VMS Software

# VSI OpenVMS

# VSI Reliable Transaction Router C++ Foundation Classes

**Operating System and Version:** VSI OpenVMS IA-64 Version 8.4-1H1 or higher
VSI OpenVMS Alpha Version 8.4-2L1 or higher

**Software Version:** VSI Reliable Transaction Router Version 5.1

**VMS Software, Inc. (VSI)**
**Boston, Massachusetts, USA**

# VSI Reliable Transaction Router C++ Foundation Classes

VMS Software

# Table of Contents

# Preface

This document describes the C++ interface for Reliable Transaction Router (RTR) in which RTR concepts are represented as individual classes. The flexibility and extensibility of these classes enable existing as well as new applications to use features that were otherwise unavailable. This application programming interface (API) is backward-compatible with existing RTR applications.

## 1. About VSI

VMS Software, Inc. (VSI) is an independent software company licensed by Hewlett Packard Enterprise to develop and support the OpenVMS operating system.

## 2. Programming Requirements

Programs using the C++ API require the following files:

### NT platform:

| File | Description |
|---|---|
| rtrapi.h | Header file defining RTR classes |
| rtrapicpp.lib | Library file that applications link against to make use of the RTRAPI |
| rtrapicpp.dll | Library used by all RTR applications using RTR V4.0 |

### UNIX platforms:

| File | Description |
|---|---|
| rtrapi.h | Header file defining RTR classes |
| rtrapicpp.so | File used by all RTR applications using RTR V3.2 or later |

### OpenVMS platforms:

| File | Description |
|---|---|
| rtrapi.h | Header file defining RTR classes |
| rtrapicpp_shr.exe | File used by all RTR applications using RTR V3.2 or later |

## 3. Document Structure

- Chapter 1, C++ API Concepts

  Overview of the C++ Foundation classes and introduction to RTR application concepts. Includes C++ Foundation Class concepts and terminology and introduces RTR transactional messaging concepts for C++ API client and server applications.

There is also a section on using the C++ API with existing applications. This section describes how to use C++ Foundation classes with legacy applications.

- Chapter 2, Design and Implementation

  Covers client and server application design and implementation. Provides foundation class overloading examples and design concepts.

- Chapter 3, Application Classes

  Lists all foundation class application classes and their inherited methods. Includes separate sections for server, client, and common data classes.

- Chapter 4, Management Classes

  Lists all C++ API management classes and their inherited methods.

- Chapter 5, Sample Application Tutorial

  Provides a walkthrough of the basics of RTR with the C++ API book processing sample application included in the examples directory of the RTR kit.

- Appendices

  There are three appendices that offer class design diagrams, sample C++ API client and server application code examples, and a multithreading example.

## 4. Related Documentation

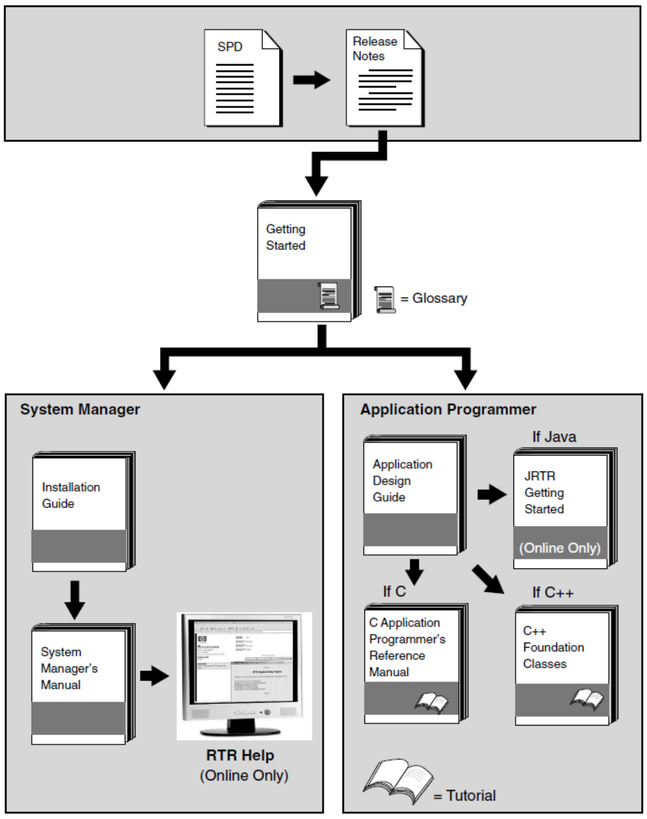
Additional resources in the RTR documentation kit include:

| **Document** | **Description** |
|---|---|
| *VSI Reliable Transaction Router Getting Started* | Provides an overview of RTR technology concepts and solutions. |
| *VSI Reliable Transaction Router Application Design Guide* | Provides design guidelines for implementing RTR client and server applications. |
| *VSI Reliable Transaction Router C Application Programmer's Reference Manual* | Explains how to design and code RTR applications; contains full descriptions of the RTR API calls for the C programming language. |
| *VSI Reliable Transaction Router System Manager's Manual* | Describes how to configure, manage, and monitor RTR. |
| *Reliable Transaction Router Migration Guide* | Explains how to migrate from RTR Version 2 to RTR Version 3. |
| *VSI Reliable Transaction Router Installation Guide* | Describes how to install RTR. |
| *VSI Reliable Transaction Router Release Notes* | Describes new features, changes, and known restrictions for RTR. |

## 5. Reading Path

The reading path to follow when using the Reliable Transaction Router information set is shown in *Figure 1, "RTR Reading Path"*.

**Figure 1. RTR Reading Path**



## 6. VSI Encourages Your Comments

You may send comments or suggestions regarding this manual or any VSI document by sending electronic mail to the following Internet address: `<docinfo@vmssoftware.com>`. Users who have VSI OpenVMS support contracts through VSI can contact `<support@vmssoftware.com>` for help with this product.

## 7. OpenVMS Documentation

The full VSI OpenVMS documentation set can be found on the VMS Software Documentation webpage at https://docs.vmssoftware.com.

## 8. Conventions

VMScluster systems are now referred to as OpenVMS Cluster systems. Unless otherwise specified, references to OpenVMS Cluster systems or clusters in this document are synonymous with VMScluster systems.

The contents of the display examples for some utility commands described in this manual may differ slightly from the actual output provided by these commands on your system. However, when the behavior of a command differs significantly between OpenVMS Alpha and Integrity servers, that behavior is described in text and rendered, as appropriate, in separate examples.

In this manual, every use of DECwindows and DECwindows Motif refers to DECwindows Motif for OpenVMS software.

The following conventions are also used in this manual:

| Convention | Meaning |
|---|---|
| **Ctrl/** *x* | A sequence such as **Ctrl/** *x* indicates that you must hold down the key labeled Ctrl while you press another key or a pointing device button. |
| PF1 *x* | A sequence such as PF1 *x* indicates that you must first press and release the key labeled PF1 and then press and release another key or a pointing device button. |
| **Return** | In examples, a key name enclosed in a box indicates that you press a key on the keyboard. (In text, a key name is not enclosed in a box.) |
| ... | A horizontal ellipsis in examples indicates one of the following possibilities:<br>● Additional optional arguments in a statement have been omitted.<br>● The preceding item or items can be repeated one or more times.<br>● Additional parameters, values, or other information can be entered. |
| .<br>.<br>. | A vertical ellipsis indicates the omission of items from a code example or command format; the items are omitted because they are not important to the topic being discussed. |
| ( ) | In command format descriptions, parentheses indicate that you must enclose the options in parentheses if you choose more than one. |
| [ ] | In command format descriptions, brackets indicate optional choices. You can choose one or more items or no items. Do not type the brackets on the command line. However, you must include the brackets in the syntax for OpenVMS directory specifications and for a substring specification in an assignment statement. |
| [ \|] | In command format descriptions, vertical bars separate choices within brackets or braces. Within brackets, the choices are options; within braces, at least one choice is required. Do not type the vertical bars on the command line. |
| { } | In command format descriptions, braces indicate required choices; you must choose at least one of the items listed. Do not type the braces on the command line. |
| **bold text** | This typeface represents the introduction of a new term. It also represents the name of an argument, an attribute, or a reason. |
| *italic text* | Italic text indicates important information, complete titles of manuals, or variables. Variables include information that varies in system output (Internal error *number*), in command lines (/PRODUCER= *name*), and in command parameters in text (where *dd* represents the predefined code for the device type). |
| UPPERCASE TEXT | Uppercase text indicates a command, the name of a routine, the name of a file, or the abbreviation for a system privilege. |
| `Monospace type` | Monospace type indicates code examples and interactive screen displays.<br><br>In the C programming language, monospace type in text identifies the following elements: keywords, the names of independently compiled external functions and files, syntax summaries, and references to variables or identifiers introduced in an example. |
| - | A hyphen at the end of a command format description, command line, or code line indicates that the command or statement continues on the following line. |

| **Convention** | **Meaning** |
|---|---|
| numbers | All numbers in text are assumed to be decimal unless otherwise noted. Nondecimal radixes—binary, octal, or hexadecimal—are explicitly indicated. |

# Chapter 1. C++ API Concepts

This chapter provides an overview of the RTR C++ foundation classes and describes concepts that apply to application development using the this application programming interface (API). It includes conceptual descriptions of client and server interaction and application processing. Detailed information is provided on each class and its associated methods in later chapters of this manual. For code examples and implementation information, see the Design and Implementation chapter of this manual.

## 1.1. Overview

The C++ foundation classes enable you to implement new RTR client and server applications, or to integrate specific classes into existing applications to add additional functionality.

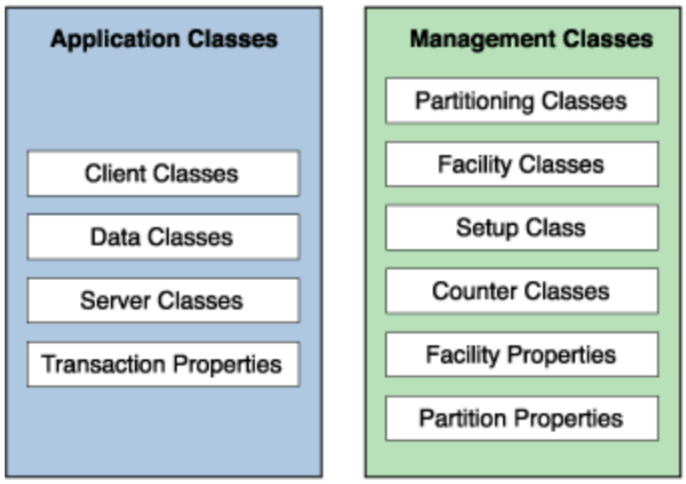
RTR concepts have been mapped to and implemented by the set of foundation classes for handling system management and the needs of business applications.

*Figure 1.1, "C++ Foundation Classes"* shows the C++ foundation classes. Management classes represent RTR, facilities, partitions, and key segments (part of the partitioning classes in *Figure 1.1, "C++ Foundation Classes"* whereas application classes represent transactions, data, messages and events.

The primary application classes include client classes, server classes, and data classes that are common to both client and server classes. There are also server and client transaction property classes.

You use management classes to implement applications that can help manage RTR. You use application classes to implement client and server applications. However, client and server applications can also use the management classes to dynamically set up RTR facilities and partitions.

**Figure 1.1. C++ Foundation Classes**



Facility, partition, and transaction property classes include methods that provide access to facilities, partitions, and transactions. These classes enable a program to obtain additional information on a facility, partition, or a transaction. Transaction property classes are useful for transaction recovery and for obtaining and setting transaction states.

Property classes work with other foundation classes in new applications; they can also be used independently in legacy RTR applications. They do this by using information that existing RTR applications already have, including transaction IDs (tids), facility names, and partition names.

## 1.2. Application Classes

RTR C++ foundation application classes include:

- Client application classes
- Server application classes
- Data classes

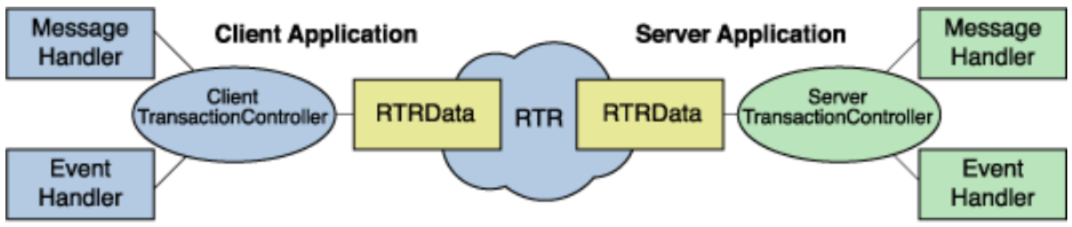
  Data classes are common classes for passing data between client and server applications.

(Client and server transaction property classes are included within the client application classes and server application classes, respectively.)

To use RTR application classes, it is useful to understand RTR concepts necessary for implementing application solutions with the C++ API, C++ API- specific information, and object-oriented concepts.

Figure 1-2 illustrates the client and server classes and the paths through which they typically communicate. (There are design alternatives to the illustrated path.) TransactionController objects control transactions. Communication between client and server applications is through messages and events sent and received by the RTR application. Data objects (instances of data classes) carry these messages and events between RTR clients and servers.

**Figure 1.2. C++ API Classes**



The principal application classes are the transaction controller classes and data classes. A transaction controller object manages a transaction. The RTRData-derived data object is the common means through which client and server applications interact. A message handler encapsulates the data. Most events are not related to transactions. A message is sent from a client to a server or a server to a client (1-to-1). An event can go from one client or server to many clients and servers.

### 1.2.1. Transaction Classes

In RTR, a transaction is a logical grouping of messages.

A transaction is controlled by a TransactionController object. The client transaction controller class (RTRClientTransactionController) creates single instances of a transaction. The server transaction controller class (RTRServerTransactionController) manages single instances of a transaction.

A transaction controller object:

- Handles the sending and receiving of a specific data object.
- Votes to accept or reject a transaction.

Typically, a transaction controller object processes multiple consecutive transactions, but there is at most one active transaction in a transaction controller object at any one time.

A transaction controller:

- Contains at most one transaction at a time (0 or 1).
- Is typically constructed once and reused for each transaction.
- Controls the transaction.
- Processes one transaction at a time. For example, if you need 50 concurrent transactions (at the same time), you need 50 transaction controllers.
- Comes in a client and a server version.

### 1.2.2. Data Classes

Applications use data objects to carry data between RTR clients and servers. Thus, the data classes are common to both client and server applications. RTRData is the base class from which four kinds of data are derived:

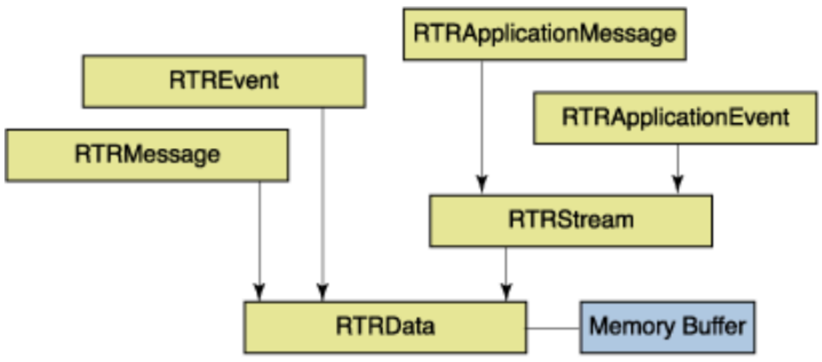
- RTRMessage
- RTREvent
- RTRApplicationMessage
- RTRApplicationEvent

The class factory, RTRClassFactory, creates instances of data classes based on the content of a transaction controller Receive call for a message or event.

Communication between client and server applications is through messages and events. Data objects contain these messages and events sent and received by RTR clients and servers.

Figure 1-3 illustrates the data classes and their relationships to the RTRData base class and a memory buffer. For example, the base class of RTRStream is RTRData and the base class of RTRApplicationMessage is RTRStream.

**Figure 1.3. Data Classes**



An application wanting to send or receive data specifies an RTRData object. The mechanism for sending and receiving is different as follows:

- Sending

When calling SendApplicationMessage on a transaction controller, the caller specifies an RTRApplicationMessage.

- Receiving

  When calling the Receive method, the application supplies an RTRData pointer to NULL. When the transaction controller determines the type of data which is about to be obtained it calls a class factory to create an instance of the appropriate object.

After a successful call to the Receive method, the RTRData pointer contains one of the following kinds of objects:

- RTRMessage, containing an RTR-generated message
- RTREvent, containing an RTR-generated event
- RTRApplicationMessage, containing an application-generated message
- RTRApplicationEvent, containing an application-generated event

The RTRClassFactory class creates the above data objects. Based on the type of message contained on the transaction controller Receive call the class factory creates an instance of the appropriate data class. The class factory also enables you to customize the behavior of data object creation. An application may derive its own RTRClassFactory class and register it with the transaction controller. In this case, the transaction controller calls the application's class factory to create the data object.

## Dispatch Methods and Handlers

Since all Data classes are derived from RTRData, an application can treat the data polymorphically, especially when receiving data on the server.

For example:

```
ServerTransactionController ServerTransactionController;
RTRData *pDataBeingReceived = NULL;
while (true)
{
// Receive some data
ServerTransactionController.Receive(&pDataBeingReceived);
// No need to determine what we received.
// Just call Dispatch()
pDataBeingReceived->Dispatch();
}
```

The RTRData class has a pure virtual method named Dispatch(). This means that all classes derived from RTRData provide an implementation of Dispatch(). This implementation of Dispatch, which is provided by the derived class, determines the exact message or event number that it contains and calls the appropriate method in the handler.

The message and event handler classes are:

- RTRServerMessageHandler
- RTRServerEventHandler
- RTRClientMessageHandler

- RTRClientEventHandler

An application may derive its own class from any or all of these handlers to provide its own custom handling of the specific messages and events.

## 1.2.3. Messages

Data objects carry messages between clients and servers. These messages are of two types:

- RTR-generated messages such as `rtr_mt_rejected` (the transaction has been rejected).
- Application-generated messages (the protocol that drives application business logic).

## 1.2.4. Events

Data objects carry events between clients and servers. These events are of two types:

- RTR-generated events such as RTR_EVTNUM_SRPRIMARY (server is in primary state for a registered partition).
- Application-generated events (the protocol that drives application business logic).

Application events can be transmitted only within the RTR facility in which they are defined. Application events cannot be sent between facilities or outside RTR.

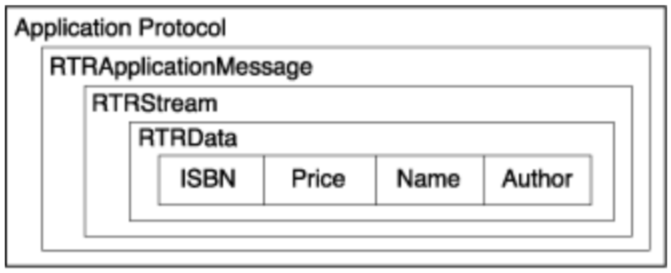
## 1.2.5. Client and Server Interaction

For client and server applications to work together, you create a ClientTransactionController in the client application and a ServerTransactionController in the server application. These transaction objects communicate by using objects derived from the RTRData class.

RTR applications need to define an application-level protocol to pass data between client and server. From the point of view of a client or server application, the application protocol is just data. The data object encapsulates the application protocol as shown in Figure 1-4. In this example, a protocol is defined for sending data between a client and server application that processes book orders, as in the book ordering sample application. This data protocol includes fields for ISBN number, book-price, book-name, and author. These fields are contained in a buffer in an RTRData object.

The data protocol is encapsulated in a user-defined ApplicationProtocol class. The ApplicationProtocol class is an (derives from) RTRApplicationMessage, which is an (derives from) RTRStream, which is an (derives from) RTRData object that contains the application protocol in its buffer.

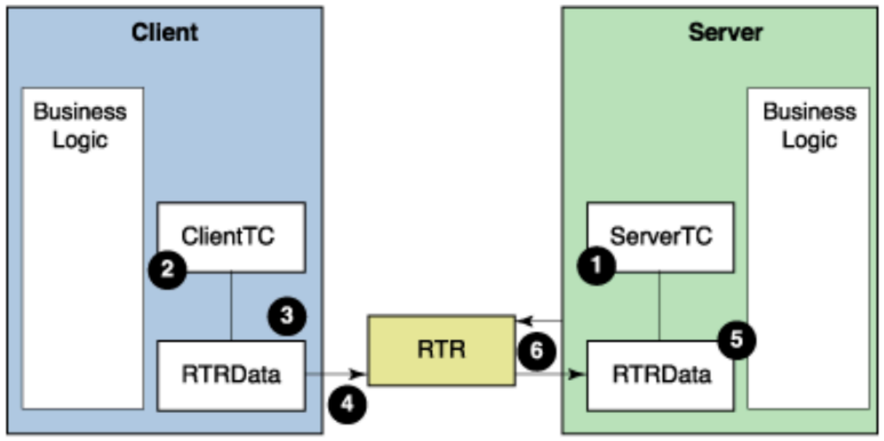
**Figure 1.4. RTRData Encapsulation**



The data classes are used by both client and server RTR applications. When applications want to send or receive data, they specify an RTRData-derived object.

Figure 1-5 illustrates client/server deployment and interaction. The numbered steps represent client logic within the client application and server logic within the server application. For a more detailed description of transactional messaging, see the *VSI Reliable Transaction Router Application Design Guide*.

**Figure 1.5. Client/Server Interaction**



An RTR transaction processing system consists of separate client applications and server applications. This example demonstrates a client sending a message to the server and the server responding, but the server calls the first Receive. The logical interaction between client and server is as follows:

1. Call Receive from an RTRServerTransactionController object to obtain an RTRApplicationMessage object from the client. (The server first creates an RTRServerTransactionController object and then calls Receive.)

2. Create a transaction in the RTRClientTransactionController object by calling StartTransaction.

3. Create an RTRApplicationMessage object (or one derived from RTRApplicationMessage).

4. Send the RTRApplicationMessage object to the server and wait for a message from the server by calling SendApplicationMessage.

5. Process the data in the server application.

6. Send an RTRApplicationMessage object from the server back to the client by calling SendApplicationMessage.

For more information on client and server messaging, see the *VSI Reliable Transaction Router Application Design Guide*.

## 1.2.6. The Class Factory

An instance of the class RTRClassFactory is an object that creates other data objects. The class RTRClassFactory has four methods:

- RTRMessage * CreateRTRMessage()
- RTREvent * CreateRTREvent()
- RTRApplicationMessage * CreateRTRApplicationMessage()
- RTRApplicationEvent * CreateRTRApplicationEvent()

Client and server transaction controllers use the class factory when receiving a message or event. Every transaction controller has a class factory. If the application does not register its own, a default is provided.

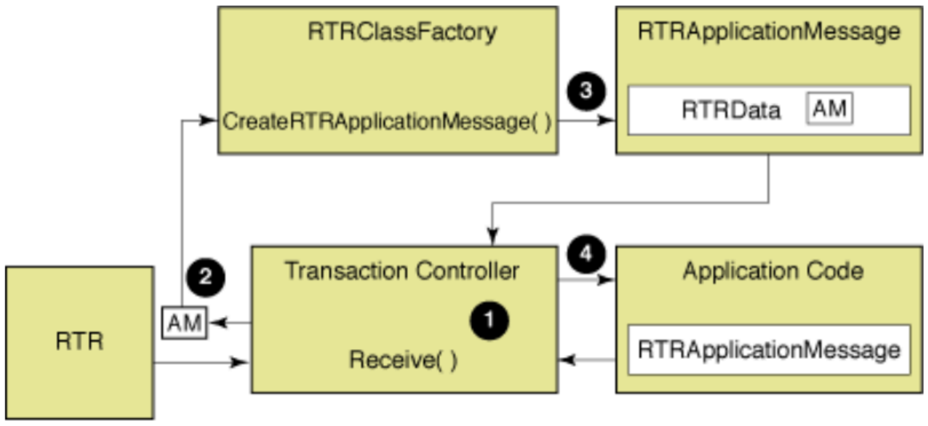
When the application calls Receive, the transaction controller determines what kind of message or event is about to be received, and then calls the appropriate method in the application-derived and registered class factory object (for example, CreateRTRMessage). This method creates the appropriate data object (for example, an RTRMessage object) and returns it to the transaction controller. The transaction controller copies the incoming data into the data object returned from the class factory and returns back to the application's call to Receive().

Applications can override the methods of the RTRClassFactory and return their own customized versions of the data classes.

## Receiving an Application Message

Typical client requests processed by a server application are sent and received as RTRApplicationMessage objects. The most common method for implementing business logic data protocols is deriving from the RTRApplicationMessage class. In Figure 1-6, AM represents an incoming application message.

**Figure 1.6. Receiving an Application Message**



In Figure 1-6:

1. An application calls a transaction controller Receive (for example, RTRServerTransactionController::Receive) to receive a message or event (AM, in the above figure).

2. The transaction controller determines what kind of message or event is to be received (in this case, an application message) and calls the appropriate method in the registered RTRClassFactory object (for example, CreateRTRApplicationMessage).

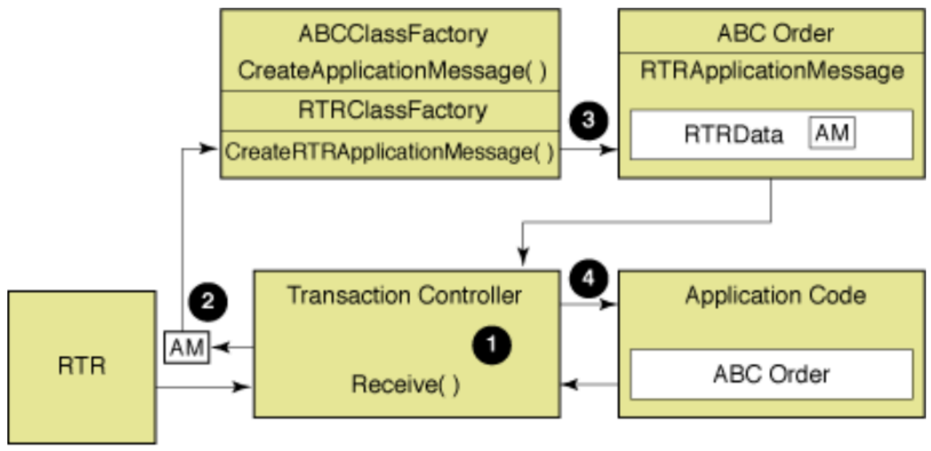
3. The RTRClassFactory object CreateRTRApplicationMessage method creates the appropriate data object (in this case, an RTRApplicationMessage object) and returns it to the transaction controller.

4. The application processes the message according to the application implementation.

## Receiving a User-Defined Application Message

User-defined application messages are sent and received as RTRApplicationMessage objects. In Figure 1-7: Receiving a User-Defined Application Message, AM represents an incoming application message.

**Figure 1.7. Receiving a User-Defined Application Message**



In Figure 1-7:

1. An application calls a transaction controller Receive (for example, RTRServerTransactionController::Receive) to receive a message or event (AM, in the above figure).

