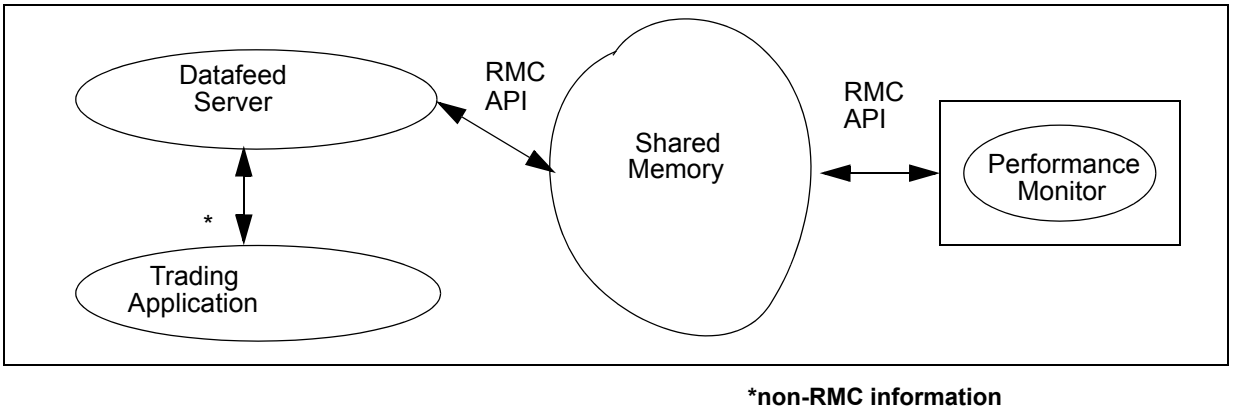


Figure 2. Managing Directly via Refinitiv Management Classes API

3.4 Programming Constructs

3.4.1 How Publishers Publish Information

3.4.1.1 Managed Objects

Managed applications are perceived as a collection of components represented by managed objects. Each managed object provides some number of managed variables, in which a managing application may have interest. The managed objects within a managed application have relationships with other managed objects. These relationships form one or more trees (whose nodes are managed objects). The type, or class, of a managed object determines the set of managed variables it will contain. This class name is referred to as the class ID.

Managed objects contained by other objects are children, and if they are not contained by another managed object, they are the roots of the trees (see Figure 3). The tree provides a means to access and possibly modify managed variables within the managed application.

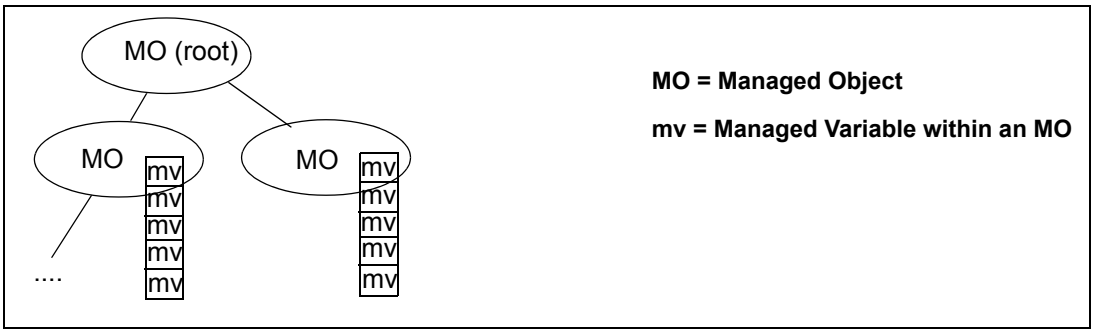


Figure 3. Managed Object Tree

3.4.1.2 Managed Variables

Managed variables are data items which are associated with a particular managed object, and which are exposed to managing applications for monitoring or changing. For example, if a datafeed server managed application contained a managed object dealing with user connections, the object might expose managed variables describing the current number of users and the maximum number of possible users.

Table 1 lists the eleven types of managed variables that a managed application can make available.

VARIABLE CATEGORY	VARIABLE TYPE	DESCRIPTION
boolean	boolean	A boolean variable
boolean	booleanConfig	Configuration and default values for a boolean variable
numeric	numeric	A numeric value
numeric	largeNumeric	A 64-bit numeric value
numeric	numericRange	A min/max value
numeric	numericConfig	Configuration and default values for a min/max value
numeric	counter	A counter
numeric	gauge	Min/max values with low/high water marks
numeric	gaugeConfig	Configuration and default values for a gauge
string	string	A string variable
string	stringConfig	Configuration and default values for a string variable

Table 2: Managed Variables

3.4.1.3 Managed Object Server

The component which makes available the collection of managed objects maintained is referred to as the managed object server. The managed objects are organized as hierarchical trees of objects, comprising the various managed applications running on the node. The managed object server then is a collection of several trees of objects, typically one per shared memory key.

3.4.2 How Consumers Use Proxies and Handles

The managing application uses a “clone on demand” policy to access the managed application information. This means the Refinitiv Management Classes only clones what the managing application is interested in monitoring. The reason for this is that it eliminates unnecessary processing that would be required to replicate the entire tree.

Proxy objects and proxy variables are used to represent the cloned managed objects and managed variables on the consumer side. These proxies are created within the managing application, and mirror the objects and variables from the managed application. The proxy objects contain references to the descriptions of their children and variables instead of the actual children and variables. The children and variables can be replicated as needed from this information.

The concept of a proxy managed object handle is used as a lightweight reference to a proxy managed object. This allows lists of these references to objects to be traversed or searched without incurring the overhead of cloning each of the actual objects. A handle contains the name and class ID information of the object.

3.4.3 Browser-Based Views

3.4.3.1 Proxy Managed Object Server Pool

The shared memory implementation of a proxy managed object server pool creates a proxy managed object server (described below) for each shared memory segment it wishes to monitor. This essentially is the consumer's access to all information from any publisher which it monitors on the same node. From this pool, a consumer may choose to manage applications via one or more of the shared memory keys.

3.4.3.2 Proxy Managed Object Servers

The shared memory implementation of a proxy managed object server makes all the roots of the trees of managed objects from a shared memory segment. The consumer can then locally clone the objects corresponding to applications it wishes to manage. The proxy managed object server also keeps the consumer aware of changes to the state of the shared memory segment by registering as client.

3.4.3.3 Proxy Managed Objects

Managed objects are manageable application components which are used as "containers" for managed variables. Proxy managed objects are the managing application's cloned representation of a managed application's managed objects. Just as in the managed application, these objects reflect an object hierarchy, with the root object typically at the highest level for each managed application, and children objects reflecting components within the application.

3.4.3.4 Proxy Managed Variables

Managed variables are used by managed applications to "expose" manageable data. For example, a managed application may decide to "publish" a set of variables related to a particular socket connection (e.g., number of writes/reads, number of overflows, number of messages sent/received). Proxy managed variables represent the managed variables that are published by a managed application.

3.4.4 Directory-Based Views

3.4.4.1 Proxy Managed Object Class Directories

The shared memory implementation of proxy managed object class directory is useful for an application that is interested in a specific category of proxy managed objects. It is used in conjunction with a shared memory implementation of proxy managed object server pool so that it has access to all the managed object trees within a node. Its purpose is to make visible all those managed objects within a node of a specified type, in order to allow getting or setting some attribute of all the instances of that type of object.

Creating a proxy managed object class directory requires specifying a shared memory implementation of proxy managed object server pool and the desired class of objects (the class ID). The directory then locates all objects of that type, and manages this collection, watching for additions and deletions as well as all updates to the current members of the set. This includes new objects from the new proxy managed object server pool that may come into being after the class directory is created. The references that the class directory returns are lightweight proxy managed object handles, for which the proxy managed objects can be instantiated if desired, using the proxy managed object pool. If the only information needed about the objects is names, this can be obtained directly from the handles without cloning the objects.

A managing application may create more than one shared memory proxy managed object class directory if needed, each looking at the same or different class types.

3.4.4.2 Proxy Managed Object Pool

The proxy managed object class directory provided a means to obtain the handles of all the available shared memory proxy managed objects of a desired type. The proxy managed object pool provides a method that will take the handles from a class directory and request all of the respective proxy managed objects. The pool also watches for additions and deletions from the class directory, and notifies its clients about changes to the set of objects currently in the pool.

Developers do not need to use the proxy managed object pool if they are only interested in some of the proxy managed objects in a class directory. Alternatively, they can write their own class which only clones selected objects.

3.5 Application Architectures

3.5.1 Managed vs. Managing Applications

Given the above programming constructs, we can envision what a typical application might look like, both on the managed and managing sides. Taking the example used earlier, of the datafeed server and its performance monitor, the major software components might look as shown in the following diagrams.

The first (Figure 4) shows two managed datafeed server applications, running on the same host. On the consumer side, there is one managing application, a shared memory shallow clone client application. It will create a proxy managed object server pool, which creates proxy managed object servers.

It creates one proxy managed object server for each shared memory segment (identified by its key) it wishes to monitor. It then is able to search through the proxy managed object handles available via this proxy managed object server, and clone those objects it cares about - for example, those relating to key performance measures. These cloned objects are the proxy managed objects, and the application can display or process the proxy managed variables it selects.

The second diagram (Figure 5) shows a similar arrangement of applications, but with a different functionality in the consumer application. This arrangement adds the use of a class directory along with a proxy managed object server pool to monitor a proxy managed object server for each of the managed object servers simultaneously. A proxy managed object class directory is used to retrieve a subset of like-typed proxy managed object handles from these proxy managed object servers. A proxy managed object pool is then used in conjunction with the class directory to access each of the proxy managed object handles and clone those it is interested in into proxy managed objects. The proxy managed variables are then accessed or displayed as before.

The architecture of a particular system should be chosen based on the type of visibility that is needed for the managing application.

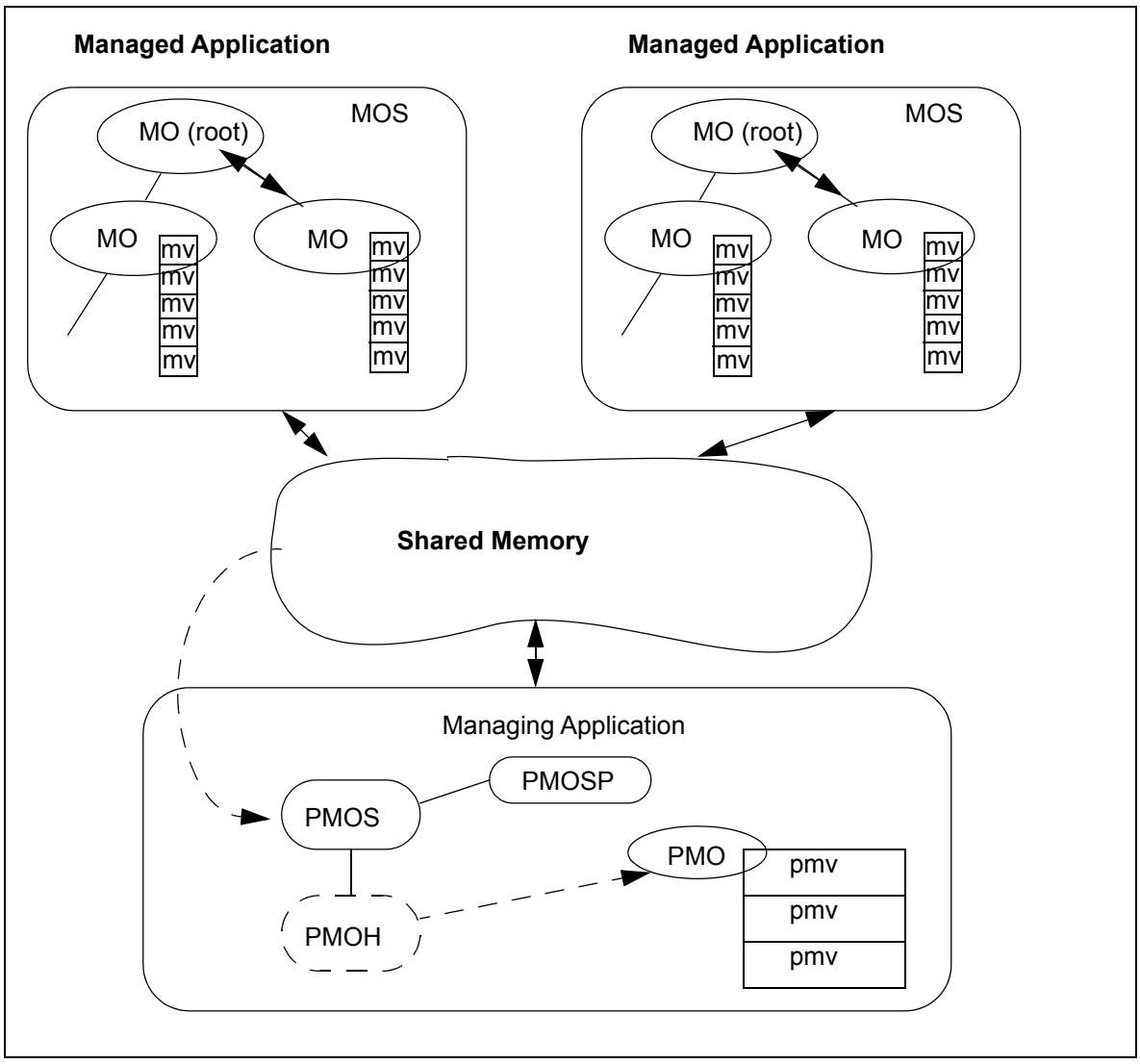


Figure 4. Direct Access to Proxy Managed Objects Without Class Directory

Legend for Figure 4:

- MO: Managed object
- MOS: Managed object server
- mv: Managed variable
- PMO: Proxy managed object
- PMOH: Proxy managed object handle
- PMOS: Proxy managed object server
- PMOSP: Proxy managed object server pool
- pmv: Proxy managed variable

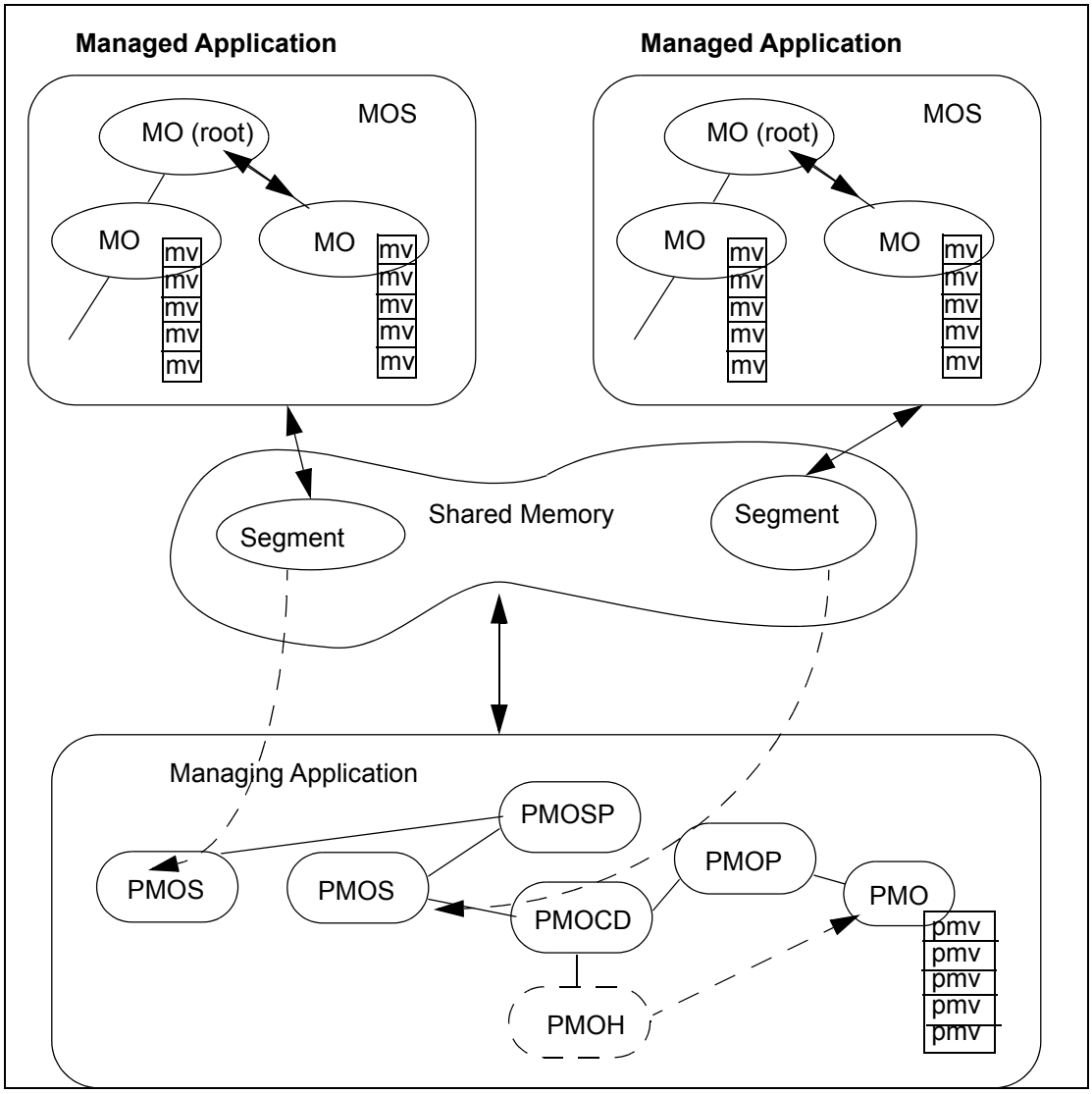


Figure 5. Access to Proxy Managed Objects Via the Class Directory

Legend for Figure 5:

- MO: Managed object
- MOS: Managed object server
- mv: Managed variable
- PMO: Proxy managed object
- PMOCD: Proxy managed object class directory
- PMOH: Proxy managed object handle
- PMOP: Proxy managed object pool
- PMOS: Proxy managed object server
- PMOSP: Proxy managed object server pool
- pmv: Proxy managed variable

3.5.2 Browser vs. Directory-based Access

Another feature of the example applications diagrammed above is their style of access. In the first example, no proxy managed object class directory is used. This implies that the application will either: 1) access variables directly by known names, or 2) browse through the proxy managed objects hierarchy, using or displaying objects as some criteria dictates. The former approach is appropriate for dedicated managing applications which are monitoring or setting a specific variable. The latter approach is useful for tools which are displaying everything which is accessible on a system, such as a generic object browser. Such a tool may be particularly helpful as managed applications are being developed and the developer wishes to see each object being published without writing custom consumer applications. Such a browser might look as follows:

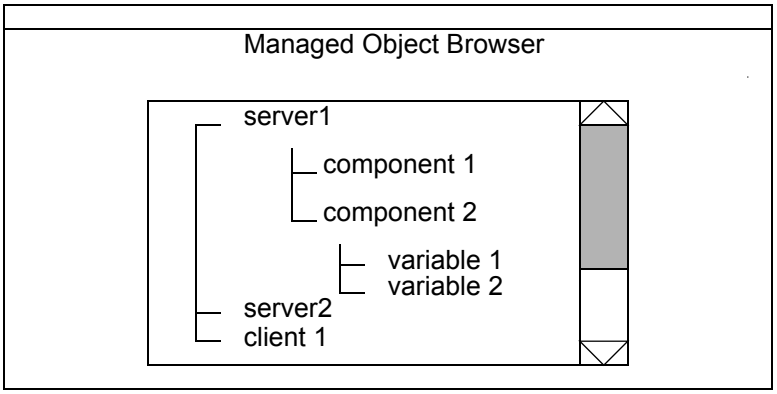


Figure 6. Example of a Generic Browser Interface

A different style of access was presented in the second example, using the class directory to narrow the field of objects to a specific class, though including ones drawn from multiple managed object servers. This style of access is suited to applications which have some domain knowledge, such as an application that knows specifically about datafeed servers. This application may retrieve the object which contains information about user connections from each running datafeed server, in order to display a table showing which servers are most heavily utilized. Such an application might look as follows:

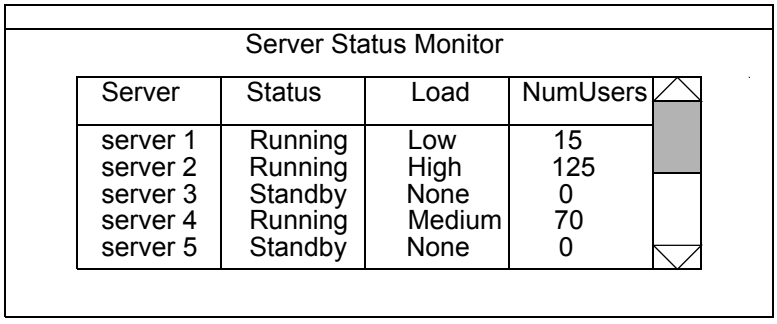


Figure 7. Example of a Table-based Interface

3.6 Pseudo Code Example

This section presents the pseudo code for a minimal Refinitiv Management Classes consumer application that retrieves a known variable from a known object.

```
invoke the application with shared memory keys to be monitored
create a Shared Memory Proxy Managed Object Server Pool (SPMOSP)
become a client of the SPMOSP
upon notification of Proxy Managed Object Server (PMOS) being added to the Pool
    become a client of the PMOS
    upon notification of synchronization of the PMOS
        retrieve desired proxy managed object (PMO) from the PMOS
        become a client of the PMO
        upon notification of synchronization of the PMO
            retrieve desired proxy managed variable (PMV) from the PMO
            become a client of the PMV
            upon notification of synchronization of the PMV
                Display the value of the PMV
            end
        end
    end
end
```

This example omits the other notifications available from the various client interfaces in order to focus on the direct route to retrieving the current variable value. It also ignores routines for error handling and cleanup (dropping a client relationship, etc.). The full source code of the examples will show all of this; the purpose here is merely to provide a high-level view.

Depending on the purpose of a real application, the developer may choose to maintain a list of the proxy managed object servers currently available, the proxy managed objects within one of them, or the proxy managed objects pertaining only to a specific application. An application will likely be a client of several proxy managed object servers, proxy managed objects, and proxy managed variables in order to be notified of the status of all relevant parts of a managed application. The specific client notifications pertaining to each of these constructs are described later in this document.

4 Managing Applications

4.1 Overview

The following topics are included in this chapter:

- Proxy Managed Variable (see Section 4.3)
- Proxy Managed Object (see Section 4.4)
- Proxy Managed Object Server (see Section 4.5)
- Proxy Managed Object Server Pool (see Section 4.6)
- Shared Memory Proxy Managed Object Server Pool (see Section 4.7)
- Proxy Managed Object Class Directory (see Section 4.8)
- Proxy Managed Object Class Directory Factory (see Section 4.9)
- Shared Memory Proxy Managed Object Class Directory Factory (see Section 4.10)
- Proxy Managed Object Pool (see Section 3.4.4.2)

In addition to an analysis of each topic, there is a description of how the abstract analysis translates into specific class functions. Example code fragments are listed C++. Each topic also includes simplified class diagrams depicting classes, their structure, and the static relationships between them. An example of a managing application is included at the end of this chapter (see Section 4.12).

4.2 Common Concepts

4.2.1 Accessing Handles

Figure 8 shows how an iterator is used to traverse all the members of a set. The **`start()`** method is invoked to initiate a new traversal. This method makes the first member of a set (e.g. the handle for a variable) available for inspection via the **`item()`** method of the iterator. The returned reference contains the identity and type of the first member. Invoking the **forth()** method makes available the next member, in this case “Variable Handle 2”, and so on. After the call to **`start()`** and after each call to **`forth()`**, the **`off()`** condition should be tested. When **`off()`** returns **RTRTRUE**, the iteration has been completed.

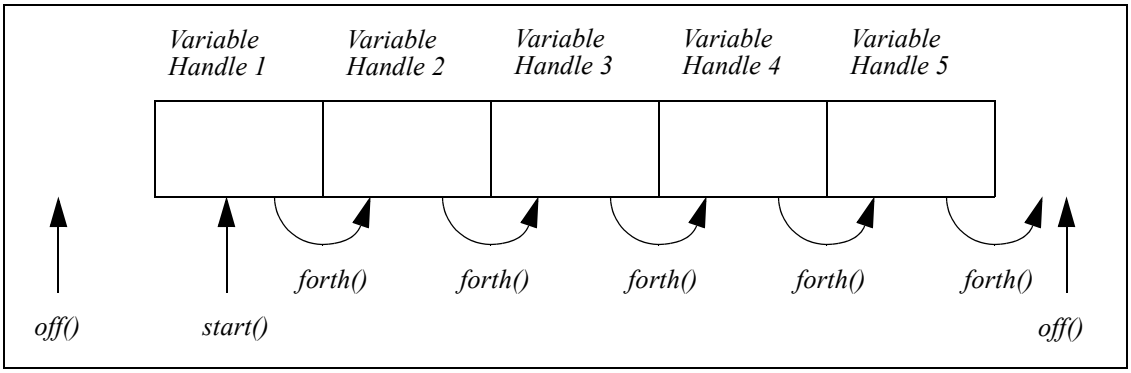


Figure 8. Accessing all Handles using Corresponding Iterator

4.2.2 Client Management

Events can be generated for managed variables for any of the following occurrences:

- variable was deleted
- variable value was changed
- variables is in an error state

Clients have to register with a managed variable in order to receive any events corresponding to that variable. When clients are no longer interested in receiving any more events for a managed variable, they have to deregister with it (see Section 4.3.4).

Event generating components may have multiple clients, and those clients may be registered with multiple event generating components. Figure 9 illustrates, for example, the n-to-n cardinality that exists between managed variables and their registered clients.

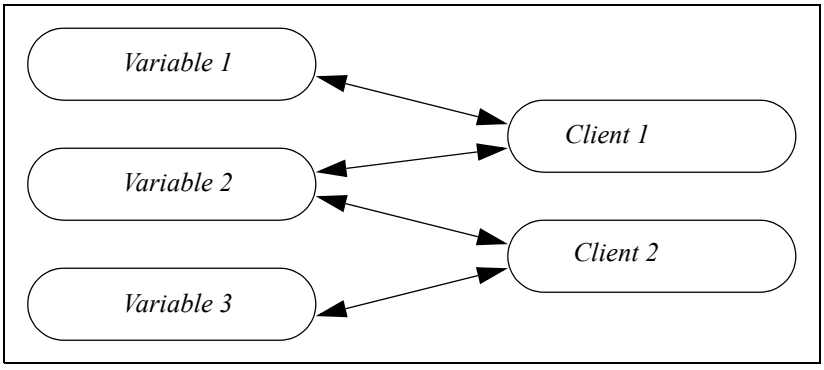


Figure 9. Run-time Relationships Example

In a typical application, these relationships are defined dynamically, i.e., on the basis of user input or other events, meaning that clients may be dynamically added (allocated) or removed (deleted) at any time.

4.3 Proxy Managed Variables

Managed variables are used by managed applications to “expose” manageable data. For example, a managed application may decide to “publish” a set of variables related to a particular socket connection (e.g., number of writes/reads, number of overflows, number of messages sent/received). Proxy managed variables represent the managed variables that are published by a managed application.

Types of managed variables supported are as follows:

VARIABLE CATEGORY	VARIABLE TYPE	PROXY MANAGED VARIABLE C++ CLASS NAME
boolean	boolean	RTRProxyManagedBoolean
boolean	booleanConfig	RTRProxyManagedBooleanConfig
numeric	counter	RTRProxyManagedCounter
numeric	gauge	RTRProxyManagedGauge
numeric	gaugeConfig	RTRProxyManagedGaugeConfig
numeric	numeric	RTRProxyManagedNumeric
numeric	largeNumeric	RTRProxyManagedLargeNumeric
numeric	numericConfig	RTRProxyManagedNumericConfig
numeric	numericRange	RTRProxyManagedNumericRange
string	string	RTRProxyManagedString
string	stringConfig	RTRProxyManagedStringConfig

Table 3: Supported Types of Managed Variables

`RTRProxyManagedVariable` is the base class for all eleven of the proxy managed variable classes. This class includes transformation methods for each of the eleven variable types (see Section 4.3.2.7).

4.3.1 Identity

Every proxy managed variable is identified by a name. The name is a string (**`RTRString`**) which uniquely identifies one proxy managed variable from another. The names are only unique within the context of a given proxy managed object.

RTRString& name()
The name of the proxy managed variable.

For example, an instance of a ServiceAttributes proxy managed object has several proxy managed variables each with a different name. A second instance of a ServiceAttributes proxy managed object will have the same set of proxy managed variables but are distinguished from the first set because they are "contained" in a different proxy managed object instance (with a different name). Figure 10 shows two instances of a ServiceAttributes type of proxy managed object. Each instance "contains" proxy managed variables with the same variable name ("Service Name", "Service State", etc.) but are distinguished from one another by being "contained" in proxy managed objects with different names (instance identifiers).

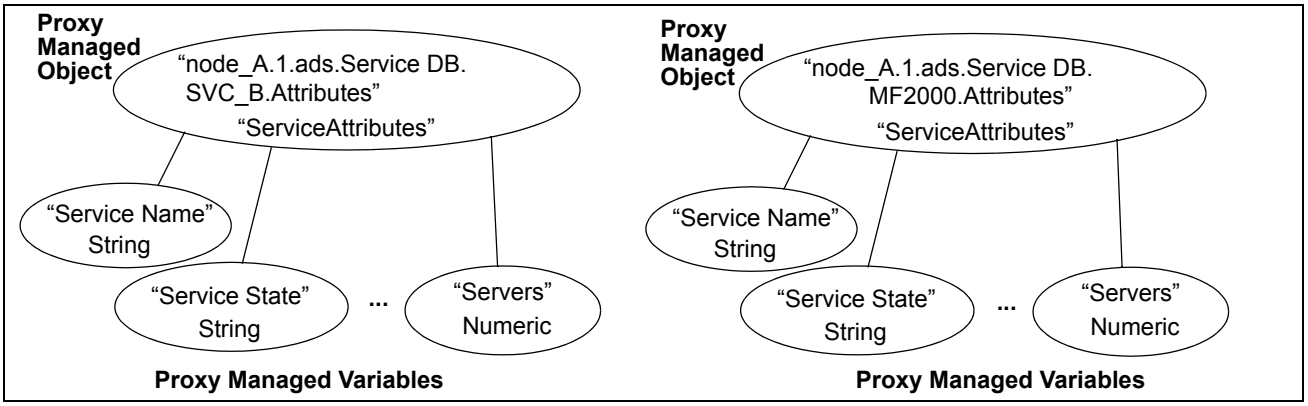


Figure 10. Example of Proxy Managed Object/Variable Name Space

4.3.2 Attributes

4.3.2.1 Context

RTRProxyManagedObject& context()
The proxy managed object that contains this proxy managed variable.

RTRString& description()
A textual description of the proxy managed variable.

A proxy managed variable exists within the context of a single proxy managed object. These functions are implemented in its base class - **`RTRProxyManagedVariable`**.

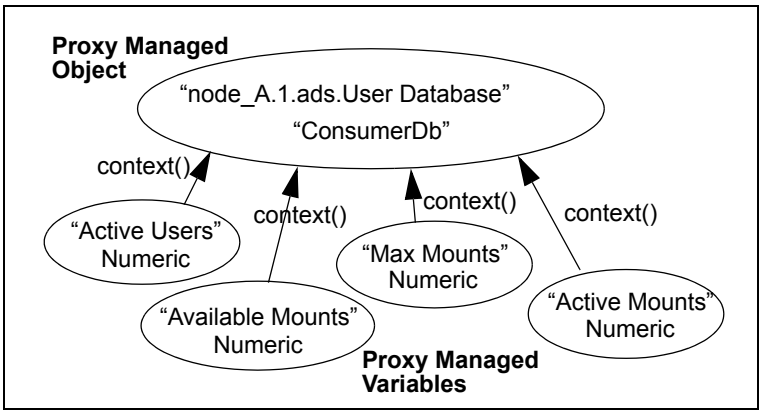


Figure 11. Example of Context Access

Refer to the following code example:

```
// RTRProxyManagedVariable pmv;
RTRProxyManagedObject& pmo = pmv.context();
cout << "The context of spmv is: " << pmo.instanceId() << ":" << pmo.classId() << endl;
```

4.3.2.2 Client Manageability

RTRBOOL modifyEnabled()
Is the managing application permitted to modify this proxy managed variable?

Depending on the type of proxy managed variable, its current value may be modified by the managing application. The managed application will accept this modification.

Refer to the following code example:

```
//RTRProxyManagedBoolean pmb;
if ( pmb.modifyEnabled() )
{
    if( pmb.inSync() && !pmb.error() )
        pmb = RTRTRUE;
}
```

4.3.2.3 Variable Types

The following managed variables are available to managed applications for the purpose of publishing its operational data. All 11 types fall into one of three categories (boolean, numeric, or string). These functions are implemented in **`RTRProxyManagedVariableHandle`**.

RTRProxyManagedVariableHandle::MVType type()
Returns the type of proxy managed variable. Its value will be one of the following enumerated types:

- RTRProxyManagedVariableHandle::Boolean
- RTRProxyManagedVariableHandle::BooleanConfig
- RTRProxyManagedVariableHandle::Counter
- RTRProxyManagedVariableHandle::Numeric
- RTRProxyManagedVariableHandle::LargeNumeric
- RTRProxyManagedVariableHandle::NumericConfig
- RTRProxyManagedVariableHandle::NumericRange
- RTRProxyManagedVariableHandle::Gauge
- RTRProxyManagedVariableHandle::GaugeConfig
- RTRProxyManagedVariableHandle::String
- RTRProxyManagedVariableHandle::StringConfig

4.3.2.4 Boolean Variable Types

The boolean category contains two managed variables (**boolean** and **booleanConfig**). Each of these variable types provides different levels of manageability and attribute characteristics.

TYPE	ATTRIBUTES	CLIENT MANAGEABILITY	DESCRIPTION
boolean	active value	Write-Enabled	Managing applications are allowed to set or clear the active value.
booleanConfig	active value store value factory default	Write-Enabled or Read-Only	Specifies a numeric value as a configuration parameter.

Table 4: Boolean Variable Types

4.3.2.5 Numeric Variables

The numeric category contains seven managed variables (**numeric**, **largeNumeric**, **counter**, **gauge**, **gaugeConfig**, **numericRange**, and **numericConfig**). Each of these variable types provides different levels of manageability and attribute characteristics.

TYPE	ATTRIBUTES	CLIENT MANAGEABILITY	DESCRIPTION
numeric	active value	Read-Only	Basic read-only numeric variable
largeNumeric	active value	Read-Only	Basic read-only 64-bit numeric variable
counter	active value	Write-Enabled	Active value starts at 0 and increments only. Managing application is allowed to reset the active value (to 0).
gauge	active value minimum value maximum value	Write-Enabled or Read-Only	Read-only variable whose active value will be between the min and max values. Min and max values may change dynamically. Usually used to represent current utilization of a limited resource (capacity).
gaugeConfig	active value minimum value maximum value store value factory default	Write-Enabled or Read-Only	Specifies a gauge as a configuration parameter.
numericRange	active value minimum value maximum value	Write-Enabled or Read-Only	Managing applications are allowed to set the active value. The managed application will accept/reject the new value at its earliest convenience. The new value will not be persistent. The min and max values are determined when the variable is instantiated and will not change during its life-cycle.
numericConfig	active value store value factory default minimum value maximum value	Write-Enabled or Read-Only	Specifies the variable as a configuration parameter.

Table 5: Numeric Category Variables

4.3.2.6 String Variables

The string category contains two managed variables (**string** and **stringConfig**). Each of these variable types provides different levels of manageability and attribute characteristics.

TYPE	ATTRIBUTES	CLIENT MANAGEABILITY	DESCRIPTION
string	active value	Write-Enabled or Read-Only	Basic read-only string variable
stringConfig	active value stored value factory default	Write-Enabled or Read-Only	Specifies a string as a configuration parameter.

Table 6: String Category Variables

4.3.2.7 Transformation

operator RTRProxyManagedBoolean()
Transform a proxy managed variable to a proxy managed boolean.

operator RTRProxyManagedBooleanConfig()
Transform a proxy managed variable to a proxy managed booleanConfig.

operator RTRProxyManagedCounter()
Transform a proxy managed variable to a proxy managed counter.

operator RTRProxyManagedGauge()
Transform a proxy managed variable to a proxy managed gauge.

operator RTRProxyManagedGaugeConfig()
Transform a proxy managed variable to a proxy managed gaugeConfig.

operator RTRProxyManagedNumeric()
Transform a proxy managed variable to a proxy managed numeric.

operator RTRProxyManagedLargeNumeric()
Transform a proxy managed variable to a proxy managed largeNumeric.

operator RTRProxyManagedNumericConfig()
Transform a proxy managed variable to a proxy managed numericConfig.

operator RTRProxyManagedNumericRange()
Transform a proxy managed variable to a proxy managed numericRange.

operator RTRProxyManagedString()
Transform a proxy managed variable to a proxy managed string.

operator RTRProxyManagedStringConfig()
Transform a proxy managed variable to a proxy managed stringConfig.

RTRString typeString()
The proxy managed variable type represented as a string.

RTRString toString()
The value of the proxy managed variable represented as a string.

Often a method or an event will return a reference to a proxy managed variable and you may need to transform it into a reference to the particular type of proxy managed variable in order to utilize that variable's methods. You can only perform this downcast operation on a proxy managed variable reference to the actual variable type or to any of its inherited types.

For example, if the proxy managed variable type is a counter, then a reference to the proxy managed variable can only be downcast to a counter. Obviously, it makes no sense to downcast the proxy managed variable to a string or gauge. The following table shows all the allowed downcasts of all the proxy managed variable types.

ACTUAL PROXY MANAGED VARIABLE TYPE	ALLOWED DOWNCASTS
boolean	boolean
booleanConfig	boolean, booleanConfig
counter	counter
numeric	numeric
largeNumeric	largeNumeric
numericConfig	numeric, numericConfig
numericRange	numeric, numericRange
gauge	gauge, numeric

Table 7: Variable Types and Allowed Downcasts

ACTUAL PROXY MANAGED VARIABLE TYPE	ALLOWED DOWNCASTS
gaugeConfig	gaugeConfig, gauge, numeric
string	string
stringConfig	string, stringConfig

Table 7: Variable Types and Allowed Downcasts (Continued)

4.3.2.8 Example: Transformation Code Fragment

```
// RTRProxyManagedVariable pmv;
if(pmv.type() == RTRProxyManagedVariableHandle::Gauge)
{
    RTRProxyManagedGauge& pmg = (RTRProxyManagedGauge&)pmv;
    long min = pmg.minValue();
    cout << "gauge min value   = " << min << endl;
    long max = pmg.maxValue();
    cout << "gauge max value   = " << max << endl;
    long lowWM = pmg.lowWaterMark();
    cout << "gauge lowWaterMark  = " << lowWM << endl;
    long highWM = pmg.highWaterMark();
    cout << "gauge highWaterMark = " << highWM << endl;
}
```

4.3.3 State

RTRBOOL error()
Indicates if this proxy managed variable is in an (unrecoverable) error state.

RTRBOOL inSync()
Indicates if this proxy managed variable has been successfully cloned (in the **InSync_Ok** state).

RTRString& text()
Provides textual explanation of the current proxy managed variable state (especially if in the **error** state).

A proxy managed variable's value cannot be queried until it has been successfully cloned. The cloning process is an asynchronous event. The returned proxy managed variable's state must be checked to determine if the proxy managed variable was previously cloned (by a previous request for the proxy managed variable).

The state of a proxy managed variable represents the state of the proxy and not the managed variable. The actual variable does not have a state attribute, it either exists or it doesn't. The state of the proxy managed variable can be queried at any time and clients will be notified of changes in the state.

Variable state is defined by the values of two variable attributes (inSync and error). The three variable states are defined as follows.

VARIABLE STATE	VARIABLE ATTRIBUTE	DESCRIPTION
OutOfSync	inSync = False	Variables in this state may transition to either of the other two states. Once out of this state, a variable will never return to it.
InSync_Error	inSync = True error = True	Once in this state, a variable will never transition to another state.
InSync_Ok	inSync = True error = False	Once in this state, a variable can only transition to the Error state.

Table 8: Variable States

4.3.3.1 State Transitions

State transitions are propagated to clients as events. The events triggered by state transitions are as follows:

- Sync - This event indicates that the managed variable has been successfully cloned. This event will only occur once for a given proxy managed variable instance. The event triggers a transition to the **`InSync_Ok`** state. Proxy managed variable clients will receive a **`processProxyManagedVariableSync()`** event.
- Error - The transition to the InSync_Error state triggers the Error event. Proxy managed variable clients will receive a **`processProxyManagedVariableError()`** event.

NOTE: A common mistake is to request a variable from its managed object, become a client of the variable and wait for a Sync event. This is a potential problem because it is possible that the requested variable was previously cloned and is currently in the **`InSync_Ok`** state; therefore, the client will never receive the Sync event. After registering as a client of the variable, the state of the requested variable must be checked. If it is not **`InSync`**, wait for the Sync event.

Figure 12 shows the variable states and state transitions.

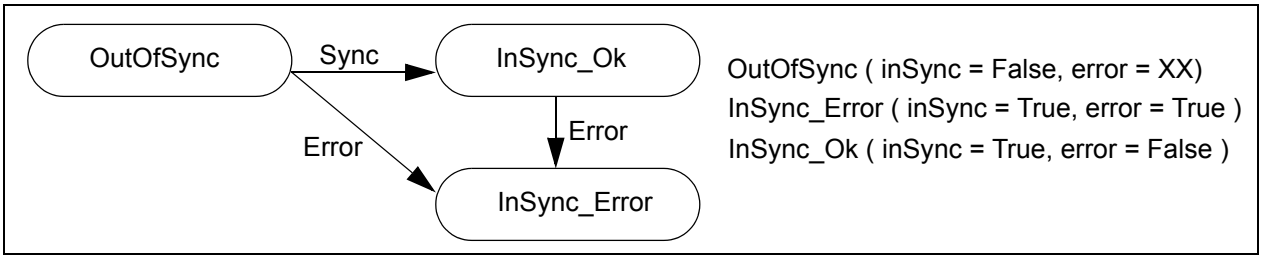


Figure 12. Proxy Managed Variable States and State Transitions

4.3.3.2 State Code Fragment Example

A typical example of checking the state is when a variable is requested (or accessed) from its containing proxy managed object.

```
//RTRProxyManagedObject pmo;
RTRProxyManagedVariable *pmv = pmo.variableByName ( "MaxMounts" );
pmv->addClient(*this);
if ( pmv->inSync() )
{
    if ( !pmv->error() )
    {
        //process Proxy Managed Variable
    }
    else
        //wait for sync event
}
```

4.3.4 Proxy Managed Variable Client Management

void addClient (RTRProxyManagedVariableClient&)
Register a client with an individual proxy managed variable so that the given client will receive all events subsequently generated by that proxy managed variable.

void dropClient (RTRProxyManagedVariableClient&)
Un-register a client with an individual proxy managed variable so that the given client will no longer receive events generated by that proxy managed variable.

RTRBOOL hasClient (RTRProxyManagedVariableClient&)
Indicates whether a given client is currently registered to receive events from a particular proxy managed variable. Allows application developer to verify precondition of corresponding **`addClient()`** and **`dropClient()`** methods.

Figure 9 shows the proxy managed variable and client relationship.

The following example illustrates a Proxy Managed Variable Client Management Code Fragment:

```
//Monitor mvClient;
//      (Monitor is a Proxy Managed Variable client class)
//RTRProxyManagedVariable mv;

if(!mv.hasClient(mvClient))   //are we registered to receive events from mv?
  mv.addClient(mvClient);     //if not, then register
```

4.3.5 Proxy Managed Variable Clients

virtual void processProxyManagedVariableSync (RTRProxyManagedVariable&) = 0
The given proxy managed variable has made the transition out of the OutOfSync state. The client should check for an error condition before processing the proxy managed variable.

virtual void processProxyManagedVariableError (RTRProxyManagedVariable&) = 0
The given proxy managed variable has made the transition to the InSync_Error state. This represents an unrecoverable error condition. The client should un-register from the proxy managed variable (using **`dropClient()`**) and ensure that no further references are made to that proxy managed variable. The **`text()`** method provides information text indicating the reason for this event.

virtual void processProxyManagedVariableUpdate (RTRProxyManagedVariable&) = 0
One or more of the given variable's attributes have changed. For example, in the case where the proxy managed variable is a gauge, then this could indicate that either the minimum, maximum, or current value has changed. It is left up to the application developer to decide which attribute changed and how to process the change.

virtual void processProxyManagedVariableDeleted (RTRProxyManagedVariable&) = 0
The given proxy managed variable has been deleted by the managed application. This represents an unrecoverable condition. The client should un-register from the proxy managed variable (using **`dropClient()`**) and ensure that no further references are made to that proxy managed variable. Managed variables may come and go at the discretion of the managed application, and it does not necessarily indicate a problem with the managed application.

To use the above four methods, proxy managed variable clients should be descendants of (inherit from) either **`RTRProxyManagedVariableClient`**. As clients, they can register to receive events from one or more instances of **`RTRProxyManagedVariable`**. Once registered with a proxy managed variable they will receive events generated by that proxy managed variable instance. Events are propagated by means of class member functions; i.e., when a proxy managed variable needs to "generate an event", it will invoke the appropriate member function of each client currently registered.

The "sync" event is one example of an event which can be generated by a proxy managed variable. The corresponding member function of **`RTRProxyManagedVariableClient`** is **`processProxyManagedVariableSync()`**. Thus, when a proxy managed variable transitions from the **OutOfSync** state to, say, the **InSync_Ok** state, it will invoke the **`processProxyManagedVariableSync()`** member function of each registered client.

As shown in the code fragment example, in the **`processProxyManagedVariableSync()`** member function you can now transform the proxy managed variable to its corresponding type and access any of its member functions.

Events which can be generated by instances of **`RTRProxyManagedVariable`** have a corresponding processing method in **`RTRProxyManagedVariableClient`**. In the cases of **`RTRProxyManagedVariableClient`** and there is no default implementation (behavior) defined. All processing methods have been declared as pure virtual. So the application developer must implement all four member functions, even if the client is not interested in certain events.

4.3.5.1 Proxy Managed Variable Client Event Categories

The events that can be generated by instances of **`RTRProxyManagedVariable`** fall into one of two categories:

- Events which signal state (of the cloned managed variable) transitions:
  - virtual void processProxyManagedVariableSync (RTRProxyManagedVariable&) = 0
  - virtual void processProxyManagedVariableError (RTRProxyManagedVariable&) = 0
- Events which signal variable attribute or life-cycle transitions:
  - abstract void processProxyManagedVariableUpdate (ProxyManagedVariable)
  - abstract void processProxyManagedVariableDeleted (ProxyManagedVariable)

4.3.5.2 Proxy Managed Variable Clients Code Fragment Example

```
void processProxyManagedVariableSync(RTRProxyManagedVariable& pmv)
{
    cout << "pmv event: Sync" << endl;
    cout << "cloned mv information:   name    = " << pmv.name() << endl;
    cout << "                         type    = " << pmv.typeString() << endl
         << "                     description = " << pmv.description() << endl;

    //check if the cloned managed variable is of type Boolean
    if(pmv.type() == RTRProxyManagedVariableHandle::Boolean)
    {
        //Boolean transformation
        RTRProxyManagedBoolean& pmb = (RTRProxyManagedBoolean&)pmv;

        //change value
        if(pmb.value() == RTRTRUE)
        {
            if(pmb.modifyEnabled() && !pmb.error())
             pmb.clear();
        }
        else
        {
            if(pmb.modifyEnabled() && !pmb.error())
             pmb.set();
        }
    }

    //check if the cloned managed variable is of type Counter
    if(pmv.type() == RTRProxyManagedVariableHandle::Counter)
    {
        //BooleanConfig transformation
         RTRProxyManagedCounter& pmc = (RTRProxyManagedCounter&)pmv;
        unsigned long cnt = pmc.value();
        cout << "           value   = " << cnt << endl;
        //reset counter
        pmc.reset();
    }
}
```

4.3.6 Class Diagrams (Simplified)

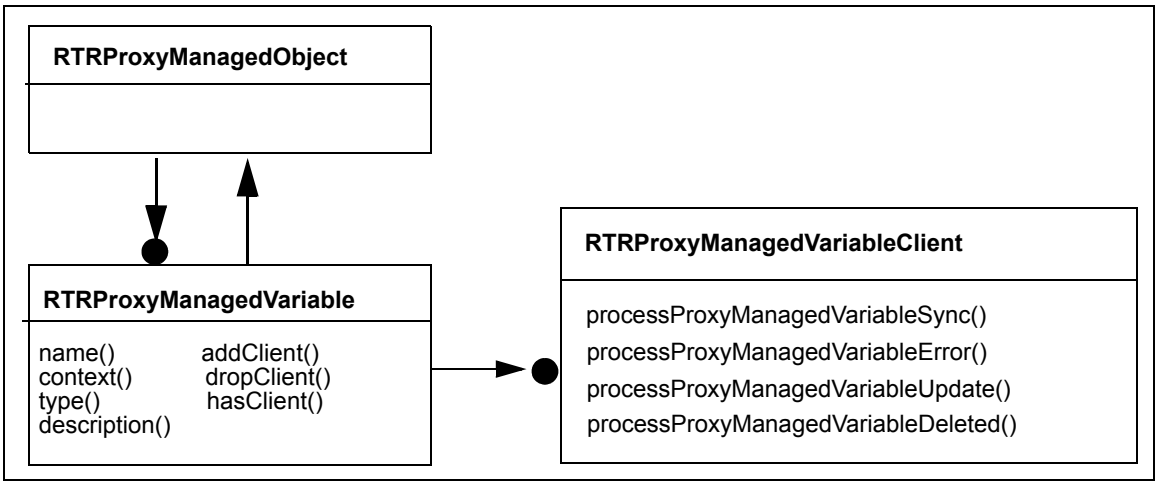


Figure 13. Proxy Managed Variable's Association with Context and Clients

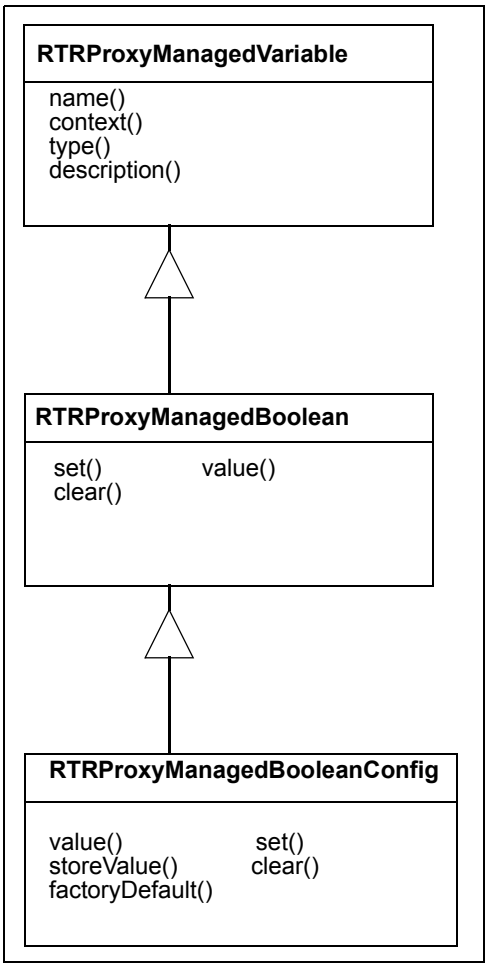


Figure 14. Boolean Category of Proxy Managed Variables - Simplified Class Diagram

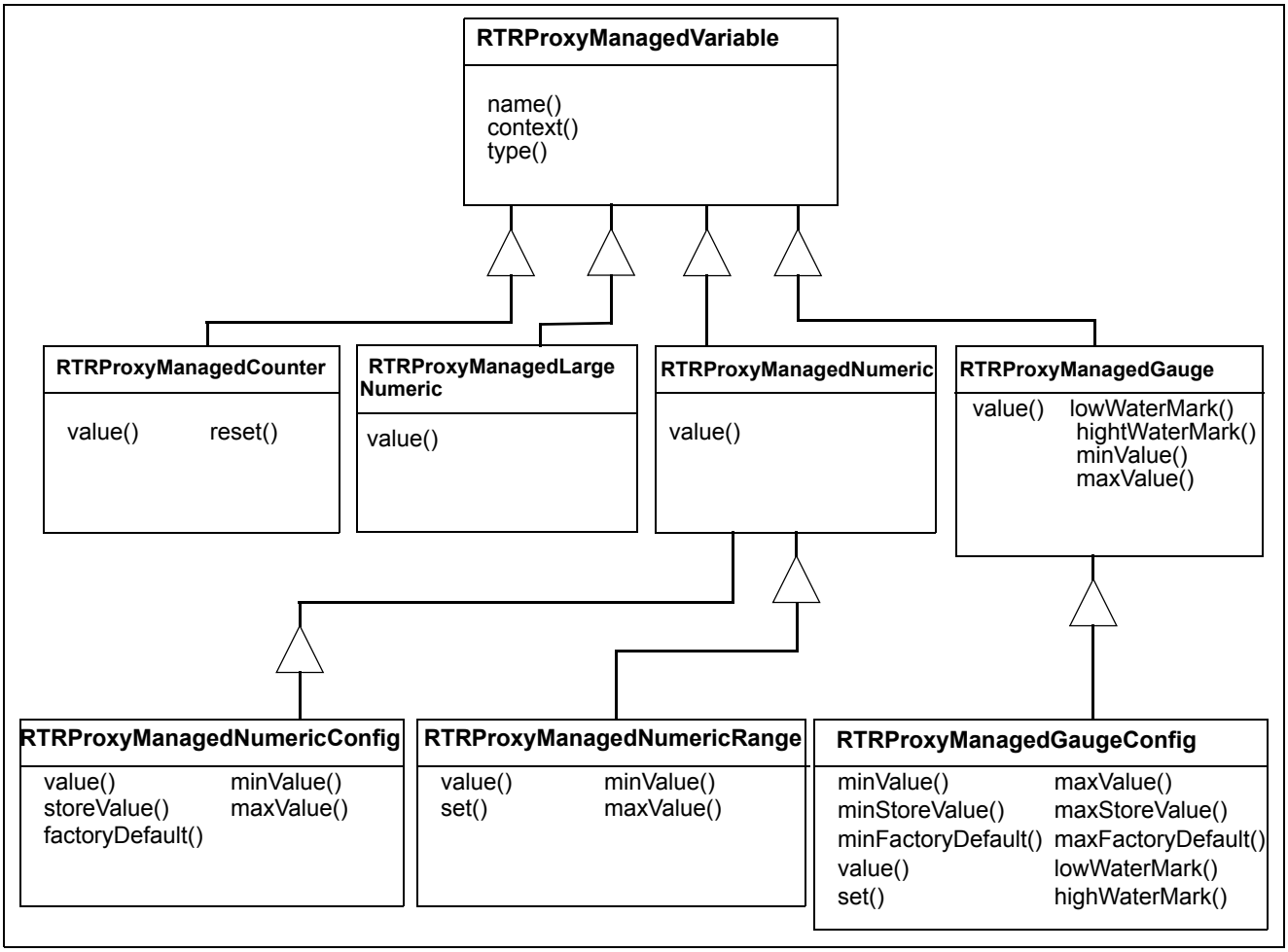


Figure 15. Numeric Category of Proxy Managed Variables - Simplified Class Diagram

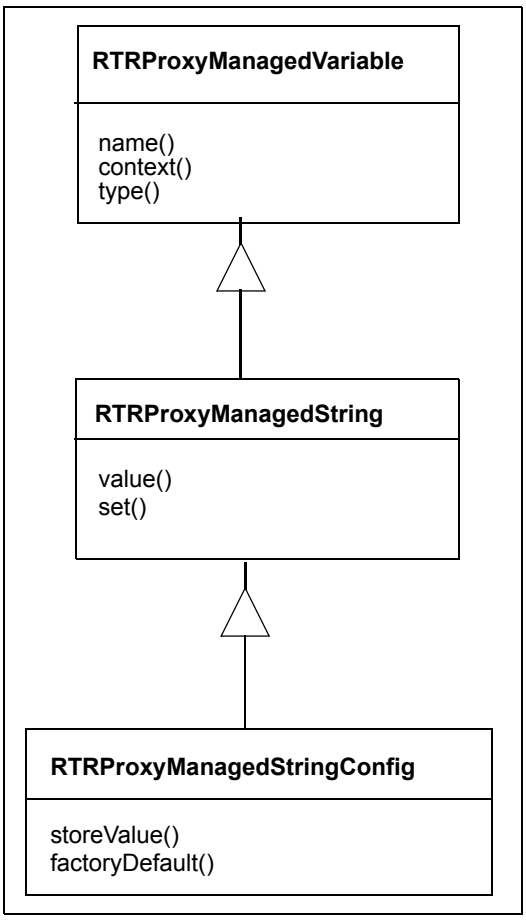


Figure 16. String Category of Proxy Managed Variables - Simplified Class Diagram

4.4 Proxy Managed Object

Managed objects are manageable application components which are used as a “container” for managed variables. Proxy managed objects are the managing application’s cloned representation of a managed application’s managed objects.

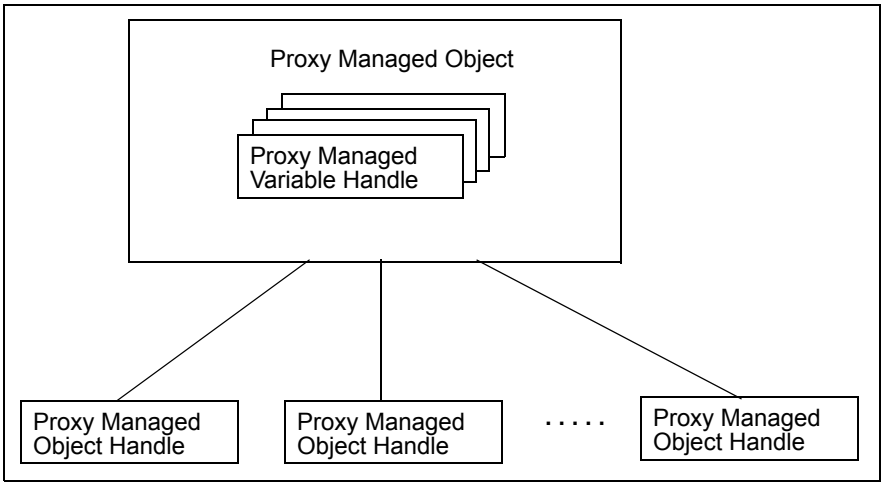


Figure 17. Proxy Managed Object Layout

As shown in Figure 17, a proxy managed object has:

- The handles (name and type) of all of its (contained) proxy managed variables (if any), and
- The handles (name, instance identifier and class identifier) of all of its child proxy managed objects (if any).