2. The transaction controller determines what kind of message or event is to be received (in this case, an application message) and calls the appropriate method in the application-derived and registered ABCClassFactory object (for example, CreateApplicationMessage).

3. The ABCClassFactory object CreateApplicationMessage method creates the appropriate data object (in this case, an ABCOrder object) and returns it to the transaction controller.

4. The application processes the message according to the application implementation.

Note that the derived class factory does not have to handle the all messages. It is only handling the application message (AM) and taking all of the other default methods (for example, CreateRTRMessage).

## 1.2.7. Stream Classes

For an added level of functionality, the RTRStream data class allows for easier access to the data passed between the client and server applications. This class provides methods with which you can read from and write to the data buffer contained in RTRData. With these methods, maintaining offset into the buffer is automatic.

The RTRStream class allows the serialization and deserialization of objects. For example, if a client application called,

```
RTRStream::WriteToStream("WarandPeace");
 RTRStream::WriteToStream("Tolstoy");
```

and a server then called,

```
RTRStream::ReadFromStream(pString1); RTRStream::ReadFromStream(pString2);
```

pString1 would point to "WarandPeace" and after the second read, pString2 would point to "Tolstoy."

For large amounts of data to be sent and received, a WriteToStream method takes a void pointer to the length of the buffer.

## 1.2.8. Application Classes Summary

Figure 1-8 illustrates the client, data, and server classes in the application classes and shows their parallelism. Data classes are common to both client and server applications.

**Figure 1.8. Application Classes**

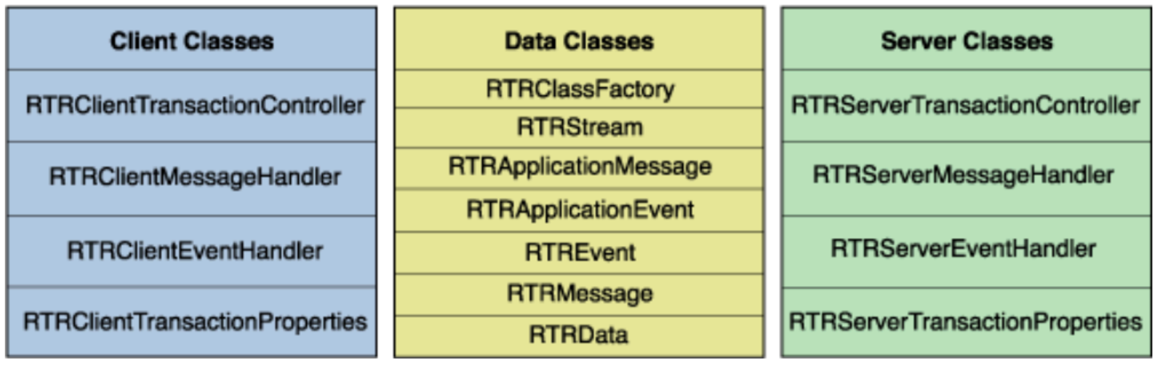


*Table 1.1, "Application Class Category Descriptions"* shows application class categories and their descriptions. *Table 1.2, "Data Class Descriptions"* lists the application data classes that are common to both client and server applications. Except for data classes, the class categories describe the characteristics of the associated client and server classes (for example, the Transaction Controller class category in *Table 1.1, "Application Class Category Descriptions"* describes the RTRServerTransactionController and RTRClientTransactionController classes). For detailed descriptions of individual foundation classes and their associated methods, see the Application Classes chapter of this manual.

**Table 1.1. Application Class Category Descriptions**

| Class Category | Description |
|---|---|
| Transaction Controller | The transaction controller manages each transaction and also manages the channels, messages, and events associated with that transaction.<br>● Has client and server versions.<br>● Manages each RTR transaction (1 transaction controller for each transaction.).<br>● Controls at most one active transaction at a time.<br>● Can process many sequential transactions. |
| Transaction Properties | The RTRTransactionProperties class:<br>● Has client and server versions.<br>● Can be used by new or existing applications.<br>● Includes:<br>• GetTransactionState. |

| Class Category | Description |
|---|---|
| | • SetTransactionState.<br>• GetInvocationType. |
| Event Handlers | Use RTREventHandler classes to obtain information about a transaction such as whether a server is primary, standby or shadow.<br><br>The RTREventHandler class:<br>● Has client and server versions.<br>● Provides default implementation for every event.<br>● Enables the application to override only the events it wants to process.<br>● Can be extended to have application-specific handlers such as OnProcessOrder.<br><br>You must register an event handler with the RegisterHandlers method in the TransactionController class. |
| Message Handlers | Message handlers can be used for all transactions and all application data.<br><br>The RTRMessageHandler class:<br>● Has client and server versions.<br>● Provides default implementation for every message.<br>● Enables an application to override only the messages it wants to process.<br>● Can be extended to have application-specific handlers.<br><br>RTRMessageHandler lets you override only messages you want to use. For example, OnApplicationMessage can be implemented with business-logic-specific objects such as OnStockBuy or OnStockSell. |

### Table 1.2. Data Class Descriptions

| Class Category | Description |
|---|---|
| RTRMessage | The RTRMessage class:<br>● Holds an RTR Message. |

| Class Category | Description |
|---|---|
| | ● Derives from the RTRData class.<br><br>● Is generated internally by RTR.<br><br>If an application has not registered a class factory, the application calls the default class factory to allocate this object. The application:<br><br>● Calls the Dispatch method to send this message to the appropriate handler.<br><br>● Can optionally derive from RTRMessage to create a more business-specific class. |
| RTREvent | The RTREvent class:<br><br>● Holds an RTR Event.<br><br>● Derives from the class RTRData.<br><br>● Is generated internally by RTR.<br><br>If an application has not registered a class factory, the application calls the default class factory to allocate this object. The application:<br><br>● Calls the Dispatch method to send this message to the appropriate handler.<br><br>● Can optionally derive from RTREvent to create a more business-specific class. |
| RTRData | The RTRData class is used to send and receive messages and events. It is the abstract base class for the following four data classes:<br><br>● RTREvent<br><br>● RTRMessage<br><br>● RTRApplicationEvent<br><br>● RTRApplicationMessage |
| RTRApplicationMessage | The RTRApplicationMessage class:<br><br>● Holds an Application Message.<br><br>● Derives from class RTRStream.<br><br>● Is generated by a C++ API application.<br><br>● Can be treated as a stream to write and read the state of a higher level object.<br><br>The application: |

| Class Category | Description |
|---|---|
| | ● Calls the Dispatch method to send this message to the appropriate handler.<br><br>● Can optionally derive from RTRApplicationMessage to create a more business-specific class. |
| RTRApplicationEvent | The RTRApplicationEvent class:<br><br>● Holds an application Event.<br><br>● Derives from the class RTRStream.<br><br>● Is generated by a C++ API application.<br><br>● Can be treated as a stream to write and read the state of a higher level object.<br><br>The application:<br><br>● Calls the Dispatch method to send this message to the appropriate handler.<br><br>● Can optionally derive from RTRApplicationMessage to create a more business-specific class. |
| RTRStream | The RTRStream class:<br><br>● Derives from and extends the RTRData class<br><br>● Allows RTR applications to issue multiple read and write requests to the memory buffer (managed by RTR).<br><br>● Automatically handles buffer pointer management<br><br>● Can be used to serialize and deserialize objects through RTR. |
| RTRClassFactory | The RTRClassFactory class creates instances of the data classes:<br><br>● RTRMessage<br><br>● RTREvent<br><br>● RTRApplicationMessage<br><br>● RTRApplicationEvent<br><br>An application registers its own class that is derived from the RTRClassFactory and returns its own business level objects. If an application does |

| Class Category | Description |
|---|---|
| | not register a customized version, by default, a class factory object is internally created. |

## 1.3. Management Classes

Management classes manage the environment in which an RTR application executes, not the business-logic infrastructure of the application. This allows you to do in a program what formerly had to be done at the system management command level.

### Facility Management

Managing facilities is based on three concepts provided as separate foundation classes:

- Facility manager (RTRFacilityManager class)

  A facility manager creates and deletes facilities, and adds and removes facility members based on facility name.

- Facility properties (RTRFacilityProperties class)

  Facility properties represent the information and properties of a single facility.

- Facility member (RTRFacilityMember class)

  Facility members represent the individual members of a particular facility. A facility member is both a role and a node combined, because a node can have more than one role. For example, a nodename can represent three members by being defined three times with the same node but with different roles (backend, frontend, router).

For general information on RTR facilities, see *VSI Reliable Transaction Router Getting Started* and the *VSI Reliable Transaction Router System Manager's Manual*.

### Partitions and Key Segments

One of the benefits of the routing capability in RTR is that it enables you to partition your data across multiple servers and nodes for increased performance. Within an application, the partition determines how messages are routed from clients to servers. RTR routes messages to the correct partition on the basis of an application-defined key.

The contents of a message determine its destination. The router tracks the location of data partitions and sends client messages to the appropriate server for processing. The routing key, or key segment, is embedded within the RTR message.

The foundation classes provide the object-oriented framework to implement data partitioning with the following classes:

- Partition manager (RTRPartitionManager class)

  A partition manager creates and deletes partitions, and returns properties for individual partitions based on partition name.

- Partition properties (RTRBackendPartitionProperties class)

  Partition properties represent individual partitions within RTR and provide statistics for a partition.

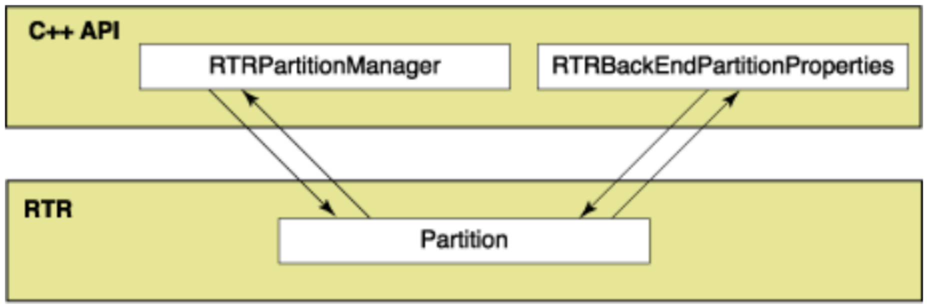
- Key segment (RTRKeySegment class)

  A key segment object defines the range of a partition.

  An RTRKeySegment object specifies a data key range and is associated with a partition when a Partition Manager creates a partition.

*Figure 1.9, "Partition Objects and RTR"* illustrates the relationship between RTR entities and partition classes. Partition classes refer to an RTR partition. As the figure illustrates, the actual partition resides in RTR, not in the foundation class objects. Methods within the partition classes can create and delete partitions, and get partition properties for the RTR partitions.

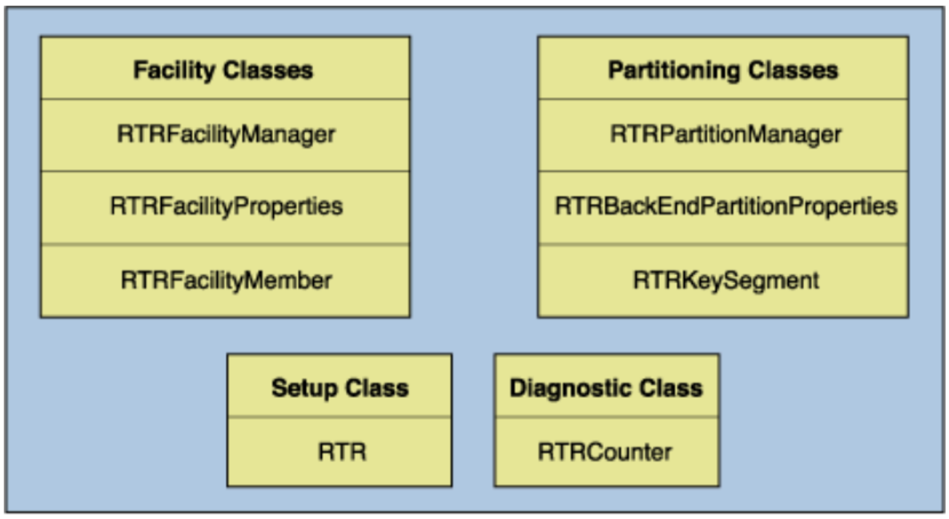
**Figure 1.9. Partition Objects and RTR**



## 1.3.1. Management Classes Descriptions

Figure 1-10 shows the management class categories and their classes. These classes can be used in new applications or integrated into existing legacy applications.

**Figure 1.10. Management Classes**



With the management classes, you can create a facility or a partition programmatically instead of using the command language interface (CLI). For legacy applications, you can write management routines to create your application environment in an existing RTR C-language application.

Facility, management, and partition information exists in RTR. The management classes access the information from RTR.

*Table 1.3, "Management Class Descriptions"* describes the management classes. For detailed descriptions of individual classes and their associated methods, see the Management Classes chapter of this manual.

**Table 1.3. Management Class Descriptions**

| Class | Description |
|---|---|
| RTRFacilityManager | Is used to manage the creation, deletion, and viewing of facilities based on facility name (existing RTR programs use facility names). |
| RTRFacilityMember | Represents a member of a particular facility. The member can be anynode in the facility, including the local node.<br><br>Knows the relationship to the local node.<br><br>Provides member functions to evaluate connectivity. For example, IsConnectedToLocalNode returns a boolean return to a query such as: "Is node A connected to me?" |
| RTRFacilityProperties | Represents a single facility that exists within RTR.<br><br>Knows other members in the facility. |
| RTRPartitionManager | Manages the creation and deletion of partitions based on partition name. |
| RTRKeySegment | Defines and represents the key range of a partition associated with an RTR server. |
| RTR | The RTR class represents RTR on the local node and performs actions that apply to RTR as a whole including:<br><br>● Starting RTR.<br>● Stopping RTR.<br>● Creating a journal.<br>● Deleting a journal.<br>● Starting a web server.<br>● Stopping a web server. |
| RTRCounter | Enables an application to define and manipulate a counter within RTR. They can be used within monitor screens to mix RTR and application diagnostic information. RTRCounter is the base class for:<br><br>● RTRStringCounter<br>● RTRSignedCounter |

| Class | Description |
|---|---|
| | ● RTRUnsignedCounter |
| RTRBackendPartitionProperties | Supplies information about a partition, once it has been created.<br><br>Can be used by new or existing applications.<br><br>Can be used to obtain information on partitions created at the command line or by the RTRPartitionManager.<br><br>Represents a single partition that exists within RTR. Since a partition property object is not an actual partition but an object that knows the properties of an RTR partition, if the partition is deleted, the partition class points to nothing and returns an error.<br><br>Provides statistics for a partition. |

## 1.4. Processing Models

You can use either of two processing models to implement client and server applications. Depending on which processing model you use, you implement the classes differently. The two processing models are:

- Event-driven
- Polling

Processing mechanisms are different for the polling and event-driven models. With the polling model, when receiving the data object, obtaining the RTR message value requires a GetMessageType call. With event-driven processing, if you are using the handlers, a Receive returns your states. Event-driven is an addition to the primitive polling mechanism. By adding a call to Dispatch in the polling mechanism in the application, you can enable default processing for all messages and events.

*Table 1.4, "Transaction Processing Models Compared"* compares the two processing models. These comparisons apply to both client and server. The sample application and code examples in this book use event-driven processing in server applications, and polling in client applications.

**Table 1.4. Transaction Processing Models Compared**

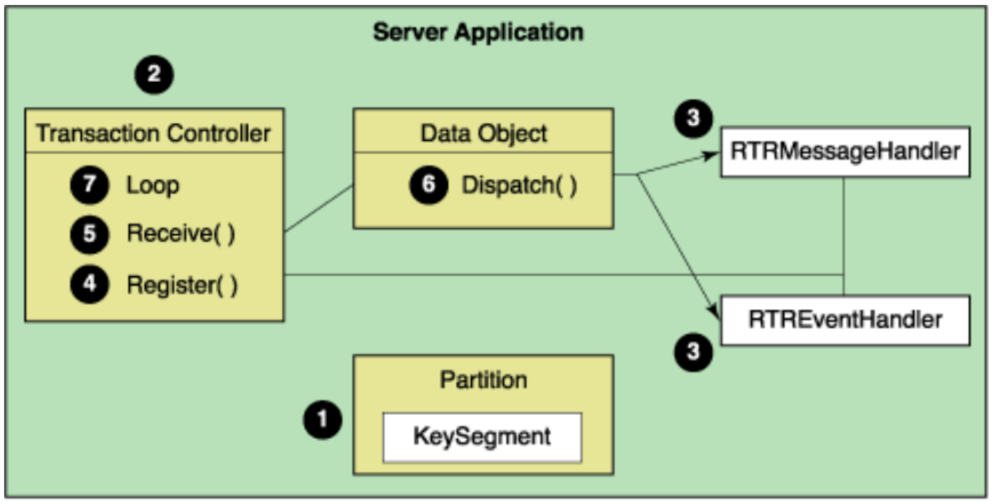
| Processing Method | What You Get | Programming Logic | Message and Event Handling |
|---|---|---|---|
| Event-Driven | Default handling of all RTR messages and events. | Create a loop containing Receive() and Dispatch() calls. | Messages and Events are handled by the MessageHandler and EventHandler objects. |
| Polling | RTRData methods that allow for user-implemented detection of incoming data and | Use RTRData methods to detect incoming data types. Develop logic to handle all possible messages and events. | User-implemented logic in place of MessageHandler and EventHandler classes. |

| Processing Method | What You Get | Programming Logic | Message and Event Handling |
|---|---|---|---|
| | development of message and event handling. | | |

## 1.4.1. Event-Driven Model

Figure 1-11 shows the steps in the event-driven model of transaction processing as used in a server application.

**Figure 1.11. Event-Driven Server Processing**



In the event-driven model, the application is informed when there is something for it. RTR automatically sends messages to the server and the server runs a transaction, using the Receive and Dispatch methods within a while loop. Business logic resides in the message and event handlers. The event-driven model is the recommended method for implementing server applications.

As shown in *Figure 1.11, "Event-Driven Server Processing"*, the sequence of operations is as follows:

1. Create an environment that has one or more partitions that are defined in key segments.

2. Create a TransactionController object.

3. Create the handler classes derived from base classes. Business logic resides in the message and event handlers.

4. Call Register methods to register facility, class factory, partition, and handlers. This internal hookup creates a mapping to the message and event handlers.

5. Start to receive information (messages or events) for the partition registered to the ServerTransactionController by calling Receive, a method on the ServerTransactionController. The class factory creates a data object on the Receive call. The Transaction Controller receives the data object.

6. Call Dispatch. Dispatch knows which handler to go to.

   User-implemented logic and methods are stored in the data object. Checking for RTR-generated data, retrieving messages, and retrieving events are all done for you automatically, if you call Dispatch. For example, on a Receive call, if the message is `rtr_mt_msgn`, then calling

Dispatch calls OnApplicationMessage by default. OnApplicationMessage is a method in the RTRServerMessageHandler and RTRClientMessageHandler classes.

Business logic is typically implemented in the server message handler. However, you can implement business logic in other ways as well.

7. Loop for next event.

## Event-Driven Processing

Using the event-driven model implements the following mechanism:

1. Receive within a loop to receive a message or event.

2. Call Dispatch.

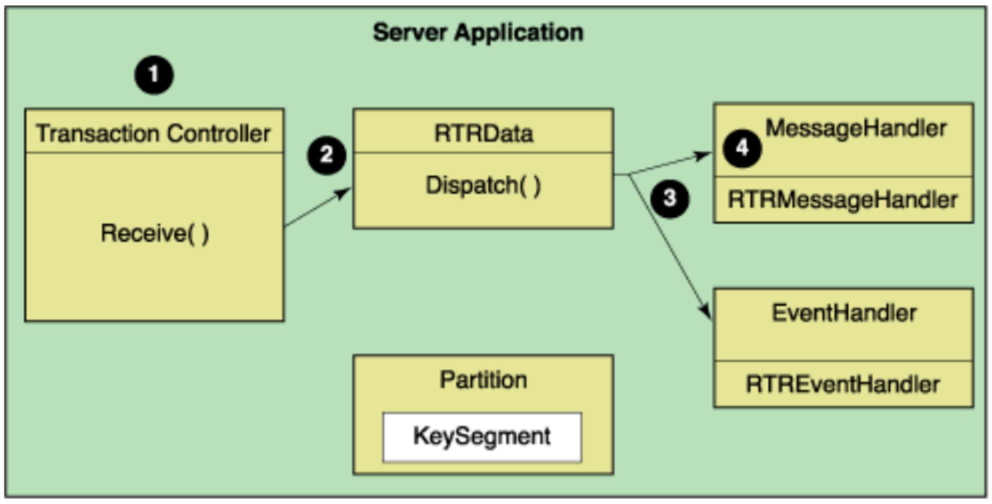
   The Data Object is passed on this call. All handler methods have two parameters: a message type, and a pointer to the TransactionController from which the message came. Data Objects, which are stateless, are not tied to a TransactionController; they can be handled by different TransactionControllers. Thus, using a TransactionController does not restrict client applications.

3. By default, the RTRData object automatically accesses the appropriate Handler by the appropriate method, depending on the message or event with the RTRClassFactory class. For example, if RTRData contains RTR message type `rtr_mt_msgn` , then Dispatch calls OnApplicationMessage(RTRApplicationMessage).

4. The Data Object is processed within the appropriate Handler. For example, the RTRData object containing `rtr_mt_msgn` is processed by OnApplicationMessage. This is where the business logic is typically implemented.

This sequence is shown in Figure 1-12.

**Figure 1.12. Event-Driven Processing Example**



## Message and Event Handling

This section provides event and message handling examples that are processed based on what RTR message or event is received on a Receive call. Depending on the message received, the subsequent process is different, as shown in *Table 1.5, "Message and Event Handling Examples"*.

**Table 1.5. Message and Event Handling Examples**

| If the message received is: | Then: |
|---|---|
| rtr_mt_msgn | The Data Object goes to the Message Handler by the OnApplicationMessage(Data Object) method. Then, in the Message Handler, the Data Object is processed by OnApplicationMessage. |
| rtr_mt_rejected | The Data Object goes to the Message Handler using the OnRejected(Data Object) method. Then, in the Message Handler, the Data Object is processed by OnRejected. |
| rtr_mt_prepare | The Data Object is dispatched to be handled internally. Application business logic does not need to know about RTR Prepares. In the C++ API, Prepares are transparent. |
| EVTNUM_SRPRIMARY | The Data Object goes to the Event Handler using the OnServerIsPrimary(Data Object) method. Then, in the Event Handler, the Data Object is processed by OnServerIsPrimary. |

RTRMessageHandler and RTREventHandler are the default handlers. Processing is done by an application's derived business logic. Default handlers do not keep state, so the application must return to BackendPartitionProperties to get state.

## Event-Driven Example

Example 1-1 illustrates the looping implementation for event-driven processing.

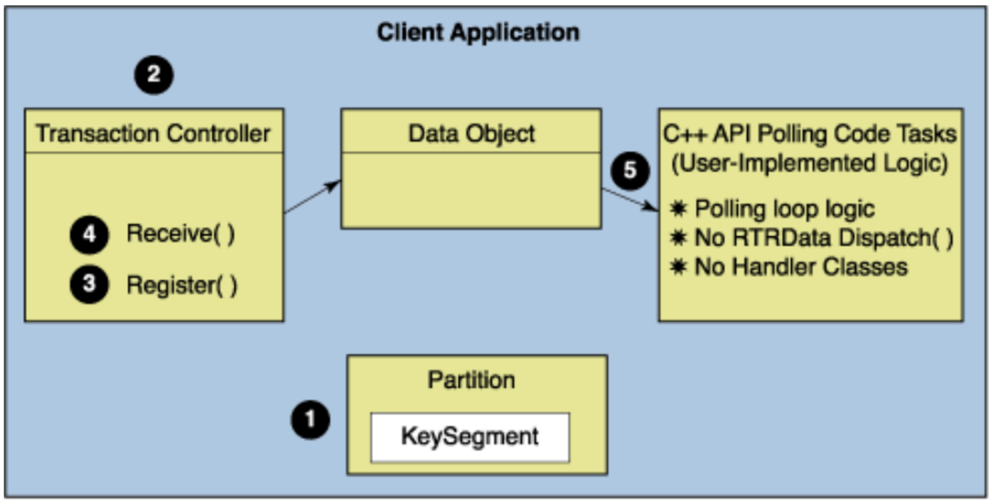
**Example 1.1. Example 1-1: Receive Loop**

```
ServerTransactionController ServerTransactionController;
RTRData *pDataBeingReceived = NULL;
while (true)
{
// Receive some data
ServerTransactionController.Receive(&pDataBeingReceived);
// No need to determine what we received.
// Just call Dispatch();
pDataBeingReceived->Dispatch();
}
```

## 1.4.2. Polling Model

Figure 1-13 shows the steps in the polling model of transaction processing as used in a client application.

**Figure 1.13. Polling Processing Model**



The polling model processing steps are:

1. Create an environment that has one or more partitions defined in key segments.
2. Create a transaction controller object.
3. Call Register methods to register facility (for client) and partitions (for server) and class factory.
4. Call Receive to check the data object.
5. In place of Dispatch, start gathering information for the partition on RTR by calling RTRData methods such as IsApplicationMessage, IsMessage, and IsEvent (for a full listing of boolean RTRData methods, see the Application Classes chapter of this manual) to determine what type of data is being received in order to process it.

   User-implemented logic handles all possible messages, events, and serialized objects using RTRData methods.
6. Call Receive again.

In the polling model, you create a receive loop to poll for incoming data. Messages or events are received one at a time, and Register does not connect message and event handlers. The server asks RTR for a request.

You can still check the data object and code tasks as follows:

- Create the polling loop logic.
- In place of Dispatch, detect the incoming data type using the RTRData methods IsApplicationData, IsMessage, and IsEvent. If you call Dispatch, RTR responds that there are no handlers.
- In place of the Handler classes, you must develop logic to handle all possible RTR and application messages, events, and serialized objects using RTRData methods such as GetBuffer, GetMessageType, and GetEventNumber. These methods are used in the Dispatch call.

As *Figure 1.13, "Polling Processing Model"* illustrates, in common with the event driven model, you use a subset of the same objects, but Register does not connect the message and event handlers.

### Polling Model Example

*Example 1.2, "Example 1-2: Polling Model Example"* illustrates an implementation for the polling model of processing. As this example illustrates, the flow is controlled by the object that is polling for a message or event from RTR with Receive.

#### Example 1.2. Example 1-2: Polling Model Example

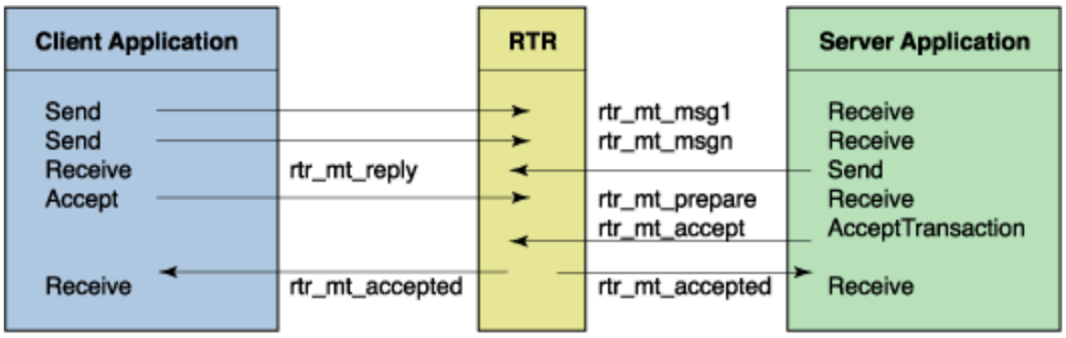
```
ServerTransactionController ServerTransactionController;
RTRData *pDataBeingReceived = NULL;
while (true){
// Receive some data
ServerTransactionController.Receive(&pDataBeingReceived);
// Since handlers are not being used, determine what is
  // received. Application-generated message or event.
  // RTR-generated message or event.
if (true = pDataBeingReceived->IsApplicationMessage())
{
  // Process accordingly
}
else
if (true = pDataBeingReceived->IsApplicationEvent())
{
  // Process accordingly
}
else
if (true = pDataBeingReceived->IsRTRMessage())
{
  // Process accordingly
}
else
if (true = pDataBeingReceived->IsRTREvent())
{
  // Process accordingly
}
}
```

## 1.5. Base Classes Message and Event Mapping

The foundation class message and event handler methods are provided in base classes. You derive from them and choose which ones to use in your implementation. Base handlers are used by default, if you do not derive from them.

*Figure 1.14, "RTR Messaging Between Client and Server Applications"* illustrates RTR messaging between client and server. RTR messages are contained in the data object passing between the client and server. With event-driven processing, the class factory creates the appropriate data object.

**Figure 1.14. RTR Messaging Between Client and Server Applications**



To connect, the client registers a facility and the server registers a facility and a partition. The client transaction ends with rtr_mt_accepted or rtr_mt_rejected. The server transaction ends with the AcknowledgeTransactionOutcome method.

When Dispatch is called, certain handlers are called for transactions on the client and server, as shown in the following tables.

## 1.5.1. Client Messages

When Dispatch is called, certain handlers are called for transactions on the client, as shown in *Table 1.6, "Client Handlers by Message Type"*.

**Table 1.6. Client Handlers by Message Type**

| When the RTR Message Type is: | Contained in: | The RTRClientMessageHandler Call is: |
|---|---|---|
| rtr_mt_accepted | RTRMessage | OnAccepted |
| rtr_mt_reply | RTRApplicationMessage | OnApplicationMessage |
| rtr_mt_rttosend | RTRMessage | OnReturnToSender |
| rtr_mt_prepared | RTRMessage | OnAllPreparedTransaction |
| rtr_mt_rejected | RTRMessage | OnRejected |

For example, using the event-driven processing model, when RTRData contains rtr_mt_reply, by default, the RTRApplicationMessage Dispatch method calls OnApplicationMessage.

## 1.5.2. Client Events

When Dispatch is called, certain handlers are called for transactions on the client, as listed in *Table 1.7, "Client Handlers by Event for RTREvent"*.

**Table 1.7. Client Handlers by Event for RTREvent**

| When the RTR Event Number is: | The RTRClientEventHandler Call is: |
|---|---|
| RTR_EVTNUM_FACDEAD | OnFacilityDead |
| RTR_EVTNUM_FACREADY | OnFacilityReady |
| RTR_EVTNUM_FERTRGAIN | OnFrontendGainedLinkToRouter |

| When the RTR Event Number is: | The RTRClientEventHandler Call is: |
|---|---|
| RTR_EVTNUM_FERTRLOSS | OnFrontendLostLinkToRouter |
| RTR_EVTNUM_RTRBEGAIN | OnRouterGainedLinkToBackend |
| RTR_EVTNUM_RTRBELOSS | OnRouterLostLinkToBackend |
| RTR_EVTNUM_KEYRANGEGAIN | OnNewKeyRangeAvailable |
| RTR_EVTNUM_KEYRANGELOSS | OnKeyRangeNoLongerAvailable |

For example, with the event-driven processing model, when RTRData contains RTR_EVTNUM_FACREADY, by default, the RTREvent Dispatch method calls OnFacilityReady.

## 1.5.3. Server Messages

When Dispatch is called, certain handlers are called for transactions on the server side, as listed in *Table 1.8, "Server Handlers by Message Type"*.

**Table 1.8. Server Handlers by Message Type**

| When the RTR Message Type is: | Contained in: | The RTRServerMessageHandler Call is: |
|---|---|---|
| rtr_mt_accepted | RTRMessage | OnAccepted |
| rtr_mt_msg1 | RTRApplicationMessage | OnInitialize |
| | | OnApplicationMessage |
| rtr_mt_msg1_uncertain | RTRApplicationMessage | OnUncertainTransaction |
| rtr_mt_msgn | RTRApplicationMessage | OnApplicationMessage |
| rtr_mt_prepare | RTRMessage | OnPrepareTransaction |
| rtr_mt_rejected | RTRMessage | OnRejected |

For example, with the event-driven processing model, by default when RTRData contains rtr_mt_msg1, the RTRServerMessageHandler first calls OnInitialize and then calls OnApplicationMessage. With the polling model, use IsMessage in place of Dispatch and implement GetMessageType to handle the message.

A typical series of Server messages processed for a Transaction in an RTRTransactionController object would be as follows:

- Start the loop and execute the following receives:

  ```
  OnInitialize
  OnApplicationMessage
  OnPrepareTransaction
  OnAccepted
  ```

- Loop again and get another transaction.

## 1.5.4. Server Events for RTREvent

When Dispatch is called, certain handlers are called for transactions on the server side, as listed in *Table 1.9, "Server Handlers by Event"*.

**Table 1.9. Server Handlers by Event**

| When the RTR Event Number is: | The RTRServerEventHandler Call is: |
|---|---|
| RTR_EVTNUM_BERTRLOSS | OnBackendGainedLinkToRouter |
| RTR_EVTNUM_BERTRGAIN | OnBackendGainedLinkToRouter |
| RTR_EVTNUM_FACDEAD | OnFacilityDead |
| RTR_EVTNUM_FACREADY | OnFacilityReady |
| RTR_EVTNUM_RTRFEGAIN | OnFrontendGainedLinkToRouter |
| RTR_EVTNUM_RTRFELOSS | OnFrontendLostLinkToRouter |
| RTR_EVTNUM_SRSHADOWGAIN | OnServerGainedShadow |
| RTR_EVTNUM_SRSHADOWLOST | OnServerLostShadow |
| RTR_EVTNUM_SRRECOVERCMPL | OnServerRecoveryComplete |
| RTR_EVTNUM_SRPRIMARY | OnServerIsPrimary |
| RTR_EVTNUM_SRSECONDARY | OnServerIsSecondary |
| RTR_EVTNUM_SRSTANDBY | OnServerIsStandby |

For example, with the event-driven processing model, by default, when RTRData contains RTR_EVTNUM_FACREADY, the RTRServerEventHandler calls OnFacilityReady. With the polling model, use IsEvent in place of Dispatch, and implement GetEventNumber to handle the event.

For more information, see the state diagrams in Appendix C of the *VSI Reliable Transaction Router Application Design Guide*.

## 1.6. Using the C++ API with Existing Applications

When working with existing RTR applications, you can integrate individual C++ foundation classes into existing client or server applications and also write new management routines that work with existing applications. With the C++ foundation classes, there are no migration issues. There is no need to rewrite existing code to integrate C++ foundation classes. Existing client and server applications are linked transparently by RTR.

In existing applications, objects defined in the application can point to instances of foundation classes.

These classes are designed to be used:

- With other foundation classes
- Independently in legacy RTR applications (property classes are constructable from legacy applications)

Objects defined in your application can point to instances of the foundation classes, and inherit the rich functionality within these base classes.

The C++ foundation classes provide a method for implementing RTR solutions that is easier to use than the C API. The C++ foundation classes:

- Replaces RTR structures with easily manageable classes. You no longer need to master complex structures and flags, a common cause of programming errors. These structures and flags are not part of the foundation classes.

- Replaces all flags with Get/Set methods. This completely eliminates the use of channels in new implementations.
- Provides for transparent creation and use of channels using the transaction controller classes, RTRServerTransactionController and RTRClientTransactionController.
- Provides default handling code for all messages and events where appropriate. Formerly, an application had to provide handling for all messages and events and could not write common processing code.
- Abstracts the sending and receiving of data to a higher level. Sending and receiving data is no longer handled at a low level. The foundation classes eliminate coding for buffers and links.
- Transforms the features of `rtr_request_info()` and `rtr_set_info()` into simple methods of RTR classes. `rtr_request_info()` and `rtr_set_info()` calls require internal knowledge of RTR data structures. The C++ API obtains this information without the application needing to know the internals of RTR.
- Represents each major RTR concept in its own individual class.

## 1.6.1. Classes that Legacy Applications Can Use

*Table 1.10, "Foundation Classes for Legacy Applications"* lists the classes that legacy RTR server applications can use to create and manage the environment in which RTR applications run. The second column of *Table 1.10, "Foundation Classes for Legacy Applications"* lists the information required to implement instances of these classes.

**Table 1.10. Foundation Classes for Legacy Applications**

| Class | Requires: |
|---|---|
| Setup class: | |
| ###RTR | Nothing |
| Facility classes: | |
| ###RTRFacilityManager | Facility Name |
| ###RTRFacilityMember | Facility Name |
| Partition classes: | |
| ###RTRPartitionManager | Nothing |
| ###RTRKeySegment | Nothing |
| Property classes: | |
| ###RTRClientTransactionProperties | TID (Transaction ID) |
| ###RTRServerTransactionProperties | TID |
| ###RTRBackendPartitionProperties | Partition Name |
| ###RTRFacilityProperties | Facility Name |
| Diagnostic class: | Nothing (for new applications) |
| ###RTRCounter | Group Name |

Facility, management, and partition information exists in RTR. The methods within the management and property classes rely on attributes that RTR applications have.

For example, the diagnostic class RTRCounter relies on the following attributes for getting information:

- Group name
- Counter name
- Data type

These are all attributes found in RTR.

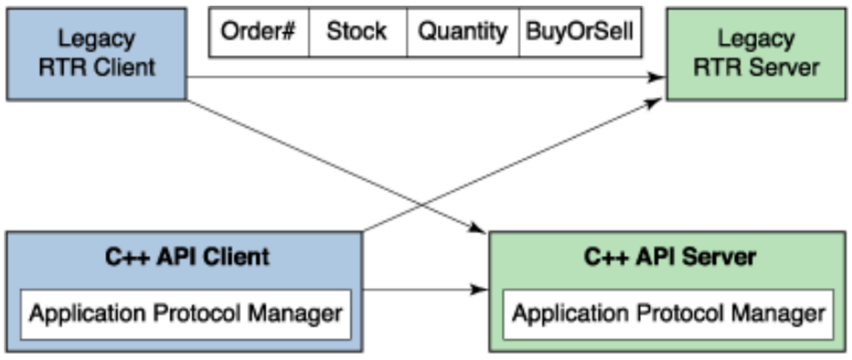
The RTRBackendPartitionProperties class relies on partition name and facility name; existing applications already know the partition name. This enables you to call methods, such as GetRetryCount, in this class by passing in a partition. For example:

```
RTRBackendPartitionProperties
MyPartition("MyPartitionName");
MyPartition.GetRetryCount();
```

## 1.6.2. Encapsulating Application Protocols

Since foundation classes work with existing applications, the protocol for passing data has not changed. As Figure 1-15 illustrates, legacy applications and new applications both use the same protocol for passing data. Thus, all combinations of old and new clients and servers can communicate with each other.

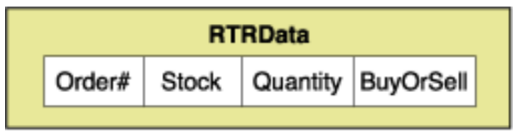
**Figure 1.15. C++ API into Existing Applications**



The protocol manager represents an object that knows how to send and receive a protocol defined by the application. The protocol *is* your data. This is achieved by deriving a class from RTRData that knows how to store information (data) in it. RTRData does this by pointing to a buffer.

*Figure 1.16, "The Protocol Manager Object"* illustrates an example of data encapsulation in the protocol manager objects that are shown in *Figure 1.15, "C++ API into Existing Applications"*.

**Figure 1.16. The Protocol Manager Object**

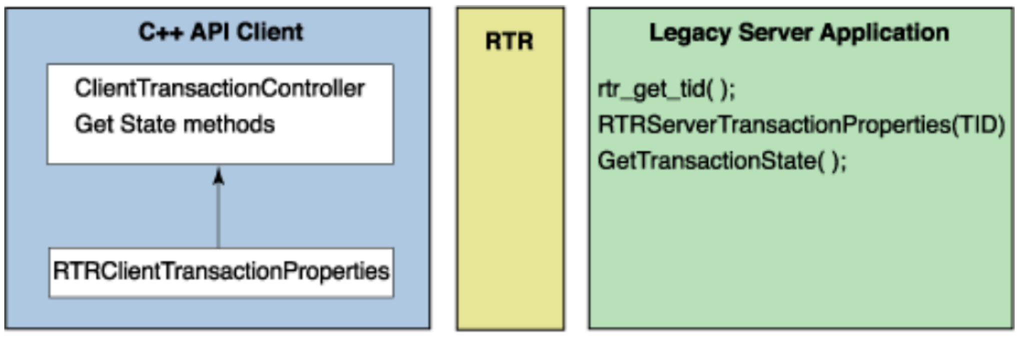


For more information on defining a class that encapsulates an application protocol, see the Design and Implementation chapter of this manual.

### 1.6.3. Implementation Example

In the example shown in *Figure 1.17, "Legacy Application Example"*, there is an existing server application and a new client application. To have your existing RTR legacy server application communicate with and obtain information from a new C++ client application, you do not need to integrate C++ foundation classes into your server application.

**Figure 1.17. Legacy Application Example**


C++ API Client
ClientTransactionController
Get State methods
RTRClientTransactionProperties
RTR
Legacy Server Application
rtr_get_tid( );
RTRServerTransactionProperties(TID)
GetTransactionState( );

Legacy applications do not have a TransactionController but with the RTRServerTransactionProperties and RTRClientTransactionProperties classes, you need only a TID (transaction identifier) to get state information. You obtain the TID with the existing RTR C API using the rtr_get_tid method. You can pass this TID into the RTRServerTransactionProperties and RTRClientTransactionProperties classes.

By using the rtr_get_tid method of the RTR C API to get the TID, you can pass this value to the new C++ API to construct a ServerTransactionProperties object, with this TID as the parameter (ServerTransactionProperties(TID)). Creating an application with this ServerTransactionProperties object enables you to call any member functions within the ServerTransactionProperties class, such as GetTransactionState and GetFacility.

## 1.7. Compiling and Linking your Application

All client and application programs must be written using C, C++, or a language that can use RTR C++ API calls. Include the RTR data types and error messages file rtrapi.h in your compilation so that it will be appropriately referenced by your application. For each client and server application, your compilation/ link process is as follows:

1. Write your application code using RTR calls.

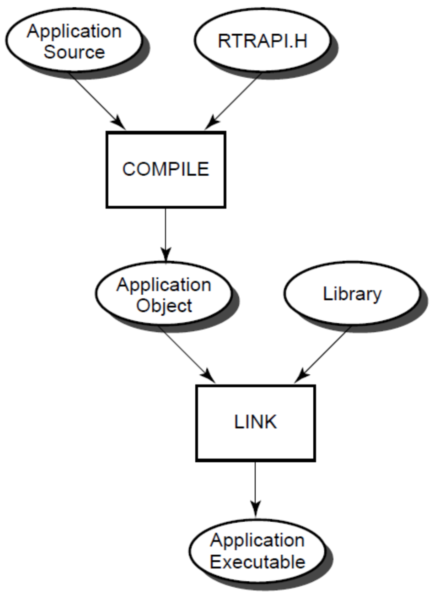
2. Use RTR data and status types for cross-platform interoperability.

3. Compile your application code calling in rtrapi.h using ANSI C include rules. For example, if rtrapi.h is in the same directory as your C++ code, use with the following statement: #include "rtrapi.h".

4. Link your object code with the RTR library to produce your application executable.

This process is illustrated in *Figure 1.18, "RTR Compile Sequence"*. In this figure, *Library* represents the RTR C++ API shareable images (OpenVMS), DLLs (Win32), and shared libraries (UNIX).

**Figure 1.18. RTR Compile Sequence**



The following command lines show the sort of command lines that are needed to compile and link a C ++ RTR application that uses the C++ foundation classes, with and without Posix or Microsoft threads. The parts of the command relating to RTR and the parts relating to threads are shown. You may need to specify library directories explicitly if the RTR header files and libraries are not installed in the same directory or in system directories. Note that the exact name of the RTR foundation classes shared library, DLL or shareable image file, and how it is referenced in a command line, varies slightly according to the conventions for the particular platform. Most compilers recognize the extensions .cc .cpp and .cxx for C+ + source files.

- VSI C++ for OpenVMS, single threaded application:

  ```
  $ cxx yourapp.cxx
  $ cxxlink yourapp,sys$input/opt
  librtrcpp/share
  $
  ```

- VSI C++ for OpenVMS 7 Alpha, multi-threaded application:

  ```
  $ cxx yourapp.cxx
  $ cxxlink yourapp,sys$input/opt
  librtrcpp_r/share
  $
  ```

- Windows MSVC (always multithreaded):

  ```
  > cl /c  -D_MT yourapp.cpp
  > link yourapp.obj /out:yourapp.exe rtrcppapi.lib
  ```

- VSI C++ for Tru64 UNIX (also available for Linux Alpha), single threaded:

  ```
  % cxx yourapp.cc -o yourapp -lrtrcpp
  ```

- VSI C++ for Tru64 UNIX (also available for Linux Alpha), multi-threaded:

  ```
  % cxx -pthread yourapp.cc -o yourapp -lrtrcpp
  ```

Compilers commonly used in developing RTR applications include those in the following table. For additional information, see the Reliable Transaction Router *Software Product Description*.

| **Operating System** | **Compiler** | **Compiler Version** |
|---|---|---|
| Microsoft Windows | Microsoft Visual C++ | Version 6.0 SP4 |
| OpenVMS Alpha | Compaq C | Version 6.2-006 |
| | Compaq C++ | Version 6.2-035 |
| OpenVMS VAX | Compaq C | Version 6.2-003 |
| Sun | Workshop Compilers | Version 4.2 |
| Tru64 UNIX | Compaq C | Version 6.3-126 |
| | Compaq C++ | Version 6.2-033 |

# Chapter 2. Design and Implementation

This chapter contains suggestions for designing and implementing a new client or server application using the C++ foundation classes. It also includes code examples from the C++ book order and processing sample application included in the examples directory of the RTR kit. This sample application shows how to implement a derived-receive model. Topics include:

- Design steps
- Implementation steps
- Implementation example
- Derived receive models
- Sample application walkthrough

## 2.1. Design Steps

When creating a new client or server application:

1. Analyze your application requirements.

   Consider your business functions and map them to C++ classes. In the sample application, the client application accepts orders to purchase books, and the server application processes these orders from the client application.

2. Define your data protocol.

   The data protocol defines the data that is passed between client and server applications. In the sample application, orders are passed between client and server. These orders can be books or magazines. Book is a type of Order.

3. Determine if your application should use the default Message and Event handlers.

   A properly designed RTR application must handle all the possible messages and events that it may receive. To make this task easier Handler classes are provided, RTRServerMessageHandler and RTRServerEventHandler. These two classes provide a separate method for each potential message and event that an application may receive. The methods provide a default implementation for the application.

   Most applications will benefit from using the default handlers. Using these handlers simplifies your design by allowing you to derive your own handlers from the default handlers and override only the messages and events which are of interest to your application. The messages and events, which are not overridden, are processed using the default implementation supplied in the base class.

   Review: The Receive() method on a Transaction Controller returns an object derived from RTRData. This may be a Message or Event sent by the application or RTR itself. To process this unknown message or event the application simply needs to call the Dispatch() method on the RTRData derived object.

In rare situations an application may decide that it does not wish to use handlers. Unless a handler is registered with the Transaction controller it will not be used. In this case calling Dispatch would return an error.

4. Determine if your application should derive from RTRClassFactory.

   When the Receive() method of a Transaction Controller is called RTR needs to create an object derived from RTRData to hold the data being received. More specifically, it creates one of the following objects:

   - RTRApplicationMessage
   - RTRApplictionEvent
   - RTRMessage
   - RTREvent

   An application may wish to have its own object returned when Receive is called. This is easily achieved by registering its own class factory object, which is derived from RTRClassFactory. RTR will call the appropriate method in the class factory and the application may return its own class, which is derived from the base class being created. This allows the application great flexibility when processing incoming data.

   Many applications will find it very valuable to derive their own class(es) from RTRApplicationMessage and return instances of this class from their custom class factory.

   RTR calls the CreateRTRApplicationMessage() method of the class factory with the data being received. This allows the application to parse the data before it is received and return the correct object for the application. For example, the sample application looks at the application message being received, determines if it is receiving a book or magazine and returns an instance of the correct object.

   In some circumstances an application may always pass only one type of data, in this case it may chose not to register a class factory.

## 2.2. Implementation Steps

The steps described in this section for client and server applications implement a polling client application and an event-driven server application. These steps include code examples that are part of the book processing sample application for ordering books and magazines.

While the steps in this section are representative of client and server applications, there are design alternatives. A sampling of these design alternatives is provided in later sections of this chapter.

### 2.2.1. Implementing a Server

To implement a server application, you:

- Create an environment for the application to run. This includes starting RTR, creating a facility, defining one or more key segments and creating one or more partitions.
- Create an RTRServerTransactionController within your server code.

- Register a facility by name.
- Register a partition by name.
- Register a class derived from RTRClassFactory [optional].
- Register a class derived from RTRServerMessageHandler [optional].
- Register a class derived from RTRServerEventHandler [optional].
- Create the control loop that includes receive and dispatch methods.
- Accept the transaction when your business logic succeeds.
- Acknowledge the outcome of the transaction.

For example, the typical steps for implementing a server are the following:

1. Create an environment for the application to run by registering a partition for the server.
   - Create an RTRKeySegment class. For example, the following sample creates an RTRKeySegment for all ASCII values between "A" and "z":

     ```
     // Create a partition that processes ISBN numbers in the
     // range 0-99
         unsigned int low = 0;
         unsigned int max = 99;
         RTRKeySegment KeyZeroTo99(           rtr_keyseg_unsigned,
                                              sizeof(int),
                                              0,
                                              &low,
                                              &max );
     ```

   - Create an RTR partition with the above KeySegment. The following example includes constants for the names of the partition and facility.

     ```
     RTRPartitionManager PartitionManager;
     sStatus = PartitionManager.CreateBackendPartition( ABCPartition1,
     ABCFacility,
     KeyZeroTo99,false,true,false);
     print_status_on_failure(sStatus);
     ```

     While the above example shows only the RTR_STS_OK return value, typical applications must check for other status returns.

2. Instantiate the RTRServerEventHandler and RTRServerMessageHandler classes.

   ```
    SimpleServerEventHandler *pEventHandler = new
                            SimpleServerEventHandler();
    SimpleServerMessageHandler *pMessageHandler = new
                            SimpleServerMessageHandler();
   ```

3. Create an RTRServerTransactionController to receive incoming messages and events from a client.

   ```
     RTRServerTransactionController *pTransaction = new
                                 RTRServerTransactionController();
   ```

4. Register the facility, partition and both handlers with the transaction controller.

   ```
     ....sStatus = pTransaction->RegisterFacility( pFacilityName );
   ```

```
        assert(RTR_STS_OK == sStatus);
        ....sStatus = pTransaction->RegisterPartition( pPartitionName );
        assert(RTR_STS_OK == sStatus);
            sStatus = pTransaction->RegisterHandlers( pMessageHandler,
                                                      pEventHandler );
        assert(RTR_STS_OK == sStatus);
```

5. Create a RTRData pointer. This pointer is assigned a pointer to a message or event when RTRSoerverTransactionController::Receive is called.

```
        RTRData *pDataReceived = NULL;
```

6. Create a control loop to continually receive messages and dispatch them to the message and event handlers.

```
    while (true)
        {
        sStatus = pTransaction->Receive(pDataReceived);
        print_status_on_failure(sStatus);
        sStatus = pDataReceived->Dispatch();
        print_status_on_failure(sStatus);
        }
```

7. Accept the Transaction when your business logic succeeds. When the server has successfully finished its work, tell RTR that it is willing to accept the transaction.

```
        RTRServerTransactionController * pController;
        pController->AcceptTransaction();
```

   Note the default behavior supplied by the OnPrepareTransaction method of the RTRServerMessage handler is to call AcceptTransaction on behalf of the application.

   The sample application overrides this default behavior to reject the transaction if the order could not be processed.

```
        void ABCSHandlers::OnPrepareTransaction( RTRMessage *pRTRMessage,
        RTRServerTransactionController *pController )
        // Check to see if anything has gone wrong. If so, reject the
        // transaction, otherwise accept it.

            if (true == m_bVoteToAccept)
            {
                  pController->AcceptTransaction();
            }
            else
            {
                 pController->RejectTransaction();
            }
```

8. Acknowledge the outcome of the transaction. A server must tell RTR that it has received the outcome of the transaction. This explicitly tells RTR that it is ok for this Transaction Controller to process the next transaction.

   The default behavior in RTRServerMessageHandler::OnAccepted is to acknowledge the outcome of the transaction.

```
        void ABCSHandlers::OnAccepted( RTRMessage *pRTRMessage,
        RTRServerTransactionController *pController )
        {    pController->AcknowledgeTransactionOutcome();
             return;
        }
```

## 2.2.2. Implementing a Client

To implement a client application, you:

- Create an RTRClientTransactionController.
- Register a facility
- Send a message
- Accept the transaction

In more detail:

1. Create a ClientTransactionController to receive the incoming messages and events from a server.

   ```
   RTRClientTransactionController *pTransaction = new
                                   RTRClientTransactionController();
   ```

2. Register the facility with the TransactionController object.

   ```
   sStatus = RegisterFacility(ABCFacility);
           print_status_on_failure(sStatus);
   if(RTR_STS_OK == sStatus)
           {
              m_bRegistered = true;
           }
   ```

3. Create an RTRApplicationMessage derived class that adds to the Data object the information that is to be sent to the server. Usually the data is added within the derived class by calling RTRStream::WriteToStream.

   ```
       class MyApplicationMessage : public RTRApplicationMessage
       MyApplicationMessage *pMessage1 = new  MyApplicationMessage()
   ```

4. Send a message to the server.

   ```
       sStatus = pTransaction->SendApplicationMessage(pMessage1);
       print_status_on_failure(sStatus);
   ```

5. Accept the transaction. When the application has successfully finished the transaction, the client tells RTR that it votes to accept the transaction.