4.4.1 Identity

RTRString& name()
The name of the proxy managed object.

RTRObjectId& instanceId()
The instance identifier of the proxy managed object.

RTRObjectId& classId()
The class identifier of the proxy managed object.

4.4.1.1 Instance Identifier

Every instance of a managed object in a managed application will have a unique identifier. This identifier (instanceId) consists of two parts:

- The instanceId of its parent managed object and
- The name of the managed object.

The exception to this is the root managed objects which do not have parents so their identifier is the same as their name.

For example, the partial set of managed objects taken from a Refinitiv Real-Time Advanced Distribution Server (as shown in Figure 18) forms a tree. The “node_A.1.ads” is a root managed object so its identifier is the same as its name. The identifier of its child is the concatenation of the parent’s identifier and the child name with a ‘.’ separator. Since this “dot” notation has a special purpose/meaning, the symbol ‘.’ is not used within the names of child managed objects.

These naming rules guarantee unique instance identifiers for each “tree”. The application developer should adopt a notation of uniquely named roots.

To have uniquely-named root objects, the following naming convention is used:

<*host_name*>.<*instance_id*>.<*executable_name*>

where:

- <*host_name*> is the machine name where the process is running on,
- <*instance_id*> is a numeric value (starting with 1) to uniquely identify multiple instances of the same executable running on the same machine and,
- <*executable_name*> is the name of the executable.

Some examples are:

node_a.1.ads
node_a.1.adh
node_b.1.adh
node_b.2.adh

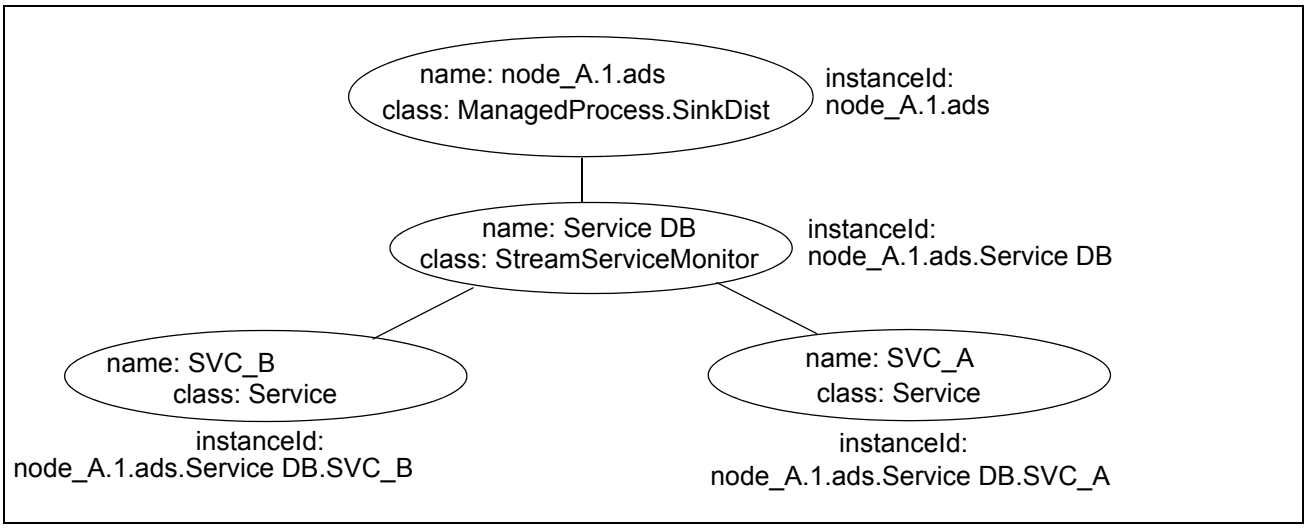


Figure 18. Refinitiv Real-Time Advanced Distribution Server Managed Object Tree

4.4.1.2 Class Identifier

Managed objects also have a class identifier which identifies the type (semantics) of that object. In the previous example (Figure 18), one class identifier is “Service”. All class identifiers should be unique throughout the whole system. It is left up to the application developer to select the appropriate identifiers.

4.4.2 State

RTRBOOL error()
Indicates if this proxy managed object is in an (unrecoverable) error state.

RTRBOOL inSync()
Indicates if this proxy managed object has been successfully cloned (in the **InSync_Ok** state).

RTRString& text()
Provides textual explanation of the current proxy managed object state (especially if in the error state).

The proxy managed object state is defined by the values of two variable attributes (**inSync** and **error**). The three object states are defined as follows:

MANAGED OBJECT STATE	VARIABLE ATTRIBUTE	DESCRIPTION
OutOfSync	inSync = False	Proxy managed objects in this state may transition to either of the other two states. Once out of this state, a variable will never return to it.
InSync_Error	inSync = True error = True	Once in this state, a proxy managed object will never transition to another state.
InSync_Ok	inSync = True error = False	The managed object cloning is complete.

Table 9: Managed Object States and Associated Variable Attributes

4.4.2.1 State Transitions

State transitions are propagated to clients as events. The events triggered by state transitions are described as follows:

- Sync - This event indicates that the managed object have been cloned. This event will only occur once for a given managed instance. The event triggers a transition to either the **InSync_Error** or **InSync_Ok** state. Variable clients will receive a **`processProxyManagedObjectSync()`** event.
- Error - Managed object clients will receive a **`processProxyManagedObjectError()`** event.

4.4.2.2 State Attributes

RTRProxyManagedObject::PMOState state()
The state attribute of the proxy managed object.

RTRProxyManagedObject::PMOState previousState()
The previous state attribute of the proxy managed object.

The proxy managed object state attribute is defined by the value of **`state()`**. Values are defined as follows:

RETUN VALUE FOR STATE()	DESCRIPTION
Init	Initializing service
Normal	Normal service
ManualRecovery	In a service interrupted state, but attempting to recover normal service manually.
AutoRecovery	In a service interrupted state, but attempting to recover normal service automatically.
Dead	In an unrecoverable error state

Table 10: Proxy Managed Object State Attribute Values

State transitions are propagated to clients as events. The events triggered by state transitions are described as follows:

- Normal - The managed object cloning is complete and its managed variables can now be cloned.
- ManualRecovery - The proxy managed object is in a service interrupted state and it is attempting to recover manually.
- AutoRecovery - The proxy managed object is in a service interrupted state and it is attempting to recover automatically.
- Dead - The proxy managed object is in an unrecoverable error state.
- Invalid - The managed object cloning is incomplete.

Not every proxy managed object will support state transitions.

4.4.3 Access to Child Proxy Managed Objects

Managed objects provide both random and sequential access to the handles of their child proxy managed objects.

4.4.3.1 Sequential Access to Child Proxy Managed Objects

RTRProxyManagedObjectHandleIterator childHandles()
An iterator which provides sequential access to all child proxy managed objects contained by a proxy managed object.

Sequential access is provided by means of an “iterator” class, an instance of which is associated with a particular proxy managed object instance and provides access to each child proxy managed object handle of the proxy managed object in turn. The proxy managed object handle iterator is defined by the class **`RTRProxyManagedObjectHandleIterator`**.

The example code below shows how a proxy managed object handle iterator is used to traverse all the children of a proxy managed object.

```
//RTRProxyManagedObject pmo;
RTRProxyManagedObjectHandle pmoh;
RTRProxyManagedObjectHandleIterator iterator = pmo.childHandles();
for ( iterator.start(); !iterator.off(); iterator.forth() )
{
    //iterator.item() provides a Proxy Managed Object Handle to the current child Managed Object
    pmoh = iterator.item();
}
```

4.4.3.2 Random Access to Child Managed Objects

RTRBOOL hasChild(RTRString& name)
Tests the existence of a child proxy managed object by querying the proxy managed object for a specific child, given the child's name.

RTRProxyManagedObjectPtr childByName(RTRString& name)
Provides random access to a child proxy managed object, given the name. This will initiate the cloning of this child proxy managed object.

- Fragment example:

```
//RTRProxyManagedObject object;
RTRString Child("ChildName");
const RTRProxyManagedObjectPtr childObject = object.childByName(Child );
if ( childObject == (RTRProxyManagedObject *) NULL )
{
    // Child is not present.
}
```

- **hasChild()** method example:

```
//const RTRProxyManagedObjectPtr object;
RTRString Child("ChildName");
if ( !object->hasChild( Child ) )
{
    // Child is not present.
}
```

4.4.4 Access to Contained Managed Variables

Proxy managed objects provide both sequential and access to the handles of their contained proxy managed variables.

4.4.4.1 Sequential Access to Managed Variables

RTRProxyManagedVarHandleIterator variableHandles()
An iterator which provides sequential access to all proxy managed variable handles contained by a proxy managed object.

Sequential access is provided by means of an “iterator” class, an instance of which is associated with a particular proxy managed object instance and provides access, in turn, to each contained proxy managed variable handle of the proxy managed object. The proxy managed variable handle iterator is defined by the class **`RTRProxyManagedVarHandleIterator`**.

The following example code illustrates how a proxy managed variable handle iterator is used to traverse all the variables of a proxy managed object.

```
//RTRProxyManagedObject pmo;
RTRProxyManagedVariableHandle pmvh;
RTRProxyManagedVarHandleIterator iterator = pmo.variableHandles();
for ( iterator.start(); !iterator.off(); iterator.forth() )
{
//iterator.item() provides a Proxy Managed Variable Handle to the current contained Managed Variable
    pmvh = iterator.item();
}
```

4.4.4.2 Random Access to Managed Variables

RTRBOOL hasVariable(RTRString& name)
Tests the existence of a proxy managed variable by querying the proxy managed object for a specific proxy managed variable, given the proxy managed variable’s name.

RTRProxyManagedVariablePtr variableByName(RTRString& name)
Provides random access to a proxy managed variable, given the name. This will initiate the cloning of the managed variable.

There are also eleven more random access methods that correspond to each of the eleven possible managed variable types:

RTRProxyManagedBooleanPtr booleanByName(RTRString& name),

RTRProxyManagedBooleanConfigPtr booleanConfigByName(RTRString& name),

RTRProxyManagedCounterPtr counterByName(RTRString& name),

RTRProxyManagedGaugePtr gaugeByName(RTRString& name),

RTRProxyManagedGaugeConfigPtr gaugeConfigByName(RTRString& name),

RTRProxyManagedNumericPtr numericByName(RTRString& name),

RTRProxyManagedLargeNumericPtr largeNumericByName(RTRString& name),

RTRProxyManagedNumericConfigPtr numericConfigByName(RTRString& name),

RTRProxyManagedNumericRangePtr numericRangeByName(RTRString& name),

RTRProxyManagedStringPtr stringByName(RTRString& name), and

RTRProxyManagedStringConfigPtr stringConfigByName(RTRString& name).

4.4.5 Proxy Managed Object Client Management

void addClient (RTRProxyManagedObjectClient&)
Register a client with an individual proxy managed object so that the given client will receive all events subsequently generated by that proxy managed object.

void dropClient (RTRProxyManagedObjectClient&)
Un-register a client with an individual proxy managed object so that the given client will no longer receive events generated by that proxy managed object.

RTRBOOL hasClient (RTRProxyManagedObjectClient&)
Indicates whether a given client is currently registered to receive events from a particular proxy managed object. Allows application developer to verify precondition of corresponding **`addClient()`** and **`dropClient()`** methods.

Figure 9 shows the proxy managed object and client relationship.

For example:

```
//Monitor moClient;
// (Monitor is a Proxy Managed Object client class)
//RTRProxyManagedObject mo;

if(!mo.hasClient(moClient))   //are we registered to receive events from mo?
    mo.addClient(moClient);     //if not, then register
```

4.4.6 Proxy Managed Object Clients

virtual void processProxyManagedObjectSync (const RTRProxyManagedObject&) = 0
The given proxy managed object has made the transition out of the OutOfSync state. The client should check for an error condition before processing the proxy managed object.

virtual void processProxyManagedObjectError (const RTRProxyManagedObject&) = 0
The given proxy managed object has made the transition to the InSync_Error state. This represents an unrecoverable error condition. The client should un-register from the proxy managed object (using **`dropClient()`**) and ensure that no further references are made to that proxy managed object. The **`text()`** method provides information text indicating the reason for this event.

virtual void processProxyManagedObjectInfo (const RTRProxyManagedObject&) = 0
The given proxy managed object has additional information concerning its state attribute. It has not changed state. The **`text()`** method provides this additional information.

virtual void processProxyManagedObjectInService (const RTRProxyManagedObject&) = 0
The given proxy managed object is now in a normal service state.

virtual void processProxyManagedObjectRecovering (const RTRProxyManagedObject&) = 0
The given proxy managed object is in a service interrupted state and is attempting to recover normal service automatically.

virtual void processProxyManagedObjectWaiting (const RTRProxyManagedObject&) = 0
The given proxy managed object is in a service interrupted state and is waiting for manual intervention to restore normal service automatically.

virtual void processProxyManagedObjectDead (const RTRProxyManagedObject&) = 0
The given proxy managed object has entered an unrecoverable error state.

virtual void processProxyManagedObjectDeleted (const RTRProxyManagedObject&) = 0
The given proxy managed object has been deleted by the managed application. This represents an unrecoverable condition. The client should un-register from the proxy managed object (using **`dropClient()`**) and ensure that no further references are made to that proxy managed object. Managed objects may come and go at the discretion of the managed application; this does not necessarily indicate a problem with the managed application.

virtual void processProxyManagedObjectChildAdded (
const RTRProxyManagedObject&,
const RTRProxyManagedObjectHandle&) = 0

The given proxy managed object has a new child.

virtual void processProxyManagedObjectChildRemoved (
const RTRProxyManagedObject&,
const RTRProxyManagedObjectHandle&) = 0

The given proxy managed object has a child removed.

virtual void processProxyManagedObjectVariableAdded (
const RTRProxyManagedObject&
const RTRProxyManagedVariableHandle&) = 0

The given proxy managed object has a new managed variable.

virtual void processProxyManagedObjectVariableRemoved (
const RTRProxyManagedObject&,
const RTRProxyManagedVariableHandle&) = 0

The given proxy managed object has a managed variable removed.

To use the above twelve methods, proxy managed object clients should be descendants of (inherit from) **`RTRProxyManagedObjectClient`**. As clients, they can register to receive events from one or more instances of **`RTRProxyManagedObject`**. Once registered with a proxy managed object, they will receive events generated by that proxy managed object instance. Events are propagated by means of class member functions; i.e., when a proxy managed object needs to "generate an event", it will invoke the appropriate member function of each client currently registered.

The "sync" event is one example of an event which can be generated by a proxy managed object. The corresponding member function of **`RTRProxyManagedObjectClient`** is **`processProxyManagedObjectSync()`**. Thus, when a proxy managed object transitions from the **OutOfSync** state to, say, the **InSync_Ok** state, it will invoke the **`processProxyManagedObjectSync()`** member function of each registered client.

As shown in the code fragment example, in the **`processProxyManagedObjectSync()`** member function you can now:

- Iterate through all of the proxy managed variable handles contained by this proxy managed object or check if it contains a specific proxy managed variable.
- Access the proxy managed object by executing any of its member functions.

Events that can be generated by instances of **`RTRProxyManagedObject`** have a corresponding processing method in **`RTRProxyManagedObjectClient`**. In the cases of **`RTRProxyManagedObjectClient`** there is no default implementation (behavior) defined. All processing methods have been declared as pure virtual. So the application developer must implement all twelve member functions, even if the client is not interested in certain events.

Code Fragment Example:

```
void processProxyManagedObjectSync(const RTRProxyManagedObject& pmo)
{
    cout << "pmo event: Sync" << endl;
    cout << "cloned mo information:   name    = " << pmo.name() << endl
         << "             instance identifier = " << pmo.instanceId() << endl
         << "                class identifier = " << pmo.classId() << endl;
    if(!pmo.error())
    {
        cout << "                   description = " << pmo.description() << endl;

        //sequentially access all pmv handles
        if(get_all_pmv())
          cout << "successfully accessed all pmv handles" << endl;

        //sequentially access all child pmo handles
        if(get_all_child_pmo())
          cout << "successfully accessed all child pmo handles" << endl;
    }
```

```
}

RTRBOOL get_all_pmv(const RTRProxyManagedObject& pmo)
{
   if(!pmo.error())
   {
      //iterate through all contained managed variable handles
      RTRProxyManagedVarHandleIterator pmvIterator = pmo.variableHandles();
      for( pmvIterator.start(); !pmvIterator.off(); pmvIterator.forth() )
      {
         RTRString pmvh_type = pmvIterator.item().typeString();
         cout << "   found a variable of type  " << pmvh_type << endl
              << "                    with name " << pmvIterator.item().name() << endl;

         //now start the cloning of  this managed variable
         RTRProxyManagedVariablePtr pmv = pmo.variableByName(pmvIterator.item().name());
         if(pmv->inSync() == RTRTRUE)
           cout << "pmv is inSync" << endl;
         else
           cout << "pmv is not inSync" << endl;
         pmv->addClient(*this);
         //we now wait for the Sync event for this managed variable
      }
   }
   else
   {
      //check explanation why in error state
      cout << "Proxy Managed Object " << pmo.name() << " error: " << pmo.text() << endl;
      return RTRFALSE;
   }

   return RTRTRUE;
}

RTRBOOL get_all_child_pmo(const RTRProxyManagedObject& pmo)
{
   if(!pmo.error())
   {
      //iterate through all child managed object handles
      RTRProxyManagedObjectHandleIterator child_pmoIterator = pmo.childHandles();
      for( child_pmoIterator.start(); !child_pmoIterator.off(); child_pmoIterator.forth() )
      {
        cout << "  found a child managed object: name   = " << child_pmoIterator.item().name() << endl
             << " instance identifier = " << child_pmoIterator.item().instanceId() << endl
             << " class identifier = " << child_pmoIterator.item().classId() << endl;

        //now we can start the cloning of  this child managed object
        RTRProxyManagedObjectPtr child_pmo = pmo.childByName(child_pmoIterator.item().name());
        if(child_pmo->inSync() == RTRTRUE)
                  cout << "child pmo is inSync" << endl;
        else
                  cout << "child pmo is not inSync" << endl;
        child_pmo->addClient(*this);
        //we now wait for the Sync event for this managed object
      }
   }
   else
   {
      //check explanation why in error state
       cout << "Proxy Managed Object " << pmo.name() << " error: " << pmo.text() << endl;
```

```
        return RTRFALSE;
    }

    return RTRTRUE;
}
```

4.4.7 Class Diagram (Simplified)

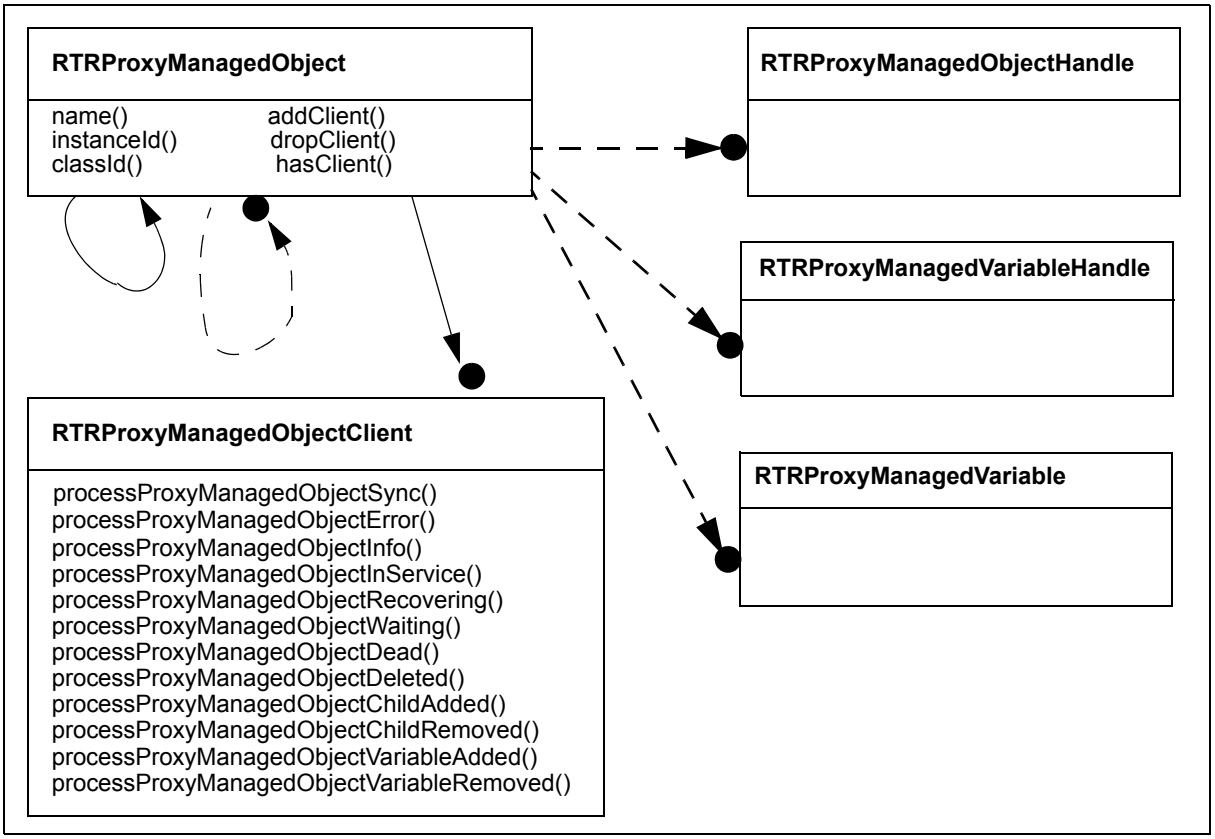


Figure 19. Proxy Managed Object - Simplified Class Diagram

4.5 Proxy Managed Object Server

If the managing application intends to manage/monitor a managed application on a system, then a convenient method is needed to "establish and maintain a connection" to the managed applications.

A proxy managed object server performs several services for a managing application:

- The proxy managed object server establishes a "connection" with the shared memory. This "connection" is used to keep the server informed about the availability of the shared memory segments.
- The proxy managed object server provides access to all the root managed objects of all managed applications that the client is monitoring.
- The proxy managed object server provides a cloning service for managed objects.

Figure 20 illustrates the services provided by the proxy managed object server.

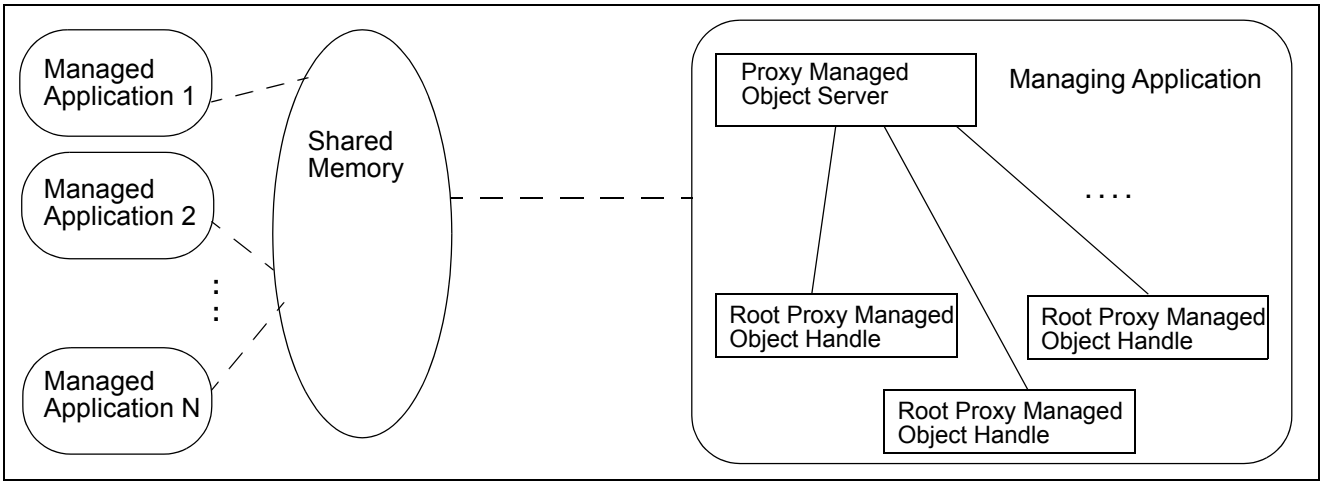


Figure 20. System View of Proxy Managed Object Server

4.5.1 State

RTRBOOL error()
Indicates if the proxy managed object server is in an (unrecoverable) error state.

RTRBOOL inSync()
Indicates if the proxy managed object server is ready to use its services. Application has to wait for this state.

RTRString& text()
Provides textual explanation of the proxy managed object server state (especially if in the error state).

The state of the proxy managed object server can be queried at any time and clients will be notified of changes in the state.

The proxy managed object server state is defined by the values of two variable attributes (**inSync** and **error**). Three variable states are defined as follows.

MANAGED OBJECT SERVER STATE	VARIABLE ATTRIBUTE	DESCRIPTION
OutOfSync	inSync = False	Proxy managed object server in this state may transition to either of the other two states. Once out of this state, a server will never return to it.
InSync_Error	inSync = True error = True	Once in this state, a proxy managed object server will never transition to another state.
InSync_Ok	inSync = True error = False	Proxy managed object server is ready to use its services. Application has to wait for this state.

Table 11: Managed Object Server States

State transitions are propagated to clients as events. The events triggered by state transitions are described as follows:

- Sync - This event indicates that the proxy managed object server has successfully established a connection with the shared memory segment, i.e., all the root handles have been received. This event will only occur once for a given proxy managed object server instance. The event triggers a transition to the **InSync_Ok** state. Server clients will receive a **`processObjectServerSync()`** event.
- Error - Proxy managed object server clients will receive a **`processObjectServerError()`** event. The event triggers a transition to the **InSync_Error** state. Once in this state, the proxy managed object server will never transition to another state.

Figure 21 shows the various variable states and state transitions.

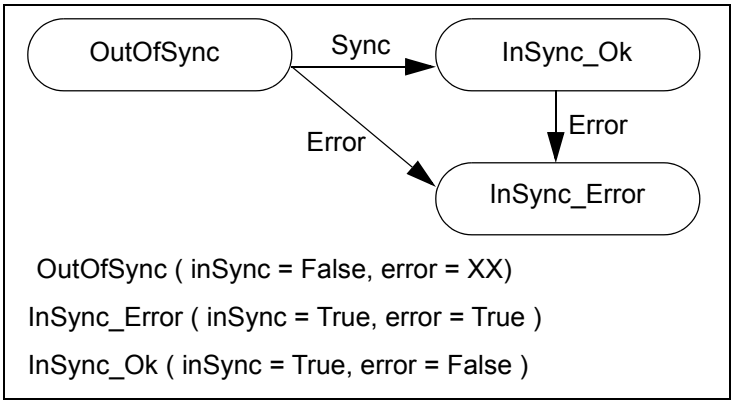


Figure 21. Proxy Managed Object Server States and State Transitions

4.5.2 Sequential Access to Root Proxy Managed Object Handles

RTRProxyManagedObjectHandleIterator roots()
An iterator which provides sequential access to handles of all root proxy managed objects available.

The following example code illustrates how a proxy managed object handle iterator is used to traverse all the proxy managed objects.

```
//RTRProxyManagedObjectServer pmos;
RTRProxyManagedObjectHandle pmoh;
RTRProxyManagedObjectHandleIterator iterator = pmos.roots();
for ( iterator.start(); !iterator.off(); iterator.forth() )
{
    //iterator.item() provides a Proxy Managed Object Handle to the current Proxy Managed Object
    pmoh = iterator.item();
}
```

4.5.3 Access to Proxy Managed Objects

RTRProxyManagedObjectPtr object(const RTRProxyManagedObjectHandle&)
Provides random access to a proxy managed object, given the handle to a proxy managed object. This will initiate the cloning of the proxy managed object.

The cloning of the managed object is complete when the proxy managed object is in the **InSync_Ok** state (i.e., the **`processProxyManagedObjectSync()`** event method is received by clients or the proxy managed object is polled).

4.5.4 Proxy Managed Object Server Client Management

void addClient (RTRProxyManagedObjectServerClient&)
Register a client with an individual proxy managed object server so that the given client will receive events subsequently generated by that proxy managed object server.

void dropClient (RTRProxyManagedObjectServerClient&)
Un-register a client with an individual proxy managed object server so that the given client will no longer receive events generated by that proxy managed object server.

RTRBOOL hasClient (RTRProxyManagedObjectServerClient&)
Indicates whether a given client is currently registered to receive events from a particular proxy managed object server. Allows application developer to verify precondition of **`addClient()`** and **`dropClient()`** methods.

Figure 9 shows the proxy managed object server and client relationship.

Code Fragment Example:

```
//Monitor mosClient;
//      (Monitor is a Proxy Managed Object Server client class)
//RTRProxyManagedObjectServer mos;

if(!mos.hasClient(mosClient))   //are we registered to receive events from mos?
    mos.addClient(mosClient);     //if not, then register
```

4.5.5 Proxy Managed Object Server Clients

virtual void processObjectServerError (RTRProxyManagedObjectServer&)
The given **RTRProxyManagedObjectServer** has made the transition to the InSync_Error state. This represents an unrecoverable error condition. The client should un-register from the proxy managed object server (using **dropClient()**) and ensure that no further references are made to that server. The **text()** method provides information text indicating the reason for this event.

virtual void processObjectServerSync (RTRProxyManagedObjectServer&)
The given proxy managed object server has made the transition into the InSync_Ok state. The services of the proxy managed object server can now be used.

virtual void processObjectServerRootAdded (
RTRProxyManagedObjectServer&,
const RTRProxyManagedObjectHandle&)
The given root proxy managed object handle has been added to the given proxy managed object server.

virtual void processObjectServerRootRemoved (
RTRProxyManagedObjectServer&,
const RTRProxyManagedObjectHandle&)
The given root proxy managed object handle has been removed from the given proxy managed object server.

To use the above four event processing methods, proxy managed object server clients should be descendants of (inherit from) **RTRProxyManagedObjectServerClient**. As clients, they can register to receive events from one or more instances of **RTRProxyManagedObjectServer**. Once registered with a proxy managed object server, they will receive events generated by that proxy managed object server instance. Events are propagated by means of class member functions; i.e., when a proxy managed object server needs to “generate an event”, it will invoke the appropriate member function of each client currently registered.

The “sync” event is one example of an event that can be generated by a proxy managed object server. The corresponding member function of **RTRProxyManagedObjectServerClient** is **processObjectServerSync()**. Thus, when a variable transitions from the **OutOfSync** state to, say, the **InSync_Ok** state, it will invoke the **processObjectServerSync()** member function of each registered client.

As shown in the code fragment example, in the **processObjectServerSync()** member function you can now access the proxy managed object server by executing any of its member functions.

Events which can be generated by instances of **RTRManagedObjectServer** have a corresponding processing method in **RTRManagedObjectServerClient**. In the case of **RTRManagedObjectServerClient** there is a default implementation (behavior) defined. All four event processing methods have been declared as virtual and the default behavior is to do nothing.

Code Fragment Example:

```
void processObjectServerSync(RTRProxyManagedObjectServer& pmos)
{
    cout << "pmos event: in Sync  " << "@" << pmos.text() << endl;
    RTRProxyManagedObjectHandleIterator pmosIterator = pmos.roots();
    for(pmosIterator.start(); !pmosIterator.off(); pmosIterator.forth())
    {
        //pmosIterator.item() is the handle to an mo
        cout << " found handle to an mo: " << endl
             << " instance identifier is " << pmosIterator.item().instanceId() << endl
             << "                 name is " << pmosIterator.item().name() << endl
             << "    class identifier is " << pmosIterator.item().classId() << endl;

        //now start the cloning of this managed object
        RTRProxyManagedObjectPtr_pmo = pmos.object(pmosIterator.item());
        if(_pmo->inSync() == RTRTRUE)
            cout << "_pmo is inSync" << endl;
        else
        {
```

```
            cout << "_pmo is not inSync" << endl;
            _pmo->addClient(*this);    //we now wait for the Sync event for this managed object
        }
    }
}
```

4.5.6 Class Diagram (Simplified)

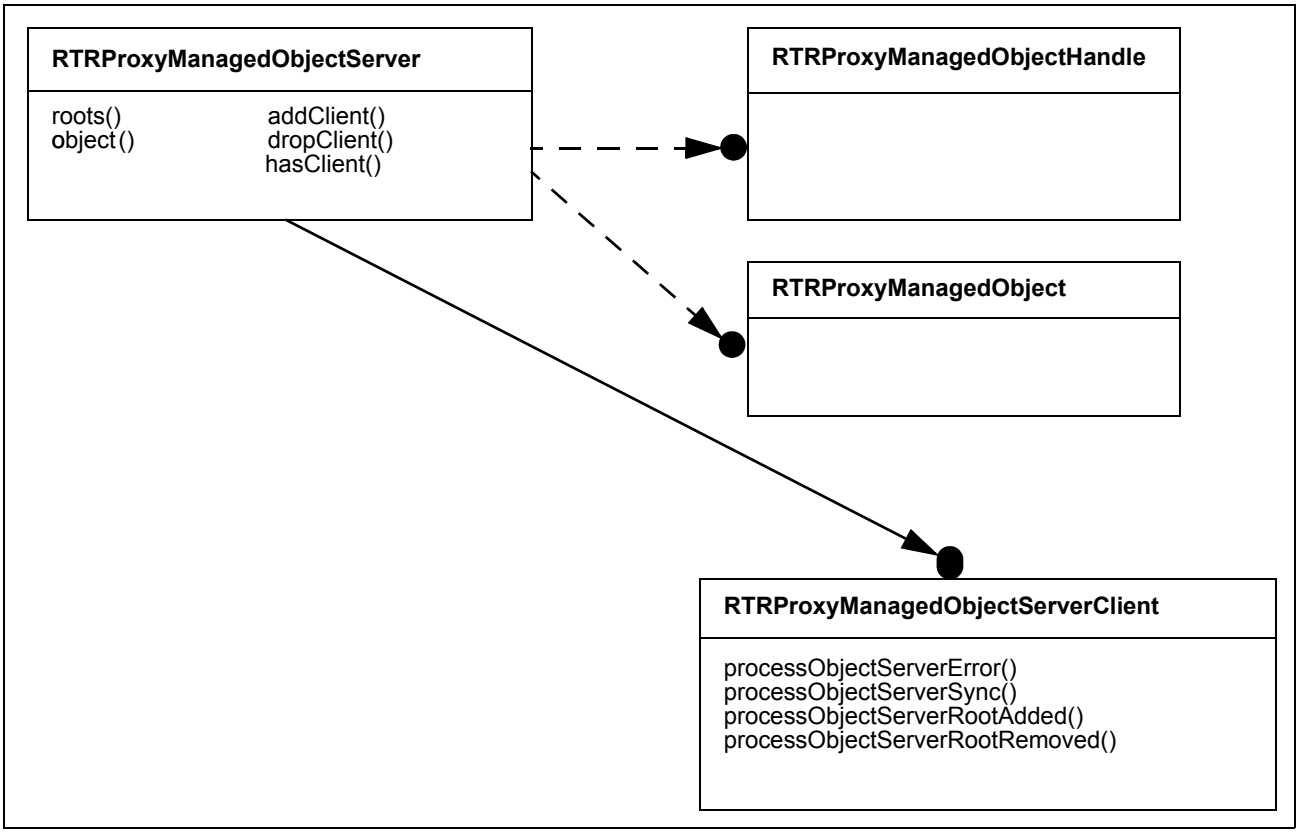


Figure 22. Proxy Managed Object Server - Simplified Class Diagram

4.6 Proxy Managed Object Server Pool

A shared memory implementation of proxy managed object server pool creates a proxy managed object server for each shared memory segment being monitored. If the managing application intends to manage/monitor all of the managed applications on a node, then a proxy managed object server pool can be used.

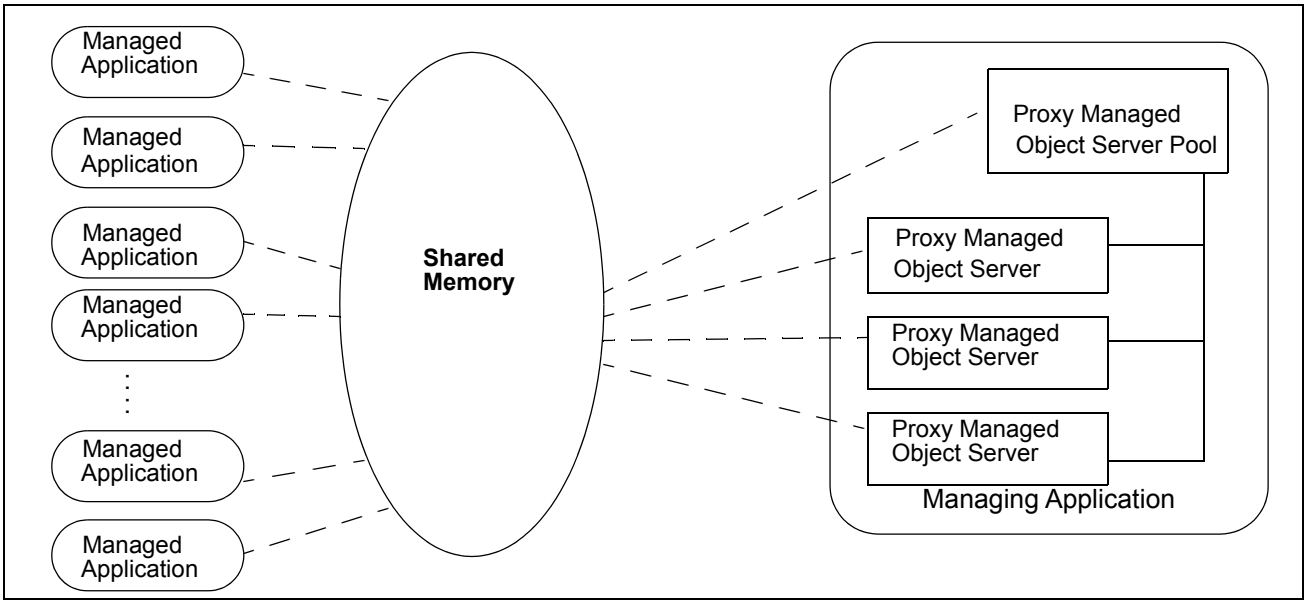


Figure 23. System View of Proxy Managed Object Server Pool

For example, a managing application can be developed which provides a list of all the available managed applications on the system. A single table of the managed applications on the system can be created and automatically updated.

HOST	INSTANCE	MANAGED APPLICATION
Node_A	1	Refinitiv Real-Time Advanced Distribution Server
Node_A	1	Refinitiv Real-Time Advanced Data Hub

Table 12: Host Table

A proxy managed object server pool performs several services for a managing application:

- The proxy managed object server pool acts as a “factory” for creating a proxy managed object server as each memory key becomes available and removes (deletes) proxy managed object servers if the corresponding memory key is no longer being monitored.
- The proxy managed object server pool acts as a “pool” for maintaining the list of proxy managed object servers that were created by the “factory”. The list is maintained as the availability of individual proxy managed object servers change. In other words, if a proxy managed object server is created it will be added to the list, and if one is deleted it will be removed from the list. The managing application has access to this list, and clients are notified of changes (additions/deletions) in this list.

Figure 23 illustrates the services provided by the proxy managed object server pool; i.e., the managed applications publish data into shared memory, the pool of proxy managed object servers creates one proxy managed server for each available memory key.

4.6.1 Access to Proxy Managed Object Servers

RTRLinkedListCursor<RTRProxyManagedObjectServer> servers()
An iterator which provides sequential access to all the proxy managed object servers contained in the managed object server pool.

For example:

```
//RTRProxyManagedObjectServerPool pool;
RTRProxyManagedObjectServer pmos;
RTRLinkedListCursor<RTRProxyManagedObjectServer> iterator = pool.servers();
for ( iterator.start(); !iterator.off(); iterator.forth() )
{
    //iterator.item() provides a Proxy Managed Object Server
    pmos = iterator.item();
}
```

4.6.2 Proxy Managed Object Server Pool Client Management

void addClient (const RTRProxyManagedObjectServerPoolClient&)
Registers a client with an individual proxy managed object server pool so that the given client will receive all events subsequently generated by that server pool.

void dropClient (const RTRProxyManagedObjectServerPoolClient&)
Un-registers a client with a particular proxy managed object server pool so that the given client will no longer receive events from that server pool.

RTRBOOL hasClient (const RTRProxyManagedObjectServerPoolClient&)
Indicates whether a given client is currently registered to receive events from a particular proxy managed object server pool. Allows application developer to verify precondition of corresponding **`addClient()`** and **`dropClient()`** methods.

Figure 9 shows the proxy managed object server pool and client relationship.

As an example, suppose an application wants to create a handle directory of all proxy managed objects of class **`UserDb`**. A class could be written which monitors all available proxy managed object servers and obtains the handles of all proxy managed objects of the desired type.

For example:

```
//Monitor mosPoolClient;
//      (Monitor is a Proxy Managed Object Server Pool client class)
//RTRProxyManagedObjectServerPool mosPool;

//are we registered to receive events from mosPool?
if(!mosPool.hasClient(mosPoolClient))
    mosPool.addClient(mosPoolClient);     //if not, then register
```

4.6.3 Proxy Managed Object Server Pool Clients

virtual void processProxyManagedObjectServerAdded (
RTRProxyManagedObjectServerPool&,
RTRProxyManagedObjectServer&)

The given proxy managed object server has been added to the proxy managed object server pool.

virtual void processProxyManagedObjectServerRemoved (
RTRProxyManagedObjectServerPool&,
RTRProxyManagedObjectServer&)

The given proxy managed object server has been removed from the proxy managed object server pool.

To use the above two event processing methods, proxy managed object server pool clients should be descendants of (inherit from) **RTRProxyManagedObjectServerPoolClient**. As clients, they can register to receive events from one or more instances of **RTRProxyManagedObjectServerPool**. Once registered with a proxy managed object server pool, they will receive events generated by that proxy managed object server pool instance. Events are propagated by means of class member functions; i.e., when a proxy managed object server pool needs to “generate an event”, it will invoke the appropriate member function of each client currently registered.

The “server added” event is one example of an event that can be generated by a proxy managed object server pool. The corresponding member function of **RTRProxyManagedObjectServerPoolClient** is **processProxyManagedObjectServerAdded()**. Thus, when a proxy managed object server is added to the proxy managed object server pool, it will invoke the **processProxyManagedObjectServerAdded()** member function of each registered client.

All events that can be generated by instances of **RTRProxyManagedObjectServerPool** have a corresponding processing method in **RTRProxyManagedObjectServerPoolClient**. In the case of **RTRManagedObjectServerPoolClient** there is a default implementation (behavior) defined. Both event processing methods have been declared as virtual and the default behavior is to do nothing.

For example:

```
//RTRProxyManagedObjectServerPtr _pmos;
//event processing for RTRProxyManagedObjectServerPoolClient
void processProxyManagedObjectServerAdded(RTRProxyManagedObjectServerPool& pmosp,
        RTRProxyManagedObjectServer& pmos)
    cout << "pmosp event: Added a pmos  @ " << pmos.text() << endl;
    _pmos = &pmos;
    //register to receive events from pmos
    _pmos->addClient(*this);
}
```

4.7 Shared Memory Proxy Managed Object Server Pool

Also refer to Section 4.6.

4.7.1 Constructor

RTRShmProxyManagedObjectServerPool(const RTRObjectId& context, const char* name)

4.7.2 Operations

void RTRShmProxyManagedObjectServerPool::addServer(const char *key)
Add a proxy managed object server to the shared memory proxy managed object server pool using the shared memory key.

void RTRShmProxyManagedObjectServerPool::dropServer(const char *key)
Remove a proxy managed object server from the shared memory proxy managed object server pool given the shared memory key.

For Example:

```
RTRObjectId instanceId("ShmPool");
RTRShmProxyManagedObjectServerPool pool(instanceId, "pool");

char *keyPtr[] = {"456", "234", "80", "81", "82"};

int len = 5;
for (int i = 0; i < 5; i++)
    pool.addServer(keyPtr[i]);
..
```

4.8 Proxy Managed Object Class Directory

A proxy managed object class directory is useful for an application that is interested in a subset of proxy managed objects. Suppose a managing application is needed which will display all of the managed variables contained in a managed object of a certain type, and this type of managed object is present in all of the managed applications on the node. For example, you want to display all of the managed objects of type “User” for a node. This would involve establishing a “connection” to every managed application, querying each managed application for all instances of managed object of type “User”, and monitoring each managed application for additions/deletions of managed objects of type “User”. This is the functionality that is provided by the proxy managed object class directory and proxy managed object server pool (refer to section 4.5). A proxy managed object server pool is specified when creating the proxy managed object class directory.

The following is a list of services that the proxy managed object class directory provides. Figure 24 shows the proxy managed object class directory view.

- The proxy managed object class directory maintains a “master” list of proxy managed object handles of the desired class type.
- The proxy managed object class directory queries each proxy managed object server to obtain a list of proxy managed object handles that match the specified class type (classId) and adds these lists to the master list.
- The proxy managed object class directory monitors (is a registered client of) each of the proxy managed object servers provided by the proxy managed object server pool. As new proxy managed objects of the desired type are added, their handles will be added to the master list. As proxy managed objects of the desired type are removed, their handles will be removed from the master list.
- The proxy managed object class directory monitors (is a registered client of) the proxy managed object server pool. As new proxy managed object servers are added, they will be queried for a list of proxy managed object handles of the desired class type. This list will be added to the master list. As proxy managed object servers are removed, the list of proxy managed object handles obtained from that server will be removed from the master list.
- The proxy managed object class directory will notify its registered clients about changes (additions/deletions) to the master list.

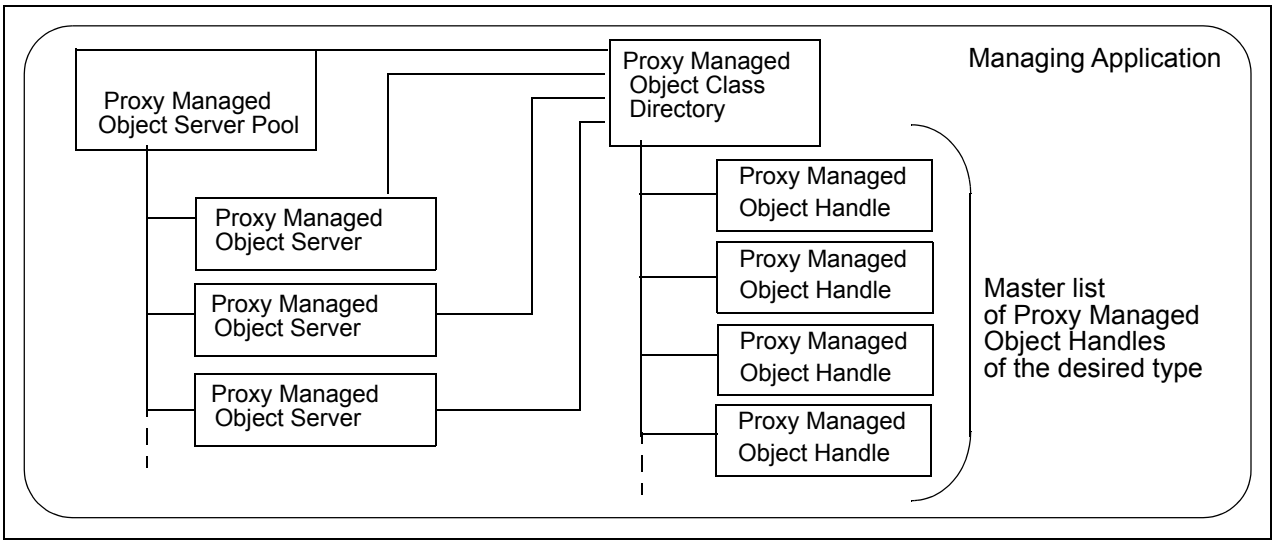


Figure 24. Proxy Managed Object Class Directory view

4.8.1 Attributes

RTRObjectId& classFilter()
The class filter that was specified when creating the proxy managed object class directory. Once the proxy managed object class directory has been created, this method can be used to get the filter class type.

4.8.2 Access to Proxy Managed Object Handles

The proxy managed object class directory provides sequential and random access to the handles of the proxy managed objects which match the given class filter.

4.8.2.1 Sequential Access to Proxy Managed Object Handles

RTRProxyManagedObjectHandleIterator handles()
An iterator which provides sequential access to all proxy managed object handles contained by a proxy managed object class directory.

Sequential access is provided by means of an “iterator” class, an instance of which is associated with a particular proxy managed object class directory instance and provides access to each handle of the proxy managed objects in the proxy managed object class directory. The proxy managed object handle iterator is defined by the class **`RTRProxyManagedObjectHandleIterator`**.

The example code below shows how a proxy managed object handle iterator is used to traverse all the proxy managed object handles of a proxy managed object class directory.

For example:

```
//RTRProxyManagedObjectServerPool pool;
RTRObjectId filter(“User”);
RTRProxyManagedObjectClassDirectory Directory(pool, filter);
RTRProxyManagedObjectHandleIterator  Iterator = Directory.handles();
for ( Iterator.start(); !Iterator.off(); Iterator.forth() )
{
    //Iterator.item() returns the current Proxy Managed Object Handle
    RTRProxyManagedObjectHandle handle = Iterator.item();
}
```

4.8.2.2 Random Access to Proxy Managed Object Handles

const RTRProxyManagedObjectHandle* handle(const RTRObjectId&)
Provides random access to a proxy managed object handle, given the instance identifier.

RTRBOOL hasHandle(const RTRObjectId&)
Tests the existence of proxy managed object handle by querying the proxy managed object class directory for a specific proxy managed object handle, given the instance identifier.

RTRBOOL hasHandle(const RTRProxyManagedObjectHandle&)
Tests the existence of proxy managed object handle by querying the proxy managed object class directory for a specific proxy managed object handle, given a reference to the proxy managed object handle.

The **handle()** method takes an RTRObjectId & as an argument which represents the instance identifier (instanceId) of the proxy managed object that you are looking for and returns a pointer to the proxy managed object handle. This method returns a NULL pointer if the desired proxy managed object handle does not exist.

- Code fragment example:

```
//RTRProxyManagedObjectServerPool pool;
//RTRProxyManagedObject pmo;
RTRObjectId filter("User");
RTRProxyManagedObjectClassDirectory Directory(pool, filter);
const RTRProxyManagedObjectHandle* moh = Directory.handle( pmo.instanceId() );
if ( moh == (RTRProxyManagedObjectHandle *) NULL )
{
    // Managed Object pmo is not present in Directory.
}
```

- **`hasHandle()`** method example:

```
//RTRProxyManagedObjectClassDirectory classDir;
//const RTRProxyManagedObjectHandle moh;
if ( classDir.hasHandle( mo ) )
{
    // moh is not present in classDir.
}
```

4.8.3 Proxy Managed Object Class Directory Client Management

void addClient (const RTRProxyManagedObjectClassDirectoryClient&)
Registers a client with an individual proxy managed object class directory so that the given client will receive all events subsequently generated by that class directory.

void dropClient (const RTRProxyManagedObjectClassDirectoryClient&)
Un-registers a client with a particular proxy managed object class directory so that the given client will no longer receive events from that class directory.

RTRBOOL hasClient (const RTRProxyManagedObjectClassDirectoryClient&)
Indicates whether a given client is currently registered to receive events from a particular proxy managed object class directory. Allows application developer to verify precondition of corresponding **`addClient()`** and **`dropClient()`** methods.

Figure 9 shows the proxy managed object class directory and client relationship.

For example:

```
//Monitor DirectoryClient;
//      (Monitor is a Proxy Managed Object Class Directory client class)
//RTRProxyManagedObjectServerPool pool;
RTRObjectId filter("User");
RTRProxyManagedObjectClassDirectory Directory(pool, filter);

//are we registered to receive events from Directory?
if(!classDir.hasClient(DirectoryClient))
    classDir.addClient(DirectoryClient);     //if not, then register
```

4.8.4 Proxy Managed Object Class Directory Clients

virtual void processDirectoryHandleAdded (
RTRProxyManagedObjectClassDirectory&,
RTRProxyManagedObjectServer&,
const RTRProxyManagedObjectHandle&) = 0

The given managed object handle has been added to the class directory.

virtual void processDirectoryHandleRemoved (
RTRProxyManagedObjectClassDirectory&,
RTRProxyManagedObjectServer&,
RTRProxyManagedObjectHandle&) = 0

The given managed object handle has been removed from the class directory.

To use the above two event processing methods, proxy managed object class directory clients should be descendants of (inherit from) **`RTRProxyManagedObjectClassDirectoryClient`**. As clients, they can register to receive events from one or more instances of **`RTRProxyManagedObjectClassDirectory`**. Once registered with a class directory they will receive events generated by that class directory instance. Events are propagated by means of class member functions; i.e., when a variable needs to "generate an event", it will invoke the appropriate member function of each client currently registered.

The "handle added" event is one example of an event which can be generated by a proxy managed object class directory. The corresponding member function of **`RTRProxyManagedObjectClassDirectoryClient`** is **`processDirectoryHandleAdded()`**. Thus, when a proxy managed object handle is added to the proxy managed object class directory, it will invoke the **`processDirectoryHandleAdded()`** member function of each registered client.

All events that can be generated by instances of **`RTRProxyManagedObjectClassDirectory`** have a corresponding processing method in **`RTRProxyManagedObjectClassDirectoryClient`**. In the cases of **`RTRManagedObjectClassDirectoryClient`** there is no default implementation (behavior) defined. Both processing methods have been declared as pure virtual. So the application developer must implement both member functions, even if the client is not interested in certain events.

For Example:

```
//event processing for RTRProxyManagedObjectClassDirectoryClient

void processDirectoryHandleAdded(RTRProxyManagedObjectClassDirectory& Directory,
        RTRProxyManagedObjectServer& Server,
        RTRProxyManagedObjectHandle& ObjectHandle)
{
    cout << "Directory event: Added a managed object handle  " << "@" << Server.text() << endl;

    cout << "managed object handle added to pool: " << endl
         << "     instance identifier is " << ObjectHandle.instanceId() << endl
         << "     name is " << ObjectHandle.name() << endl
         << "     class identifier is " << ObjectHandle.classId() << endl;
}

void processDirectoryHandleRemoved(RTRProxyManagedObjectClassDirectory& Directory,
        RTRProxyManagedObjectServer& Server,
        RTRProxyManagedObjectHandle& ObjectHandle)
{
    cout << "Directory event: Removed a managed object handle  " << "@" << Server.text() << endl;

    cout << "managed object handle removed from pool: " << endl
         << "     instance identifier is " << ObjectHandle.instanceId() << endl
         << "     name is " << ObjectHandle.name() << endl
         << "     class identifier is " << ObjectHandle.classId() << endl;
```

```
}
```

4.8.5 Class Diagram (Simplified)

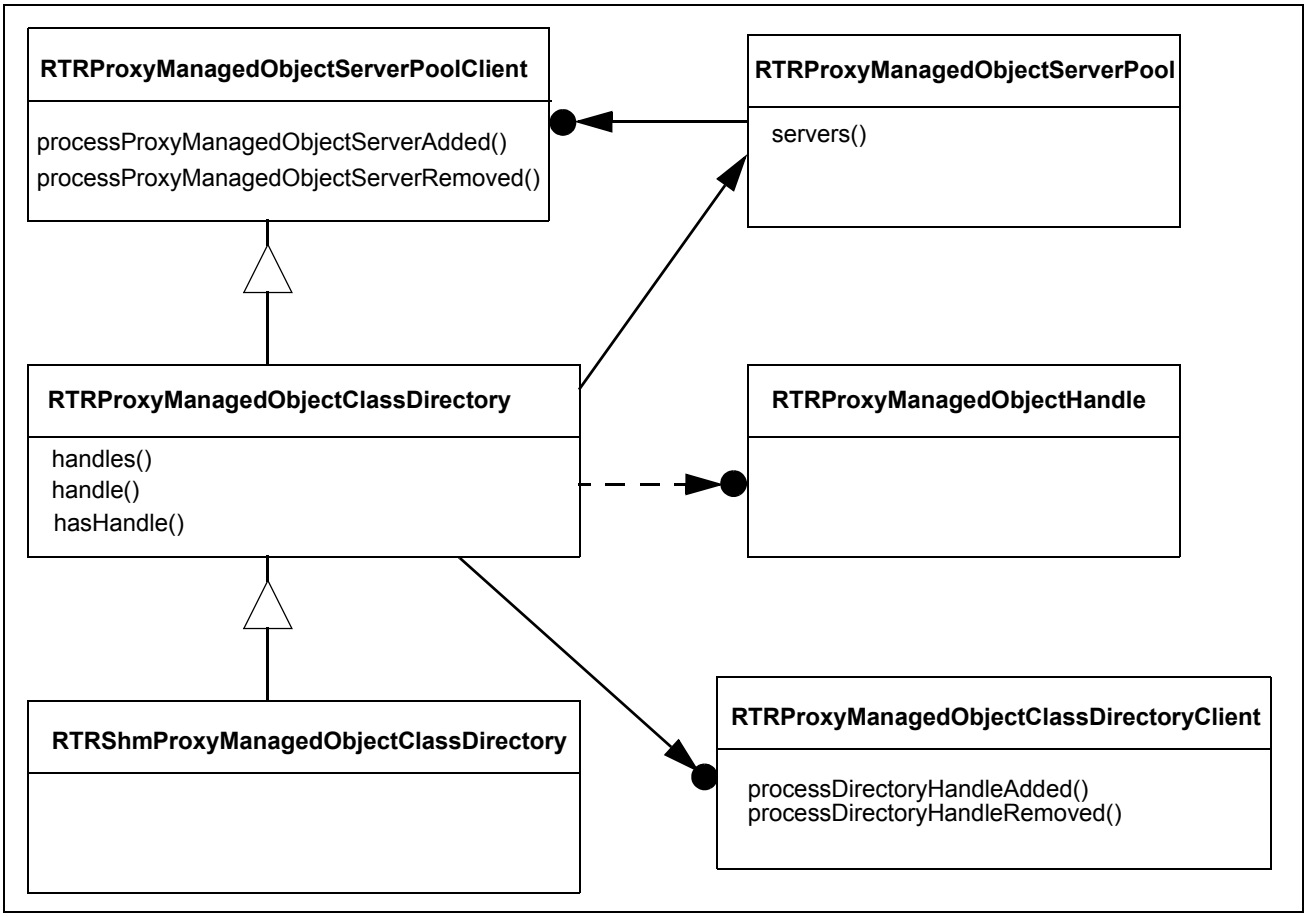


Figure 25. Proxy Managed Object Class Directory - Simplified Class Diagram

4.9 Proxy Managed Object Class Directory Factory

For a given proxy managed object server pool, proxy managed object class directories can be created by a factory (**`RTRShmProxyManagedObjectClassDirFactory`**). This factory instantiates proxy managed object class directories, given a class identifier. Figure 24 shows the proxy managed object class directory factory view.