   ```
        pTransaction->AcceptTransaction();
   ```

   The client application's business logic operates between sending a first message to the server and accepting the transaction (Step 5).

## 2.2.3. Implementation Example

The following server application example shows the steps in setting up the infrastructure for running an application to process transactional requests. There is a server.h and a server.cpp file.

In the server.h file, after including the necessary header files and defining pointers to RTR facility and partition names, the business class, deriving from the RTRServerTransactionController class is defined. This includes declaring a transaction controller constructor and destructor.

```
#include <iostream.h>
#include <rtrapi.h>
#include <assert.h>

const     char *ABCFacility = "MyFacility";
const     char *ABCPartition = "MyPartition";

class SRVTransactionController: public
RTRServerTransactionController
{
public:
   SRVTransactionController();
   ~SRVTransactionController();
private:
};
SRVTransactionController::SRVTransactionController()
{
cout << "In Server Transaction Controller constructor " << endl;
}
SRVTransactionController::~SRVTransactionController()
{
cout << "In Server Transaction Controller destructor " << endl;
}
```

The server message and event handlers are then declared and defined. MySRVMessageHandler derives from RTRServerMessageHandler and MySRVEventHandler derives from RTRServerEventHandler. In this example, the RTRServerMessageHandler methods OnAccepted, OnPrepareTransaction and the RTRServerEventHandler method OnServerIsPrimary are overridden. Both handler classes also define constructors and destructors.

```
class MySRVMessageHandler: public RTRServerMessageHandler
{
public:
   MySRVMessageHandler();
   ~MySRVMessageHandler();
   rtr_status_t  OnPrepareTransaction( RTRMessage *pmyMsg,
                            RTRServerTransactionController *pTC);
   rtr_status_t  OnAccepted(  RTRMessage *pmyMsg,
                           RTRServerTransactionController *pTC);
private:
};
MySRVMessageHandler::MySRVMessageHandler()
{
}
MySRVMessageHandler::~MySRVMessageHandler()
{
}
rtr_status_t    MySRVMessageHandler::OnPrepareTransaction(
                            RTRMessage *pmyMsg,
                            RTRServerTransactionController *pTC)
{
    cout << "prepare txn " << endl;
    pTC->AcceptTransaction();
    return RTR_STS_OK;
}
rtr_status_t    MySRVMessageHandler::OnAccepted(
                            RTRMessage *pmyMsg,
                            RTRServerTransactionController *pTC)
{
    cout << "accepted txn " << endl;
    pTC->AcknowledgeTransactionOutcome();
    return RTR_STS_OK;
}
class MySRVEventHandler: public RTRServerEventHandler
{
public:
   MySRVEventHandler();
   ~MySRVEventHandler();
   rtr_status_t OnServerIsPrimary( RTREvent *pRTREvent,
                           RTRServerTransactionController *pTC );
};
MySRVEventHandler::MySRVEventHandler()
{
}
MySRVEventHandler::~MySRVEventHandler()
{
}
MySRVEventHandler::OnServerIsPrimary( RTREvent *pRTREvent,
                              RTRServerTransactionController *pTC )
{
   cout << "This server is primary " <<endl;
   return RTR_STS_OK;
}
```

In the server.cpp file, after including the server.h file and instantiating the SRVTransactionController class (myTC), the management class steps for setting up the RTR infrastructure take place. These steps create the RTR environment for client and server transactional messaging. This includes:

- Starting RTR (myRTR.Start)
- Creating a journal (myRTR.CreateJournal(true))
- Creating a facility (myFac)
- Defining a partition (myPartition)
- Defining a key segment (mySegment)
- Creating a server partition (myPartition.CreateBackendPartition)

```
       #include  "srv.h"
       int main(void)
       {
           rtr_status_t sStatus;
           SRVTransactionController myTC;
        // start rtr
           RTR    myRTR;
           sStatus = myRTR.Start();
           cout << myRTR.GetErrorText(sStatus) << endl;
        // create journal
           sStatus = myRTR.CreateJournal(true);
```

```
            cout << myRTR.GetErrorText(sStatus) << endl;
        // create facility
            RTRFacilityManager myFac;
        // get nodes names for facility
            char *pszBackendNodes = "dejavu";
            char *pszRouterNodes = "dejavu";
            char *pszFrontendNodes = "dejavu";
            char *nodename = "dejavu";
            sStatus = myFac.CreateFacility(ABCFacility,pszRouterNodes,
                       pszFrontendNodes,pszBackendNodes,false,false);
            cout << myRTR.GetErrorText(sStatus) << endl;
            RTRPartitionManager myPartition;
            char   *low="A";
            char   *high="Z";
            RTRKeySegment mySegment(rtr_keyseg_string,1,
                                            0,low,high);
            sStatus = myPartition.CreateBackendPartition(ABCPartition,
                                 ABCFacility,mySegment,false,true,true);
            cout << myRTR.GetErrorText(sStatus) << endl;
```

Then register the facility, partition and handler classes and instantiate a pointer to a data object (*myData).

```
    sStatus = myTC.RegisterFacility(ABCFacility);
    cout << myRTR.GetErrorText(sStatus) << endl;
    sStatus = myTC.RegisterPartition(ABCPartition);
    cout << myRTR.GetErrorText(sStatus) << endl;
    MySRVMessageHandler myHandler;
    MySRVEventHandler myEventHandler;
    myTC.RegisterHandlers(&myHandler,&myEventHandler);
    RTRData *myData;
```

Finally, create control loop logic with the Receive and Dispatch methods.

```
    while(true)
        {
        sStatus = myTC.Receive(&myData);
        cout << "message received " << myRTR.GetErrorText(sStatus) <<
        endl;
        if ( sStatus != RTR_STS_OK)
        {
            assert(false);
        }
        sStatus = myData->Dispatch();
        cout << myRTR.GetErrorText(sStatus) << endl;
        delete myData;
        }
        cout << "hey I am done" <<endl;
        return 0;
    }
```

## 2.3. Sample Application Walkthrough

This section uses the sample application included in the RTR kit as an example of implementing both a client and a server application using the C++ foundation classes.

The sample application is a simple client and server for ordering books and magazines.

The client takes orders and creates the corresponding Book or Magazine object. This object is told to serialize itself (write its state to a stream) and the client then sends the serialized object to a server.

The server application creates and registers two partitions. These partitions represent orders with ISBN numbers from 1-99 and 100-199. The server will register a custom class factory to peek at the object, which it is about to receive and determine its type, book or magazine. When the object has been created by the class factory and returned to the application the server will tell the object to deserialize itself and then to process itself. Processing means to carry out the business logic of buying the book or magazine.

The sample application demonstrates the following features:

- Serializing and Deserializing an application-defined object with RTR.
- Creating multiple partitions, each with a different key segment, including ABCPartition1 and ABCPartition2.
- Using default handlers for RTR messages, for example, default calls to methods such as OnAccepted and OnRejected.
- Using default handlers for RTR events, for example, default calls to methods such as OnServerIsPrimary.
- Dispatching RTRData-derived objects, for example, pOrder->Dispatch().

In this sample there are three server classes and one client class. Each class is declared in its own .h file and implemented in a .cpp file.

The server classes are:

- ABCOrderProcessor: this class derives from RTRServerTransactionController.
- ABCSClassFactory: this class derives from RTRClassFactory.
- ABCSHandlers: this class derives from RTRServerEventHandler and RTRServerMessageHandler.

The client classes are:

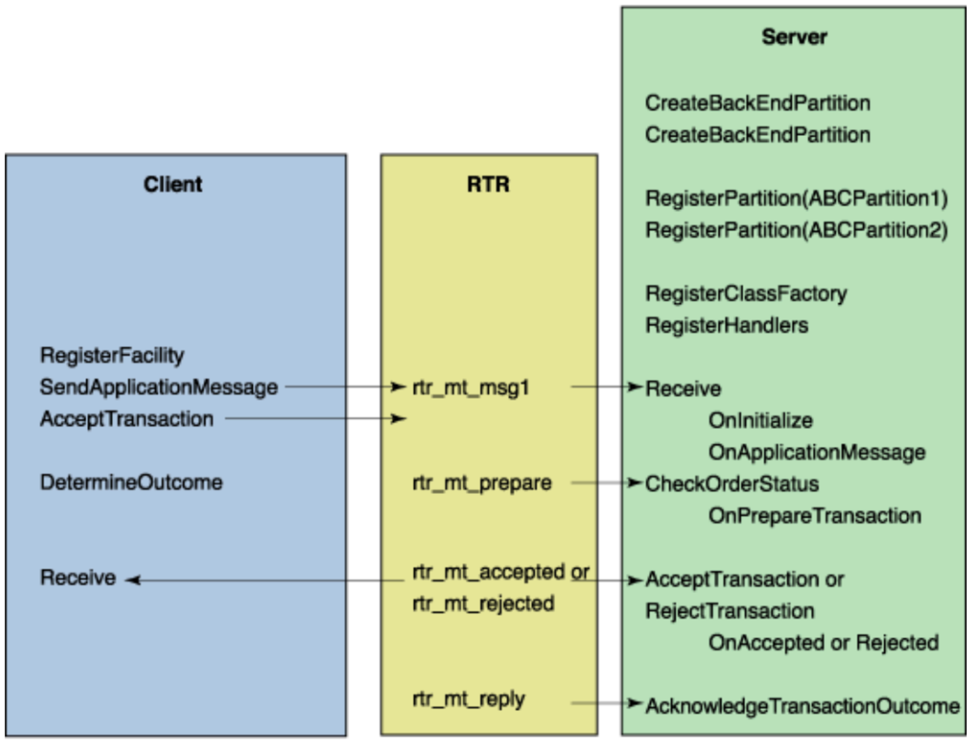
- ABCOrderTaker: this class derives from RTRClientTransactionController.
- ABCCHandlers this class derives from RTRClientEventHandler and RTRClientMessageHandler.

There are three common data classes:

- ABCOrder: this class derives from RTRApplicationMessage.
- ABCBook: this class derives from ABCOrder.
- ABCMagazine: this class derives from ABCOrder.

*Figure 2.1, "Sample Application Messaging"* illustrates the messaging between the sample client and server applications.

**Figure 2.1. Sample Application Messaging**



## 2.3.1. Deriving from Base Classes in the Sample Application

This section provides examples of creating derived classes in the book- ordering sample application for implementing additional functionality in client and server application code by:

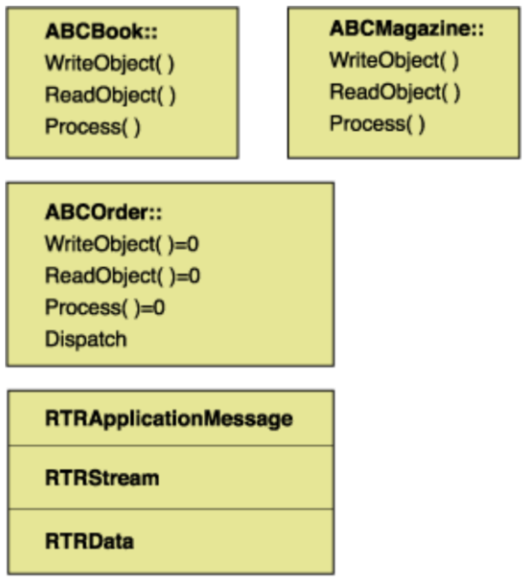
- Adding functionality to RTRData-derived data objects
- Encapsulating data
- Examining RTRData objects

## 2.3.2. Adding Functionality to Data Objects

You can add functionality to an RTRData object without changing any code in the Message or Event handlers or the Receive loop, by deriving a class from RTRData.

*Figure 2.2, "Adding Functionality to RTRData"* illustrates the base class relationships to the ABCOrder data class. This class adds functionality to the RTRApplicationMessage class by defining three additional methods.

**Figure 2.2. Adding Functionality to RTRData**

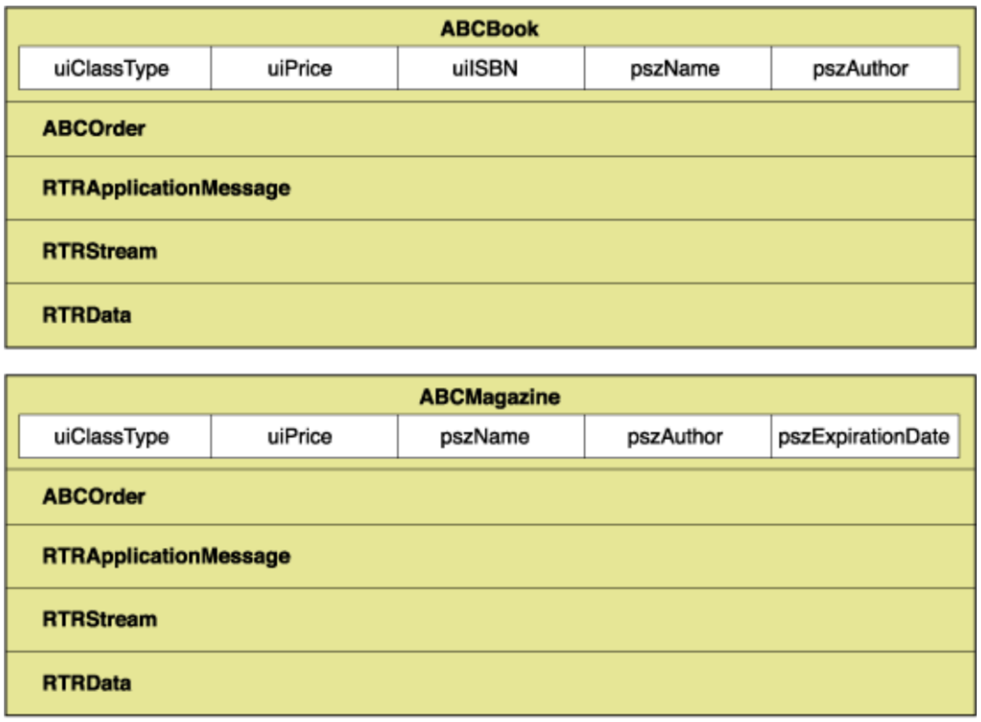


For example, a book is represented as an ABCBook object with its inherited Dispatch method from ABCOrder. This class overrides the WriteObject, ReadObject, and Process methods. A magazine is represented as an ABCMagazine object with overridden WriteObject ReadObject, and Process methods, and the Dispatch method inherited from ABCOrder.

## 2.3.3. Encapsulating Data with RTRData

The following example illustrates the protocol class that encapsulates application-level data with an RTRData-derived class. In this sample application, two kinds of orders are processed by the server application, book orders and magazine orders. An order is defined as an ABCOrder object which derives from RTRApplicationMessage. All data sent between the client and server applications represents either a magazine order or a book order. As *Figure 2.3, "Encapsulating Data with RTRData"* shows, there are two kinds or orders, book orders and magazine orders. This information is represented in a buffer organized for sending to the server from the client.

**Figure 2.3. Encapsulating Data with RTRData**



These two classes have been derived from the application's base class, ABCOrder. Book and Magazines are kinds of Orders. The order class tells its derived classes when to serialize their data. When this happens, the data in stored in the RTRData class via the methods of the RTRStream class.

When the client application is to make a request, the user enters the data for the fields illustrated above. The client application then stores this information in the corresponding book or magazine object and sends it to the server using SendOrder. The server then calls Receive to obtain the Book or Magazine order. Note that a Book (or magazine) is an RTRApplicationMessage.

In addition to RTRApplicationMessage data objects, three other kinds of RTR data can exist in the RTR application:

- RTRApplicationEvent
- RTRMessage
- RTREvent

The application must be set up to handle these data classes, even if an application chooses to ignore them. In the sample application, if an order is an RTRApplicationMessage, then the object (an order) is processed by the Dispatch method. If the data is an RTRMessage or an RTREvent, then default handling occurs, and the event and message handler methods are called. The default Dispatch methods then execute, as each RTRData-derived data class has its own Dispatch method.

When ABCOrderProcessor calls its derived Receive method, one of the four types of data objects is assigned. The server can receive RTRMessage and RTREvent or can overwrite code in the class factory class to receive book or magazine orders. The class factory returns a pointer to incoming data (as an RTRData pointer) and knows what kind of object to return.

## 2.3.4. Examining RTRData Objects

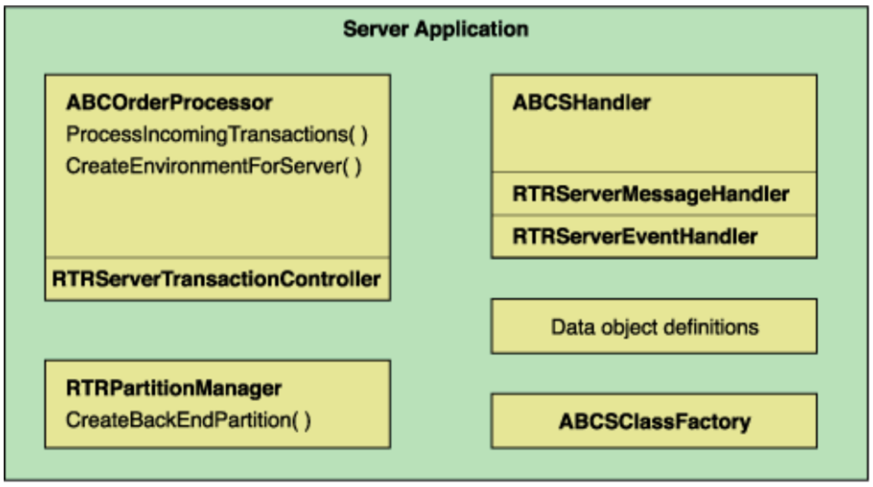
You can check the contents of an RTRData object by calling any RTRData method such as IsMessage. The following example from the client application ABCOrderTaker illustrates how an application can retrieve and use the message from an RTRData derived object.

```
while (OrderBeingProcessed == eTxnResult)
{
sStatus = Receive(&pResult);
print_status_on_failure(sStatus);
if ( true == pResult->IsRTRMessage())
{
// Check to see if we have a status for the transaction.
// rtr_mt_accepted = Server successfully processed our request.
// rtr_mt_rejected = Server could not process our request.
sStatus = ((RTRMessage*)pResult)->GetMessageType(mtMessageType);
print_status_on_failure(sStatus);
if (rtr_mt_accepted == mtMessageType) return eTxnResult =
OrderSucceeded;
if (rtr_mt_rejected == mtMessageType) return eTxnResult =
OrderFailed;
      }
    }
    return eTxnResult;
```

## 2.3.5. Sample Server Application

The following figure illustrates the objects within the ABCOrderProcessor server application. Each of the four server classes derives from the associated base classes.

**Figure 2.4. Sample Server Application**



The implementation of ABCOrderProcessor uses default construction and destruction and then follows the steps described earlier in this chapter to create a server application.

### Processing Method

The sample server application implements the event-driven processing model in ProcessIncomingTransactions. Implementation of ProcessIncomingTransactions is as follows:

1. Create a transaction controller to receive incoming messages and events from a client.

2. Create an environment where the server can run, then Register with RTR the partitions, handler classes, class factory and objects using the transaction controller:

```
    sStatus = RegisterFacility(ABCFacility);
    print_status_on_failure(sStatus);
    // ABC Partition
    sStatus = RegisterPartition(ABCPartition1);
    print_status_on_failure(sStatus);
    sStatus = RegisterPartition(ABCPartition2);
    print_status_on_failure(sStatus);
    // ABC Class Factory
    sStatus = RegisterClassFactory(&m_ClassFactory);
    print_status_on_failure(sStatus);
    // ABC Handlers
    sStatus = RegisterHandlers(&m_rtrHandlers,&m_rtrHandlers);
    print_status_on_failure(sStatus);
    return;
// Create the environment :
void ABCOrderProcessor::CreateRTREnvironment()
{
       rtr_status_t sStatus;
        // If RTR is not already started then start it now.
    StartRTR();
        // Create a Facility if not already created.
    CreateFacility();
// Create a partition that processes ISBN numbers in the range 0 -
// 99
        unsigned int low = 0;
        unsigned int max = 99;
    RTRKeySegment KeyZeroTo99(  rtr_keyseg_unsigned,
                                sizeof(int),
                                0,
                                &low,
                                &max );
RTRPartitionManager PartitionManager;
sStatus = PartitionManager.CreateBackendPartition( ABCPartition1,
                                                   ABCFacility,
                                                   KeyZeroTo99,
                                                   false,
                                                   true,
                                                   false);
        print_status_on_failure(sStatus);
// Create a partition that processes ISBN numbers in the range 100 -
// 199
        low = 100;
        max = 199;
    RTRKeySegment Key100To199(  rtr_keyseg_unsigned,
                                sizeof(int),
                                0,
                                &low,
                                &max );
    sStatus = PartitionManager.CreateBackendPartition(  ABCPartition2,
                                                        ABCFacility,
                                                        Key100To199,
                                                        false,
                                                        true,
                                                        false);
        print_status_on_failure(sStatus);
    }
```

3. Instantiate the handler class ABCSHandlers.

4. Create an RTRData object to hold each incoming message or event. This object will be reused.

```
    // Start processing orders.
    abc_status sStatus;
    RTRData *pOrder = NULL;
```

5. Continually loop, receiving messages and dispatching them to the handlers:

```
    while (true)
        {
    // Receive an Order
            sStatus = Receive(&pOrder);
               print_status_on_failure(sStatus);
            if(ABC_STS_SUCCESS != sStatus) break;
    // if we can't get an Order then stop processing.
    // Dispatch the Order to be processed
            sStatus = pOrder->Dispatch;
            print_status_on_failure(sStatus);
    // Exception handling:
    // Check to see if there were any problems processing the order.
    // If so, let the handler know to reject this transaction when
    // asked to vote.
           CheckOrderStatus(sStatus);
    ...
    }
```

6. Check to see if there were any problems processing the order. If so, let the handler know that this transaction is to be rejected when asked to vote.

```
    void ABCOrderProcessor::CheckOrderStatus (abc_status sStatus)
    if (sStatus == ABC_STS_ORDERNOTPROCESSED)
        {
    // Let the handler know that the current txn should be rejected
            GetHandler()->OnABCOrderNotProcessed();
        };
```

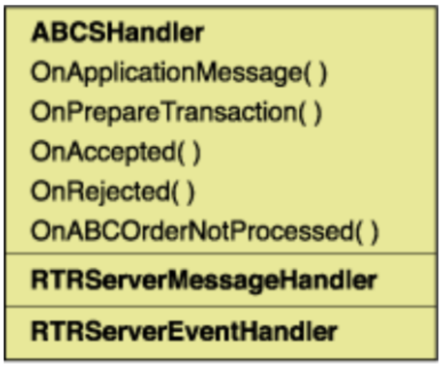
7. Cleanup. Delete this order that was allocated by the class factory. In the sample application, the class factory returns a separate instance of an order each time it is called.

```
     delete pOrder;
```

## Server Message and Event Handler

The ABCOrderProcessor server application includes the derived class ABCSHandler for event-driven message and event handling. As *Figure 2.5, "Figure 2-5: Sample Server-Handler-Derived Class"* illustrates, it combines both handlers into one handler class by deriving from both RTRServerEventHandler and RTRServerMessageHandler classes.

**Figure 2.5. Figure 2-5: Sample Server-Handler-Derived Class**



The ABCSHandler class overrides the following four handler methods:

- OnApplicationMessage
- OnPrepareTransaction
- OnAccepted
- OnRejected

It uses the default handler methods for:

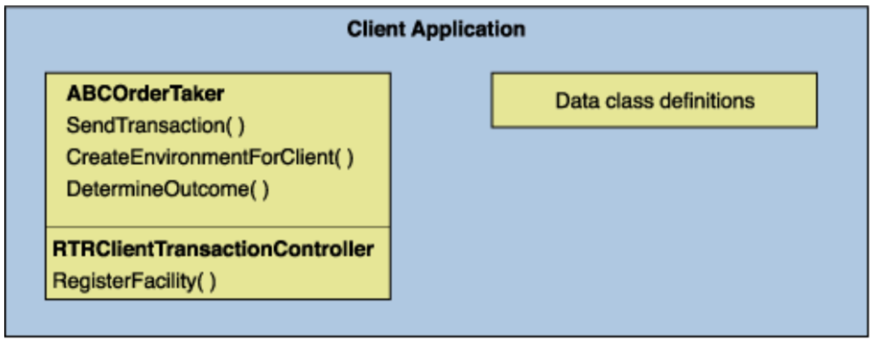
- OnInitialize
- OnUncertainTransaction

In addition to the above over-ridden methods, it also contains an application-defined method to handle exceptions, OnABCOrderNotProcessed().

## 2.3.6. Sample Client Application

*Figure 2.6, "Figure 2-6: Sample Client Application"* illustrates the ABCOrderTaker sample application. This example uses the polling receive processing model, not message or event handlers.

**Figure 2.6. Figure 2-6: Sample Client Application**



The client application header file ABCOrderTaker.h declares the interface for the ABCOrderTaker class. The file ABCOrderTaker.cpp provides the implementation.

In addition to the default constructor and destructor, there are two methods within class ABCOrderTaker:

- SendOrder

- Register
- DetermineOutcome

### SendOrder

1. Create the environment where ABCOrderTaker is to run by registering a facility:

   ```
   sStatus = RegisterFacility(ABCFacility);
          print_status_on_failure(sStatus);
          if(RTR_STS_OK == sStatus)
          {
              m_bRegistered = true;
          }
   ```

2. Create a Transaction Controller to receive incoming messages and events from a client.

   ```
    ABCOrderTaker OrderTaker;
   ```

3. Send the server a message:

   ```
   sStatus = SendApplicationMessage(pOrder);
       print_status_on_failure(sStatus);
   ```

4. Since we have successfully finished our work, tell RTR that we are willing to accept the transaction. Let RTR know that this is the object being sent and that we are done with our work:

   ```
   sStatus = AcceptTransaction();
   print_status_on_failure(sStatus);
   ```

5. Determine if the server successfully processed the request

   ```
   eTxnResult = DetermineOutcome();
   return true;
   ```

## 2.4. RTR Applications in a Multiplatform Environment

Applications using RTR in a multiplatform (mixed endian) environment with non-string application data must tell RTR how to marshall the data both for the destination of the application data being sent and the application data itself. This description is supplied as the `rtr_const_msgfmt_t` argument to:

- RTRClientTransactionController::SendApplicationMessage
- RTRServerTransactionController::SendApplicationMessage
- RTRClientTransactionConrtroller::SendApplicationEvent
- RTRServerTransactionController::SendApplicationEvent

The default (that is, when `rtr_const_msgfmt_t` is supplied) is to assume the application message is string data.