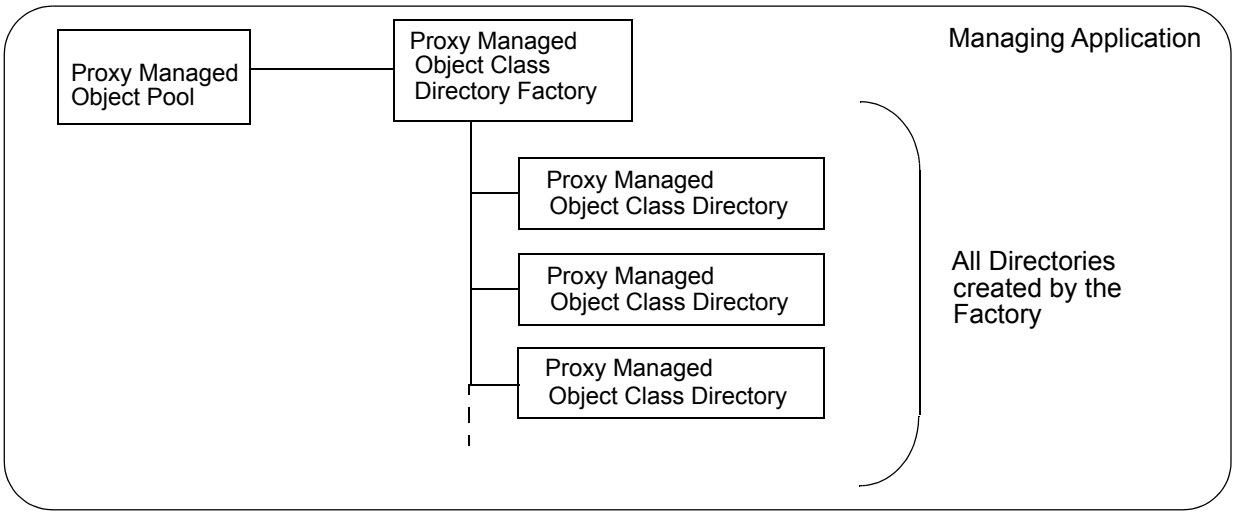


Figure 26. Proxy Managed Object Class Directory View

The proxy managed object class directory factory provides no access or any iterator for the proxy managed object class directories it instantiates. The purpose of the proxy managed object class directory factory is to instantiate the appropriate implementation of the proxy managed object class directory.

The rest of the program can instantiate proxy managed object class directories without caring about client-specific implementations. This keeps all implementation parts in one location.

The following allocates a new proxy managed object class directory:

RTRProxyManagedObjectClassDirectoryPtr newClassDirectory (const RTRObjectId& classFilter)

4.10 Shared Memory Proxy Managed Object Class Directory Factory

This is the shared memory implementation of Proxy Managed Object Class Directory Factory.

RTRProxyManagedObjectClassDirectoryPtr newClassDirectory(const RTRObjectId&) const

For example:

```
//RTRShmProxyManagedObjectServerPool shmObjectServerPool;
//RTRObjectId filter("User");

RTRShmProxyManagedObjectClassDirFactory factory(shmObjectServerPool)

RTRProxyManagedObjectPool objPool(*factory.newClassDirectory(filter));
```

4.11 Proxy Managed Object Pool

The proxy managed object class directory provided a means to obtain the handles of all the available proxy managed objects of a desired type. In order to complete the picture in this problem domain, a method is needed that will take the handles from a class directory and request the proxy managed objects. This is the responsibility of the proxy managed object pool.

The following is a list of services that the proxy managed object pool provides. Figure 27 shows the proxy managed object pool view.

- The proxy managed object pool maintains a set of proxy managed objects (of a particular type/class).
- The proxy managed object pool monitors (is a registered client of) a proxy managed object class directory. As new proxy managed object handles are added to the proxy managed object class directory, the proxy managed object pool will request the new proxy managed object. As proxy managed object handles are removed from the proxy managed object class directory, the proxy managed object pool will remove the proxy managed object from its list.
- The proxy managed object pool will notify its registered clients about changes (additions/deletions) to the proxy managed object pool.

Developers do not need to use the provided class **`RTRProxyManagedObjectPool`**. If they want only some of the proxy managed objects in a proxy managed object class directory, they can write their own class. With **`RTRProxyManagedObjectPool`** we get all the proxy managed objects.

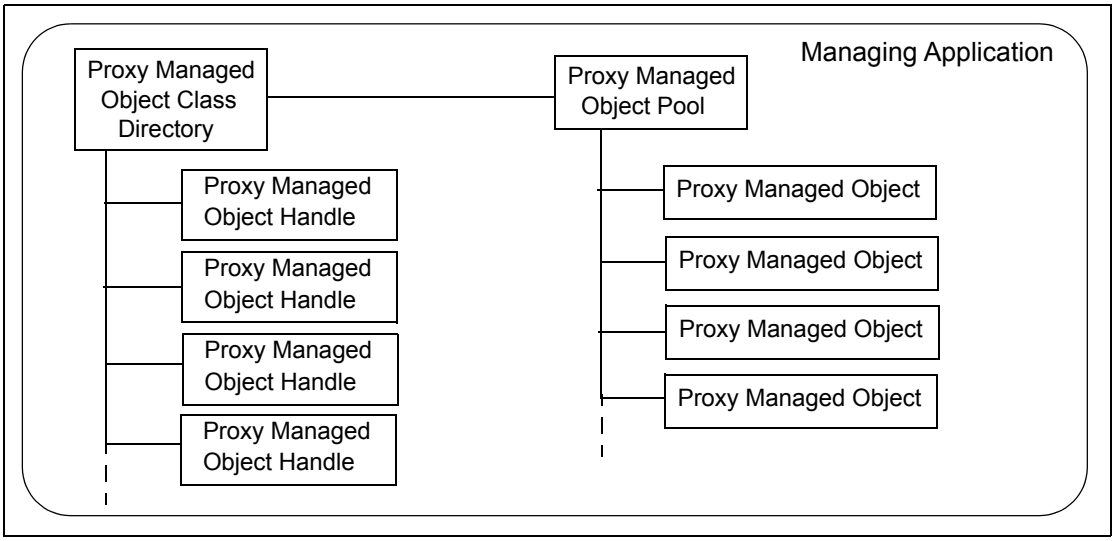


Figure 27. Proxy Managed Object Pool view

4.11.1 Access to Proxy Managed Objects

The proxy managed object pool provides sequential and random access to the available proxy managed objects.

4.11.1.1 Sequential Access to Proxy Managed Objects

RTRLinkedListCursor<RTRProxyManagedObjectPtr> objects()
An iterator which provides sequential access to all the proxy managed objects in the proxy managed object pool.

Sequential access is provided by the class **`RTRLinkedListCursor`**.

The following example code illustrates how a proxy managed object iterator is used to traverse all the proxy managed objects in a proxy managed object pool.

```
//RTRProxyManagedObjectClassDirectory Directory;
RTRProxyManagedObjectPtr mo;
RTRProxyManagedObjectPool pool(Directory);
RTRLinkedListCursor<RTRProxyManagedObjectPtr> Iterator = pool.objects();
for ( Iterator.start(); !Iterator.off(); Iterator.forth() )
{
    //Iterator.item() returns the current Proxy Managed Object
    mo = Iterator.item();
}
```

4.11.1.2 Random Access to Proxy Managed Objects

RTRBOOL hasObject(const RTRObjectId&) const
Tests the existence of a proxy managed object by querying the proxy managed object pool, given the instance identifier of the proxy managed object.

RTRProxyManagedObject* object(const RTRObjectId&) const
Provides random access to a proxy managed object, given the instance identifier.

For Example:

```
//Monitor ObjectPoolClient;
//      (Monitor is a Proxy Managed Object Pool client class)
//RTRProxyManagedObjectClassDirectory Directory;
//RTRProxyManagedObject object;
RTRProxyManagedObjectPool pool(Directory);

if( pool.hasObject( object.instanceId() )
{
    RTRProxyManagedObjectPtr object1 = pool.object( object.instanceId() );

    //register this client (ObjectPoolClient) with this Managed Object
    object1->addClient(ObjectPoolClient);
}
else
    cout << "object does not exist in pool" << endl;
```

4.11.2 Proxy Managed Object Pool Client Management

void addClient (const RTRProxyManagedObjectPoolClient&)
Registers a client with an individual proxy managed object pool so that the given client will receive all events subsequently generated by that proxy managed object pool.

void dropClient (const RTRProxyManagedObjectPoolClient&)
Un-registers a client with a particular proxy managed object pool so that the given client will no longer receive events from that proxy managed object pool.

RTRBOOL hasClient (const RTRProxyManagedObjectPoolClient&)
Determines whether a given client is currently registered to receive events from a particular proxy managed object pool.

Figure 9 shows the proxy managed object class pool and client relationship.

For example:

```
//Monitor ObjectPoolClient;
//      (Monitor is a Proxy Managed Object Pool client class)
//RTRProxyManagedObjectServerPool mosPool;
RTRObjectId filter("User");
RTRProxyManagedObjectClassDirectory Directory(mosPool, filter);
RTRProxyManagedObjectPool moPool(Directory);

//are we registered to receive events from moPool?
if(!moPool.hasClient(ObjectPoolClient))
    moPool.addClient(ObjectPoolClient);     //if not, then register
```

4.11.3 Proxy Managed Object Pool Clients

virtual void processProxyManagedObjectAdded (
RTRProxyManagedObjectPool&,
RTRProxyManagedObject&) = 0
The given proxy managed object has been added to the proxy managed object pool.

virtual void processProxyManagedObjectRemoved (
RTRProxyManagedObjectPool&,
RTRProxyManagedObject&) = 0
The given proxy managed object has been removed from the proxy managed object pool.

To use the above two event processing methods, proxy managed object pool clients should be descendants of (inherit from) **`RTRProxyManagedObjectPoolClient`**. The clients can register to receive events from one or more instances of **`RTRProxyManagedObjectPool`**. Once registered with a proxy managed object pool, they will receive events generated by that proxy managed object pool instance. Events are propagated by means of class member functions; i.e., when a variable needs to “generate an event”, it will invoke the appropriate member function of each client currently registered.

The “proxy managed object added” event is one example of an event that can be generated by a proxy managed object pool. The corresponding member function of **`RTRProxyManagedObjectPoolClient`** is **`processProxyManagedObjectAdded()`**. Thus, when a proxy managed object is added to the proxy managed object pool, it will invoke the **`processProxyManagedObjectAdded()`** member function of each registered client.

All events that can be generated by instances of **`RTRProxyManagedObjectPool`** have a corresponding processing method in **`RTRProxyManagedObjectPoolClient`**. In the cases of **`RTRManagedObjectPoolClient`** there is no default implementation (behavior) defined. Both processing methods have been declared as pure virtual. So the application developer must implement both member functions, even if the client is not interested in certain events.

For Example:

```
//event processing for RTRProxyManagedObjectPoolClient
void processProxyManagedObjectAdded(RTRProxyManagedObjectPool& pool,
        RTRProxyManagedObject& mo)
{
    cout << "pool event: Added a proxy managed object  " << "@" << mo.text() << endl;

    //register this client (*this) with this Managed Object
    mo.addClient(*this);
}

void processProxyManagedObjectRemoved(RTRProxyManagedObjectPool& pool,
        RTRProxyManagedObject& mo)
{
    cout << "pool event: Removed a proxy managed object  " << "@" << pmo.text() << endl;
    if(mo.hasClient(*this))     //are we registered to receive events from mo?
        mo.dropClient(*this);     //if yes, then un-register
}
```

4.11.4 Class Diagram (Simplified)

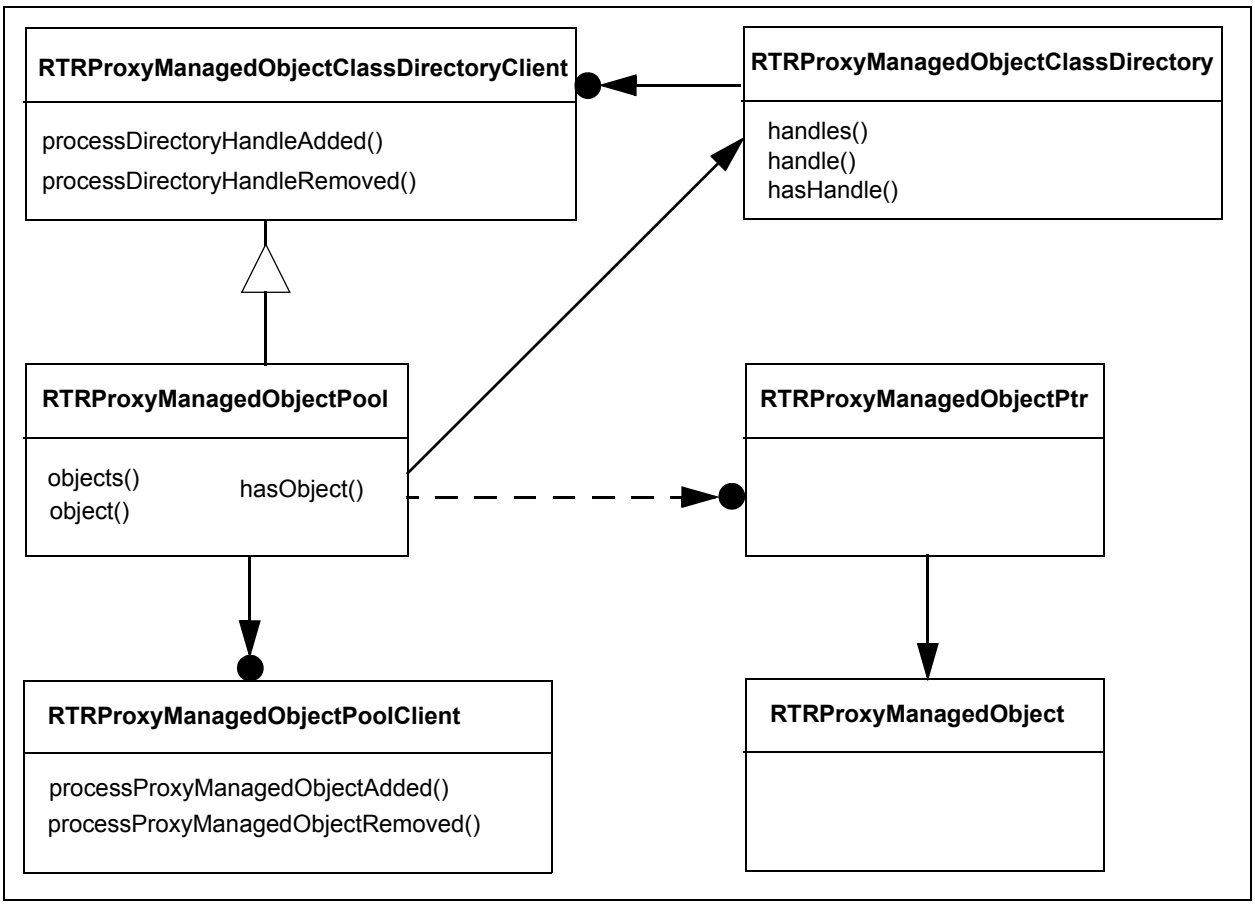


Figure 28. Proxy Managed Object Pool -Simplified Class Diagram

4.12 Managing Application Example

4.12.1 Managing Application Example

This section presents an example of a managing application. This application creates a shared memory proxy managed object server pool first, then it adds one proxy managed object server for each shared memory segment it wishes to monitor. It also monitors all the root managed objects, all the managed variables in the root managed objects, and all the managed variable updates as they are generated. All information is displayed on the standard output device.

All the monitoring — of the proxy managed object server pool, the proxy managed object servers, the root proxy managed objects, and the proxy managed variables — is done in the Monitor class.

4.12.1.1 mainRoots.C

This file provides the main section of the application. Here we use the Monitor class, which is implemented in the file **monitorRoots.C** (see the following section).

```
#include <iostream.h>

#include "rtr/selectni.h"// Event loop implementation
#include "rtr/shmpmosp.h"     // RTRShmProxyManagedObjectServerPool
#include "monitorRoots.h"

int main(int argc, char **argv)
{
   RTRObjectId instanceId("shmApp");
   RTRShmProxyManagedObjectServerPool pool(instanceId, "pool");

   // Create a Server Pool
   // add server for each shared memory segment identified by its key
   char *keyPtr[] = {"456", "234", "80", "81", "82"};
   int len = 5;
   for (int i = 0; i < len; i++)
      pool.addServer(keyPtr[i]);

   // Monitor the server pool
   Monitor mon(pool);

   // This application uses the select() based event loop.
   // You may choose a different implementation (e.g. Windows main loop
   // XWindows Event Notifier, you're own implementation etc.).
    RTRSelectNotifier::run();

   return 0;
}
```

4.12.1.2 monitorRoots.C

This file provides the implementation (Monitor class) of monitoring all of the root managed objects and their managed variables.

```
#include <iostream.h>

#include "monitorRoots.h"


Monitor::Monitor( RTRProxyManagedObjectServerPool& p )
   : _pmosp(p)
{
   // Register with the Server Pool to receive events.
```

```
    _pmosp.addClient(*this);
    RTRLinkedListCursor<RTRProxyManagedObjectServer> iter = _pmosp.servers();
    for (iter.start(); !iter.off(); iter.forth())
        processProxyManagedObjectServerAdded(_pmosp, (RTRProxyManagedObjectServer&)*iter);
}

Monitor::~Monitor()
{
    // Be sure to drop client
    if ( _pmosp.hasClient(*this) )
        _pmosp.dropClient(*this);
}


// Event processing for RTRProxyManagedObjectServerPoolClient
void Monitor::processProxyManagedObjectServerAdded(
        RTRProxyManagedObjectServerPool& pmosp,
        RTRProxyManagedObjectServer& pmos
        )
{
    cout << "pmosp event: Added a pmos  @" << pmos.text() << endl;
    if(pmos.inSync() == RTRTRUE)
    {
        cout << "pmos is inSync" << endl;
        processObjectServerSync(pmos);
    }
    else
    {
        cout << "pmos is not inSync" << endl;
    }
    // Register with each Server to receive events
    pmos.addClient(*this);
}

void Monitor::processProxyManagedObjectServerRemoved(
        RTRProxyManagedObjectServerPool& pmosp,
        RTRProxyManagedObjectServer& pmos
        )
{
    cout << "pmosp event: Removed a pmos  @" << pmos.text() << endl;
    // I know that I'm a client (since I received this event)
    // so be sure to drop client when the Server is no longer valid.
    pmos.dropClient(*this);
}


//event processing for RTRProxyManagedObjectServerClient
void Monitor::processObjectServerError(
        RTRProxyManagedObjectServer& pmos
        )
{
    cout << "pmos event: Error  @" << pmos.text() << endl;
    // Drop client since this Server is no longer valid.
    pmos.dropClient(*this);
}

void Monitor::processObjectServerSync(
        RTRProxyManagedObjectServer& pmos
        )
{
```

```
    cout << "pmos event: in Sync  @" << pmos.text() << endl;

    // Now that the Server is in Sync I can
    // iterate through all Root Proxy Managed Objects in pmos
    // for each root Proxy Managed Object I will become its client.
    RTRProxyManagedObjectHandleIterator pmosIterator = pmos.roots();
    for ( pmosIterator.start(); !pmosIterator.off(); pmosIterator.forth() )
    {
        // pmosIterator.item() is the handle to the current mo
        cout << "found handle to an mo: " << endl
            << "     instance identifier is " << pmosIterator.item().instanceId() << endl
            << "     name is " << pmosIterator.item().name() << endl
            << "     class identifier is " << pmosIterator.item().classId() << endl;

        //now clone the current Proxy Managed Object
        RTRProxyManagedObjectPtr pmoPtr = pmos.object(pmosIterator.item());

        // Maintain a smart pointer reference to the Proxy Managed Object
        // or else the object will be garbage collected.
        addToList(pmoPtr);

        pmoPtr->addClient( (RTRProxyManagedObjectClient &) *this );
        /* If the Object is already inSync() then I will not receive
         * the 'Sync' event, so I need to check it here.
         */
        if ( pmoPtr->inSync() == RTRTRUE )
        {
            cout << "pmoPtr is inSync" << endl;
            processProxyManagedObjectSync(*pmoPtr);
        }
        else
        {
            cout << "pmoPtr is not inSync" << endl;
        }
    }
}

void Monitor::processObjectServerRootAdded(
        RTRProxyManagedObjectServer& pmos,
        const RTRProxyManagedObjectHandle& pmoH
        )
{
    cout <<"pmos event: Added a root on pmos  @" << pmos.text() << endl;

     //now clone the current Proxy Managed Object
    RTRProxyManagedObjectPtr pmoPtr = pmos.object(pmoH);

    // Maintain a smart pointer reference to the Proxy Managed Object
    // or else the object will be garbage collected.
    addToList(pmoPtr);

    pmoPtr->addClient( (RTRProxyManagedObjectClient &) *this );
    /* If the Object is already inSync() then I will not receive
    * the 'Sync' event, so I need to check it here.
    */
    if ( pmoPtr->inSync() == RTRTRUE )
    {
        cout << "pmoPtr is inSync" << endl;
        processProxyManagedObjectSync(*pmoPtr);
    }
```

```
    else
    {
        cout << "pmoPtr is not inSync" << endl;
    }
}

void Monitor::processObjectServerRootRemoved(
       RTRProxyManagedObjectServer& pmos,
       const RTRProxyManagedObjectHandle& pmoh
       )
{
    cout <<"pmos event: Removed a root from pmos  @" << pmos.text() << endl;
}


//
// Event processing for RTRProxyManagedObjectClient
//
void Monitor::processProxyManagedObjectError(
       const RTRProxyManagedObject& pmo
       )
{
    cout << "pmo event: Error  @" << pmo.text() << endl;
}

void Monitor::processProxyManagedObjectSync(
    const RTRProxyManagedObject& pmo
    )
{
    cout << "pmo event: Sync" << endl;

    //iterate through all Proxy Managed Variables
    RTRProxyManagedVarHandleIterator pmvIterator = pmo.variableHandles();
    for ( pmvIterator.start(); !pmvIterator.off(); pmvIterator.forth() )
    {
        cout << "      found a variable of type  " << pmvIterator.item().typeString() << endl
            << "                        with name " << pmvIterator.item().name() << endl;

        //clone this Proxy Manged Variable
        RTRProxyManagedVariablePtr pmvPtr = pmo.variableByName(pmvIterator.item().name());

        if (pmvPtr->error())
            return;

        //add to pmvList (so there will be no garabage collection with pmvPtr)
        addToList(pmvPtr);

        pmvPtr->addClient(*this);

         /* If the Variable is already inSync() then I will not receive
         * the 'Sync' event, so I need to check it here.
         */
        if ( pmvPtr->inSync() == RTRTRUE )
        {
            cout << "pmvPtr is inSync" << endl;
            processProxyManagedVariableSync(*pmvPtr);
        }
        else
        {
            cout << "pmvPtr is not inSync" << endl;
```

```
        }
    }
}

void Monitor::processProxyManagedObjectDeleted(
        const RTRProxyManagedObject& pmo
        )
{
    cout << "pmo event: Deleted" << endl;
}

void Monitor::processProxyManagedObjectInfo(
        const RTRProxyManagedObject& pmo
        )
{
    cout << "pmo event: Info   @" << pmo.text() << endl;
}

void Monitor::processProxyManagedObjectInService(
        const RTRProxyManagedObject& pmo
        )
{
    cout << "pmo event: InService" << endl;
}

void Monitor::processProxyManagedObjectRecovering(
        const RTRProxyManagedObject& pmo
        )
{
    cout << "pmo event: Recovering" << endl;
}

void Monitor::processProxyManagedObjectWaiting(
        const RTRProxyManagedObject& pmo
        )
{
    cout << "pmo event: Waiting" << endl;
}

void Monitor::processProxyManagedObjectDead(
        const RTRProxyManagedObject& pmo
        )
{
    cout << "pmo event: Dead" << endl;
}

void Monitor::processProxyManagedObjectChildAdded(
        const RTRProxyManagedObject& pmo,
        const RTRProxyManagedObjectHandle& pmoh
        )
{
    cout << "pmo event: ChildAdded" << endl;
}

void Monitor::processProxyManagedObjectChildRemoved(
        const RTRProxyManagedObject& pmo,
        const RTRProxyManagedObjectHandle& pmoh
        )
{
    cout << "pmo event: ChildRemoved" << endl;
```

```
}

void Monitor::processProxyManagedObjectVariableAdded(
       const RTRProxyManagedObject& pmo,
       const RTRProxyManagedVariableHandle& pmoh
       )
{
   cout << "pmo event: VariableAdded" << endl;
}

void Monitor::processProxyManagedObjectVariableRemoved(
       const RTRProxyManagedObject& pmo,
       const RTRProxyManagedVariableHandle& pmoh
       )
{
   cout << "pmo event: VariableRemoved" << endl;
}



//Event processing for RTRProxyManagedVariableClient
void Monitor::processProxyManagedVariableError(
       RTRProxyManagedVariable& pmv
       )
{
   cout << "pmv event: Error  @" << pmv.text() << endl;
}

void Monitor::processProxyManagedVariableSync(
       RTRProxyManagedVariable& pmv
       )
{
   cout << "pmv event: Sync" << endl;
   cout << "cloned mv information:   name        = " << pmv.name() << endl;
   cout << "                         type        = " << pmv.typeString() << endl;
   cout << "                         description = " << pmv.description() << endl;

   //show variable specific values
   showPMVdata(pmv);
}

void Monitor::processProxyManagedVariableUpdate(
       RTRProxyManagedVariable& pmv
       )
{
   cout << "pmv event: Update" << endl;

   cout << "new value of " << pmv.name()
        << " (with instance Id " << pmv.context().instanceId() << ") is "
        << pmv.toString() << endl;
}

void Monitor::processProxyManagedVariableDeleted(
       RTRProxyManagedVariable& pmv
       )
{
   cout << "pmv event: Deleted" << endl;
   pmv.dropClient(*this);
}
```

```
// showPMVdata calls the specific variable member function, which
//                              shows some specific variable data
void Monitor::showPMVdata(
       RTRProxyManagedVariable& pmv
       )
{
   switch ( pmv.type() )
   {
      case RTRProxyManagedVariableHandle::Boolean:
         showBoolean_data(pmv);
         break;

      case RTRProxyManagedVariableHandle::BooleanConfig:
         showBooleanConfig_data(pmv);
         break;

      case RTRProxyManagedVariableHandle::Counter:
         showCounter_data(pmv);
         break;

      case RTRProxyManagedVariableHandle::Numeric:
         showNumeric_data(pmv);
         break;

      case RTRProxyManagedVariableHandle::LargeNumeric:
         showLargeNumeric_data(pmv);
         break;

      case RTRProxyManagedVariableHandle::NumericConfig:
         showNumericConfig_data(pmv);
         break;

      case RTRProxyManagedVariableHandle::NumericRange:
         showNumericRange_data(pmv);
         break;

      case RTRProxyManagedVariableHandle::Gauge:
         showGauge_data(pmv);
         break;

      case RTRProxyManagedVariableHandle::GaugeConfig:
         showGaugeConfig_data(pmv);
         break;

      case RTRProxyManagedVariableHandle::String:
         showString_data(pmv);
         break;

      case RTRProxyManagedVariableHandle::StringConfig:
         showStringConfig_data(pmv);
         break;

      case RTRProxyManagedVariableHandle::Invalid:
         cout << "Invalid Proxy Managed Variable" << endl;
         break;

      default:
         cout << "Unknown Proxy Managed Variable" << endl;
   }
}
```

```
void Monitor::showBoolean_data(RTRProxyManagedVariable& pmv)
{
   RTRProxyManagedBoolean& pmb = (RTRProxyManagedBoolean&)pmv;
   if ( pmb.value() == RTRTRUE )
      cout << "                          value       = RTRTRUE" << endl;
   else
      cout <<"                          value       = RTRFALSE" << endl;
}

void Monitor::showBooleanConfig_data(RTRProxyManagedVariable& pmv)
{
   RTRProxyManagedBooleanConfig& pmbc = (RTRProxyManagedBooleanConfig&)pmv;
   if ( pmbc.activeValue() == RTRTRUE )
      cout << "                          active value = RTRTRUE" << endl;
   else
      cout <<"                          active value = RTRFALSE" << endl;
   if ( pmbc.storeValue() == RTRTRUE )
      cout << "                          store value  = RTRTRUE" << endl;
   else
      cout <<"                          store value  = RTRFALSE" << endl;
   if ( pmbc.factoryDefault() == RTRTRUE )
      cout << "                          factory default = RTRTRUE" << endl;
   else
      cout <<"                          factory default = RTRFALSE" << endl;
}

void Monitor::showCounter_data(RTRProxyManagedVariable& pmv)
{
   RTRProxyManagedCounter& pmc = (RTRProxyManagedCounter&)pmv;
   unsigned long cnt = pmc.value();
   cout << "                          value       = " << cnt << endl;
}

void Monitor::showNumeric_data(RTRProxyManagedVariable& pmv)
{
   RTRProxyManagedNumeric& pmn = (RTRProxyManagedNumeric&)pmv;
   cout << "                          value       = " << pmn.value() << endl;
}

void Monitor::showLargeNumeric_data(RTRProxyManagedVariable& pmv)
{
   RTRProxyManagedLargeNumeric& pmln = (RTRProxyManagedLargeNumeric&)pmv;
   #if defined (_WIN32) || defined (_WIN64)
      char buf[30];//There is no operator << for __int64 type defined for the ostream class.
      sprintf(buf, "%I64d", pmln.value());
      cout << "                          value       = " << buf << endl;
   #else
      cout << "                          value       = " << pmln.value() << endl;
   #endif
}

void Monitor::showNumericConfig_data(RTRProxyManagedVariable& pmv)
{
   RTRProxyManagedNumericConfig& pmnc = (RTRProxyManagedNumericConfig&)pmv;
   cout << "                          active value = " << pmnc.activeValue() << endl
   << "                          min value    = " << pmnc.minValue() << endl
   << "                          max value    = " << pmnc.maxValue() << endl
   << "                          store value  = " << pmnc.storeValue() << endl
   << "                          factory default = " << pmnc.factoryDefault() << endl;
```

```
}

void Monitor::showNumericRange_data(RTRProxyManagedVariable& pmv)
{
   RTRProxyManagedNumericRange& pmnr = (RTRProxyManagedNumericRange&)pmv;
   cout << "                          min value   = " << pmnr.minValue() << endl
   << "                        max value   = " << pmnr.maxValue() << endl;
}

void Monitor::showGauge_data(RTRProxyManagedVariable& pmv)
{
   RTRProxyManagedGauge& pmg = (RTRProxyManagedGauge&)pmv;
   cout << "                          min value   = " << pmg.minValue() << endl
   << "                        max value   = " << pmg.maxValue() << endl
   << "                        lowWaterMark  = " << pmg.lowWaterMark() << endl
   << "                        highWaterMark = " << pmg.highWaterMark() << endl;
}

void Monitor::showGaugeConfig_data(RTRProxyManagedVariable& pmv)
{
   RTRProxyManagedGaugeConfig& pmgc = (RTRProxyManagedGaugeConfig&)pmv;
   cout << "                          min store value     = " << pmgc.minStoreValue() << endl
   << "                        min factory default = " << pmgc.minFactoryDefault() << endl
   << "                        max store value     = " << pmgc.maxStoreValue() << endl
   << "                        max factory default = " << pmgc.maxFactoryDefault() << endl;
}

void Monitor::showString_data(RTRProxyManagedVariable& pmv)
{
   RTRProxyManagedString& pms = (RTRProxyManagedString&)pmv;
   cout << "                          value       = " << pms.value() << endl;
}

void Monitor::showStringConfig_data(RTRProxyManagedVariable& pmv)
{
   RTRProxyManagedStringConfig& pmsc = (RTRProxyManagedStringConfig&)pmv;
   RTRString active = pmsc.activeValue();
   cout << "                          active value = " << pmsc.activeValue() << endl;
   RTRString store = pmsc.storeValue();
   cout << "                          store value  = " << pmsc.storeValue() << endl;
   RTRString fDflt = pmsc.factoryDefault();
   cout << "                          factory default = " << pmsc.factoryDefault() << endl;
}


//create a linked list that contains
//all the proxy managed objects and their children
void Monitor::addToList( const RTRProxyManagedObjectPtr obj )
{
   RTRProxyManagedObjectPtr objPtr;
   RTRBOOL done = RTRFALSE;  //used to maintain correct managed object tree structure

   RTRProxyManagedObjectPtr *nPtr =  new RTRProxyManagedObjectPtr;
   *nPtr = obj;
   if ( _pmoList.empty() )
   {
      _pmoList.addRight( nPtr );
   }
   else
   {
```

```
      for ( _pmoList.start(); !_pmoList.off(); _pmoList.forth() )
      {
         objPtr = *(_pmoList.item());

         if ( obj->instanceId().parent() == objPtr->instanceId() )
         {
            _pmoList.addRight( nPtr );
            done = RTRTRUE;
            break;
         }
      }
      if ( !done )
      {
         _pmoList.start();
         _pmoList.addRight( nPtr );
      }
   }
}


//create a linked list that contains
//all the proxy managed variables
void Monitor::addToList( const RTRProxyManagedVariablePtr var )
{
   RTRProxyManagedVariablePtr   ptr;
   RTRBOOL variable_found = RTRFALSE;  //used to ensure that the variables
                     //added in the same order as we see them

   RTRProxyManagedVariablePtr *nPtr =  new RTRProxyManagedVariablePtr;
   *nPtr = var;

   if ( _pmvList.empty() )
   {
      _pmvList.addRight( nPtr );
   }
   else
   {
      for ( _pmvList.start(); !_pmvList.off(); _pmvList.forth() )
      {
         ptr = *(_pmvList.item());

         if (ptr->error()) {
            if (ptr->hasClient(*this))
               ptr->dropClient(*this);
            _pmvList.remove();
            return;
         }

         if ( var->context().instanceId() == ptr->context().instanceId() )
            variable_found = RTRTRUE;

         if ( ( variable_found ) &&
              (var->context().instanceId() != ptr->context().instanceId()) )
         {
            _pmvList.addLeft( nPtr );
            return;
         }
      }

      // New variable added to list
```

```
        _pmvList.extend( nPtr );
    }
}
```

4.12.1.3 monitorRoots.h

This file provides the declaration for the Monitor class (in **monitorRoots.C**), which monitors all of the root managed objects and their managed variables.

```
#ifndef _monitorRoots_h
#define _monitorRoots_h

#include "rtr/pmosp.h"// RTRProxyManagedObjectServerPool
#include "rtr/pmospc.h"// RTRProxyManagedObjectServerPoolClient
#include "rtr/prxymos.h"// RTRProxyManagedObjectServer
#include "rtr/pmosc.h"// RTRProxyManagedObjectServerClient
#include "rtr/proxymo.h"// RTRProxyManagedObject and all Variables

/* This is a simple class which monitors (by becoming a client)
 * all of the root Managed Objects and their Managed Variables.
 * More complex examples may prefer to have separate classes
 * for each type of client and separate instances of those
 * classes for each Managed Object/Managed Variable.
 */
class Monitor :
    public RTRProxyManagedObjectServerPoolClient,
    public RTRProxyManagedObjectServerClient,
         public RTRProxyManagedObjectClient,
         public RTRProxyManagedVariableClient
{
public:
// Constructor
         Monitor(RTRProxyManagedObjectServerPool& p);

// Destructor
    ~Monitor();

// Event processing -- RTRProxyManagedObjectServerPoolClient
    // The events generated by a RTRProxyManagedObjectServerPool
    void processProxyManagedObjectServerAdded(
           RTRProxyManagedObjectServerPool& pmosp,
                    RTRProxyManagedObjectServer& pmos);
    void processProxyManagedObjectServerRemoved(
           RTRProxyManagedObjectServerPool& pmosp,
                    RTRProxyManagedObjectServer& pmos);

// Event processing -- RTRProxyManagedObjectServerClient
    // The events generated by a RTRProxyManagedObjectServer
    void processObjectServerError(
           RTRProxyManagedObjectServer& pmos);
    void processObjectServerSync(
           RTRProxyManagedObjectServer& pmos);
    void processObjectServerRootAdded(
           RTRProxyManagedObjectServer& pmos,
                    const RTRProxyManagedObjectHandle& pmoh);
    void processObjectServerRootRemoved(
           RTRProxyManagedObjectServer& pmos,
                    const RTRProxyManagedObjectHandle& pmoh);

// Event processing -- RTRProxyManagedObjectClient
```

```
   // The events generated by a RTRProxyManagedObject
        void processProxyManagedObjectError(
          const RTRProxyManagedObject& pmo);
        void processProxyManagedObjectSync(
          const RTRProxyManagedObject& pmo);
        void processProxyManagedObjectDeleted(
          const RTRProxyManagedObject& pmo);
        void processProxyManagedObjectInfo(
          const RTRProxyManagedObject& pmo);
        void processProxyManagedObjectInService(
          const RTRProxyManagedObject& pmo);
        void processProxyManagedObjectRecovering(
          const RTRProxyManagedObject& pmo);
        void processProxyManagedObjectWaiting(
          const RTRProxyManagedObject& pmo);
        void processProxyManagedObjectDead(
          const RTRProxyManagedObject& pmo);
        void processProxyManagedObjectChildAdded(
          const RTRProxyManagedObject& pmo,
                        const RTRProxyManagedObjectHandle& pmoh);
        void processProxyManagedObjectChildRemoved(
          const RTRProxyManagedObject& pmo,
                        const RTRProxyManagedObjectHandle& pmoh);
        void processProxyManagedObjectVariableAdded(
          const RTRProxyManagedObject& pmo,
                        const RTRProxyManagedVariableHandle& pmoh);
        void processProxyManagedObjectVariableRemoved(
          const RTRProxyManagedObject& pmo,
                        const RTRProxyManagedVariableHandle& pmoh);


// Event processing -- RTRProxyManagedVariableClient
   // The events generated by a RTRProxyManagedVariable
        void processProxyManagedVariableError(RTRProxyManagedVariable& pmv);
        void processProxyManagedVariableSync(RTRProxyManagedVariable& pmv);
        void processProxyManagedVariableUpdate(RTRProxyManagedVariable& pmv);
        void processProxyManagedVariableDeleted(RTRProxyManagedVariable& pmv);

private:
// Utilities
        void showPMVdata(RTRProxyManagedVariable& pmv);

        void showBoolean_data(RTRProxyManagedVariable& pmv);
        void showBooleanConfig_data(RTRProxyManagedVariable& pmv);
        void showCounter_data(RTRProxyManagedVariable& pmv);
        void showNumeric_data(RTRProxyManagedVariable& pmv);
        void showLargeNumeric_data(RTRProxyManagedVariable& pmv);
        void showNumericConfig_data(RTRProxyManagedVariable& pmv);
        void showNumericRange_data(RTRProxyManagedVariable& pmv);
        void showGauge_data(RTRProxyManagedVariable& pmv);
        void showGaugeConfig_data(RTRProxyManagedVariable& pmv);
        void showString_data(RTRProxyManagedVariable& pmv);
        void showStringConfig_data(RTRProxyManagedVariable& pmv);

   void addToList( const RTRProxyManagedObjectPtr obj );
   void addToList( const RTRProxyManagedVariablePtr var );

// Data
    RTRProxyManagedObjectServerPool& _pmosp;
   // Maintain a reference (Smart pointer) to all of the
```

```
    // root managed objects and managed variables so that
    // they will not be deleted (garbage collected).
    RTRLinkedList<RTRProxyManagedObjectPtr> _pmoList;
    RTRLinkedList<RTRProxyManagedVariablePtr> _pmvList;
};

#endif
```

5 Managed Applications

5.1 Overview

The following topics are included in this chapter:

- Managed Variable (see Section 5.3)
- Public Variable Type (see Section 5.4)
- Managed Object (see Section 5.5)
- Public Object (see Section 5.6)
- Managed Process (see Section 5.7)
- Server Shared Memory Root (see Section 5.8)
- Shared Memory Managed Object Server Memory Pool (see Section 5.9)
- Shared Memory Managed Object Server (see Section 5.10)
- Shared Memory Server (see Section 5.11)

In addition to an analysis of each topic, there is a description of how the abstract analysis translates into specific class functions. Example code fragments are listed C++. Each topic also includes simplified class diagrams depicting classes, their structure, and the static relationships between them.

5.2 Common Concepts

5.2.1 Client Management

Events can be generated for managed variables for any of the following occurrences:

- variable was deleted
- variable value was changed
- variables are in an error state

Clients have to register with a managed variable in order to receive any events corresponding to that variable. When clients are no longer interested in receiving any more events for a managed variable, they have to deregister with it (see Section 5.3.2.8).

Event generating components may have multiple clients, and those clients may be registered with multiple event generating components. Figure 29 illustrates, for example, the n-to-n cardinality that exists between managed variables and their registered clients.

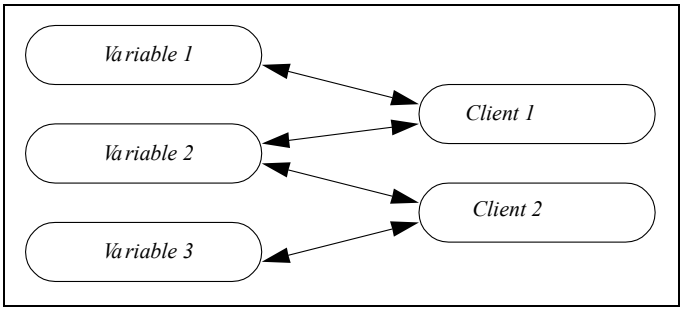


Figure 29. Run-time Relationships Example

In a typical application, these relationships are defined dynamically, i.e., on the basis of user input or other events, meaning that clients may be dynamically added (allocated) or removed (deleted) at any time.

5.3 Managed Variables

Managed variables are used by managed applications to “expose” manageable data. For example, a managed application may decide to “publish” a set of variables related to a particular socket connection (e.g., number of writes/reads, number of overflows, number of messages sent/received). Managed variables represent the managed variables that are published by a managed application.

The eleven types of managed variables supported are listed in Table 12.

VARIABLE CATEGORY	VARIABLE TYPE	MANAGED VARIABLE C++ CLASS NAME
Boolean	boolean	RTRManagedBoolean
Boolean	booleanConfig	RTRManagedBooleanConfig
Numeric	counter	RTRManagedCounter
Numeric	gauge	RTRManagedGauge
Numeric	gaugeConfig	RTRManagedGaugeConfig
Numeric	numeric	RTRManagedNumeric
Numeric	largeNumeric	RTRManagedLargeNumeric
Numeric	numericConfig	RTRManagedNumericConfig
Numeric	numericRange	RTRManagedNumericRange
String	string	RTRManagedString
String	stringConfig	RTRManagedStringConfig

Table 13:

`RTRManagedVariable` is the base class for all eleven of the managed variable classes. This class includes transformation methods for each of the eleven variable types (see Section 5.3.2.7).

5.3.1 Identity

RTRString& name()

The name of the managed variable.

Every managed variable is identified by a name. The name is a string (**`RTRString`**) which uniquely identifies one managed variable from another. The names are only unique within the context of a given managed object.

For example, an instance of a “ServiceAttributes” managed object has several managed variables each with a different name. A second instance of a “ServiceAttributes” managed object will have the same set of managed variables but are distinguished from the first set because they are “contained” in a different managed object instance (with a different name). Figure 30 shows two instances of a “ServiceAttributes” type of managed object. Each instance “contains” managed variables with the same variable name (“Service Name”, “Service State”, etc.) but are distinguished from one another by being “contained” in managed objects with different names (instance identifiers).

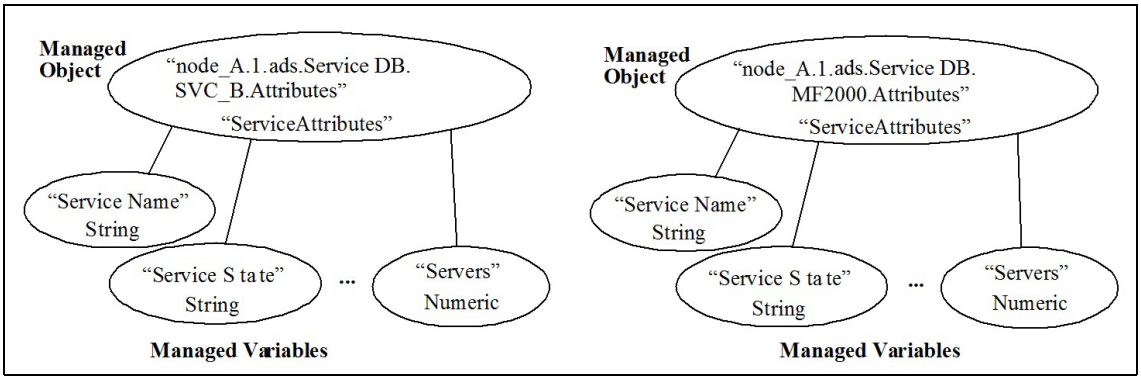


Figure 30. Example of Managed Object/Variable Name Space

5.3.2 Attributes

5.3.2.1 Context

RTRManagedObject& context()

The managed object that contains this managed variable.

RTRString& description()

A textual description of the managed variable.

A managed variable exists within the context of a single managed object. These functions are implemented in its base class- **`RTRManagedVariable`**.

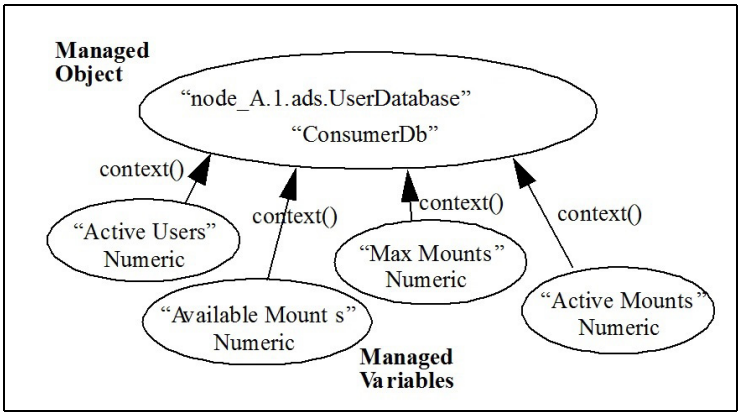


Figure 31. Example of Context Access

Code Fragment Example:

```
// RTRManagedVariable mv;
RTRManagedObject& mo = mv.context();
cout << "The context of smv is: " << mo.instanceId() << ":" << mo.classId() << endl;
```

5.3.2.2 State

RTRBOOL modifyEnabled()
Is the managing application permitted to modify this managed variable?

Depending on the type of managed variable, its current value may be modified by the managing application. The managed application will accept this modification.

Code Fragment Example:

```
//RTRPublicBoolean pb;
if ( pb.modifyEnabled() )
{
    if( pb.inSync() && !pb.error() )
        pb = RTRTRUE;
}
```

5.3.2.3 Variable Type

MVType type()
Returns the type of managed variable. Its value will be one of the following enumerated types:

- Boolean
- BooleanConfig
- Counter
- Numeric
- LargeNumeric
- NumericConfig
- NumericRange
- Gauge
- GaugeConfig
- String
- StringConfig

These managed variables are available to managed applications for the purpose of publishing its operational data. 11 types of managed variables can be divided into the categories: boolean, numeric, or string. These functions are intermediate base class of **`RTRManagedVariable`**.

Managed variables that can be modified are as follows:

- Boolean
- BooleanConfig
- NumericConfig
- Gauge

- GaugeConfig
- String
- StringConfig

5.3.2.4 Boolean Category Variables

The boolean category contains two managed variables (boolean and booleanConfig). Each of these variable types provides different levels of manageability and attribute characteristics.

TYPE	ATTRIBUTES	CLIENT MANAGEABILITY	DESCRIPTION
boolean	active value	Write-Enabled or Read-Only	Managing applications are allowed to set or clear the active value.
booleanConfig	active value store value factory default	Write-Enabled or Read-Only	Specifies a numeric value as a configuration parameter.

Table 14: Boolean Category Variables

5.3.2.5 Numeric Category Variables

The numeric category contains seven managed variables (numeric, largeNumeric, counter, gauge, gaugeConfig, numericRange, and numericConfig). Each of these variable types provides different levels of manageability and attribute characteristics.

TYPE	ATTRIBUTES	CLIENT MANAGEABILITY	DESCRIPTION
numeric	active value	Read-Only	Basic read-only numeric variable
largeNumeric	active value	Read-Only	Basic read-only 64-bit numeric variable
counter	active value	Read-Only	Active value starts at 0 and increments only.
gauge	active value minimum value maximum value	Write-Enabled or Read-Only	Read-only variable whose active value will be between the min and max values. Min and max values may change dynamically. Usually used to represent current utilization of a limited resource (capacity).
gaugeConfigType	active value store value factory default minimum value maximum value	Write-Enabled or Read-Only	Specifies the variable as a configuration parameter.
numericRange	active value minimum value maximum value	Read-Only	The min and max values are determined when the variable is instantiated and will not change during its life-cycle.
numericConfig	active value store value factory default minimum value maximum value	Write-Enabled or Read-Only	Specifies the variable as a configuration parameter.

Table 15: Numeric Category Variables

5.3.2.6 String Category Variables

The string category contains two managed variables (string and stringConfig). Each of these variable types provides different levels of manageability and attribute characteristics.

TYPE	ATTRIBUTES	CLIENT MANAGEABILITY	DESCRIPTION
string	active value	Write-Enabled or Read-Only	Managing applications are allowed to modify the active value.
stringConfig	active value stored value factory default	Write-Enabled or Read-Only	Specifies a string as a configuration parameter.

Table 16: String Category Variables

5.3.2.7 Transformation

operator RTRManagedBoolean()
Transform a managed variable to a managed boolean.

operator RTRManagedBooleanConfig()
Transform a managed variable to a managed booleanConfig.

operator RTRManagedCounter()
Transform a managed variable to a managed counter.

operator RTRManagedGauge()
Transform a managed variable to a managed gauge.

operator RTRManagedGaugeConfig()
Transform a managed variable to a managed gaugeConfig.

operator RTRManagedNumeric()
Transform a managed variable to a managed numeric.

operator RTRManagedLargeNumeric()
Transform a managed variable to a managed large numeric.

operator RTRManagedNumericConfig()
Transform a managed variable to a managed numericConfig.

operator RTRManagedNumericRange()
Transform a managed variable to a managed numericRange.

operator RTRManagedString()
Transform a managed variable to a managed string.

operator RTRManagedStringConfig()
Transform a managed variable to a managed stringConfig.

RTRString toString()
The value of the managed variable represented as a string.

Often a method or an event will return a reference to a managed variable and you may need to transform it into a reference to the particular type of managed variable in order to utilize that variable's methods. You can only perform this downcast operation on a managed variable reference to the actual variable type or to any of its inherited types.

For example, if the managed variable type is a counter, then a reference to the managed variable can only be downcast to a counter. Obviously, it makes no sense to downcast the managed variable to a string or gauge. The following table shows all the allowed downcasts of all the managed variable types.

ACTUAL MANAGED VARIABLE TYPE	ALLOWED DOWNCASTS
boolean	Boolean
booleanConfig	boolean, booleanConfig
counter	counter
numeric	numeric
largeNumeric	largeNumeric
numericConfig	numeric, numericConfig
numericRange	numeric, numericRange
gauge	gauge, numeric
gaugeConfig	gaugeConfig, gauge, numeric
string	string
stringConfig	string, stringConfig

Table 17: Managed Variable Types and Allowed Downcasts

Code Fragment Example:

```
RTRPublicGauge& pg;
long min = pg.minValue();
cout << "gauge min value   = " << min << endl;
long max = pg.maxValue();
cout << "gauge max value   = " << max << endl;
long lowWM = pg.lowWaterMark();
cout << "gauge lowWaterMark  = " << lowWM << endl;
long highWM = pg.highWaterMark();
cout << "gauge highWaterMark = " << highWM << endl;
}
```

5.3.2.8 Managed Variable Client Management

virtual void addClient (RTRManagedVariableClient&)
Register a client with an individual managed variable so that the given client will receive all events subsequently generated by that managed variable.

virtual void dropClient (RTRManagedVariableClient&)
Un-register a client with an individual managed variable so that the given client will no longer receive events generated by that managed variable.

RTRBOOL hasClient (RTRManagedVariableClient&)
Indicates whether a given client is currently registered to receive events from a particular managed variable. Allows application developer to verify precondition of corresponding **`addClient()`** and **`dropClient()`** methods.

RTRBOOL hasClients ()
Indicates whether this managed variable has any clients or not. Allows application developer to verify precondition of corresponding **`addClient()`** and **`dropClient()`** methods.

Figure 29 shows the managed variable and client relationship.

Code Fragment Example:

```
//Monitor mvClient;
//      (Monitor is a Proxy Managed Variable client class)
//RTRManagedVariable mv;

if(!mv.hasClient(mvClient))   //are we registered to receive events from mv?
    mv.addClient(mvClient);     //if not, then register
```

5.3.2.9 Managed Variable Clients

virtual void processVariableChange (RTRManagedVariable&) = 0
One or more of the given managed variable's attributes have changed.

virtual void processVariableDelete (RTRManagedVariable&) = 0
The given managed variable has been deleted by the managed application. This represents an unrecoverable condition. The client should un-register from the managed variable (using **`dropClient()`**) and ensure that no further references are made to that managed variable. Managed variables may come and go at the discretion of the managed application, and it does not necessarily indicate a problem with the managed application.

To use the above two methods, managed variable clients should be descendants of (inherit from) **`RTRManagedVariableClient`**. As clients, they can register to receive events from one or more instances of **`RTRManagedVariable`**. Once registered with a managed variable, they will receive events generated by that managed variable instance. Events are propagated by means of class member functions; i.e., when a managed variable needs to "generate an event", it will invoke the appropriate member function of each client currently registered.

Events that can be generated by instances of **`RTRManagedVariable`** have a corresponding processing method in **`RTRManagedVariableClient`**. In the case of **`RTRManagedVariableClient`** there is no default implementation (behavior) defined. All processing methods have been declared as pure virtual. So the application developer must implement both **`processVariableChangeand processVariableDelete`**, even if the client is not interested in certain events.