### 2.4.1. Defining a Message Format

The `rtr_const_msgfmt_t` string is a null-terminated ASCII string consisting of a number of field-format specifiers: `[field-format-specifier, ...]`

The field-format specifier is defined as: `%[dimension]field-type`

where:

| Field | Description | Meaning |
|---|---|---|
| % | Indicates a new field description is starting. | |
| dimension | Is an optional integer denoting array cardinality (default 1). | |
| field-type | Is one of the following codes: | |
| | UB | 8 bit unsigned byte |
| | SB | 8 bit signed byte |
| | UW | 16 bit unsigned |
| | SW | 16 bit signed |
| | UL | 32 bit unsigned |
| | SL | 32 bit signed |
| | C | 8 bit signed char |
| | UC | 8 bit unsigned char |
| | B | boolean |

For example, consider a data object containing the following:

```
unsigned int m_uiISBN;
unsigned int m_uiPrice;
char m_szTitle[ABCMAX_STRING_LEN];
char m_szAuthor[ABCMAX_STRING_LEN];
```

The `rtr_const_msgfmt_t` for this object could be ("%UL%SL%12C%12C").

The transparent data-type conversion of RTR does not support certain conversions (for example, floating point). These should be converted to another format, such as character string.

# Chapter 3. Application Classes

The RTR C++ API major application classes are:

- Server classes
- Client classes
- Data classes

This chapter describes these major classes in the above order. Within each major class, each class is described in alphabetical order. Within each class, all of its inherited methods are described in alphabetical order.

RTRData-derived classes are used for passing data between client and server applications.

An application can send two data categories:

- Application-defined messages
- Application-defined events.

An application can receive four data categories:

- Application-defined messages
- Application-defined events
- RTR-defined messages
- RTR-defined events

The four RTRData-derived classes are:

- RTRApplicationMessage
- RTRApplicationEvent
- RTRMessage
- RTREvent

An RTRClassFactory object creates these four classes. The RTR application does not need to register a class factory with a transaction controller, but if it does, it can customize how the objects are allocated including allocating a class that is derived from any of the four data classes above.

## 3.1. Server Classes

The server application classes are:

- RTRServerEventHandler
- RTRServerMessageHandler
- RTRServerTransactionController

- RTRServerTransactionProperties

## 3.2. RTRServerEventHandler

This class defines event handlers for all potential events that an RTR server application can receive. Each handler has a default behavior. Applications should override those member functions for which they want to perform application-specific processing.

### Note

Applications can extend this class by deriving from it and adding their own application-level event handlers.

For further information see RTRData::Dispatch().

### Construction

| Method | Description |
|---|---|
| RTRServerEventHandler() | Constructor. |
| ~RTRServerEventHandler() | Destructor. |

### Operations

| Method | Description |
|---|---|
| OnApplicationEvent(RTRApplicationEvent, RTRServerTransactionController) | The application has generated an event for the server. |
| OnBackendGainedLinkToRouter(RTREvent, RTRServerTransactionController) | Default handler for the event where a backend link to the current router has been established. |
| OnBackendLostLinkToRouter(RTREvent, RTRServerTransactionController) | Default handler for the event where the backend link to the current router has been lost. |
| OnFacilityDead(RTREvent,<br><br>RTRServerTransactionController) | Default handler for the event where the facility is no longer operational. |
| OnFacilityReady(RTREvent,<br><br>RTRServerTransactionController) | Default handler for the event where the facility has become operational. |
| OnServerGainedShadow(RTREvent,<br><br>RTRServerTransactionController,<br><br>(rtr_const_parnam_t )) | The server gained its shadow partner. |
| OnServerIsPrimary(RTREvent,<br><br>RTRServerTransactionController) | Default handler for the event where the server is in primary mode. |
| OnServerIsSecondary(RTREvent,<br><br>RTRServerTransactionController) | Default handler for the event where the server is in secondary mode. |
| OnServerIsStandby(RTREvent, | Default handler for the event where the server is in standby mode. |

| Method | Description |
|---|---|
| RTRServerTransactionController) | |
| OnServerLostShadow(RTREvent,<br><br>RTRServerTransactionController,<br><br>(rtr_const_parnam_t )) | The server lost its shadow partner. |
| OnServerRecoveryComplete(RTREvent, RTRServerTransactionController) | Default handler for the event where the server has completed recovery. |

## OnApplicationEvent()

OnApplicationEvent() — RTRServerMessageHandler::OnApplicationEvent();

### Prototype

```
virtual rtr_status_t OnApplicationEvent (RTRApplicationEvent
                                         *pRTRApplicationEvent,
                                         RTRServerTransactionController
                                         *pController)
{
  return RTR_STS_OK;
}
```

### Parameters

## pRTRApplicationEvent

Pointer to an RTRApplicationEvent object that describes the message which is being processed.

## pController

Pointer to the transaction controller within which this event was received.

### Description

The pRTRApplicationEvent parameter contains an application event sent to it by the client application.

Override this method if your application is to receive an indication that this event has occurred.

The default behavior is that the handler dismisses the notification.

### Example

```
void CombinationOrderProcessor::OnApplicationEvent ( RTRApplicationEvent
*pApplicationEvent, RTRServerTransactionController *pController)
{
// This handler is called by RTR when the client has sent an event.
}
```

## OnBackendGainedLinkToRouter()

OnBackendGainedLinkToRouter() — RTRServerEventHandler::OnBackendGainedLinkToRouter();

### Prototype

```
virtual rtr_status_t OnBackendGainedLinkToRouter(RTREvent * pRTREvent,
                                    RTRServerTransactionController
                                    *pController)
{
return RTR_STS_OK;
}
```

### Parameters

## pRTREvent

Pointer to an RTREvent object that describes the RTR-generated event being processed.

## pController

Pointer to the transaction controller within which this event was received.

### Description

This method provides the default handler for the event where a backend link to the current router has been established.

The server application is receiving an RTR-generated event. RTREvent contains the RTR-defined event number RTR_EVTNUM_BERTRGAIN (104) and any associated data.

Override this method if your application is to receive an indication that this event has occurred.

### Example

```
void RTRServerEventHandler::OnBackendGainedLinkToRouter( RTREvent
           *pEvent, RTRServerTransactionController *pController )
{
}
```

## OnBackendLostLinkToRouter()

OnBackendLostLinkToRouter() — RTRServerEventHandler::OnBackendLostLinkToRouter();

### Prototype

```
virtual rtr_status_t OnBackendLostLinktToRouter(RTREvent * pRTREvent,
                         RTRServerTransactionController *pController)
{
return RTR_STS_OK;
}
```

### Parameters

## pRTREvent

Pointer to an RTREvent object that describes the RTR-generated event being processed.

## pController

Pointer to the transaction controller within which this event was received.

### Description

This method provides the default handler for the event where the backend link to the current router has been lost.

The server application is receiving an RTR-generated event. RTREvent contains the RTR-defined event number RTR_EVTNUM_BERTRLOSS (105) and any associated data.

Override this method if your application is to receive an indication that this event has occurred.

### Example

```
void RTRServerEventHandler::OnBackendLostLinkToRouter( RTREvent
           *pEvent, RTRServerTransactionController *pController )
{
}
```

## OnFacilityDead()

OnFacilityDead() — RTRServerEventHandler::OnFacilityDead();

### Prototype

```
virtual rtr_status_t OnFacilityDead(RTREvent * pRTREvent,
                               RTRServerTransactionController *pController)
{
 return RTR_ST_OK;
}
```

### Parameters

## pRTREvent

Pointer to an RTREvent object that describes the RTR-generated event being processed.

## pController

Pointer to the transaction controller within which this event was received.

### Description

This method provides the default handler for the event where the facility is no longer operational.

The server application is receiving an RTR-generated event. RTREvent contains the RTR-defined event number RTR_EVTNUM_FACDEAD (97) and any associated data.

Override this method if your application wants to receive an indication that this event has occurred.

### Example

```
void RTRServerEventHandler::OnFacilityDead( RTREvent *pEvent,
RTRServerTransactionController *pController )
{
}
```

## OnFacilityReady()

OnFacilityReady() — RTRServerEventHandler::OnFacilityReady();

### Prototype

```
virtual rtr_status_t OnFacilityReady(RTREvent * pRTREvent,
                                 RTRServerTransactionController *pController)
{
 return RTR_STS_OK;
}
```

### Parameters

## pRTREvent

Pointer to an RTREvent object that describes the RTR-generated event being processed.

## pController

Pointer to the transaction controller within which this event was received.

### Description

This method provides the default handler for the event where the facility has become operational.

The server application is receiving an RTR-generated event. RTREvent contains the RTR-defined event number RTR_EVTNUM_FACREADY (96) and any associated data.

Override this method if your application is to receive an indication that this event has occurred.

### Example

```
void RTRServerEventHandler::OnFacilityReady( RTREvent *pEvent,
RTRServerTransactionController *pController )
{
}
```

## OnRouterGainedLinkToFrontend()

OnRouterGainedLinkToFrontend() — RTRServerEventHandler::OnFrontendGainedLinkToRouter();

### Prototype

```
virtual rtr_status_t OnRouterGainedLinkToFrontend(RTREvent * pRTREvent,
                             RTRServerTransactionController *pController)
{
 return RTR_STS_OK;
}
```

### Parameters

### pRTREvent

Pointer to an RTREvent object that describes the RTR-generated event being processed.

### pController

Pointer to the transaction controller within which this event was received.

### Description

This method provides the default handler for the event where the router gained link to a frontend.

The server application is receiving an RTR-generated event. RTREvent contains the RTR-defined event number RTR_EVTNUM_RTRFEGAIN (106) and any associated data.

Override this method if your application is to receive an indication that this event has occurred.

### Example

```
void RTRServerEventHandler::OnRouterGainedLinkToFrontend( RTREvent *pEvent,
RTRServerTransactionController *pController )
{
}
```

## OnRouterLostLinkToFrontend()

OnRouterLostLinkToFrontend() — RTRServerEventHandler::OnRouterLostLinkToFrontend();

### Prototype

```
virtual rtr_status_t OnRouterLostLinkToFrontend(RTREvent * pRTREvent,
                            RTRServerTransactionController *pController)
{
 return RTR_STS_OK;
}
```

### Parameters

### pRTREvent

Pointer to an RTREvent object that describes the RTR-generated event being processed.

### pController

Pointer to the transaction controller within which this event was received.

### Description

This method provides the default handler for the event where the router lost link to the current frontend.

The server application is receiving an RTR-generated event. RTREvent contains the RTR-defined event number RTR_EVTNUM_RTRFELOSS (107) and any associated data.

Override this method if your application is to receive an indication that this event has occurred.

### Example

```
void RTRServerEventHandler::OnRouterLostLinkToFrontend(
                             RTREvent *pEvent,
                             RTRServerTransactionController *pController )
{
}
```

## OnServerGainedShadow()

OnServerGainedShadow() — RTRServerEventHandler::OnServerGainedShadow();

### Prototype

```
virtual rtr_status_t OnServerGainedShadow(RTREvent * pRTREvent,
                              RTRServerTransactionController *pController
                              rtr_const_parnam_t pszPartitionName)
{
 return RTR_STS_OK;
}
```

### Parameters

## pRTREvent

Pointer to an RTREvent object that describes the RTR-generated event being processed.

## pController

Pointer to the partition name on which the message or event was received.

### Description

This method provides the default handler for the event where the server gained its shadow partner.

The server application is receiving an RTR generated event. RTREvent contains the RTR defined event number RTR_EVTNUM_SRSHADOWGAIN (112) and any associated data.

Override this method if your application is to receive an indication that this event has occurred.

### Example

```
void RTRServerEventHandler::OnServerGainedShadow (*pEvent, *pController,
pszPartitionName)
{
}
```

## OnServerIsPrimary()

OnServerIsPrimary() — RTRServerEventHandler::OnServerIsPrimary();

### Prototype

```
virtual rtr_status_t OnServerIsPrimary(RTREvent * pRTREvent,
                              RTRServerTransactionController *pController
                              rtr_const_parnam_t pszPartitionName)
{
 return RTR_STS_OK;
}
```

### Parameters

## pRTREvent

Pointer to an RTREvent object that describes the RTR-generated event being processed.

## pController

Pointer to the transaction controller within which this event was received.

## pszPartitionName

Pointer to the partition name on which the message or event was received.

### Description

This method provides the default handler for the event where the server is in primary mode.

The server application is receiving an RTR-generated event. RTREvent contains the RTR-defined event number RTR_EVTNUM_SRPRIMARY (108) and any associated data.

Override this method if your application is to receive an indication that this event has occurred.

### Example

```
void RTRServerEventHandler::OnServerIsPrimary(*pEvent, *pController,
pszPartitionName )
{   }
```

## OnServerIsSecondary()

OnServerIsSecondary() — RTRServerEventHandler::OnServerIsSecondary();

### Prototype

```
virtual rtr_status_t OnServerIsSecondary(RTREvent * pRTREvent,
                               RTRServerTransactionController *pController
                               rtr_const_parnam_t pszPartitionName)
{
 return RTR_STS_OK;
}
```

### Parameters

## pRTREvent

Pointer to an RTREvent object that describes the RTR-generated event being processed.

## pController

Pointer to the transaction controller within which this event was received.

## pszPartitionName

Pointer to a partition name that is registered for the server transaction controller.

### Description

This method provides the default handler for the event where the server is in secondary mode.

The server application is receiving an RTR-generated event. RTREvent contains the RTR-defined event number RTR_EVTNUM_SRSECONDARY (110) and any associated data.

Override this method if your application is to receive an indication that this event has occurred.

### Example

```
void RTRServerEventHandler::OnServerIsSecondary(*pEvent,
 *pController, pszPartitionName )
{   }
```

## OnServerIsStandby()

OnServerIsStandby() — RTRServerEventHandler::OnServerIsStandby();

### Prototype

```
virtual rtr_status_t OnServerIsStandby(RTREvent * pRTREvent,
                            RTRServerTransactionController *pController
                            rtr_const_parnam_t pszPartitionName)
{
 return RTR_STS_OK;
}
```

### Parameters

## pRTREvent

Pointer to an RTREvent object that describes the RTR-generated event being processed.

## pController

Pointer to the transaction controller within which this event was received.

## pszPartitionName

Pointer to a partition name that is registered for the server transaction controller.

### Description

This method provides the default handler for the event where the server is in standby mode.

The server application is receiving an RTR-generated event. RTREvent contains the RTR-defined event number RTR_EVTNUM_SRSTANDBY (109) and any associated data.

Override this method if your application is to receive an indication that this event has occurred.

### Example

```
void RTRServerEventHandler::OnServerIsStandby(*pEvent,
 *pController, pszPartitionName )
{
}
```

## OnServerLostShadow()

OnServerLostShadow() — RTRServerEventHandler::OnServerLostShadow();

### Prototype

```
virtual rtr_status_t OnServerLostShadow(RTREvent * pRTREvent,
                             RTRServerTransactionController *pController
                             rtr_const_parnam_t pszPartitionName)
{
 return RTR_STS_OK;
}
```

### Parameters

## pRTREvent

Pointer to an RTREvent object that describes the RTR-generated event being processed.

## pController

Pointer to the transaction controller within which this event was received.

## pszPartitionName

Pointer to a partition name that is registered for the server transaction controller.

### Description

This method provides the default handler for the event where the server lost its shadow partner

The server application is receiving an RTR-generated event. RTREvent contains the RTR defined event number RTR_EVTNUM_SRSHADOWLOST (111) and any associated data.

Override this method if your application is to receive an indication that this event has occurred.

### Example

```
void RTRServerEventHandler::OnServerShadowLost(*pEvent,
                                     *pController, pszPartitionName )
{
}
```

## OnServerRecoveryComplete()

OnServerRecoveryComplete() — RTRServerEventHandler::OnServerRecoveryComplete();

### Prototype

```
virtual rtr_status_t OnServerRecoveryComplete(RTREvent * pRTREvent,
                              RTRServerTransactionController *pController
                              rtr_const_parnam_t pszPartitionName)
{
 return RTR_STS_OK;
}
```

### Parameters

## pRTREvent

Pointer to an RTREvent object that describes the RTR-generated event being processed.

## pController

Pointer to the transaction controller within which this event was received.

## pszPartitionName

Pointer to a partition name that is registered for the server transaction controller.

### Description

This method provides the default handler for the event where the server has completed recovery.

The server application is receiving an RTR-generated event. RTREvent contains the RTR-defined event number RTR_EVTNUM_SRRECOVERCMPL (113) and any associated data.

Override this method if your application is to receive an indication that this event has occurred.

### Example

```
void RTRServerEventHandler::OnServerRecoveryComplete(*pEvent, *pController,
pszPartitionName )
{    }
```

## RTRServerEventHandler()

RTRServerEventHandler() — RTRServerEventHandler::RTRServerEventHandler();

### Prototype

```
    RTRServerEventHandler();
    virtual ~RTRServerEventHandler();
```

### Return Value

None

### Parameters

None

### Description

Call this constructor to create and RTRServerEventHandler object.

### Example

```
class MySRVEventHandler: public RTRServerEventHandler
{
public:
       MySRVEventHandler();
       ~MySRVEventHandler();
    rtr_status_t OnServerIsPrimary( RTREvent *pRTREvent,
                              RTRServerTransactionController *pTC );
private:
};

MySRVEventHandler::MySRVEventHandler()
{
}
MySRVEventHandler::~MySRVEventHandler()
{
}

MySRVEventHandler::OnServerIsPrimary( RTREvent *pRTREvent,
                             RTRServerTransactionController *pTC )
{
 cout << "This server is primary " <<endl;
 return RTR_STS_OK;
}
```

## 3.3. RTRServerMessageHandler

This class defines message handlers for all potential messages that an RTR server application can receive. Each handler has a default behavior. Applications should override those member functions for which they want to perform application-specific processing.

### Note

Applications can extend this class by deriving from it and adding their own application-level message handlers.

For further information see RTRData::Dispatch().

**Table 3.1. Construction**

| Method | Description |
|---|---|
| RTRServerMessageHandler() | Constructor |
| ~RTRServerMessageHandler() | Destructor |

| Method | Description |
|---|---|
| OnAccepted(RTRMessage,<br><br>RTRServerTransactionController) | The specified transaction has been accepted by all participants. |
| OnApplicationMessage<br><br>RTRApplicationMessage,<br><br>RTRServerTransactionController | The client has sent the server this message. |
| OnInitialize(RTRApplicationMessage, RTRServerTransactionController) | A new transaction is being processed. |
| OnPrepareTransaction(RTRMessage, RTRServerTransactionController) | The specified transaction is complete (that is, all messages from the client have been received). |
| OnRejected(RTRMessage,<br><br>RTRServerTransactionController) | The specified transaction has been rejected by a participant. |
| OnUncertainTransaction<br><br>(RTRApplicationMessage,<br><br>RTRServerTransactionController) | RTR is replaying a transaction which may or may not have been completed. |

## OnAccepted()

OnAccepted() — RTRServerMessageHandler::OnAccepted();

### Prototype

```
virtual rtr_status_t OnAccepted(RTRMessage *pRTRMessage,
                          RTRServerTransactionController *pController)
{
        pController->AcknowledgeTransactionOutcome();
};
```

### Return Value

**rtr_status_t** Interpret value for the success or failure of this call.

| Status | Message |
|---|---|
| RTR_STS_NOMESSAGE | The data object does not contain a message. |
| RTR_STS_OK | Normal successful completion. |

### Parameters

## pRTRMessage

Pointer to an RTRMessage object that describes the message which is being processed.

## pController

Pointer to the transaction controller within which this message was received.

### Description

The specified transaction has been accepted by all participants.

RTR is informing the application that the current transaction has been accepted by all parties of the transaction and successfully completed. RTRMessage will contain rtr_mt_accepted.

The default behavior is the handler dismisses the notification.

### Example

```
rtr_status_t MySRVMessageHandler::OnAccepted(RTRMessage *pmyMsg,
                             RTRServerTransactionController *pTC)
{
 cout << "accepted txn " << endl;
 pTC->AcknowledgeTransactionOutcome();
 return RTR_STS_OK;
}
```

## OnApplicationMessage()

OnApplicationMessage() — RTRServerMessageHandler::OnApplicationMessage();

### Prototype

```
virtual rtr_status_t OnApplicationMessage(RTRApplicationMessage
                                                  *pRTRApplicationMessage,
                                RTRServerTransactionController *pController
                                rtr_const_parnam_t pszPartitionName)
{
 return RTR_STS_OK;
}
```

### Parameters

## pRTRApplicationMessage

Pointer to an RTRApplicationMessage object that describes the message which is being processed.

## pController

Pointer to the transaction controller within which this event was received.

## pszPartitionName

Pointer to a partition name that is registered for the server transaction controller.

### Description

The RTRApplicationMessage parameter contains application data sent to it by an RTR client. RTRApplicationMessage will contain `rtr_mt_msg1` or `rtr_mt_msgn` and associated data.

The default behavior is the handler dismisses the notification.

### Example

```
void ClassDerivedFromHandler::OnApplicationMessage(*pApplicationMessage,
*pController, pszPartitionName )
{
// This handler is called by RTR when the client has sent a message.
// This is where you process the application's business logic
return RTR_STS_OK;
}
```

## OnInitialize()

OnInitialize() — RTRServerMessageHandler::OnInitialize(RTRApplicationMessage);

### Prototype

```
virtual rtr_status_t OnInitialize(RTRApplicationMessage
                             *pRTRApplicationMessage,
                             RTRServerTransactionController
                             *pController)
{
 return RTR_STS_OK;
}
```

### Parameters

## pRTRApplicationMessage

Pointer to an RTRApplicationMessage object that describes the message which is being processed.

## pController

Pointer to the transaction controller within which this event was received.

### Description

The OnInitialize member function is called by the RTR framework at the beginning of every new transaction this object processes. Your application should override this member function to perform any special logic for each transaction processed.

### Example

```
rtr_status_t ABCSHandlers::OnInitialize( RTRApplicationMessage
*pRTRApplicationMessage, RTRServerTransactionController *pController )
{
// This message notifies the RTR application that a new transaction
// is about to begin. Do any per-transaction state handling here.
cout << endl << endl << endl << "New Transaction being received..."
<< endl;
    m_bVoteToAccept = true;
       return RTR_STS_OK;
}
```

## OnPrepareTransaction()

OnPrepareTransaction() — RTRServerMessageHandler::OnPrepareTransaction();

### Prototype

```
virtual rtr_status_t OnPrepareTransaction(RTRMessage *pRTRMessage,
                        RTRServerTransactionController *pController)
{
 return RTR_STS_OK;
}
```

### Parameters

## pRTRMessage

Pointer to an RTRMessage object that describes the message which is being processed.

## pController

Pointer to the transaction controller within which this event was received.

### Description

The current transaction is complete (that is, all messages from the client have been received). RTRMessage will contain `rtr_mt_prepare`.

The default behavior is the handler dismisses the notification. Note that if you must override the defaults with a vote to accept or reject the transaction being processed so the transaction is successfully completed.

### Example

```
rtr_status_t ABCSHandlers::OnPrepareTransaction( RTRMessage *pRTRMessage,
RTRServerTransactionController *pController )
{
// This handler is called by RTR when the client has accepted the
// transaction. This is our notification that we have all orders
// for this transaction.
// We must now give RTR a vote for this transaction. A vote means
// either calling Accept or Reject.
// We simply check to see if anything has gone wrong. If so, reject
// the transaction, otherwise accept it.
        rtr_status_t sStatus;
    if (true == m_bVoteToAccept)
    {
         cout << "Voting to Accept..." << endl;
             sStatus = pController->AcceptTransaction();
    }
    else
    {
       cout << "Voting to Reject..." << endl;
       sStatus = pController->RejectTransaction();
    }
       return sStatus;
}
```

## OnRejected()

OnRejected() — RTRServerMessageHandler::OnRejected();

### Prototype

```
virtual rtr_status_t OnRejected(RTRMessage * pRTRMessage,
                         RTRServerTransactionController *pController)
{
           pController->AcknowledgeTransactionOutcome();
};
```

### Return Value

**rtr_status_t** Interpret value for the success or failure of this call.

| **Status** | **Message** |
|---|---|
| RTR_STS_NOMESSAGE | The data object does not contain a message. |
| RTR_STS_OK | Normal successful completion. |

### Parameters

## pRTRMessage

Pointer to an RTRMessage object that describes the message which is being processed.

## pController

Pointer to the transaction controller within which this event was received.

### Description

The specified transaction has been rejected by a participant. RTRMessage will contain `rtr_mt_rejected`.

The default behavior is the handler dismisses the notification.

### Example

```
 rtr_status_t ABCSHandlers::OnRejected( RTRMessage *pRTRMessage,
 RTRServerTransactionController *pController) {     cout
```

```
<
< "Entire Transaction Rejected..."
<
< endl;     return
 RTRServerMessageHandler::OnRejected(pRTRMessage,pController); }
```

## OnUncertainTransaction()

OnUncertainTransaction() — RTRServerMessageHandler::OnUncertainTransaction();

### Prototype

```
virtual rtr_status_t OnUncertainTransaction(RTRMessage
                           *pRTRApplicationMessage,
                           RTRServerTransactionController *pController
                            rtr_const_parnam_t pszPartitionName)
{
 return RTR_STS_OK;
}
```

### Parameters

## pRTRApplicationMessage

Pointer to an RTRMessage object that describes the message which is being processed.

## pController

Pointer to the transaction controller within which this event was received.

## pszPartitionName

Pointer to a partition name that is registered for the server transaction controller.

### Description

The OnUncertainTransaction() member function is called by the RTR framework when RTR is replaying or recovering a transaction. The user's application should override this member function to perform any special logic for each transaction processed. OnInitialize is also called when the server receives an `rtr_mt_msg1_uncertain` message.

This member function is only called for transactions whose RTRServerEnvironment object has set bXAManaged = FALSE.

The default behavior is the handler dismisses the notification.

### Example

```
ABCSHandlers::OnUncertainTransaction( RTRApplicationMessage
*pRTRApplicationMessage, RTRServerTransactionController
*pController, rtr_const_parnam_t pszPartitionName )
{
    return RTR_STS_OK;
}
```

## RTRServerMessageHandler()

RTRServerMessageHandler() — RTRServerMessageHandler::RTRServerMessageHandler();

### Prototype

```
RTRServerMessageHandler();
virtual ~RTRServerMessageHandler();
```

### Return Value

None

### Parameters

None

### Description

Call this constructor to create an RTRServerMessageHandler object.

### Example

```
class MySRVMessageHandler: public RTRServerMessageHandler
{
public:
        MySRVMessageHandler();
        ~MySRVMessageHandler();
        rtr_status_t OnPrepareTransaction( RTRMessage *pmyMsg,
                          RTRServerTransactionController *pTC);
        rtr_status_t  OnAccepted( RTRMessage *pmyMsg,
                         RTRServerTransactionController *pTC);
private:
};
MySRVMessageHandler::MySRVMessageHandler()
{
}
MySRVMessageHandler::~MySRVMessageHandler()
{
}
```

## 3.4. RTRServerTransactionController

RTRServerTransactionController is the class most commonly used to create an RTR server application. Typically, one instance of this class is used to process multiple consecutive transactions. A transaction controller object is used to send and receive all data between RTR clients and servers.