5.3.3 Class Diagrams (Simplified)

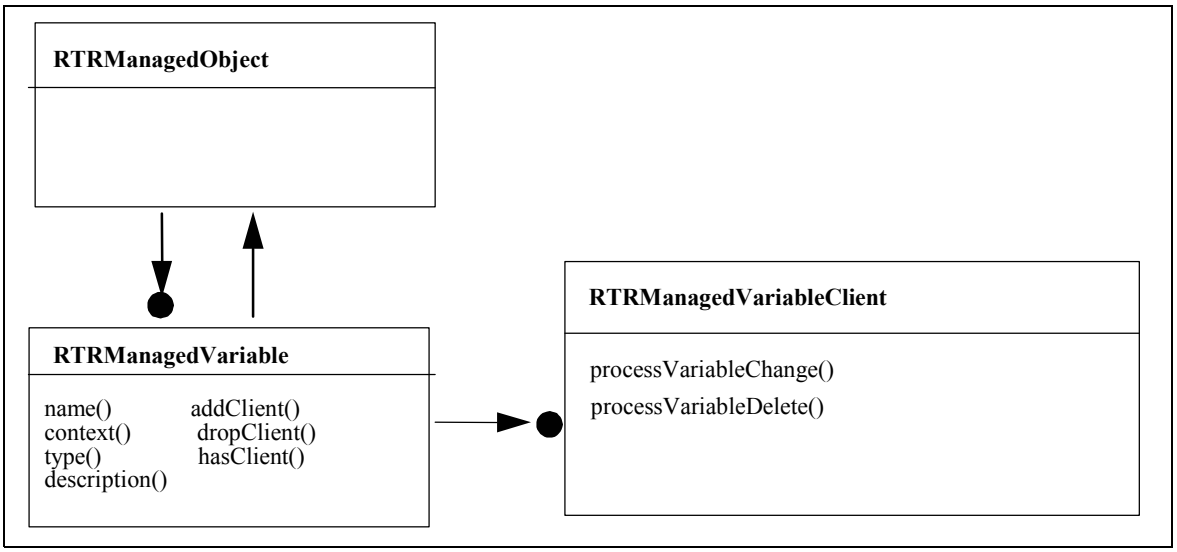


Figure 32. Managed Variable's Association with Context and Clients

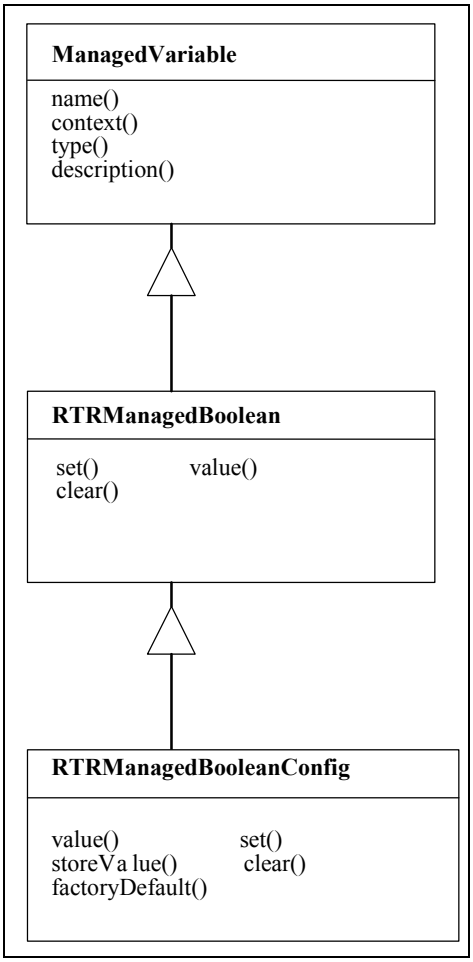


Figure 33. Boolean Category of Managed Variables - Simplified Class Diagram

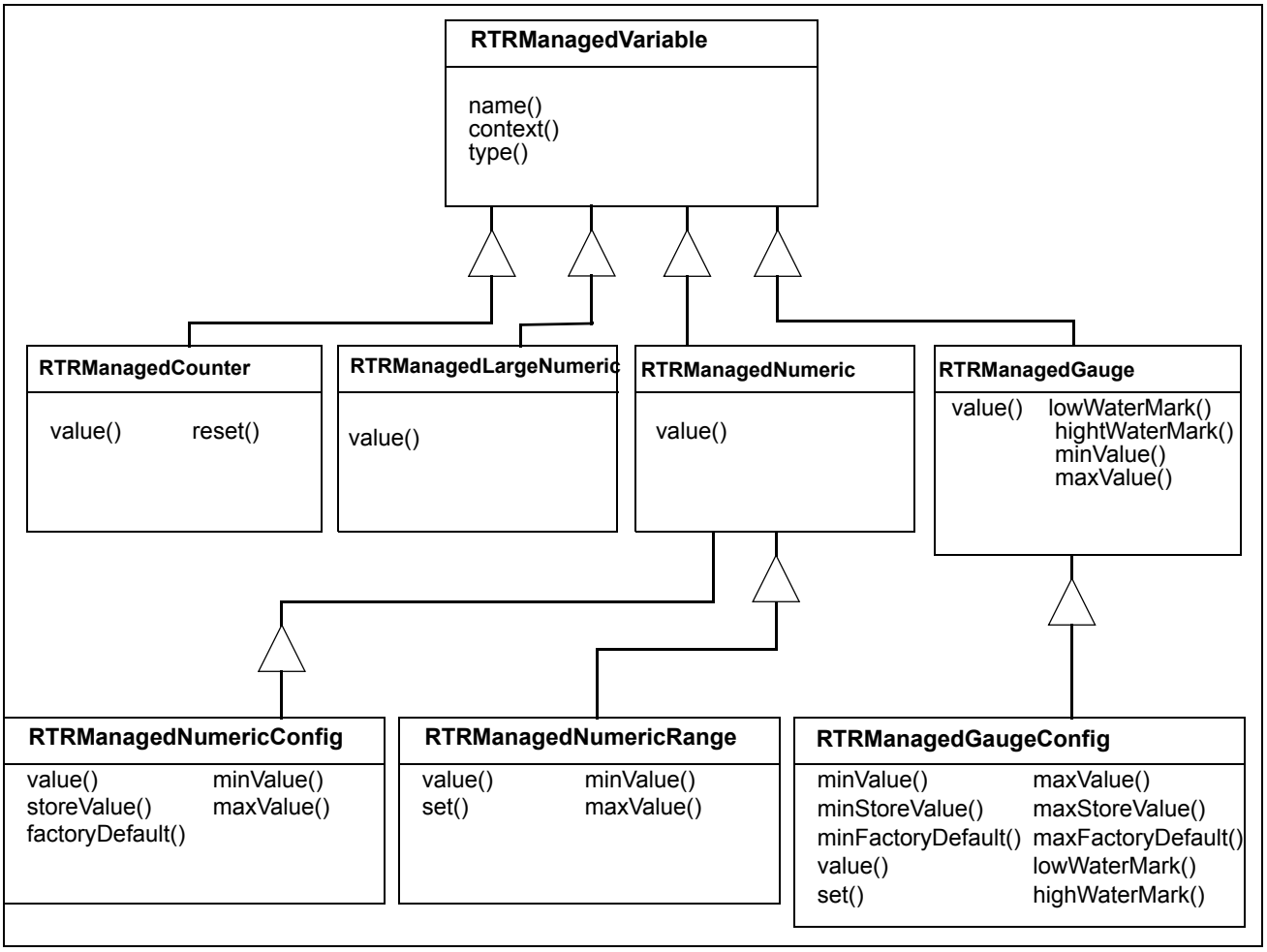


Figure 34. Numeric Category of Managed Variables - Simplified Class Diagram

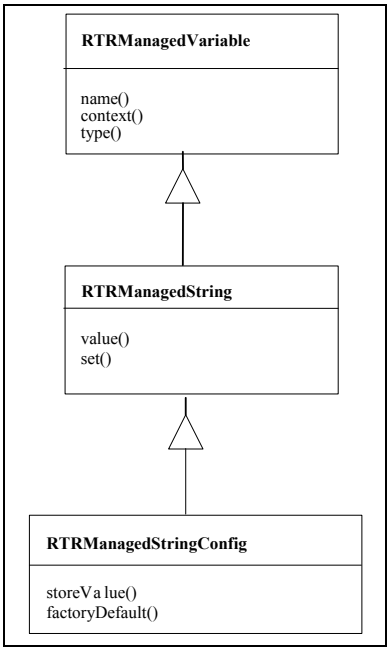


Figure 35. String Category of Managed Variables - Simplified Class Diagram

5.4 Public Variable Types

Each public variable class inherits its corresponding managed variable class. Types of supported public variables are as follows:

VARIABLE CATEGORY	VARIABLE TYPE	PUBLIC VARIABLE C++ CLASS NAME	INHERIT FROM
Boolean	boolean	RTRPublicBoolean	RTRManagedBoolean
Boolean	booleanConfig	RTRPublicBooleanConfig	RTRManagedBooleanConfig
Numeric	counter	RTRPublicCounter	RTRManagedCounter
Numeric	gauge	RTRPublicGauge	RTRManagedGauge
Numeric	gaugeConfig	RTRPublicGaugeConfig	RTRManagedGaugeConfig
Numeric	numeric	RTRPublicNumeric	RTRManagedNumeric
Numeric	largeNumeric	RTRPublicLargeNumeric	RTRManagedLargeNumeric
Numeric	numericConfig	RTRPublicNumericConfig	RTRManagedNumericConfig
Numeric	numericRange	RTRPublicNumericRange	RTRManagedNumericRange
String	string	RTRPublicString	RTRManagedString
String	stringConfig	RTRPublicStringConfig	RTRManagedStringConfig

Table 18: Supported Public Variable Types

RTRPublicBooleanConfig

RTRPublicBooleanConfig& **RTRPublicBooleanConfig::operator=** (RTRBOOL rhs)

Sets the value to rhs.

RTRPublicBoolean

RTRPublicBoolean& **RTRPublicBoolean::operator=** (RTRBOOL rhs)

Sets the current value to rhs and notifies client.

RTRPublicGaugeConfig

RTRPublicGaugeConfig& **RTRPublicGaugeConfig::operator=** (long rhs)

Sets the value to rhs.

RTRPublicGauge

RTRPublicGauge& **RTRPublicGauge::operator=** (long rhs)

Sets the value to rhs.

RTRPublicNumericConfig

RTRPublicNumericConfig& **RTRPublicNumericConfig::operator=** (long rhs)

Sets the active value to newValue.

RTRPublicNumericRange

RTRPublicNumericRange& **RTRPublicNumericRange::operator=** (long rhs)

Sets the current value to rhs.

RTRPublicNumeric

RTRPublicNumeric& **operator=** (long)

Sets the current value.

RTRPublicLargeNumeric

RTRPublicLargeNumeric& **operator=** (RTR_164)

Sets the current value.

5.4.1 Operations

RTRPublicBooleanConfig

void **RTRPublicBooleanConfig::internalSet**()

Sets the value to newValue.

void **RTRPublicBooleanConfig::internalClear**()

Sets the value to newValue.

void **RTRPublicBooleanConfig::setStore**()

Sets the store value to RTRTRUE. The new value is not persistent.

void **RTRPublicBooleanConfig::clearStore**()

Sets the store value to RTRFALSE. The new value is not persistent.

RTRPublicBoolean

void **RTRPublicBoolean::internalSet()**

Sets the value to RTRTRUE.

void **RTRPublicBoolean::internalClear()**

Sets the value to RTRFALSE.

RTRPublicCounter

virtual void **RTRPublicCounter::reset** ()

Resets this counter to 0.

RTRPublicGaugeConfig

void **RTRPublicGaugeConfig::operator+=** (long)

void **RTRPublicGaugeConfig::operator-=** (long)

RTRPublicGaugeConfig& **RTRPublicGaugeConfig::operator++**

RTRPublicGaugeConfig& **RTRPublicGaugeConfig::operator++** (int)

RTRPublicGaugeConfig& **RTRPublicGaugeConfig::operator--**

RTRPublicGaugeConfig& **RTRPublicGaugeConfig::operator--** (int)

void **RTRPublicGaugeConfig::set** (long newValue)

void **RTRPublicGaugeConfig::set** (long newMin, long newMax, long newValue)

Sets the minimum assignment value for this gauge to **newMin**, the maximum assignment value for this gauge to **newMax**, and the current value of this gauge to **newValue**.

void **RTRPublicGaugeConfig::internalSetRange** (long newMin, long newMax)

Sets the values for min and max. Clients are notified but the context is not notified.

void **RTRPublicGaugeConfig::setStore** (long newMin, long newMax)

RTRPublicGauge

void **RTRPublicGauge::operator+=** (long)

void **RTRPublicGauge::operator-=** (long)

RTRPublicGauge& **RTRPublicGauge::operator++** ()

RTRPublicGauge& **RTRPublicGauge::operator++** (int)

RTRPublicGauge& **RTRPublicGauge::operator--** ()

RTRPublicGauge& **RTRPublicGauge::operator--** (int)

void **RTRPublicGauge::set** (long newValue)

void **RTRPublicGauge::set** (long newMin, long newMax, long newValue)

Sets the minimum assignment value for this gauge to **newMin**, the maximum assignment value for this gauge to **newMax**, and the current value of this gauge to **newValue**.

virtual void **RTRPublicGauge::internalSetRange** (long newMin, long newMax)

Sets the minimum assignment for this gauge to **newMin**, the maximum assignment value for this gauge to **newMax**.

RTRPublicNumericConfig

void **RTRPublicNumericConfig::internalSet** (long newValue)

Set the active value to newValue. Does not notify the context MO.

void **RTRPublicNumericConfig::setStore** (long newStore)

Set the store value to newStore. The value is not persistent.

RTRPublicNumericRange

void **RTRPublicNumericRange::set** (long newValue)

Sets the current value of this parameter to newValue.

void **RTRPublicNumericRange::internalSet** (long newValue)

Sets the current value of this parameter to newValue.

void **RTRPublicNumericRange::set** (long newMin, long newMax, long newValue)

Sets the minimum assignment value for this parameter to **newMin**, the maximum assignment value for this parameter to **newMax**, and the current value of this parameter to **newValue**.

RTRPublicNumeric

void **operator+=** (long)

void **operator-=** (long)

RTRPublicNumeric& **operator++** ()

RTRPublicNumeric& **operator++** (int)

RTRPublicNumeric& **operator--** ()

RTRPublicNumeric& **operator--** (int)

void **set** (long)

RTRPublicLargeNumeric

void **operator+=** (RTR_164)

void **operator-=** (RTR_164)

RTRPublicLargeNumeric& **operator++** ()

RTRPublicLargeNumeric& **operator++** (int)

RTRPublicLargeNumeric& **operator--** ()

RTRPublicLargeNumeric& **operator--** (int)

void **set** (RTR_164)

5.5 Managed Object

Managed objects are manageable application components which are used as a “container” for managed variables.

As shown in Figure 36, managed objects within an application have relationships with other managed objects, i.e. objects may refer to other objects. These relationships form one or more trees (whose nodes are objects) and can be of interest to management entities. Managed objects contained by other objects are children. Managed objects not contained by other objects are the roots of object trees. Given the set of roots in an application, or system, all other objects can be reached by traversing the trees defined by those roots.

In the following diagram, MO stands for Managed Objects.

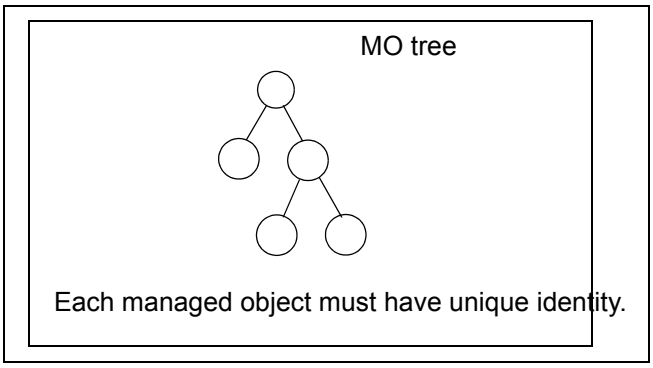


Figure 36. Managed Object Layout

5.5.1 Identity

RTRString& name()
The name of the managed object.

RTRObjectId& instanceId()
The instance identifier of the managed object.

RTRObjectId& classId()
The class identifier of the managed object.

5.5.1.1 Instance Identifier

Every instance of a managed object in a managed application will have a unique identifier. This identifier (instanceId) consists of two parts:

- the instanceId of its parent managed object and
- the name of the managed object.

The exception to this are the root managed objects which do not have parents so their identifier is the same as their name.

For example, the partial set of managed objects taken from a Refinitiv Real-Time Advanced Distribution Server (as shown in Figure 27) form a tree. The “node_A.1.ads” is a root managed object so its identifier is the same as its name. The identifier of its child is the concatenation of the parent’s identifier and the child name with a ‘.’ separator. Since this “dot” notation has a special purpose/meaning, the symbol ‘.’ is not used within the names of child managed objects.

These naming rules guarantee unique instance identifiers for each “tree”. The application developer should adopt a notation of uniquely named roots.

In order to have uniquely-named root objects, the following convention is used:

host_name.instance_id.executable_name

where:

- ***host_name*** is the machine name where the process is running on,
- ***instance_id*** is a numeric value (starting with 1) to uniquely identify multiple instances of the same executable running on the same machine and,
- ***executable_name*** is the name of the executable.

Some examples are:

node_a.1.ads
node_a.1.adh
node_b.1.adh
node_b.2.adh

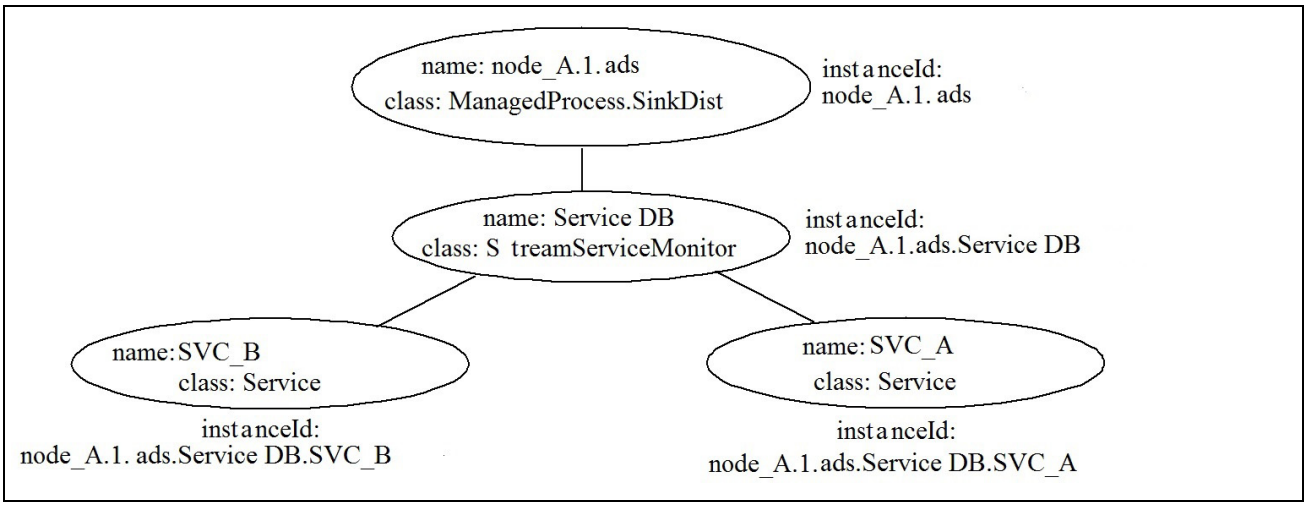


Figure 37. Refinitiv Real-Time Advanced Distribution Server Managed Object Tree

5.5.1.2 Class identifier

Managed objects also have a class identifier which identifies the type (semantics) of that object. In the previous example (Figure 37), one class identifier is “Service”. All class identifiers should be unique throughout the whole system. It is left up to the application developer to select the appropriate identifiers.

5.5.2 State

Each Managed Object uses a ‘state’ as a parameter to inform the registered managing object about its status.

RTRBOOL isInitializing()
Indicates if this managed object is in an initialization state.

RTRBOOL isNormal()
Indicates if this managed object is in a normal state.

RTRBOOL isRecovering()
Indicates if this managed object is recovering service automatically.

RTRBOOL isWaiting()
Indicates if this managed object is waiting for manual recovery.

RTRBOOL isInterupted()
Indicates if this managed object is in a service interrupted state.

RTRBOOL isDead()
Indicates if this managed object is in an (unrecoverable) error state.

char* text()
Provides textual explanation of the current managed object state.

5.5.3 State Attributes

RTRManagedObject::MOState state()
The state attribute of the managed object.

RTRManagedObject::MOState previousState()
The previous state attribute of the proxy managed object.

The managed object state attribute is defined by the value of **`state()`**. The five return values are defined as follows:

STATE	DESCRIPTION
Init	Initializing service
Normal	Normal service
ManualRecovery	In a service interrupted state, but attempting to recover normal service manually.
AutoRecovery	In a service interrupted state, but attempting to recover normal service automatically.
Dead	In an unrecoverable error state

Table 19: Managed Object State Attribute Return Values

State transitions are propagated to clients as events. The events triggered by state transitions are described as follows:

- Normal - The managed object cloning is complete and its managed variables can now be cloned.
- ManualRecovery - The managed object is in a service interrupted state and it is attempting to recover manually.
- AutoRecovery - The managed object is in a service interrupted state and it is attempting to recover automatically.
- Dead - The managed object is in an unrecoverable error state.

Not every managed object will support state transitions.

5.5.4 Access to Child Managed Objects

Managed objects provide both random and sequential access to the handles of their child managed objects.

5.5.4.1 Sequential Access to Child Managed Objects

RTRManagedObjectIterator childIterator()
An iterator which provides sequential access to all child managed objects contained by a managed object.

Sequential access is provided by means of an “iterator” class, an instance of which is associated with a particular managed object instance and provides access to each child managed object of the managed object in turn. The managed object iterator is defined by the class **`RTRManagedObjectIterator`**.

5.5.4.2 Random Access to Child Managed Objects

RTRBOOL hasChild(RTRString& name)
Tests the existence of a child managed object by querying the managed object for a specific child, given the child’s name.

RTRManagedObject* childByName(char*)
Provides random access to a child managed object, given the name.

- Code fragment example:

```
//RTRManagedObject object;
char* Child="ChildName";
const RTRManagedObject* childObject = object.childByName(Child );
if ( childObject == (RTRManagedObject *) NULL )
{
    // Child is not present.
```

```
}
```

- **`hasChild()`** method example:

```
//const RTRManagedObject* object;
RTRString Child("ChildName");
if ( !object->hasChild( Child ) )
{
    // Child is not present.
}
```

5.5.5 Access to Contained Managed Variables

Managed objects provide both sequential and random access to the handles of their contained managed variables.

5.5.5.1 Sequential Access to Managed Variables

RTRManagedVariableIterator variableIterator()
An iterator which provides sequential access to all managed variable handles contained by a managed object.

Sequential access is provided by means of an “iterator” class, an instance of which is associated with a particular managed object instance and provides access, in turn, to each contained managed variable of the managed object. The managed variable iterator is defined by the class **`RTRManagedVariableIterator`**.

5.5.5.2 Random Access to Managed Variables

RTRBOOL hasVariable(RTRString& name)
Tests the existence of a managed variable by querying the managed object for a specific managed variable, given the managed variable’s name.

RTRManagedVariable* variableByName(char*)
Provides random access to a managed variable, given the name.

There are also eleven more random access methods that correspond to each of the eleven possible managed variable types:

RTRManagedBoolean* booleanByName(char*),

RTRManagedBooleanConfig* booleanConfigByName(char*),

RTRManagedCounter* counterByName(char*),

RTRManagedGauge* gaugeByName(char*),

RTRManagedGaugeConfig* gaugeConfigByName(char*),

RTRManagedNumeric* numericByName(char*),

RTRManagedLargeNumeric* largeNumericByName(char*),

RTRManagedNumericConfig* numericConfigByName(char*),

RTRManagedNumericRange* numericRangeByName(char*),

RTRManagedString* stringByName(char*), and

RTRManagedStringConfig* stringConfigByName(char*).

5.5.6 Managed Object Client Management

void addClient (RTRManagedObjectClient&)
Register a client with an individual managed object so that the given client will receive all events subsequently generated by that managed object.

void dropClient (RTRManagedObjectClient&)
Un-register a client with an individual managed object so that the given client will no longer receive events generated by that managed object.

RTRBOOL hasClient (RTRManagedObjectClient&)
Indicates whether a given client is currently registered to receive events from a particular managed object. Allows application developer to verify precondition of corresponding **`addClient()`** and **`dropClient()`** methods.

5.5.7 Managed Object Clients

To use the following methods, managed object clients should be descendants of (inherit from) **`RTRManagedObjectClient`**. As clients, they can register to receive events from one or more instances of **`RTRManagedObject`**. Once registered with a managed object, they will receive events generated by that managed object instance. Events are propagated by means of class member functions; i.e., when a managed object needs to “generate an event”, it will invoke the appropriate member function of each client currently registered.

Events that can be generated by instances of **`RTRManagedObject`** have a corresponding processing method in **`RTRManagedObjectClient`**. In the case of **`RTRManagedObjectClient`** there is no default implementation (behavior) defined. Some processing methods have been declared as pure virtual. So the application developer must implement all ten member functions, even if the client is not interested in certain events.

virtual void processObjectInfo (RTRManagedObject&)
The given managed object has changed its informational text.

virtual void processObjectInService (RTRManagedObject&)
The given managed object is now in a normal service state.

virtual void processObjectRecovering (RTRManagedObject&)
The given managed object is in a service interrupted state and is attempting to recover normal service automatically.

virtual void processObjectWaiting (RTRManagedObject&)
The given managed object is in a service interrupted state and is waiting for manual intervention to restore normal service automatically.

virtual void processObjectDead (RTRManagedObject&)
The given managed object has entered an unrecoverable error state.

virtual void processObjectDeleted (RTRManagedObject&) = 0
The given managed object has been deleted by the managed application. This represents an unrecoverable condition. The client should un-register from the managed object (using **`dropClient()`**) and ensure that no further references are made to that managed object. Managed objects may come and go at the discretion of the managed application; this does not necessarily indicate a problem with the managed application.

virtual void processChildAdded (
const RTRManagedObject&,
const RTRManagedObject&)
The given managed object has a new child.

virtual void processChildRemoved (
const RTRManagedObject&,
const RTRManagedObject&) = 0
The given managed object has a child removed.

virtual void processVariableAdded (
const RTRManagedObject&
const RTRManagedVariable&)
The given managed object has a new managed variable.

virtual void processVariableRemoved (
const RTRManagedObject&,
const RTRManagedVariable&)
The given managed object has a managed variable removed.

5.5.8 Operations

virtual void RTRManagedObject::lock ()

virtual void RTRManagedObject::unlock ()

5.5.9 Event Processing

virtual void RTRManagedObject::processParameterChange (RTRManagedVariable &)
One of the parameter variables contained by this object has been changed.

virtual void RTRManagedObject::processConfigChange (RTRManagedVariable &)
One of the configuration variables contained by this object has been changed.

5.5.10 Class Diagram (Simplified)

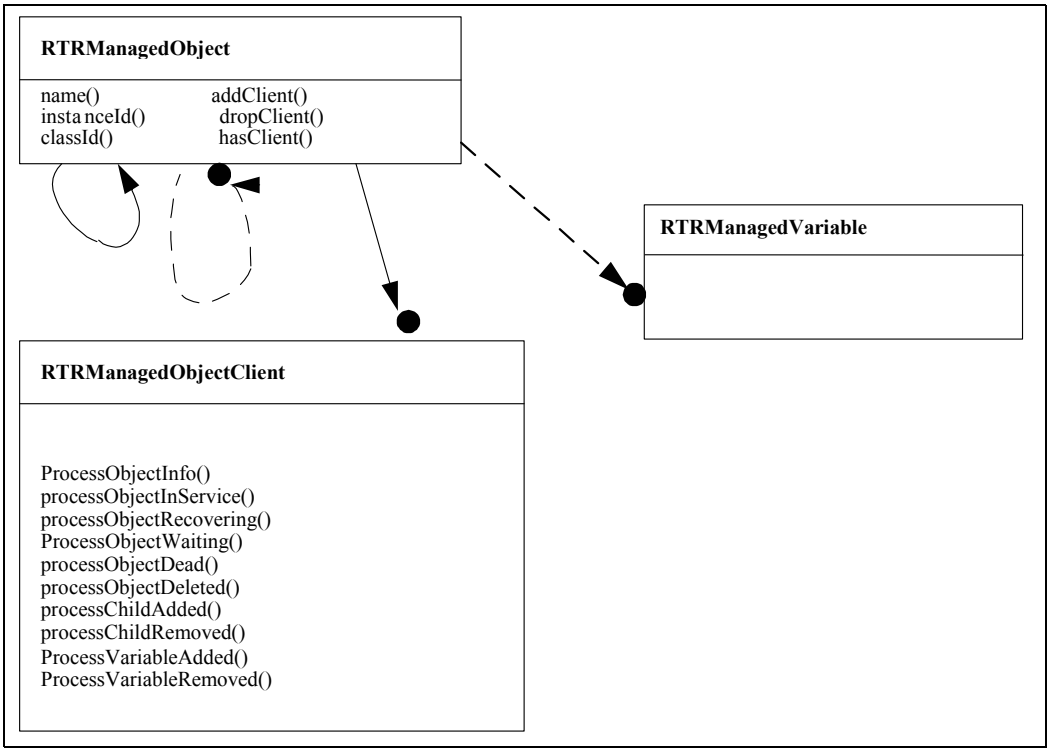


Figure 38. Managed Object - Simplified Class Diagram

5.6 Public Object

RTRPublicObject inherits **RTRManagedObject**. When application components wish to be managed or become 'public', they have to be descendants of **RTRPublicObject**. They in turn may instantiate other public objects which will be their children in the managed object tree.

RTRManagedObject has an enumerated type that represents the current state of the Managed Object :- Init, Normal, ManualRecovery, AutoRecovery and Dead. Invoking an appropriate function alters the state of the MO (for example **RTRPublicObject::markDead** alters the state to Dead). The state is used to convey the current state of a logical component of the publisher application. Not all components of the publisher application will need this ability to change states and therefore should only use the Normal state.

void markNormal(char*)
Marks this public object as normal and notify its clients.

void markRecovering(char*)
Marks this public object as recovering and notify its clients.

void markWaiting(char*)
Marks this public object as waiting and notify its clients.

void markDead(char*)
Marks this public object as dead and notify its clients.

void indicateInfo(char*)
Notifies its clients of change in text format.

5.7 Managed Process

A managed process class is a utility class to provide a minimum set of variables related to process state and to publish process statistics into shared memory segments. The following is a list of process variables that a managing application is able to access in the shared memory segments.

- The executable name and path of the managed application.
- The process ID, user ID, group ID and instance ID of the managed application.
- Description and version of the managed application.
- The time stamp and state of the managed application.

A managed process uses the following constructors:

```
RTRManagedProcess(
                    const RTRObjectId& appId,
                    const char *execName,
                    const char *subClassName,
                    const char *description,
                    const char *version,
                    MOState startState=RTRManagedObject::Normal )
RTRManagedProcess(
                    const RTRObjectId& appId,
                    const char *execName,
                    const char *description,
                    const char *version,
                    MOState startState=RTRManagedObject::Normal )
RTRManagedProcess(
                    int argc,
                    char **argv,
                    const char *subClassName,
                    const RTRApplicationId& appId,
                    const RTRString version )
```

5.8 Server Shared Memory Root

`ServerSharedMemoryRoot` is the encapsulation of the server side of a server/client shared memory relationship. An instance of **`ServerSharedMemoryRoot`** is constructed with a key and will then attempt to allocate the shared memory using that key. If memory already exists with that key then the memory server will examine that memory to determine whether or not it can safely be reinitialized. If the memory header matches that which the server would create (version, size, etc) and the memory appears to no longer be in use, then the server will re-initialize the existing memory. If the memory could not be reused then the class will go into an error state and the application or user will have to correct the problem (kill the other publisher that is using the same shared memory key) and restart the publisher.

Once memory has been successfully allocated or re-initialized then the server will implement a handshaking scheme with any clients that attach to the memory segment.

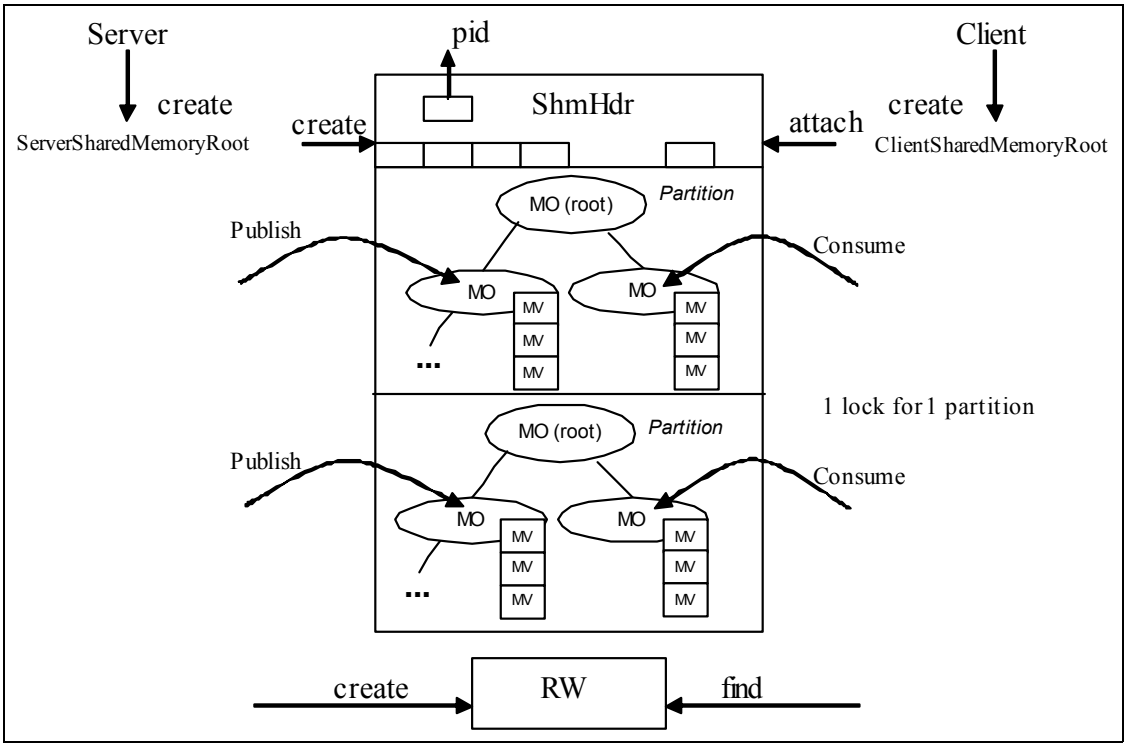


Figure 39. Structure of Shared Memory

Legend for Figure 39:

- MO: Managed object
- MV: Managed Variable
- pid: Process ID

5.8.1 State

virtual RTRBOOL error() const
Indicates if the memory segment is in an error state.

5.8.2 Example

Server shared memory root will keep determining the process ID in the shared memory header at every specified time interval. If the process ID belongs to that server shared memory root, the time stamp and the connection in the shared memory header will be updated. Otherwise, an error message will be logged.

```
#include "rtr/sharedmem.h"

RTRApplicationId appId((const RTRString&)inst, "Example");
int shmkey = 99, semkey = 99;
int maxClients = 5;
int shmsize = 30000;

RTRServerSharedMemoryRoot shm(appId, "sharedmem", shmkey, semkey, maxClients, shmsize, 2);
//maxClients is the maximum number of clients which is allowed to access Shared
//memory segment

if (shm.error(0)) return -1;
```

5.9 Shared Memory Managed Object Server Memory Pool

A shared memory managed object server memory pool is a class that uses shared memory to allocate storage for managed objects and variables allocated by the managed application (server).

The object structure defined by the managed application is reproduced in shared memory and used by clients to clone managed objects and variables in the client process.

The memory management scheme is pseudo dynamic. Shared memory storage is allocated dynamically (based on demand from the application) as storage for either an object or some type of variable. Once allocated for a particular purpose, it will never be freed for general purpose use, When storage is freed it is placed on a free list with other storage of the same type (same size and layout).

In general, modification of shared memory must be synchronized with access with other processes. Allocated objects and variables cannot be attached to or removed from the existing shared memory layout without acquiring a lock. For efficiency, allocation is done on demand but attach and remove operations are done in batches. Allocation does not need to be synchronized because there is only one reader and writer of the free lists, the server. Lists of objects and variables which need to be attached or removed are maintained by the server and serviced at regular intervals. This means that the server does not have to obtain a semaphore lock every time an object is allocated or deleted.

If no memory (or no memory of the requested type) is available the sever returns a null pointer to the caller (usually an instance of an object or variable implementation class). It is assumed that the caller will detect this and allocate memory from the heap as necessary.

The server also detects and processes parameter operations requested by clients.

5.9.1 State

RTRBOOL error() const
Indicates if the shared memory managed object server memory pool is in an error state.

5.9.2 Attributes

const RTRString& text()
Provides textual explanation of the shared memory managed object server memory pool state (especially if in the error state).

RTRServerPartition& partition()
The partition of the server partition.

5.9.3 Operation

void useStats(RTRSharedMemoryStats *)
Provides shared memory statistics.

5.9.4 Example

If a fatal error is encountered during initialization, **`error()`** will be true and **`text()`** contain an explanation of the problem.

Typically, a shared memory root is always associated with a shared memory pool. Once the pool is created, it will periodically check whether there are any changes in the root (e.g., a new managed object pool (MO) is added into the tree). If that is the case, the pool will refresh the memory in order to be in sync with the root.

```
#include "rtr/shmmosvr.h"

RTRApplicationId appId((const RTRString&)inst, "Example");
int shmkey = 99, semkey = 99;
int shmmaxClients = 5;
int shmsize = 30000;
int poolmaxClients = 3;
int poolSize = 10000;

RTRServerSharedMemoryRoot shm(appId, "sharedmem", shmkey, semkey, shmmaxClients, shmsize, 2);

if (shm.error(0)) return -1;

RTRShmMOServerMemPool moPool(shm, *(shm.semaphoreSet()), poolMaxClients ,poolsize);
// poolMaxClients is the maximum number of client that can be attached to Shared
//memory.

if (moPool.error(0))
{
cout << moPool.text();
return -1;
}
```

5.9.5 Class Diagram (Simplified)

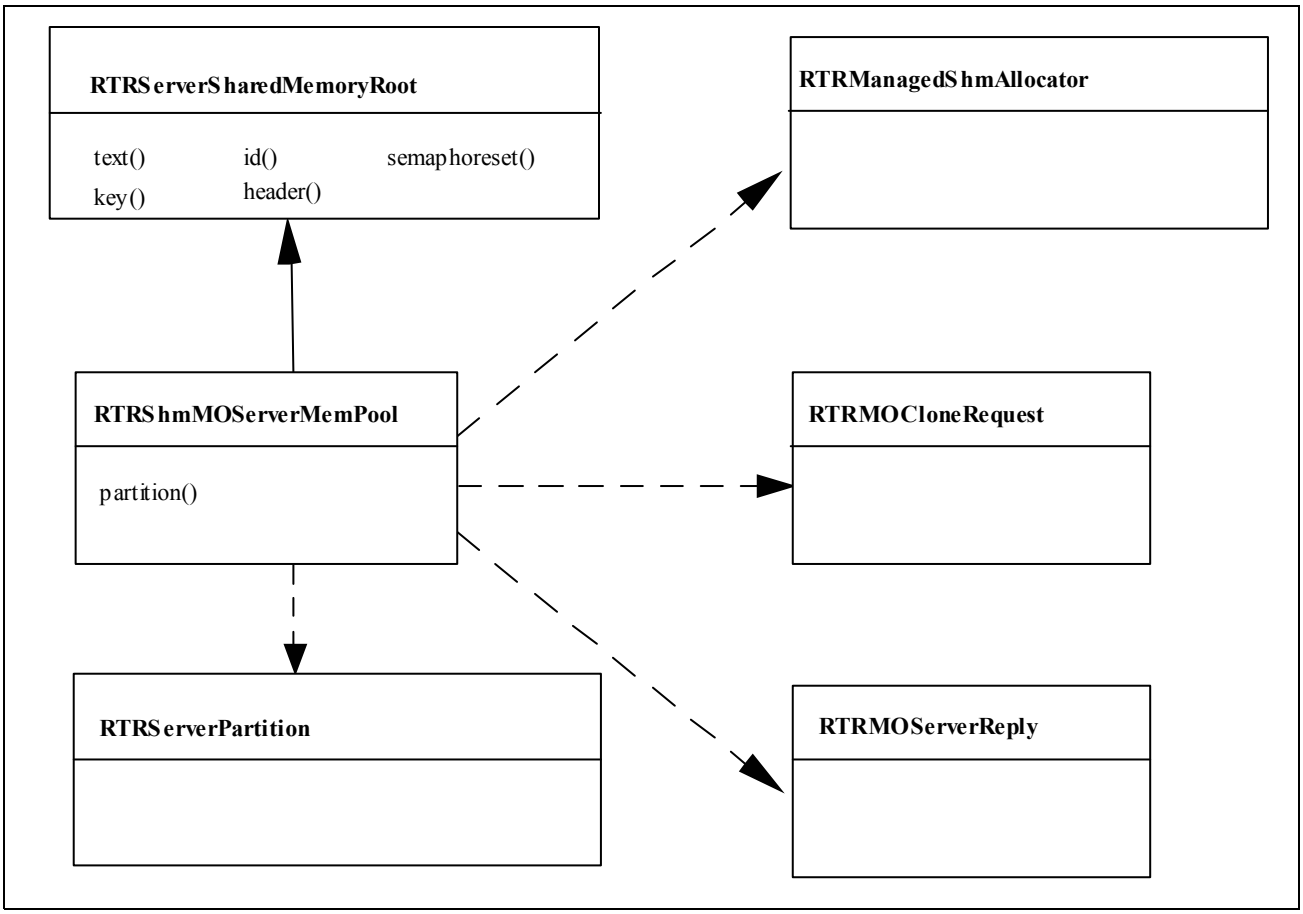


Figure 40. Shared Memory Managed Object Server Memory Pool – Simplified Class Diagram

5.10 Shared Memory Managed Object Server

A shared memory managed object server class is a utility class that utilizes a portion of the shared memory segment. This class also provides the functionality to manage the instance of the shared memory managed object server memory pool.

5.10.1 Constructors

RTRShmMOServer(
const RTRObjectId& classId,
const RTRObjectId& context,
const char *name,
RTRShmServer& shm,
RTRBOOL enable=RTRTRUE)

Creates shared memory managed object server by using RTRConfigDb:configDb for configuration.

RTRShmMOServer(
const RTRObjectId& context,
const char *name,
RTRShmServer& shm,
unsigned long sharedMemorySize,
RTRBOOL enable=RTRTRUE,
RTRBOOL enableStats=RTRFALSE)

Creates shared memory managed object server by using the constructor arguments for configuring the managed object server.

5.10.2 States

RTRBOOL enabled() const
Indicates if the shared memory managed object server memory pool has been instantiated.

RTRBOOL error() const
Indicates if the shared memory managed object server or shared memory managed object memory pool is in an error state.

5.10.3 Identity

RTRObjectId& instanceId() const
The instance identifier of the shared memory managed object server.

5.10.4 Attributes

unsigned long sharedMemorySize() const
Amount of shared memory allocated for storing managed objects.

int maxClients() const
The maximum number of clients permitted to access the shared memory segment.

const RTRString& text() const
Provides textual explanation of the shared memory managed object server state (especially if in the error state).

RTRShmMOServerMemPool *managedObjectServer() const
The managed object server in the shared memory managed object server memory pool.

RTRSharedMemoryStats *memoryStats() const
Provides memory statistic in shared memory segment.

5.10.5 Operations

void enable()
Creates an instance of **`RTRShmMOServerMemPool`**.

void disable()
Destroys an instance of **`RTRShmMOServerMemPool`**.

5.11 Shared Memory Server

A shared memory managed object server class is a utility class that is based on the server shared memory root class. This class also provides shared memory statistics from server shared memory root class.

5.11.1 Constructors

RTRShmServer(
const RTRObjectId& classId,
const RTRObjectId& context,
const char *name,
RTRBOOL enable=RTRTRUE)
Creates shared memory server by using RTRConfigDb:configDb for configuration.

RTRShmServer(
const RTRObjectId& classId,
const char name,
unsigned long sharedMemorySize,
const char *sharedMemoryKey,
const char *semaphoreKey,
RTRBOOL enable=RTRTRUE,
int maxClients=10,
int numSemaphores=8)
Creates shared memory server by using arguments provided.

5.11.2 States

RTRBOOL enabled() const
Indicates if the segment of shared memory has been created.

RTRBOOL error() const
Indicates if the shared memory server or server shared memory root is in an error state.

5.11.3 Identity

RTRObjectId& instanceId() const
The instance identifier of the shared memory server.

5.11.4 Attributes

const RTRString& sharedMemoryKey() const
The shared memory key

const RTRString& semaphoreKey() const
The semaphore key

unsigned long sharedMemorySize() const
The desired size of the shared memory segment (in bytes).

int maxClients() const
The desired maximum number of allowed clients.

int numberofSemaphores () const
The number of semaphores.

const RTRString& text() const
Provides textual explanation of the shared memory managed object server state (especially if in the error state).

RTRServerSharedMemoryRoot *sharedMemory() const
A reference to the **`RTRServerSharedMemoryRoot`**.

5.11.5 Operations

void enable()
Creates an **`RTRServerSharedMemoryRoot`**.

void disable()
Destroys an **`RTRServerSharedMemoryRoot`**.

6 Additional Topics

This chapter discusses the following topics:

- Thread safety of Refinitiv Management Classes 2.X (see Section 6.1)
- Event loops (see Section 6.2)
- I/O events (see Section 6.3)
- Timer events (see Section 6.4)
- Example class using an event loop, I/O and timing events (see Section 6.5)
- Smart pointers (see Section 6.6)
- Performance constraints (see Section 6.7)

6.1 Thread Safety of Refinitiv Management Classes 2.X

6.1.1 Thread Safe Classes

Refinitiv Management Classes has mechanisms to make Refinitiv Management Classes 2.X C++ applications thread safe.

The classes that are thread safe are those that are descendants of the **`RTRLockableObj`** class. They need to be locked externally before their public methods are called in a multi-threaded application. Here is an example of how to use this kind of class.

The class **`RTRProxyManagedObject`** is made thread safe via inheritance from **`RTRLockableObj`**. In a multi-threaded application, if an instance of **`RTRProxyManagedObject`** is accessible from different threads, its public methods should be called like this:

In one thread:

```
// RTRProxyManagedObjectPtr pmo1;
// assume pmo1 is initialized to point to an instance of RTRProxyManagedObject
pmo1->lock();
RTRString objectName = pmo1->name();
pmo1->unlock();
```

In another thread:

```
// RTRProxyManagedObjectPtr pmo2;
// assume pmo2 is initialized to point to an instance of RTRProxyManagedObject
pmo2->lock();
RTRProxyManagedObjectHandleIterator Iterator = pmo2->childHandles();
pmo2>unlock();
```

For the list of classes that are descendants of **`RTRLockableObj`**, refer to the Refinitiv Management Classes Class Reference manual.

6.1.2 Guidelines for Multi-threaded Applications to Use Refinitiv Management Classes 2.X Safely

1. Lock an object when accessing it in multiple threads.

   All classes that inherit from **`RTRLockableObj`** can be used in a multi-threaded application (usually by locking the instance before accessing it).

2. Classes that inherit from **`RTRMTGCObj`** can be referenced from multiple threads, i.e., one thread can access an instance without worrying if it is being deleted in other threads. In case of non-smart pointer classes, it is the application's responsibility to ensure the MT-safety. For example, this can be achieved by locking the container of the instance, so the thread who wishes to delete the instance will be blocked.

3. The virtual function **`lock()`**/**`unlock()`** of **`RTRLockableObj`** can be overridden to reflect the application's locking granularity. The default implementation is using the **`RTRReentMutex`** class. If locking class A always implies the need to lock class B, then A's **`lock()`**/**`unlock()`** can be implemented using B's. For example, ProxyManagedVariable's **`lock()`** can use its context ProxyManagedObject's **`lock()`**.

4. Avoid locking multiple objects at one time as much as possible.

   If more than one thread wants to hold one lock and wait for the other lock, and if they are waiting for each other, deadlock will happen. When locking multiple objects is necessary, follow a specific order. In Refinitiv Management Classes, the recommended order is "lock the container first, then the contained".

5. The client-call-back functions for lockable objects can assume that the caller (server) is locked.

   Usually the client-call-back functions are called in a server thread right after the state/value is changed there, thus the server should have already been locked if it needs to be so.

6. Don't block the client call-back functions.

   All Refinitiv Management Classes events occur in the notifier thread and propagate to clients via the call-backs (**`processSomeEvent()`**). When programming these functions, keep in mind that they are called in the "main/RMC" thread, and blocking any of them will block all other Refinitiv Management Classes activities.

6.2 Event Loops

Refinitiv Management Classes defines the abstract and implementation classes for an event dispatching loop. The Refinitiv Management Classes abstraction for an event loop is represented by the class **`RTREventNotifier`**. The purpose of the notifier is to allow application components to use system resources (I/O, timers) in a cooperative way, without being dependent on any particular implementation of a “main loop”. Application designers may choose the event loop implementation which best suits their requirements.

There can be only one event loop (**`RTREventNotifier`**) in an application. Refinitiv Management Classes accesses the notifier implementation via a static data element of type **`RTREventNotifierInit`**. This static data element is made available to an application component by including the file **rtr/rtrnotif.h**. The actual implementation used depends on how the application is constructed (what **`main()`** looks like) and how the application is linked. Usually, the **`main()`** is “notifier specific”. This is because, in most applications, **`main()`** must invoke the instructions which cause the application to enter the loop and such instructions couple the main with a particular implementation of notifier. For example, the applications described in this manual use the **`RTRSelectNotifier`**. Example source files that define **`main()`** include the header file **rtr/selectni.h** and invoke **`RTRSelectNotifier::run()`** to enable the control loop.

6.3 I/O Events

The base class for components which need to be informed of I/O events is **`RTRIOClient`**. This class provides event handling methods for each of three types of I/O event available on file descriptors.

- A Read event indicates to interested clients that there is data to be read on a given file descriptor.
- A Write event indicates that a given file descriptor is now available for output.
- An Exception event indicates that there is an exception condition on a given file descriptor.

Descendants of **`RTRIOClient`** may register for each type of I/O event with an instance **`RTREventNotifier`**. Only one client may register for a particular event on any one file descriptor.

6.4 Timer Events

RTRBOOL active()
Is the timer still activated?

void activate()
Activate timer

void deactivate()
Deactivate timer.

void setTimerOffset(long s, short m)
Set timer offset to s seconds and m milliseconds.

void processTimerEvent()=0
Descendants of RTRTimerCmd must provide specific behavior when the timer expires.

Timer events allow application components to give up control of the thread of execution in the program until a specified amount of time has elapsed. **`RTRTimerCmd`** is the base class that allows application components to initiate and receive timer events. This base class provides methods used to set the timer interval (offset) and activate or deactivate the timer. Descendants of **`RTRTimerCmd`** provide an implementation of the **`processTimerEvent()`** method. This method is called when the timer expires.

NOTE: When a managing application tries to access a segment of shared memory whose header is not up-to-date, it receives an error message “Header is old.” To prevent this, managing applications must not attempt to access a shared memory segment with a stale header. For example,

- managing applications should wait for the shared memory header to be completely initialized before accessing it.
- managed applications should not spend too much time in **`processTimerEvent()`**.

Once the message “Header is Old” is encountered, it can be solved by restarting the managed application that uses the relevant shared memory segment.

6.5 Example - Event Loops, I/O and Timing

This section presents an example class that utilizes an event loop, I/O and timing events. **`IOTimerClient`** is a descendant of both **`RTRIOClient`** and **`RTRTimerCmd`**. The purpose of this class is to read data from the standard input device. The data is interpreted as an interval, in seconds, which is used as the interval to be timed. When new data is received, the current timer event, if any, is canceled and the timer event is re-activated with the new interval. If the timer expires, a message is printed to the standard output device.

```
// IOTimerClient is a simple class, which uses both I/O and timing events.
// It reads the standard input for line of data. The data is converted
// to an integer value and is used to set a timer. When the timer expires,
// the message "HELLO WORLD, AGAIN" is printed to the standard output.
// Existing timers are canceled prior to installation of new timers.
// A timer value of 0 causes this class to stop waiting for input.

#include "rtr/timercmd.h"
#include "rtr/ioclient.h"
class IOTimerClient :
   public RTRTimerCmd,
   public RTRIOClient
{
public:
// Constructor
IOTimerClient(char *fnm)
{
   fd = open(fnm, O_RDWR);
   RTREventNotifierInit::notifier->addReadClient(*this, fd);
   cout << "Enter time:" << flush;
}
```

```
// Destructor
~IOTimerClient()
{
    RTREventNotifierInit::notifier->dropReadClient(fd);
}

// Event processing - from RTRIOClient
void processIORead(int fd)
{
    char buf[100];
    int len = read(fd, buf, 10);
    buf[len] = '\0';

    int s = atoi(buf);
    if (active())
    {
        cout << "Canceling current event" << endl;
        deactivate();
    }
    if ( s > 0 )
    {
        cout << "Adding event for " << s << " seconds" << endl;
        setTimerOffset(s, 0);
        activate();
    }
    else
    {
        if ( active() )
            deactivate();
        RTREventNotifierInit::notifier->dropReadClient(fd);
    }
}

// Event processing - from RTRTimerCmd
    void processTimerEvent()
{
    cout << "HELLO WORLD, AGAIN" << endl << "Enter time:" << flush;
}

protected:
// Implementation attributes
    int fd;
};

//
// This application creates a single instance of a IOTimerClient.
// It uses the RTRSelectNotifier as the main loop implementation.
//

#include "rtr/selectni.h"// Defines RTRSelectNotifier

void main()
{
    IOTimerClient client("/dev/tty");
    RTRSelectNotifier::run();
}
```

6.6 Smart Pointers

Smart pointers are a C++ technique, where the “smart pointers” are objects that act like pointers and in addition perform some action whenever a resource is accessed through them. This additional action is keeping a reference count, i.e., how many objects/classes hold pointers to the resource. This enables the application developer to use automatic garbage collection.

In Refinitiv Management Classes, a smart pointer to a proxy managed object (**`RTRProxyManagedObject`**) would be **`RTRProxyManagedObjectPtr`**. This is a typedef already provided with the Refinitiv Management Classes 1.0 Library. There is no need to delete the smart pointer when you are done using it.

A common mistake is not to use a smart pointer when there is a need to delete an object. Before losing reference to a smart pointer, it is a good practice to set it to NULL.

6.7 Performance Constraints

Developers of managing applications must design their applications such that they will not adversely affect the performance (node) of the system being monitored. Although the Refinitiv Management Classes does not limit the number of components and variables that a managing application can monitor, it is recommended, and expected, that managing applications restrict the number of components and variables that are monitored at any given time.

Currently the recommendation is to not exceed monitoring more than 500 variables at any given time. This means, for example, 5 variables from 100 managed applications or 100 variables from 5 managed applications. This guideline also depends on the variables that are being monitored. Monitoring 500 variables that update often will have more of a performance impact than 500 variables that seldom change their value.

Appendix A Glossary

This sections provides definitions for a number of key terms used in this manual.

Abstract Class	A class whose primary purpose is to define an interface. An abstract class has one or more functions that are declared, but not defined. Implementation classes inherit from abstract classes and provide definitions for the undefined functions of the abstract class. See also Pure Virtual Function.
Boolean Variable	A managed variable whose value is of type boolean (i.e true or false).
BooleanConfig Variable	A boolean variable whose value can also be obtained from a configuration file.
Class Diagram	A diagram that depicts classes, their internal structure and operations, and the static relationships between them.
Class Identifier	Uniquely identifies the type (semantics) of a managed object. In principle, all managed objects with a given class identifier will provide the same set of managed variables.
Client	A managing application that registered interest in changes in managed variables or managed objects.
Consumer Application	See Managing Application.
Counter Variable	A managed variable whose value can be incremented or reset (to 0).
Gauge Variable	A managed variable whose value falls between a given maximum and minimum and it also provides low/high water marks. The low/high watermarks indicate the lowest/highest values assumed by a gauge since its creation. The value and the minimum and maximum values may all be changed.
GaugeConfig Variable	A gauge variable whose value can also be obtained from a configuration file.
Implementation Class	A class that is an implementation of an abstract class provides definitions for the ancestor‘s abstract functions. Implementation classes can be instantiated while abstract classes provide one or more abstract functions (pure virtual functions, in C++ terminology) and cannot be instantiated.
Instance	An instance of a class is a physical manifestation of a class type also known as an object. An instance is created when a class in instantiated.
Instance Identifier	A unique identifier for an instance of a managed object. The Instance identifier consists of the Instance identifier of the parent managed object (if any) and the name of the managed object.
Instantiate, Instantiation	Instantiation occurs when an instance of a class type is created. Instantiation is the act of creating an instance of a particular class type.
Large Numeric Variable	A managed variable whose numeric value (64 bits) cannot be changed.
Managed Application	An application that creates and manages (i.e. publishes) a collection of managed objects. Also referred to as a Publisher Application.
Managed Object	Application components that can be accessed and managed by external management entities. A managed object provides some number of managed variables. The managed objects within an application have relationships with other managed objects. These relationships form one or more trees. Managed objects contained by other objects are children. Managed objects not contained by other objects are the roots.
Managed Variable	Variables of interest to external management entities. Managed variables are contained within managed objects. Managed variables may change over time and will propagate changes to registered clients. There are eleven types of managed variables (boolean, booleanConfig, counter, numeric, largeNumeric, numericConfig, numericRange, gauge, gaugeConfig, string, stringConfig).

Managing Application	An application that is interested in managed objects and managed variables from one or more managed applications. Also referred to as a Consumer Application.
Numeric Variable	A managed variable whose numeric value cannot be changed.
NumericConfig Variable	A numeric variable whose value can also be obtained from a configuration file.
NumericRange Variable	A managed variable whose value is within a minimum/maximum range. Both the value and the min/max range can be changed.
Proxy	Cloned.
Proxy Managed Object	A proxy representation of a managed object.
Proxy Managed Object Class Directory	A directory of all proxy managed object handles of particular type as published by each proxy managed object server in a specified proxy managed object server pool.
Proxy Managed Object Class Directory Factory	A factory for instantiating a proxy managed object class directory for a given class identifier.
Proxy Managed Object Handle	Uniquely identifies a managed object and is used to request a particular proxy managed object.
Proxy Managed Object Pool	A set of proxy managed objects that match the contents of a specified proxy managed object class directory.
Proxy Managed Object Server	Provides access (through proxy managed object handles) to all the root managed objects. Also provides the cloning service for managed objects (using the corresponding proxy managed object handle).
Proxy Managed Object Server Pool	Creates a proxy managed object server for each specified shared memory key.
Proxy Managed Variable	A proxy representation of a managed variable.
Proxy Managed Variable Threshold	Monitors a proxy managed numeric, gauge, guageConfig, numericRange, or numericConfig variable by notifying its client when the variable value is within certain threshold values.
Publisher Application	See Managed Application.
Pure Virtual Function	In the C++ language, a pure virtual function has a call signature, but no implementation. A virtual base class, or abstract class, contains one or more pure virtual functions. Pure virtual functions are filled in by descendants of the abstract class, also known as implementation classes.
Shared Memory	Shared memory provides a method for two or more applications to share a segment of virtual memory and use it as if it were actually part of each application.
Shared Memory Segment	Section of the shared memory identified by the shared memory key. In Refinitiv Management Classes 2.0, a proxy managed object server is created for each shared memory segment.
String Variable	A managed variable whose value is a string that can be changed.
StringConfig Variable	A string variable whose value can also be obtained from a configuration file.

Document ID: RMC220UM.200
Date of issue: December 2020