## RTRServerTransactionController Class Members

## Construction

| Method | Description |
|---|---|
| RTRServerTransactionController | Constructor |

| Method | Description |
|---|---|
| ~RTRServerTransactionController | Destructor |

## Operations

| Method | Description |
|---|---|
| RegisterClassFactory<br><br>(RTRClassFactory ) | Register a class factory for RTR to call when creating RTRData-derived objects. |
| RegisterHandlers<br><br>(RTRServerMessageHandler, RTRServerEventHandler ) | Register your handlers with this transaction. |
| RegisterPartition<br><br>(rtr_const_parnam_t rtr_const_rcpnam_t, rtr_const_access_t) | Add a partition to the list of partitions for which this transaction controller processes requests. |

## Basic Methods

| Method | Description |
|---|---|
| AcknowledgeTransactionOutcome() | Allow RTR to remove the current transaction from the journal and proceed with the next request from a client. |
| AcceptTransaction(rtr_reason_t, bool) | Accept the current transaction. |
| UnRegisterPartition(rtr_const_parnam_t) | Remove a partition from the list of partitions for which this transaction controller processes requests. |
| ForceTransactionRetry() | Tell RTR to cancel the current transaction and re-present it. |
| Receive(RTRData, rtr_timout_t) | Receive an RTR or application- generated message or an RTR event. |
| RejectTransaction(rtr_reason_t) | Vote to reject the current transaction. |
| SendApplicationEvent<br><br>(RTRApplicationEvent, rtr_const_rcpspc_t, rtr_const_msgfmt_t ) | Send an application-defined event within the current facility to the client. |
| SendApplicationMessage<br><br>(RTRApplicationMessage, rtr_const_msgfmt_t ) | Send an application-defined message to the client whose transaction this controller call is currently processing. |

## Get State Methods

| Method | Description |
|---|---|
| GetFacilityName (rtr_facnam_t, | Get facility name for the current transaction, if one exists. |

| Method | Description |
|---|---|
| const size_t ) | |
| GetPartitionName<br><br>(rtr_parnam_t,const size_t ) | Get partition name for the current transaction, if one exists. |
| GetProperties() | Get properties of the current transaction. |

## AcceptTransaction()

AcceptTransaction() — RTRServerTransactionController::AcceptTransaction();

### Prototype

```
virtual rtr_status_t AcceptTransaction(rtr_reason_t
                                        rtrReasonCode = RTR_NO_REASON,
                                        bool bIndependent = false);
```

### Return Value

**rtr_status_t** Interpret value for the success or failure of this call.

| Status | Message |
|---|---|
| RTR_STS_NOACCEPT | Client or Server has already voted or there is no active transaction. |
| RTR_STS_NOMESSAGE | The data object does not contain a message. |
| RTR_STS_OK | Normal successful completion. |
| RTR_STS_TXNOTACT | No transaction currently active on this channel. |

### Parameters

## rtrReasonCode

Optional reason for accepting the transaction. This reason is **OR**ed together with the reasons of the other participants in the transaction and returned to all participants of the transaction. The participants can retrieve this reason by calling RTRMessage::GetReason().

## bIndependent

If set to true, the transaction is considered independent of other transactions that RTR is processing. Independent transactions can improve performance in certain shadowing conditions because RTR will not need to maintain the order in which this transaction is processed on the shadow node.

### Description

Call this member function to accept the transaction currently being processed.

### Example

```
ABCSHandlers::OnPrepareTransaction( RTRMessage *pRTRMessage,
RTRServerTransactionController *pController )
{
// We simply check to see if anything has gone wrong. If so,
// reject the transaction, otherwise accept it.
    if (true == m_bVoteToAccept)
    {
        pController->AcceptTransaction();
    }
    else
    {
        pController->RejectTransaction();
    }
        return;
    }
```

## AcknowledgeTransactionOutcome()

AcknowledgeTransactionOutcome() —
RTRServerTransactionController::AcknowledgeTransactionOutcome();

### Prototype

```
virtual rtr_status_t AcknowledgeTransactionOutcome();
```

### Return Value

**rtr_status_t** Interpret value for the success or failure of this call.

| **Status** | **Message** |
|---|---|
| RTR_STS_ACKTXN | AcknowledgeTransactionOutcome may only be called after receiving the transaction outcome. |
| RTR_STS_NOMESSAGE | The data object does not contain a message. |
| RTR_STS_OK | Normal successful completion. |

### Parameters

None.

### Description

Call this member function after the application receives an indication of the outcome of the transaction, that is, the transaction has been either accepted or rejected.

Calling this method is mandatory. RTR will not process the next transaction until the application acknowledged that it has received the outcome of the transaction.

### Example

```
ABCSHandlers::OnAccepted( RTRMessage *pRTRMessage,
RTRServerTransactionController *pController )
{       pController->AcknowledgeTransactionOutcome();
```

```
        return;
}
```

## ForceTransactionRetry()

ForceTransactionRetry() — RTRServerTransactionController::ForceTransactionRetry();

### Prototype

```
virtual rtr_status_t ForceTransactionRetry ();
```

### Return Value

**rtr_status_t** Interpret value for the success or failure of this call.

| **Status** | **Message** |
|---|---|
| RTR_STS_NORETRYTXN | ForceRetryTransaction may only be called while processing a transaction. |
| RTR_STS_OK | Normal successful completion. |

### Parameters

None.

### Description

Call this function when your application wants the current transaction to be represented to your application without being rejected. If this call is made before the application votes to accept or reject the transaction, the maximum number of attempts will be 3. If this function is called after the application has voted, the maximum number of attempts will be determined by the current value of the Recovery Retry Count attribute of the partition. Note that this attribute can be changed by using the RTRPartitionProperties class or by issuing command to the RTR command line interface.

### Example

```
pController->ForceTransactionRetry();
```

## GetFacilityName()

GetFacilityName() — RTRServerTransactionController::GetFacilityName();

### Prototype

```
virtual rtr_status_t GetFacilityName (rtr_facnam_t pszFacilityName,
                                      size_t uiFacilityNameSize);
```

### Return Value

**rtr_status_t** Interpret value for the success or failure of this call.

| Status | Message |
|---|---|
| RTR_STS_APPBUFFTOOSMALL | The application buffer is too small. |
| RTR_STS_INVARGPTR | Invalid argument pointer. |
| RTR_STS_OK | Normal successful completion. |

### Parameters

### pszFacilityName

A null-terminated pointer to a facility name. Memory is allocated by the function call.

### uiFacilityNameSize

Specifies size of buffer passed by the facility name. If the size of the facility name intended for the `pszFacilityName` character string is greater than the size in `uiFacilityNameSize`, the error code RTR_STS_APPBUFFTOOSMALL is returned and the facility name is not copied into the character string.

### Description

Obtain the facility name, which the current transaction is executing in.

Memory is allocated by the caller and if `uiFacilityNameSize` is not big enough, an error message is returned.

### Example

```
pController->GetFacilityName(pszFacilityName, uiFacilityNameSize);
```

## GetPartitionName()

GetPartitionName() — RTRServerTransactionController::GetPartitionName();

### Prototype

```
virtual rtr_status_t GetPartitionName(rtr_parnam_t pszPartitionName,
                                      const size_t uiPartitionNameSize
                                      RTRData *pRTRData);
```

### Return Value

**rtr_status_t** Interpret value for the success or failure of this call.

| Status | Message |
|---|---|
| RTR_STS_APPBUFFTOOSMALL | The application buffer is too small. |
| RTR_STS_DATANOTAVAILABL | A required property was not available. |
| RTR_STS_INVARGPTR | Invalid argument pointer. |
| RTR_STS_OK | Normal successful completion |

| Status | Message |
|---|---|
| RTR_STS_TXNOTACT | Transaction not active. |

### Parameters

## pszPartitionName

A null-terminated pointer to a partition name.

## uiPartitionNameSize

An unsigned integer for the size of the named partition.

## pRTRData

The name of the partition on which the data object (message or event) was received.

### Description

Obtain the partition name, which the current transaction is using.

### Example

```
char szPartitionName[RTR_MAX_PARNAM_LEN+1];
sStatus = pController-> GetPartitionName(&szPartitionName[0],
                                          RTR_MAX_PARNAM_LEN+1,
                                          pRTRData);
// This call will either succeed or return RTR_STS_NOPARTITION.
// This means that the dat object has no partition associated with
// it. Only application messages and certain RTR events have a
// partition associated with them.
```

## GetProperties()

GetProperties() — RTRServerTransactionController::GetProperties();

### Prototype

```
virtual RTRServerTransactionProperties* GetProperties();
```

### Parameters

None.

### Description

This method gets a pointer to the RTRServerTransacitonProperties object describing the server transaction. If a transaction does not exist NULL is returned.

### Example

```
RTRServerTransactionProperties *pTxnProp =
                                        pController->GetProperties();
if (PTxnProp->TransactionIsOriginal())
{
.
}
```

## Receive()

Receive() — RTRServerTransactionController::Receive();

### Prototype

```
virtual rtr_status_t Receive(RTRData **pRTRData,
                               rtr_timout_t tTimeout = RTR_NO_TIMOUTMS);
```

### Return Value

**rtr_status_t** Interpret value for the success or failure of this call.

| Status | Message |
|---|---|
| RTR_STS_ACPNOTVIA | RTR ACP not a viable entity. |
| RTR_STS_INVCHANNEL | Invalid channel argument. |
| RTR_STS_INVDATPTRPTARG | Invalid pointer-to-data-pointer pointer argument |
| RTR_STS_INVFLAGS | Invalid flags argument. |
| RTR_STS_INVMSG | Invalid pmsg argument. |
| RTR_STS_INVRMNAME | Invalid resource manager name. |
| RTR_STS_NOACP | No RTRACP process available. |
| RTR_STS_NOMESSAGE | The data object does not contain a message. |
| RTR_STS_NORECEIVE | Attempting to receive at this point is not allowed. |
| RTR_STS_OK | Normal successful completion. |
| RTR_STS_TIMOUT | Call to Receive timed out. |
| RTR_STS_TRUNCATED | Buffer too short for message. Message has been truncated. |

### Parameters

## pRTRData

A pointer passed by reference, which will receive an object, derived from RTRData. This object can be any of the following:

- RTRMessage
- RTREvent
- RTRApplicationMessage
- RTRApplicationEvent

If a class factory is registered with the transaction controller, the application has the ability to have this object be any application class derived from RTRData. By calling the Dispatch() method, the most over ridden implementation of dispatch will be called.

For more information see the description of the RTR receive model in the *VSI Reliable Transaction Router Application Design Guide*.

## tTimeout

The maximum amount of time that the application is willing to wait for this receive to complete. The timeout value is in milliseconds.

### Description

This member function should be called when the application is ready to receive messages and events from the RTR framework. Typically this function is called in a loop. The RTRData object returned contains the message or event type, as well as other information useful to the application.

For more information see:

RTRData

### Example

```
// Continually loop receiving messages and dispatching them to the
 handlers.

void ABCOrderProcessor::ProcessIncomingOrders()

{
 //     Start processing orders

        abc_status sStatus = RTR_STS_OK;

        RTRData *pOrder = NULL;

        while (1)

        {

//  Receive an Order

sStatus = Receive(&pOrder);

print_status_on_failure(sStatus);

if(ABCSuccess != sStatus) break;

// If we can't get an Order then stop processing.

            delete pOrder;

        }

        return;
        }
```

## RegisterClassFactory()

RegisterClassFactory() — RTRServerTransactionController::RegisterClassFactory();

### Prototype

```
virtual rtr_status_t RegisterClassFactory( RTRClassFactory *pFactory);
```

### Return Value

**rtr_status_t** Interpret value for the success or failure of this call.

| **Status** | **Message** |
|---|---|
| RTR_STS_INVFACTORYPTARG | The factory argument pointer is invalid. |
| RTR_STS_OK | Normal successful completion |

### Parameters

## pFactory

Pointer to an RTRClassFactory object that is called, if registered, from the RTR framework when processing all Receive calls in your application.

### Description

A class factory returns an object for RTR to use (write data to) when the method RTRServerTransactionController::Receive is called. The application can register their own class factory and override the functions to return their own objects derived from the RTR data classes. The four RTR data classes are RTRApplicationMessage, RTRApplicationEvent, RTRMessage, and RTREvent.

Registering a class factory is not a requirement. An application would register a class factory only when they wish to customize the object that is being allocated.

### Example

```
sStatus = RegisterClassFactory(&m_ClassFactory);
print_status_on_failure(sStatus);
```

## RegisterFacility()

RegisterFacility() — RTRServerTransactionController::RegisterFacility();

### Prototype

```
virtual rtr_status_t RegisterFacility (rtr_const_facnam_t pszFacilityName);
```

### Return Value

**rtr_status_t** Interpret value for the success or failure of this call.

| Status | Message |
|---|---|
| RTR_STS_INVALIDFACILITY | The specified facility does not exist. |
| RTR_STS_INVFACNAMEARG | The facility name argument is invalid. |
| RTR_STS_OK | Normal successful completion |
| RTR_STS_RTRNOTRUNNING | RTR is not running. |

### Parameters

## pszFacilityName

A null-terminated pointer to a facility name. Memory is allocated by the function call. If the size of the parameter is not big enough, the return error message RTR_STS_APPBUFFTOOSMALL is returned.

### Description

Call the RegisterFacility() member function to register an existing RTR facility for your application. By registering a facility, your application informs RTR of the facility for which your application can process transactions.

### Example

```
// Register the facility with the transaction controller.
   sStatus = RegisterFacility(ABCFacility);
   print_status_on_failure(sStatus);
```

## RegisterHandlers()

RegisterHandlers() — RTRServerTransactionController::RegisterHandlers();

### Prototype

```
virtual rtr_status_t RegisterHandlers (
                             RTRServerMessageHandler *pMessageHandler,
                             RTRServerEventHandler *pEventHandler);
```

### Return Value

**rtr_status_t** Interpret value for the success or failure of this call.

| Status | Message |
|---|---|
| RTR_STS_INVEVNTHNDPTARG | The event handler pointer argument is invalid. |
| RTR_STS_INVMSGHNDLPTARG | The message handler pointer argument is invalid. |
| RTR_STS_OK | Normal successful completion |

### Parameters

## pMessageHandler

Pointer to an RTRServerMessageHandler object that will process all server messages in your application.

## pEventHandler

Pointer to an RTRServerMessageHandler object that will process all server events in your application.

### Description

Call the RegisterHandlers member function to register RTR message and event handlers for your application. By registering the handlers, your application informs RTR of the different configurations for which your application can process transactions. Your application can only use one partition at a time. The message and event handlers are called by the RTRData::Dispatch method.

Specify pMessageHandler and/or pEventHandler if your application wishes to make use of the RTR frameworks predefined handlers.

For more information on handlers see:

- RTRApplicationMessage::Dispatch
- RTRApplicationEvent::Dispatch
- RTRMessage::Dispatch
- RTREvent::Dispatch
- RTRServerMessageHandler
- RTRServerEventHandler

### Example

```
// Register the message and event handlers with the transaction controller.
   sStatus = pTransaction->RegisterHandlers(
pAppClassDerivedFromRTRMessageHandler,
pAppClassDerivedFromRTREventHandler
);
assert(RTR_STS_OK == sStatus);
```

## RegisterPartition()

RegisterPartition() — RTRServerTransactionController::RegisterPartition();

### Prototype

```
virtual rtr_status_t RegisterPartition(rtr_const_parnam_t pszPartitionName,
        rtr_const_rcpnam_t szRecipientName = RTR_NO_RCPNAM,
        rtr_const_access_t pszAccess = RTR_NO_ACCESS);
```

### Return Value

**rtr_status_t** Interpret value for the success or failure of this call.

| **Status** | **Message** |
|---|---|
| RTR_STS_DUPLPARTITION | Attempting to insert a duplicate partition. |

| Status | Message |
|---|---|
| RTR_STS_FACNOTREG | Facility is not registered. |
| RTR_STS_INVACCSTRPTRARG | The access string argument is invalid. |
| RTR_STS_INVPARTNAMEARG | The partition name argument is invalid |
| RTR_STS_INVRECPNAMPTARG | The recipient name argument is invalid. |
| RTR_STS_OK | Normal successful completion |
| RTR_STS_RTRNOTRUNNING | RTR is not running. |

### Parameters

## pszPartitionName

A null-terminated pointer to a partition name.

## szRecipientName

Name of the recipient. This null-terminated string contains the name of the recipient. This is an optional parameter.

Wildcards ("*" for any sequence of characters, and "%" for any one character) can be used in this string to address more than one recipient.

Note that *szRecipientName* is case sensitive.

## pszAccess

Pointer to a null-terminated string containing the access parameter. The default value is RTR_NO_ACCESS.

### Description

Call the RegisterPartition member function to register an RTR partition for your application. By registering a partition, your application informs RTR of the different configurations for which your application can process transactions. Your application can only use one partition at a time.

**Note**

It is mandatory to register a partition that already exists in a registered facility. RegisterPartition may be called multiple times to register multiple partitions.

### Example

```
// Register the partition with the transaction controller.
sStatus = pTransaction->RegisterPartition( "MyPartition);
assert(RTR_STS_OK == sStatus);
```

## RejectTransaction()

RejectTransaction() — RTRServerTransactionController::RejectTransaction();

### Prototype

```
virtual rtr_status_t RejectTransaction(rtr_reason_t rtrReasonCode =
                                       RTR_NO_REASON);
```

### Return Value

**rtr_status_t** Interpret value for the success or failure of this call.

| Status | Message |
|---|---|
| RTR_STS_NOMESSAGE | The data object does not contain a message. |
| RTR_STS_NOREJECT | Client or Server has already voted or there is no active transaction. |
| RTR_STS_OK | Normal successful completion. |
| RTR_STS_TXNOTACT | No transaction is currently active on this channel. |

### Parameters

## rtrReasonCode

Optional reason for rejecting the transaction. This reason is returned to the other participants in the transaction. The participants can retrieve this reason by calling RTRMessage::GetReason.

### Description

Call this member function to reject the transaction currently being processed by this object.

### Example

```
     sStatus = pController->RejectTransaction();
```

## RTRServerTransactionController()

RTRServerTransactionController() —
RTRServerTransactionController::RTRServerTransactionController();

### Prototype

```
RTRServerTransactionController();
virtual ~RTRServerTransactionController();
```

### Return Value

None.

### Parameters

None.

### Description

Call this constructor to create an RTRServerTransactionController object.

### Example

```
ABCOrderProcessor::ABCOrderProcessor()
{
}
```

## SendApplicationEvent()

SendApplicationEvent() — RTRServerTransactionController::SendApplicationEvent();

### Prototype

```
virtual rtr_status_t SendApplicationEvent( RTRApplicationEvent
                                           * pRTRApplicationEvent,
                        rtr_const_rcpspc_t szRecipientName = "*",
           rtr_const_msgfmt_t mfMessageFormat = RTR_NO_MSGFMT);
```

### Return Value

**rtr_status_t** Interpret value for the success or failure of this call.

| Status | Message |
|---|---|
| RTR_STS_INSVIRMEM | Insufficient virtual memory. |
| RTR_STS_INVAPPEVNTPTARG | Invalid application event pointer argument. |
| RTR_STS_INVMSGFMTPTRARG | The message format string argument is invalid. |
| RTR_STS_INVRECPNAMPTARG | The recipient name argument is invalid. |
| RTR_STS_NOEVENTDATA | There is no event data associated with the event. |
| RTR_STS_NOMESSAGE | The data object does not contain a message. |
| RTR_STS_OK | Normal successful completion. |

### Parameters

## pRTRApplicationEvent

Pointer to an RTRApplicationEvent object which contains application data to be sent to the client.

## szRecipientName

Name of the recipient. This null-terminated character string contains the name of the recipient specified with the *szRecipientName* parameter on the RTRServerTransactionController::RegisterPartition method.

Wildcards ("*" for any sequence of characters, and "%" for any one character) can be used in this string to address more than one recipient. *szRecipientName* is an optional parameter.

Note that *szRecipientName* is case sensitive.

## mfMessageFormat

Message format description. mfMessageFormat is a null-terminated character string containing the format description of the message. RTR uses this description to convert the contents of the message appropriately when processing the message on different hardware platforms. If no parameter is specified, the default is no special formatting.

### Description

This member function should be called when the application wants to send an application-defined (broadcast) event to the client. Formerly, application- defined events are only delivered to the clients that have subscribed for them and these are not related to any transaction. Only reply messages go to the client that started the transaction. Simply calling this function will not deliver the event to the client, unless it has subscribed for it. With the C++ API, you "subscribe" by overriding the event handler methods. The events are only received if they are overridden.

### Example

```
sStatus = pTransaction->SendApplicationEvent(pEventA);
assert(RTR_STS_OK == sStatus);
```

## SendApplicationMessage()

SendApplicationMessage() — RTRServerTransactionController::SendApplicationMessage();

### Prototype

```
virtual rtr_status_t SendApplicationMessage(RTRApplicationMessage
                                            *pRTRApplicationMessage,
                     rtr_const_msgfmt_t mfMessageFormat = RTR_NO_MSGFMT);
```

### Return Value

**rtr_status_t** Interpret value for the success or failure of this call.

| Status | Message |
|---|---|
| RTR_STS_INSVIRMEM | Insufficient virtual memory. |
| RTR_STS_INVAPPMSGPTAR | Invalid application message pointer argument. |
| RTR_STS_INVMSGFMTPTRARG | The message format string argument is invalid. |
| RTR_STS_NOMESSAGE | The data object does not contain a message. |
| RTR_STS_NOSEND | Attempting to send an application message at this point is not allowed. |
| RTR_STS_OK | Normal successful completion. |

### Parameters

## pRTRApplicationMessage

Pointer to an RTRApplicationMessage object which contains application data to be sent to the client.

## mfMessageFormat

Message format description. mfMessageFormat is a null-terminated character string containing the format description of the message. RTR uses this description to convert the contents of the message appropriately when processing the message on different hardware platforms. If no parameter is specified, the default is no special formatting.

### Description

This member function should be called when the application wants to send application data to the client which originally established the transaction. The RTRData object contains the data to be sent.

For more information see:

RTRData

### Example

```
// Send the Server a message
sStatus = pTransaction->SendApplicationMessage(pMessage1);
assert(RTR_STS_OK == sStatus);
```

## UnRegisterPartition()

UnRegisterPartition() — RTRServerTransactionController::UnRegisterPartition();

### Prototype

```
virtual rtr_status_t UnRegisterPartition(rtr_const_parnam_t
pszPartitionName);
```

### Return Value

**rtr_status_t** Interpret value for the success or failure of this call.

| **Status** | **Message** |
|---|---|
| RTR_STS_INVPARTNAMEARG | The partition name argument is invalid |
| RTR_STS_NOPARTITION | The partition name has not been previously registered. |
| RTR_STS_OK | Normal successful completion |

### Parameters

## pszPartitionName

A null-terminated pointer to a partition name.

### Description

Remove a partition from the list of partitions for which this transaction controller processes requests.

### Example

```
pController-> UnRegisterPartition();
```

## 3.5. RTRServerTransactionProperties

This class holds, makes available, and allows modification of the properties of its associated RTRServerTransactionController object. It provides attributes for a given transaction.

Typically, RTR C++ API applications obtain this object by calling GetProperties on the transaction controller. Other applications, including legacy applications, may create an instance of this object by calling the constructor with the TID of the transaction.

### RTRServerTransactionProperties Class Members

### Construction

| **Method** | **Description** |
|---|---|
| RTRServerTransactionProperties (const rtr_tid_t) | Constructor |
| ~RTRServerTransactionProperties() | Destructor |

### Get the Type of Transaction

| **Method** | **Description** |
|---|---|
| TransactionIsOriginal() | Tests whether the transaction is an original transaction. |
| TransactionIsReplay() | Tests whether the transaction is a replayed transaction. |
| TransactionIsRecovery() | Tests whether the transaction is a recovered transaction. |

### Get Functions

| **Method** | **Description** |
|---|---|
| GetFacilityName(rtr_facnam_t,<br><br>const size_t) | Get the facility. |
| GetTransactionState<br><br>(rtr_tx_jnl_state_t ) | Get the transaction state |
| GetTID(rtr_tid_t) | Get the TID (transaction ID). |

When setting the state of a transaction, the state transaction must be valid, or else the call will return an error. For each of the set state methods, there are two versions. The versions with no parameters attempt to transition the transaction to the requested state. The second version for each method will only transition to the requested state if the current trnasaction state matches the state passed in the stCurrentTxnState argument.

### Set the State of Transaction

| **Method** | **Description** |
|---|---|
| SetStateToAbort() | Sets the transaction state to abort. |

| Method | Description |
|---|---|
| SetStateToAbort(rtr_tx_jnl_state_t) | Sets the transaction state to abort. |
| SetStateToCommit() | Sets the transaction state to commit. |
| SetStateToCommit<br><br>(rtr_tx_jnl_state_t ) | Sets the transaction state to commit. |
| SetStateToDone() | Sets the transaction state to done. |
| SetStateToDone(rtr_tx_jnl_state_t) | Sets the transaction state to done. |
| SetStateToException() | Sets the transaction state to exception. |
| SetStateToException<br><br>(rtr_tx_jnl_state_t ) | Sets the transaction state to exception. |

## GetFacilityName()

GetFacilityName() — RTRServerTransactionProperties::GetFacilityName();

### Prototype

```
rtr_status_t GetFacilityName(rtr_facnam_t pszFacilityName,
                             size_t uiFacilityNameSize );
```

### Return Value

**rtr_status_t** Interpret value for the success or failure of this call.

| Status | Message |
|---|---|
| RTR_STS_APPBUFFTOOSMALL | The application buffer is too small. |
| RTR_STS_DATANOTAVAILABL | A required property was not available. |
| RTR_STS_INVARGPTR | Invalid argument pointer. |
| RTR_STS_OK | Normal successful completion. |

### Parameters

## pszFacilityName

A null-terminated pointer to a facility name. Memory is allocated by the function call.

## uiFacilityNameSize

Specifies size of buffer passed by the facility name. If the size of the facility name intended for the `pszFacilityName` character string is greater than the size in `uiFacilityNameSize`, the error code RTR_STS_APPBUFFTOOSMALL is returned and the facility name is not copied into the character string.

### Description

This method gets the facility name associated with the transaction and described by the RTRServerTransactionProperties object.

### Example

```
pTransaction->GetFacility(pszFacilityName);
```

## GetPartitionName()

GetPartitionName() — RTRServerTransactionController::GetPartitionName();

### Prototype

```
virtual rtr_status_t GetPartitionName(rtr_parnam_t pszPartitionName,
                                      const size_t uiPartitionNameSize
                                      RTRData *pRTRData);
```

### Return Value

**rtr_status_t** Interpret value for the success or failure of this call.

| Status | Message |
|---|---|
| RTR_STS_APPBUFFTOOSMALL | The application buffer is too small. |
| RTR_STS_DATANOTAVAILABL | A required property was not available. |
| RTR_STS_INVARGPTR | Invalid argument pointer. |
| RTR_STS_OK | Normal successful completion |
| RTR_STS_TXNOTACT | Transaction not active. |

### Parameters

## pszPartitionName

A null-terminated pointer to a partition name.

## uiPartitionNameSize

An unsigned integer for the size of the named partition.

## pRTRData

The name of the partition on which the data object (message or event) was received.

### Description

Obtain the partition name, which the current transaction is using.

### Example

```
char szPartitionName[RTR_MAX_PARNAM_LEN+1];
sStatus = pController-> GetPartitionName(&szPartitionName[0],
                                         RTR_MAX_PARNAM_LEN+1,
                                         pRTRData);
```

```
// This call will either succeed or return RTR_STS_NOPARTITION.
// This means that the dat object has no partition associated with
// it. Only application messages and certain RTR events have a
// partition associated with them.
```

## GetTID()

GetTID() — RTRServerTransactionProperties::GetTID();

### Prototype

```
rtr_status_t GetTID(rtr_tid_t &rtrTID);
```

### Return Value

**rtr_status_t** Interpret value for the success or failure of this call. RTR_STS_OK is the normal successful completion. Returns RTR_STS_NOTID on failure.

### Parameters

## rtrTID

An RTR transaction identifier.

### Description

This method copies the transaction identifier (TID) of the transaction described by the RTRServerTransactionProperties object for the current transaction.

### Example

```
rtr_tid_t tid = pController->GetTID(&rtrTID);
```

## GetTransactionState()

GetTransactionState() — RTRServerTransactionProperties:: GetTransactionState ();

### Prototype

```
rtr_status_t GetTransactionState (rtr_tx_jnl_state_t &pstCurrentTxnState);
```

### Return Value

**rtr_status_t** Interpret value for the success or failure of this call.

| **Status** | **Message** |
|---|---|
| RTR_STS_INVTXNSTATPTARG | Invalid transaction state pointer argument. |
| RTR_STS_DATANOTAVAILABL | A required property was not available. |
| RTR_STS_OK | Normal successful completion |

### Parameters

### pstCurrentTxnState

Pointer to the transaction state of type `rtr_tx_jnl_state_t`.

### Description

Get the transaction state for the current transaction.

### Example

```
rtr_tx_jnl_state_t txnState;
rtr_status_t sStatus = GetTransactionState(txnState);
if ( rtr_tx_jnl_voted == txnState)
{

}
```

## RTRServerTransactionProperties()

RTRServerTransactionProperties() — RTRServerTransactionProperties::RTRServerTransactionProperties();

### Prototype

```
RTRServerTransactionProperties(const rtr_tid_t &tid);
virtual ~RTRServerTransactionProperties();
```

### Return Value

None.

### Parameters

### tid

A transaction identifier value of type `rtr_tid_t`.

### Description

Call this constructor to create an RTRServerTransactionProperties object associated with the specified tid.

### Example

```
RTRServerTransactionProperties::RTRServerTransactionProperties
{
}
```

## SetStateToAbort()

SetStateToAbort() — RTRServerTransactionProperties::SetStateToAbort();

## Prototype

```
rtr_status_t SetStateToAbort();
```

```
rtr_status_t SetStateToAbort(rtr_tx_jnl_state_t stCurrentTxnState);
```

## Return Value

**rtr_status_t** Interpret value for the success or failure of this call. RTR_STS_OK is the normal successful completion.

## Parameters

## stCurrentTxnState

A transaction state of type `rtr_tx_jnl_state_t`.

## Description

This method is used to set the current server transaction state to abort. There are two forms:

- For the form with no parameter, the current transaction state is internally tested. If it is currently valid to transition from that state to the abort state, the call succeeds.
- For the form with the transaction state parameter, if it is valid to transition from that state to the abort state, the call succeeds.

## Example

```
rtr_status_t sStatus = SetStateToAbort(txnState);
```

## SetStateToCommit()

SetStateToCommit() — RTRServerTransactionProperties::SetStateToCommit();

## Prototype

```
rtr_status_t SetStateToCommit();
```

```
rtr_status_t SetStateToCommit(rtr_tx_jnl_state_t stCurrentTxnState);
```

## Return Value

**rtr_status_t** Interpret value for the success or failure of this call. RTR_STS_OK is the normal successful completion.

## Parameters

## stCurrentTxnState

A transaction state of type `rtr_tx_jnl_state_t`.

### Description

This method is used to set the current server transaction state to commit. There are two forms:

- For the form with no parameter, the current transaction state is internally tested. If it is currently valid to transition from that state to the commit state, the call succeeds.
- For the form with the transaction state parameter, if it is valid to transition from that state to the commit state, the call succeeds.

### Example

```
rtr_status_t sStatus = SetStateToCommit(txnState);
```

## SetStateToDone()

SetStateToDone() — RTRServerTransactionProperties::SetStateToDone();

### Prototype

```
rtr_status_t SetStateToDone();

rtr_status_t SetStateToDone(rtr_tx_jnl_state_t stCurrentTxnState);
```

### Return Value

**rtr_status_t** Interpret value for the success or failure of this call. RTR_STS_OK is the normal successful completion.

### Parameters

## stCurrentTxnState

A transaction state of type `rtr_tx_jnl_state_t`.

### Description

This method is used to set the current server transaction state to done. There are two forms:

- For the form with no parameter, the current transaction state is internally tested. If it is currently valid to transition from that state to the done state, the call succeeds.
- For the form with the transaction state parameter, if it is valid to transition from that state to the done state, the call succeeds.

### Example

```
rtr_status_t sStatus = SetStateToDone(txnState);
```

## SetStateToException()

SetStateToException() — RTRServerTransactionProperties::SetStateToException();

## Prototype

```
rtr_status_t SetStateToException();
```

```
rtr_status_t SetStateToException(rtr_tx_jnl_state_t stCurrentTxnState);
```

## Return Value

**rtr_status_t** Interpret value for the success or failure of this call. RTR_STS_OK is the normal successful completion.

## Parameters

## stCurrentTxnState

A transaction state of type `rtr_tx_jnl_state_t`.

## Description

This method is used to set the current server transaction state to exception.There are two forms:

- For the form with no parameter, the current transaction state is internally tested. If it is currently valid to transition from that state to the exception state, the call succeeds.
- For the form with the transaction state parameter, if it is valid to transition from that state to the exception state, the call succeeds.

## Example

```
rtr_status_t sStatus = SetStateToException(txnState);
```

## TransactionIsOriginal()

TransactionIsOriginal() — RTRServerTransactionProperties::TransactionIsOriginal();

## Prototype

```
bool TransactionIsOriginal();
```

## Return Value

**bool** A true or false return value.

## Parameters

None.

## Description

This method tests if the transaction is an original transaction. Note that this does not necessarily mean that the transaction has never been presented before.

### Example

```
RTRServerTransactionProperties *pstProperties =
                                pController->GetProperties();
bool bOriginal  = pTransactionController ->TransactionIsOriginal();
```

## TransactionIsRecovery()

TransactionIsRecovery() — RTRServerTransactionProperties::TransactionIsRecovery();

### Prototype

```
bool TransactionIsRecovery();
```

### Return Value

**bool** A true or false return value.

### Parameters

None.

### Description

This method tests if the transaction is a recovered transaction. A recovered transaction is one where the transaction was held in the RTR journal during a crash of a node, and has been restored and can be committed in the database.

### Example

```
rtr_status_t sStatus = TransactionIsRecovery();
```

## TransactionIsReplay()

TransactionIsReplay() — RTRServerTransactionProperties::TransactionIsReplay();

### Prototype

```
bool TransactionIsReplay();
```

### Return Value

**bool** A true or false return value.

### Parameters

None.

### Description

This method tests if the transaction is a replayed transaction.

### Example

```
rtr_status_t sStatus = TransactionIsReplay();
```

## 3.6. Client Classes

The client classes of the RTR API are:

- RTRClientEventHandler
- RTRClientMessageHandler
- RTRClientTransactionController
- RTRClientTransactionProperties

These classes are described in this section in alphabetical order.

## 3.7. RTRClientEventHandler

This class defines event handlers for all potential events that an RTR client application can receive. Each handler has a default behavior. Applications should override those member functions for which they intend to perform application-specific processing.

Applications can extend this class by deriving from it and adding their own application-level event handlers.

For further information see RTRData::Dispatch().

### RTRClientEventHandler Class Members

### Construction

| **Method** | **Description** |
|---|---|
| RTRClientEventHandler() | Constructor |
| ~RTRClientEventHandler() | Destructor |

### Operations

| **Method** | **Description** |
|---|---|
| OnApplicationEvent (RTRApplicationEvent, RTRClientTransactionController ) | There is an event generated by the application, for the client. |
| OnFacilityDead(RTREvent, RTRClientTransactionController) | Default handler for the event where the facility is no longer operational. |
| OnFacilityReady(RTREvent, RTRClientTransactionController) | Default handler for the event where the facility has become operational. |

| Method | Description |
|---|---|
| OnFrontendGainedLinkToRouter<br><br>(RTREvent, RTRClientTransactionController ) | Default handler for the event where a frontend link to the current router has been established. |
| OnFrontendLostLinkToRouter<br><br>(RTREvent, RTRClientTransactionController ) | Default handler for the event where the frontend link to the current router has been lost. |
| OnKeyRangeNoLongerAvailable<br><br>(RTREvent, RTRClientTransactionController ) | Default handler for the event where no more servers remain for a particular routing key range. |
| OnNewKeyRangeAvailable(RTREvent, RTRClientTransactionController) | Default handler for the event where one or more servers for a new key range have become available. |
| OnRouterGainedLinkToBackend<br><br>(RTREvent, RTRClientTransactionController ) | Default handler for the event where a current router established a link to a backend. |
| OnRouterLostLinkToBackend<br><br>(RTREvent, RTRClientTransactionController ) | Default handler for the event where the current router lost a link to a backend. |

## OnApplicationEvent()

OnApplicationEvent() — RTRClientEventHandler::OnApplicationEvent();

### Prototype

```
virtual rtr_status_t OnApplicationEvent(RTRApplicationEvent
                                           *pRTRApplicationEvent,
                         RTRClientTransactionController *pController)
{
return RTR_STS_OK;
}
```

### Parameters

## pRTRApplicationEvent

Pointer to an RTRApplicationEvent object that describes the message which is being processed.

## pController

Pointer to the transaction controller within which this event was received.

### Description

The RTRData parameter contains an application event sent to it by an RTR server.

The default behavior is the handler dismisses the notification.

### Example

```
MyCLIEventHandler::OnApplicationEvent( RTRApplicationEvent
                                           *pRTRApplicationEvent,
                                     RTRClientTransactionController
                                              *pCTC )
{
        cout << "An application event... " <<endl;
        return RTR_STS_OK;
}
```

## OnFacilityDead()

OnFacilityDead() — RTRClientEventHandler::OnFacilityDead();

### Prototype

```
virtual rtr_status_t OnFacilityDead(RTREvent *pRTREvent,
                              RTRClientTransactionController *pController)
{
return RTR_STS_OK;
}
```

### Parameters

## pRTREvent

Pointer to an RTREvent object that describes the RTR- generated event being processed.

## pController

Pointer to the transaction controller within which this event was received.

### Description

This method provides the default handler for the event where the facility is no longer operational.

The client application is receiving an RTR-generated event. RTREvent contains the application-defined number RTR_EVTNUM_FACDEAD (97) and any associated data.

### Example

```
MyCLIEventHandler::OnFacilityDead( RTREvent *pRTREvent,
 RTRClientTransactionController *pCTC )
{
 return RTR_STS_OK;
}
```

## OnFacilityReady()

OnFacilityReady() — RTRClientEventHandler::OnFacilityReady();

### Prototype

```
virtual rtr_status_t OnFacilityReady(RTREvent *pRTREvent,
                              RTRClientTransactionController *pController)
{
return RTR_STS_OK;
}
```

### Parameters

## pRTREvent

Pointer to an RTREvent object that describes the RTR- generated event being processed.

## pController

Pointer to the transaction controller within which this event was received.

### Description

This method provides the default handler for the event where the facility has become operational.

The client application is receiving an RTR-generated event. RTREvent contains the application-defined number RTR_EVTNUM_FACREADY (96) and any associated data.

### Example

```
MyCLIEventHandler::OnFacilityReady( RTREvent *pRTREvent,
 RTRClientTransactionController *pCTC )
{
        return RTR_STS_OK;
}
```

## OnFrontendGainedLinkToRouter()

OnFrontendGainedLinkToRouter() — RTRClientEventHandler::OnFrontendGainedLinkToRouter();

### Prototype

```
virtual rtr_status_t OnFrontendGainedLinktToRouter(RTREvent *pRTREvent,
                               RTRClientTransactionController *pController)
{
return RTR_STS_OK;
}
```

### Parameters

## pRTREvent

Pointer to an RTREvent object that describes the RTR- generated event being processed.

## pController

Pointer to the transaction controller within which this event was received.

## Description

This method provides the default handler for the event where a frontend link to the current router has been established.

The client application is receiving an RTR-generated event. RTREvent contains the application-defined event number RTR_EVTNUM_FERTRGAIN (98) and any associated data.

## Example

```
MyCLIEventHandler::OnFrontendGainedLinkToRouter( RTREvent
                                                        *pRTREvent,
                                     RTRClientTransactionController
                                                          *pCTC )
{
 return RTR_STS_OK;
}
```

## OnFrontendLostLinkToRouter()

OnFrontendLostLinkToRouter() — RTRClientEventHandler::OnFrontendLostLinkToRouter();

### Prototype

```
virtual rtr_status_t OnFrontendLostLinkToRouter(RTREvent *pRTREvent,
                        RTRClientTransactionController *pController)
{
return RTR_STS_OK;
}
```

### Parameters

## pRTREvent

Pointer to an RTREvent object that describes the RTR- generated event being processed.

## pController

Pointer to the transaction controller within which this event was received.

### Description

This method provides the default handler for the event where the frontend link to the current router has been lost.

The client application is receiving an RTR-generated event. RTREvent contains the application-defined number RTR_EVTNUM_FERTRLOSS (99) and any associated data.

### Example

```
MyCLIEventHandler:: OnFrontendLostLinkToRouter (
                                    RTREvent  *pRTREvent,
                                    RTRClientTransactionController *pCTC )
{
        return RTR_STS_OK;
}
```

## OnNewKeyRangeAvailable()

OnNewKeyRangeAvailable() — RTRClientEventHandler::OnNewKeyRangeAvailable();

### Prototype

```
virtual rtr_status_t OnKeyRangeNoLongerAvailable(
                             RTREvent * pRTREvent,
                             RTRClientTransactionController *pController)
{
return RTR_STS_OK;
}
```

### Parameters

## pRTREvent

Pointer to an RTREvent object that describes the RTR- generated event being processed.

## pController

Pointer to the transaction controller within which this event was received.

### Description

This method provides the default handler for the event where one or more servers for a new routing key range have become available.

The client application is receiving an RTR-generated event. RTREvent contains the application-defined number RTR_EVTNUM_KEYRANGEGAIN (102) and any associated data.

### Example

```
MyCLIEventHandler:: OnKeyRangeNoLongerAvailable(
                               RTREvent  *pRTREvent,
                               RTRClientTransactionController *pCTC )
{
return RTR_STS_OK;
}
```

## OnKeyRangeNoLongerAvailable()

OnKeyRangeNoLongerAvailable() — RTRClientEventHandler::OnKeyRangeNoLongerAvailable();

### Prototype

```
virtual rtr_status_t OnNewKeyRangeAvailable(RTREvent * pRTREvent,
                              RTRClientTransactionController *pController)
{
return RTR_STS_OK;
}
```

### Parameters

## pRTREvent

Pointer to an RTREvent object that describes the RTR- generated event being processed.

## pController

Pointer to the transaction controller within which this event was received.

### Description

This method provides the default handler for the event where no more servers remain for a particular routing key range.

The client application is receiving an RTR-generated event. RTREvent contains the application-defined number RTR_EVTNUM_KEYRANGELOSS (103) and any associated data.

### Example

```
MyCLIEventHandler:: OnNewKeyRangeAvailable (
         RTREvent  *pRTREvent,
         RTRClientTransactionController *pCTC )
{
 return RTR_STS_OK;
}
```

## OnRouterGainedLinkToBackend()

OnRouterGainedLinkToBackend() — RTRClientEventHandler::OnRouterGainedLinkToBackend();

### Prototype

```
virtual rtr_status_t OnRouterGainedLinkToBackend(RTREvent * pRTREvent,
                                RTRClientTransactionController *pController)
{
return RTR_STS_OK;
}
```

### Parameters

## pRTREvent

Pointer to an RTREvent object that describes the RTR- generated event being processed.

## pController

Pointer to the transaction controller within which this event was received.

### Description

This method provides the default handler for the event where the current router established a link to the backend.

The client application is receiving an RTR-generated event. RTREvent contains the application-defined event number RTR_EVTNUM_RTRBEGAIN (100) and any associated data.

### Example

```
MyCLIEventHandler::OnRouterGainedLinkToBackend(
                                    RTREvent  *pRTREvent,
                                    RTRClientTransactionController *pCTC )
{
return RTR_STS_OK;
}
```

## OnRouterLostLinkToBackend()

OnRouterLostLinkToBackend() — RTRClientEventHandler::OnRouterLostLinkToBackend();

### Prototype

```
virtual rtr_status_t OnRouterLostLinkToBackend(RTREvent * pRTREvent,
                         RTRClientTransactionController *pController)
{
return RTR_STS_OK;
}
```

### Parameters

## pRTREvent

Pointer to an RTREvent object that describes the RTR- generated event being processed.

## pController

Pointer to the transaction controller within which this event was received.

### Description

This method provides the default handler for the event where the current router lost a link to a backend.

The client application is receiving an RTR-generated event. RTREvent contains the application-defined number RTR_EVTNUM_RTRBELOSS (101) and any associated data.

### Example

```
MyCLIEventHandler::OnRouterLostLinkToBackend(
                                  RTREvent  *pRTREvent,
                                  RTRClientTransactionController *pCTC )
{
        return RTR_STS_OK;
}
```

## RTRClientEventHandler()

RTRClientEventHandler() — RTRClientEventHandler::RTRClientEventHandler();

### Prototype

```
RTRClientEventHandler();
```

### Return Value

None.

### Parameters

None.

### Description

Construct a client event handler object.

### Example

```
    RTRClientEventHandler::RTRClientEventhandler()
    {
    }
```

## 3.8. RTRClientMessageHandler

This class defines message handlers for all potential messages that an RTR client application can receive. Each handler has a default behavior. Applications should override those member functions for which they intend to perform application specific processing.

**Note**

Applications can extend this class by deriving from it and adding their own application-level message handlers.

For further information see RTRData::Dispatch().

### RTRClientMessageHandler Class Members

### Construction

| Method | Description |
|---|---|
| RTRClientMessageHandler() | Constructor |
| ~RTRClientMessageHandler() | Destructor |

### Operations

| Method | Description |
|---|---|
| OnAccepted(RTRMessage,<br><br>RTRClientTransactionController) | The specified transaction has been accepted by all participants. |

| Method | Description |
|---|---|
| OnAllPreparedTransaction<br><br>(RTRMessage, RTRClientTransactionController ) | The specified transaction has been prepared by all participants. |
| OnApplicationMessage<br><br>(RTRApplicationMessage, RTRClientTransactionController ) | The server has sent the client a message. |
| OnInitialize() | A new transaction is being processed. |
| OnRejected(RTRMessage,<br><br>RTRClientTransactionController) | The specified transaction has been rejected by a participant. |
| OnReturnToSender(RTRMessage,<br><br>RTRClientTransactionController) | The message could not be delivered and has been returned to the sender. |

## OnAccepted()

OnAccepted() — RTRClientMessageHandler::OnAccepted();

### Prototype

```
virtual rtr_status_t OnAccepted(RTRMessage *pRTRMessage,
                               RTRClientTransactionController *pController)
{
 return RTR_STS_OK;
}
```

### Return Value

None.

### Parameters

## pRTRMessage

Pointer to an RTRApplicationMessage object that describes the message which is being processed.

## pController

Pointer to the transaction controller within which this message was received.

### Description

The specified transaction has been accepted by all participants.

The default behavior is the handler dismisses the notification.

### Example

```
rtr_status_t ABCCHandlers::OnAccepted( RTRMessage *pRTRMessage,
RTRClientTransactionController *pController )
{
        return ABCOrderSucceeded;
}
```

## OnAllPreparedTransaction()

OnAllPreparedTransaction() — RTRClientMessageHandler::OnAllPreparedTransaction();

### Prototype

```
virtual rtr_status_t OnAllPreparedTransaction (RTRMessage * pRTRMessage,
                            RTRClientTransactionController *pController)
{
RTR_STS_OK;
}
```

### Parameters

## pRTRMessage

Pointer to an RTRMessage object that describes the message which is being processed.

## pController

Pointer to the transaction controller within which this message was received.

### Description

The specified transaction has been prepared by all participants.

The default behavior is the handler dismisses the notification.

### Example

```
rtr_status_t MyCLIMessageHandler::OnAllPreparedTransaction(
                                   RTRMessage *pmyMsg,
                                   RTRClientTransactionController *pTC)
{
        cout << "prepare txn " << endl;
        rtr_return RTR_STS_OK;
}
```

## OnApplicationMessage()

OnApplicationMessage() — RTRClientMessageHandler::OnApplicationMessage();

### Prototype

```
virtual rtr_status_t OnApplicationMessage(RTRApplicationMessage
                                                *pRTRApplicationMessage,
                             RTRClientTransactionController *pController)
{
```

```
RTR_STS_OK;
}
```

## Return Value

None.

## Parameters

## pRTRApplicationMessage

Pointer to an RTRApplicationMessage object that describes the message which is being processed.

## pController

Pointer to the transaction controller within which this message was received.

## Description

The RTRApplicationMessage parameter contains application data sent to it by an RTR server.

The default behavior is the handler dismisses the notification.

## Example

```
rtr_status_t MyCLIMessageHandler::OnApplicationMessage(
                                   RTRApplicationMessage *pmyMsg,
                                   RTRClientTransactionController *pTC)
{
        return RTR_STS_OK;
}
```

## OnInitialize()

OnInitialize() — RTRClientMessageHandler::OnInitialize();

## Prototype

```
virtual rtr_status_t OnInitalize()'
{
RTR_STS_OK;
}
```

## Parameters

None.

## Description

This method is called at the beginning of each transaction to prepare the server for a transaction. Allowing the application to perform any application-specific initialization necessary to process the transaction.

### Example

```
rtr_status_t MyCLIMessageHandler::OnInitialize()
{
        return RTR_STS_OK;

}
```

## OnRejected()

OnRejected() — RTRClientMessageHandler::OnRejected();

### Prototype

```
virtual rtr_status_t OnRejected(RTRMessage * pRTRMessage,
                         RTRClientTransactionController *pController)
{
return RTR_STS_OK;
}
```

### Parameters

## pRTRMessage

Pointer to an RTRMessage object that describes the message which is being processed.

## pController

Pointer to the transaction controller within which this message was received.

### Description

The specified transaction has been rejected by a participant.

The default behavior is the handler dismisses the notification.

### Example

```
rtr_status_t ABCCHandlers::OnRejected( RTRMessage *pRTRMessage,
RTRClientTransactionController *pController )
{
        return ABCOrderFailed;
}
```

## OnReturnToSender()

OnReturnToSender() — RTRClientMessageHandler::OnReturnToSender();

### prototype

```
virtual rtr_status_t OnReturnToSender(RTRMessage * pRTRMessage,
                         RTRClientTransactionController *pController)
```

```
{
return RTR_STS_OK;
}
```

### Parameters

## pRTRMessage

Pointer to an RTRMessage object that describes the message which is being processed.

## pController

Pointer to the transaction controller within which this message was received.

### Description

The message could not be delivered and has been returned to sender.

The default behavior is the handler dismisses the notification.

### Example

```
rtr_status_t MyCLIMessageHandler::OnReturnToSender(
        RTRMessage *pmyMsg,
        RTRClientTransactionController *pTC)
{
    return RTR_STS_OK;
}
```

## RTRClientMessageHandler()

RTRClientMessageHandler() — RTRClientMessageHandler::RTRClientMessageHandler();

### Prototype

```
RTRClientMessageHandler();
virtual ~RTRClientMessageHandler();
```

### Return Value

None.

### Parameters

None.

### Description

Call this constructor to create an RTRClientMessageHandler object.

### Example

```
MyCLIMessageHandler::MyCLIMessageHandler()
```

```
{

}

MyCLIMessageHandler::~MyCLIMessageHandler()

{

}
```

## 3.9. RTRClientTransactionController

RTRClientTransactionController is the main class used to create an RTR client application. The transaction controller object is used to send and receive all data between RTR clients and servers. Typically one instance of this class is used to process multiple consecutive transactions.

### RTRClientTransactionController Class Members

### Construction

| **Method** | **Description** |
|---|---|
| RTRClientTransactionController() | Constructor |
| ~RTRClientTransactionController() | Destructor |

### Basic Methods

| **Method** | **Description** |
|---|---|
| AcceptTransaction(rtr_reason_t) | Accept the current transaction. |
| Receive(RTRData, rtr_timout_t) | Receive an RTR or application- generated message or an RTR event. |
| RegsiterClassFactory<br><br>(RTRClassFactory ) | Register a class factory for RTR to call when creating RTR Data derived objects. |
| RegisterFacility(rtr_const_facnam_t, rtr_const_rcpspc_t, rtr_const_access_t) | Inform the controller that it should operate within the given facility. |
| RegisterHandlers<br><br>(RTRClientMessageHandler, RTRClientEventHandler ) | Register handlers for messages and events. |
| RejectTransaction(const rtr_reason_t) | Reject the current transaction. |
| SendApplicationEvent<br><br>(RTRApplicationEvent, rtr_const_rcpspc_t, rtr_const_msgfmt_t ) | Send an application-defined event to the server. |
| SendApplicationMessage<br><br>(RTRApplicationMessage, bool, bool, rtr_const_msgfmt_t ) | Send an application-defined message to the server. |

| Method | Description |
|---|---|
| StartTransaction(rtr_timout_t) | Start a new transaction. |

## AcceptTransaction()

AcceptTransaction() — RTRClientTransactionController::AcceptTransaction();

### Prototype

```
virtual rtr_status_t AcceptTransaction(rtr_reason_t rtrReasonCode =
                                                    RTR_NO_REASON);
```

### Return Value

**rtr_status_t** Interpret value for the success or failure of this call.

| Status | Message |
|---|---|
| RTR_STS_NOACCEPT | Client or Server has already voted or there is no active transaction. |
| RTR_STS_NOMESSAGE | The data object does not contain a message. |
| RTR_STS_OK | Normal successful completion. |
| RTR_STS_TXNOTACT | No transaction currently active on this channel. |

### Parameters

## rtrReasonCode

Optional reason for accepting the transaction. This reason is **OR** ed together with the reasons of the other participants in the transaction and returned to all participants of the transaction. The participants can retrieve this reason by calling RTRMessage::GetReason().

### Description

Call this member function to accept the transaction currently being processed by this object.

### Example

```
// Let RTR know that this is the only object being sent and that
// we are done with our work.
    cout << "AcceptTransaction..." << endl;
        sStatus = AcceptTransaction();
        print_status_on_failure(sStatus);
```

## Receive()

Receive() — RTRClientTransactionController::Receive();

### Prototype

```
virtual rtr_status_t Receive (RTRData **pRTRData,
                                  rtr_timout_t tTimeout = RTR_NO_TIMOUTMS);
```

## Return Value

**rtr_status_t** Interpret value for the success or failure of this call.

| **Status** | **Message** |
|---|---|
| RTR_STS_INVDATPTRPTARG | Invalid pointer-to-data-pointer pointer argument |
| RTR_STS_NORECEIVE | Attempting to receive at this point is not allowed. |
| RTR_STS_NOMESSAGE | The data object does not contain a message. |
| RTR_STS_OK | Normal successful completion. |
| RTR_STS_TIMOUT | Call to Receive timed out. |
| RTR_STS_TRUNCATED | Buffer too short for message. Message has been truncated. |

## Parameters

## pRTRData

A pointer passed by reference, which will receive an object, derived from RTRData. This object can be any of the following:

- RTRMessage
- RTREvent
- RTRApplictionMessage
- RTRApplicationEvent

If a class factory is registered with the transaction controller, the application has the ability to have this object be any application class derived from RTRData. By calling the Dispatch() method, the most over ridden implementation of dispatch will be called.

For more information see the description of the RTR receive model.

## tTimeout

An optional receive timeout value in milliseconds. If the timeout expires, the call completes with status RTR_STS_TIMOUT.

## Description

This member function should be called when the application is ready to receive messages and events from the RTR framework. Typically this function is called in a loop. The RTRData object returned contains the message or event type as well as other information useful to the application.

For more information see:

RTRData

### Example

```
abc_status ABCOrderTaker::DetermineOutcome()
{
        RTRData  *pResult = NULL;
        abc_status sStatus = ABCSuccess;
        bool bDone = false;
        while (!bDone)
        {
               sStatus = Receive(&pResult);
               print_status_on_failure(sStatus);
        }
delete pResult;
        return sStatus;
}
```

## RegisterClassFactory()

RegisterClassFactory() — RTRClientTransactionController::RegisterClassFactory();

### Prototype

```
virtual rtr_status_t RegisterClassFactory ( RTRClassFactory *pFactory);
```

### Return Value

**rtr_status_t** Interpret value for the success or failure of this call.

| **Status** | **Message** |
|---|---|
| RTR_STS_INVFACTORYPTARG | The factory argument pointer is invalid. |
| RTR_STS_OK | Normal successful completion |

### Parameters

## pFactory

Pointer to an RTRClassFactory object that is called, if registered, from the RTR framework when processing all Receive methods in your application.

### Description

Registering a class factory is not a requirement. An application would register a class factory only when they wish to customize the object that is being allocated.

### Example

```
    sStatus = RegisterClassFactory(*pFactory);
    print_status_on_failure(sStatus);
```

## RegisterFacility()

RegisterFacility() — RTRClientTransactionController::RegisterFacility();

## Prototype

```
virtual rtr_status_t RegisterFacility (rtr_const_facnam_t pszFacilityName,
                            rtr_const_rcpspc_t szRecipientName = "*",
                            rtr_const_access_t pszAccess = RTR_NO_ACCESS);
```

## Return Value

**rtr_status_t** Interpret value for the success or failure of this call.

| **Status** | **Message** |
|---|---|
| RTR_STS_INVACCSTRPTRARG | The access string argument is invalid. |
| RTR_STS_INVALIDFACILITY | The specified facility does not exist. |
| RTR_STS_INVFACNAMEARG | The facility name argument is invalid. |
| RTR_STS_INVRECPNAMPTARG | The recipient name argument is invalid. |
| RTR_STS_OK | Normal successful completion |
| RTR_STS_RTRNOTRUNNING | RTR is not running. |

## Parameters

## pszFacilityName

Pointer to a null-terminated facility name.

## szRecipientName

Name of the recipient. This null-terminated string contains the name of the recipient. This is an optional parameter.

Wildcards ("*" for any sequence of characters, and "%" for any one character) can be used in this string to address more than one recipient.

Note that *szRecipientName* is case sensitive.

## pszAccess

Pointer to a null-terminated string containing the access parameter. The default is RTR_NO_ACCESS.

## Description

Call the RegisterFacility() member function to register an RTR facility for your application. By registering a facility, your application informs RTR of the facility for which your application can process transactions.

## Example

```
// Register the facility with RTR.
        sStatus = RegisterFacility(ABCFacility);
        print_status_on_failure(sStatus);
if(RTR_STS_OK == sStatus)
      {
        m_bRegistered = true;
      }
```

## RegisterHandlers()

RegisterHandlers() — RTRClientTransactionController::RegisterHandlers();

### Prototype

```
virtual rtr_status_t RegisterHandlers (RTRClientMessageHandler
                                       *pMessageHandler,
                                       RTRClientEventHandler
                                       *pEventHandler);
```

### Return Value

**rtr_status_t** Interpret value for the success or failure of this call.

| **Status** | **Message** |
|---|---|
| RTR_STS_INVEVNTHNDPTARG | The event handler pointer argument is invalid. |
| RTR_STS_INVMSGHNDLPTARG | The message handler pointer argument is invalid. |
| RTR_STS_OK | Normal successful completion |

### Parameters

## pMessageHandler

Pointer to an RTRClientMessageHandler object that processes messages received.

## pEventHandler

Pointer to an RTRClientEventHandler object that processes events received.

### Description

Call the RegisterHandlers member function to register message and event handlers for your application. By registering an environment (a facility and a partition), your application informs RTR of the different configurations for which your application can process transactions. Your application will only use one environment at a time. The RTR framework picks the most efficient environment for your application depending on the number of client requests being received. If no environment is specified, RTR uses any of the previously defined environments in your applications process.

Specify pMessageHandler and/or pEventHandler in order for your application to make use of the RTR frameworks predefined handlers.

For more information on handlers see:

- RTRData::Dispatch
- RTRClientMessageHandler
- RTRClientMessageHandler

### Example

```
// ABC Handlers
// Register the both handlers with RTR
sStatus = RegisterHandlers(&m_rtrHandlers,&m_rtrHandlers);
print_status_on_failure(sStatus);
```

## RejectTransaction()

RejectTransaction() — RTRClientTransactionController::RejectTransaction();

### Prototype

```
virtual rtr_status_t RejectTransaction(const rtr_reason_t rtrReasonCode =
                                                          RTR_NO_REASON);
```

### Return Value

**rtr_status_t** Interpret value for the success or failure of this call.

| Status | Message |
|---|---|
| RTR_STS_FACNOTREG | Facility is not registered. |
| RTR_STS_NOMESSAGE | The data object does not contain a message. |
| RTR_STS_NOREJECT | Client or Server has already voted or there is no active transaction. |
| RTR_STS_OK | Normal successful completion. |
| RTR_STS_TXNOTACT | No transaction is currently active on this channel. |

### Parameters

## rtrReasonCode

Optional reason for rejecting the transaction. This reason is returned to the other participants in the transaction. The participants can retrieve this reason by calling RTRMessage::GetReason().

### Description

Call this member function to reject the transaction currently being processed by this object.

### Example

```
pController->RejectTransaction();
```

## RTRClientTransactionController()

RTRClientTransactionController() —
RTRClientTransactionController::RTRClientTransactionController();

### Prototype

```
RTRClientTransactionController();
```

```
virtual ~RTRClientTransactionController();
```

### Return Value

None.

### Parameters

None.

### Description

Call this constructor to create an RTRClientTransactionController object.

### Example

```
ABCOrderTaker::ABCOrderTaker():m_bRegistered(false)
{

}
```

## SendApplicationEvent()

SendApplicationEvent() — RTRClientTransactionController::SendApplicationEvent();

### Prototype

```
virtual rtr_status_t SendApplicationEvent(RTRApplicationEvent
                              * pRTRApplicationEvent,
                              rtr_const_rcpspc_t szRecipientName = "*",
                              rtr_const_msgfmt_t mfMessageFormat =
                              RTR_NO_MSGFMT);
```

### Return Value

**rtr_status_t** Interpret value for the success or failure of this call.

| Status | Message |
|---|---|
| RTR_STS_INSVIRMEM | Insufficient virtual memory. |
| RTR_STS_INVAPPEVNTPTARG | Invalid application event pointer argument. |
| RTR_STS_INVMSGFMTPTRARG | The message format string argument is invalid. |
| RTR_STS_INVRECPNAMPTARG | The recipient name argument is invalid. |
| RTR_STS_NOEVENT | The data object does not contain an event. |
| RTR_STS_NOMESSAGE | The data object does not contain a message. |
| RTR_STS_OK | Normal successful completion. |

### Parameters

## RTRApplicationEvent

Pointer to an RTRApplicationEvent object which contains application data to be sent to the server.

## szRecipientName

Name of the recipient. This null-terminated character string contains the name of the recipient. This is an optional parameter.

Wildcards ( "*" for any sequence of characters, and "%" for any one character) can be used in this string to address more than one recipient

Note that *szRecipientName* is case sensitive.

## mfMessageFormat

Message format description. mfMessageFormat is a null-terminated character string containing the format description of the message. RTR uses this description to convert the contents of the message appropriately when processing the message on different hardware platforms. If no parameter is specified, the default is no special formatting.

### Description

This member function should be called when the application wants to send an application-defined event to the server. Formerly, application-defined events are only delivered to the clients or servers that have subscribed for them and these are not related to any transaction. Only reply messages go to the client that started the transaction. Simply calling this function does not deliver the event to the client, unless it has subscribed for it. With the C++ API, you "subscribe" by overriding the event handler methods. The events are only received if they are overridden.

### Example

```
sStatus = SendApplicationEvent(pOrder);
print_status_on_failure(sStatus);
```

## SendApplicationMessage()

SendApplicationMessage() — RTRClientTransactionController::SendApplicationMessage();

### Prototype

```
virtual rtr_status_t SendApplicationMessage(RTRApplicationMessage
                                  *pRTRApplicationMessage,
                                  bool bReadonly = false,
                                  bool bReturnToSender = false,
                                  rtr_const_msgfmt_t
                                  mfMessageFormat = RTR_NO_MSGFMT);
```

### Return Value

**rtr_status_t** Interpret value for the success or failure of this call.

| Status | Message |
|---|---|
| RTR_STS_INSVIRMEM | Insufficient virtual memory. |
| RTR_STS_INVAPPMSGPTARG | Invalid application message pointer argument. |

| Status | Message |
|---|---|
| RTR_STS_INVMSGFMTPTRARG | The message format string argument is invalid. |
| RTR_STS_INVRECPNAMPTARG | The recipient name argument is invalid. |
| RTR_STS_NOMESSAGE | The data object does not contain a message. |
| RTR_STS_NOSEND | Attempting to send an application message at this point is not allowed. |
| RTR_STS_OK | Normal successful completion. |

### Parameters

## pRTRApplicationMessage

Pointer to an RTRApplicationMessage object which contains application data to be sent to the server.

## mfMessageFormat

Message format description. mfMessageFormat is a null-terminated character string containing the format description of the message. RTR uses this description to convert the contents of the message appropriately when processing the message on different hardware platforms. If no parameter is specified the default is no special formatting.

## bReadonly

Set this Boolean to true to indicate to RTR that this message does not perform and writes that would need to be shadowed.

## bReturnToSender

Set this Boolean to true to indicate to RTR that, if the message is not delivered to the server, it should return a to this transaction controller a message indicating that.

### Description

This member function should be called when the application wants to send application data to the server. The RTRData object contains the data to be sent.

For more information see:

RTRData

### Example

```
// Send this Book Order object to a server capable of processing it.
    cout << "SendApplicationMessage..." << endl;
        sStatus = SendApplicationMessage(pOrder);
        print_status_on_failure(sStatus);
```

## StartTransaction()

StartTransaction() — RTRClientTransactionController::StartTransaction();

### Prototype

```
virtual rtr_status_t StartTransaction(rtr_timout_t
                                  timeout = RTR_NO_TIMOUTMS);
```

### Return Value

**rtr_status_t** Interpret value for the success or failure of this call.

| Status | Message |
|---|---|
| RTR_STS_CONNECTIONLOST | An RTR connection has been lost. |
| RTR_STS_FACNOTREG | Facility is not registered. |
| RTR_STS_INVTIMOUTMS | Invalid timeout argument |
| RTR_STS_NOMESSAGE | The data object does not contain a message. |
| RTR_STS_NOSTARTTXN | A client transaction is already active. |
| RTR_STS_OK | Normal successful completion. |
| RTR_STS_TIMOUT | Timeout time expired. |

### Parameters

## timeout

Transaction timeout. This value is specified in milliseconds. If the timeout time expires, RTR aborts the transaction and returns status RTR_STS_TIMOUT. If no timeout is required, specify RTR_NO_TIMOUTMS.

### Description

Explicitly start a transaction from a client transaction controller. This method is mandatory.

The StartTransaction method is used to start a transaction explicitly. An explicit transaction start is only necessary if:

- either a join to an existing transaction is to be done
- or a transaction timeout is to be specified

Transactions are implicitly started when a message is sent on a currently inactive transaction controller. Implicitly started transactions have no timeout and are not joined to other RTR transactions.

### Example

```
    abc_status sStatus;

cout << "StartTransaction..." << endl;
    sStatus = StartTransaction();
    print_status_on_failure(sStatus);
```

## 3.10. RTRClientTransactionProperties

This class holds the properties of its associated RTRServerTransaction object.

## RTRClientTransactionProperties Class Members

## Construction

| Method | Description |
|---|---|
| RTRClientTransactionProperties() | Constructor |
| ~RTRClientTransactionProperties() | Destructor |

## RTRClientTransactionProperties()

RTRClientTransactionProperties() —
RTRClientTransactionProperties::RTRClientTransactionProperties();

### Prototype

```
RTRClientTransactionProperties(); virtual
~RTRClientTransactionProperties();
```

### Return Value

None.

### Parameters

None.

### Description

This class holds the properties of its associated RTRClientTransaction object.

### Example

```
RTRClientTransactionProperties::RTRClientTransactionProperties();
{
}
```

## 3.11. Data Classes and the Class Factory

The data classes of the C++ API are common to both server and client applications. There classes include:

- RTRData
- RTRApplicationMessage
- RTRApplicationEvent
- RTRMessage
- RTREvent
- RTRClassFactory

- RTRStream

The RTRData class is the base class of all the C++ foundation class data classes. When applications want to receive data they specify Pointer to an RTRData object. After a successful call to the Receive method in either the client or server RTRtransactionController class, RTRData contains one of the following:

- RTRMessage
- RTREvent
- RTRApplicationMessage
- RTRApplicationEvent

The data classes are common to both client and server applications.

The RTRStream class is an RTRData-derived class designed for RTRApplicationMessage and RTRApplicationEvent data objects to read from and write to a buffer.

The RTRClassFactory class creates instances of data classes based on the contents of a Receive call for a message or event. For more information on RTR message and event processing, see the *VSI Reliable Transaction Router Application Design Guide*

## 3.12. RTRApplicationEvent Class

### RTRApplicationEvent Class Members

### Construction

| **Method** | **Description** |
|---|---|
| RTRApplicationEvent () | Default constructor |
| RTRApplicationEvent<br><br>(RTRApplicationEvent & ) | Copy constructor |

### Operations

| **Method** | **Description** |
|---|---|
| Dispatch() | Basic method. |
| GetEventData( rtr_msgbuf_t ) | Retrieve the application data associated with this RTRApplicationEvent object. |
| GetEventDataLength(); | Retrieve the actual length of the buffer allocated for this RTRApplicationEvent object. |
| GetEventNumber( rtr_evtnum_t ) | Retrieve the application event associated with the data in this RTRApplicationEvent object. |
| SetEventData( rtr_msgbuf_t,<br><br>rtr_msglen_t) | Set the actual data length of the buffer allocated for this RTRApplicationEvent object. |
| SetEventNumber( const rtr_evtnum_t) | Set the application event number associated with the data in this RTRApplicationEvent object. |

## Dispatch()

Dispatch() — RTRApplicationEvent::Dispatch();

### Prototype

```
rtr_status_t Dispatch();
```

### Return Value

**rtr_status_t** Interpret value for the success or failure of this call.

| **Status** | **Message** |
|---|---|
| RTR_STS_HANDLERDELETED | The application has deleted the handler. |
| RTR_STS_NOEVENT | The data object does not contain an event. |
| RTR_STS_NOEVENTDATA | There is no event data associated with the event. |
| RTR_STS_NOHANDLRREGSTRD | The application has not registered a handler |
| RTR_STS_NOMESSAGE | The data object does not contain a message. |
| RTR_STS_OK | Normal successful completion. |
| RTR_STS_TCDELETED | The application has deleted the transaction controller. |

### Parameters

None.

### Description

This member function must be overridden by the RTR application. When called, the data contained within the object is processed. Processing the data may include performing some application specific logic and/or dispatching to a handler.

### Example

```
sStatus = pApplicationEvent->Dispatch();
{
}
```

## GetEventData()

GetEventData() — RTRApplicationEvent::GetEventData();

### Prototype

```
rtr_status_t GetEventData( rtr_msgbuf_t &evEventData );
```

### Return Value

**rtr_status_t** Interpret value for the success or failure of this call.

| Status | Message |
|---|---|
| RTR_STS_INVARGPTR | Invalid argument pointer. |
| RTR_STS_NOEVENT | The data object does not contain an event. |
| RTR_STS_NOEVENTDATA | There is no event data associated with the event. |
| RTR_STS_OK | Normal successful completion |

### Parameters

## evEventData

Pointer to event data.

### Description

Retrieve the application data associated with this RTRApplicationEvent object.

### Example

```
GetEventData(&evEventData );
```

## GetEventDataLength()

GetEventDataLength() — RTRApplicationEvent::GetEventDataLength();

### Prototype

```
rtr_msglen_t GetEventDataLength();
```

### Return Value

**rtr_msglen_t:** Returns the size of the event data length.

### Parameters

None.

### Description

Call this member function to receive the size of the application event data length.

### Example

```
rtr_msglen_t LengthOfData =
                        pRTRApplicationEvent->GetEventDataLength();
```

## GetEventNumber()

GetEventNumber() — RTRApplicationEvent::GetEventNumber();

### Prototype

```
rtr_status_t GetEventNumber (rtr_evtnum_t &evEventNumber );
```

### Return Value

**rtr_status_t** Interpret value for the success or failure of this call.

| **Status** | **Message** |
|---|---|
| RTR_STS_NOEVENT | The data object does not contain an event. |
| RTR_STS_NOMESSAGE | The data object does not contain a message |
| RTR_STS_OK | Normal successful completion. |

### Parameters

## evEventNumber

An event number.

### Description

Get the event number associated with the received application event.

### Example

```
GetEventNumber (&evEventNumber );
```

## SetEventData()

SetEventData() — RTRApplicationEvent::SetEventData();

### Prototype

```
rtr_status_t SetEventData (rtr_msgbuf_t &evEventData, rtr_msglen_t
dlDataLength);
```

### Return Value

**rtr_status_t** Interpret value for the success or failure of this call.

| **Status** | **Message** |
|---|---|
| RTR_STS_OK | Normal successful completion. |
| RTR_STS_INVARGPTR | The data object does not contain an event. |

### Parameters

## evEventData

Pointer to event data.

## dlDataLength

The length of the data.

### Description

Set the application data associated with this RTRApplicationEvent object.

### Example

```
SetEventData (&evEventData, dlDataLength);
```

## SetEventNumber()

SetEventNumber() — RTRApplicationEvent::SetEventNumber();

### Prototype

```
rtr_status_t SetEventNumber (const rtr_evtnum_t &evEventNumber );
```

### Return Value

**rtr_status_t** Interpret value for the success or failure of this call. RTR_STS_OK is the normal successful completion.

### Parameters

## evEventNumber

An event number.

### Description

Set the application event number associated with the data in this RTRApplicationEvent object.

### Example

```
SetEventNumber (&evEventNumber );
```

# 3.13. RTRApplicationMessage Class

## RTRApplicationMessage Class Members

## Construction

| **Method** | **Description** |
|---|---|
| RTRApplicationMessage() | Default constructor |
| ~RTRApplicationMessage () | Default destructor |

## Operations

| **Method** | **Description** |
|---|---|
| Dispatch() | Basic method. |

| Method | Description |
|---|---|
| GetMessage() | Retrieve the message associated with the data in this object. |
| GetMessageLength() | Retrieve the actual length of the message associated with the data in this object. |

## Dispatch()

Dispatch() — RTRApplicationMessage::Dispatch();

### Prototype

```
rtr_status_t Dispatch();
```

### Return Value

**rtr_status_t** Interpret value for the success or failure of this call.

| Status | Message |
|---|---|
| RTR_STS_HANDLERDELETED | The application has deleted the handler. |
| RTR_STS_NOHANDLRREGSTRD | The application has not registered a handler |
| RTR_STS_NOMESSAGE | The data object does not contain a message |
| RTR_STS_OK | Normal successful completion |
| RTR_STS_TCDELETED | The application has deleted the transaction controller. |

### Parameters

None.

### Description

This member function must be overridden by the RTR application. When called the data contained within the object is processed. Processing the data may include performing some application specific logic and/or dispatching to a handler.

### Example

```
void ABCOrderProcessor::ProcessIncomingOrders()
{
        abc_status sStatus = RTR_STS_OK;
        RTRData *pOrder = NULL;
        while (1)
        {
        sStatus = pOrder->Dispatch();
        print_status_on_failure(sStatus);
        delete pOrder;
        }
return;
}
```

## GetMessage()

GetMessage() — RTRApplicationMessage::GetMessage();

### Prototype

```
rtr_msgbuf_t GetMessage();
```

### Return Value

**rtr_status_t** Interpret value for the success or failure of this call.

| **Status** | **Message** |
|---|---|
| RTR_STS_OK | Normal successful completion. |
| RTR_STS_NOMESSAGE | The data object does not contain a message |

### Parameters

None.

### Description

Retrieve the message associated with the data in this object.

### Example

```
RTRApplicationMessage.GetMessage();
```

## GetMessageLength()

GetMessageLength() — RTRApplicationMessage::GetMessageLength();

### Prototype

```
rtr_msgbuf_t GetMessageLength();
```

### Return Value

**rtr_status_t** Interpret value for the success or failure of this call.

| **Status** | **Message** |
|---|---|
| RTR_STS_OK | Normal successful completion. |
| RTR_STS_NOMESSAGE | The data object does not contain a message |

### Parameters

None.

### Description

Retrieve the actual length of the message associated with the data in this object.

### Example

```
RTRApplicationMessage.GetMessageLength();
```

## 3.14. RTRClassFactory Class

The RTRClassFactory class constructs an RTR application event or message directly from an RTR message data buffer.

### RTRClassFactory Class Members

### Construction

| Method | Description |
|---|---|
| RTRClassFactory() | Default constructor |
| ~RTRClassFactory() | Default destructor |

### Operations

| Method | Description |
|---|---|
| CreateRTRApplicationEvent<br><br>(rtr_const_msgbuf_t, rtr_msglen_t, RTRApplicationEvent ) | Create an RTRApplicationEvent data object. |
| CreateRTRApplicationMessage<br><br>(rtr_const_msgbuf_t, rtr_msglen_t, RTRApplicationMessage ) | Create an RTRApplicationMessage data object. |
| CreateRTREvent(RTREvent) | Create an RTREvent data object. |
| CreateRTRMessage(RTRMessage) | Create an RTRMessage data object. |

### CreateRTRApplicationEvent()

CreateRTRApplicationEvent() — RTRClassFactory::CreateRTRApplicationEvent();

#### Prototype

```
virtual rtr_status_t CreateRTRApplicationEvent(rtr_const_msgbuf_t
                                                   pmsgCallersData,
                                rtr_msglen_t msglCallersDataLength
                                RTRApplicationEvent *&pApplicationEvent)
{
    rtr_status_t sStatus = RTR_STS_OK;
    pApplicationEvent = new RTRApplicationEvent();
    if (NULL == pApplicationEvent);
    {
      sStatus = RTR_STS_INSVIRMEM;
    }
    return sStatus;
};
```

### Return Value

**rtr_status_t** Interpret value for the success or failure of this call.

| Status | Message |
|---|---|
| RTR_STS_OK | Normal successful completion. |
| RTR_STS_INSVIREM | Insufficient virtual memory. |

### Parameters

## pmsgCallersData

Pointer to the caller's data.

## msglCallersDataLength

The length of the caller's data.

## pApplicationEvent

Pointer to the application event.

### Description

Create an RTRApplicationEvent data object if the transaction controller determines that Receive call points to a message of type RTRApplicationEvent.

### Example

```
pApplicationEvent = new ApplicationEvent();
```

## CreateRTRApplicationMessage()

CreateRTRApplicationMessage() — RTRClassFactory::CreateRTRApplicationMessage();

### Prototype

```
virtual rtr_status_t CreateRTRApplicationMessage(rtr_const_msgbuf_t
                                                 pmsgCallersData,
                                   rtr_msglen_t msglCallersDataLength,
                           RTRApplicationMessage *&pApplicationMessage)
{
    rtr_status_t sStatus = RTR_STS_OK;
    pApplicationMessage = new RTRApplicationMessage();
    if (NULL == pApplicationMessage)
    {
    sStatus = RTR_STS_INSVIRMEM;
    }
    return sStatus;
};
```

## Return Value

**rtr_status_t** Interpret value for the success or failure of this call.

| Status | Message |
|---|---|
| RTR_STS_OK | Normal successful completion. |
| RTR_STS_INSVIREM | |

## Parameters

## pmsgCallersData

Pointer to the caller's data.

## msglCallersDataLength

The length of the caller's data.

## pApplicationMessage

Pointer to the application message.

## Description

Create an RTRApplicationMessage data object if the transaction controller determines that Receive call points to a message of type RTRApplicationMessage.

## Example

```
rtr_status_t ABCSClassFactory::CreateRTRApplicationMessage(
rtr_const_msgbuf_t pmsgCallersData,
rtr_msglen_t msglCallersDataLength,
RTRApplicationMessage *&pApplicationMessage )
{ // Determine what kind of serialized object we are receiving.
  // The ABC company protocol defines the first integer of the
  // message to represent the type of the object we are receiving.
  // Book = ABC_BOOK. Magazine = ABC_MAGAZINE unsigned int
  // uiClassType = *(unsigned int*)pmsgCallersData;
switch (uiClassType)
    {
  case ABC_BOOK : pApplicationMessage = new ABCBook(); break;
  case ABC_MAGAZINE : pApplicationMessage = new ABCMagazine(); break;
  default:
    // If we ever get here then the client is sending us data that we
    // can't recognize. For some applictations this may not be an
    // issue. For the ABC company this should be impossible.
    assert(false);
    }
// Make sure we are passing back a valid address
if (NULL == pApplicationMessage)
return RTR_STS_INSVIRMEM;
return ABC_STS_SUCCESS;}
```

## CreateRTREvent()

CreateRTREvent() — RTRClassFactory::CreateRTREvent();

### Prototype

```
virtual rtr_status_t CreateRTREvent( RTREvent *&pRTREvent)
{
  rtr_status_t sStatus = RTR_STS_OK;
  pRTREvent = new RTREvent();
  if (NULL == pRTREvent)
  {
    sStatus = RTR_STS_INSVIRMEM;
  }
  return sStatus;
};
```

### Return Value

**rtr_status_t** Interpret value for the success or failure of this call.

| **Status** | **Message** |
|---|---|
| RTR_STS_OK | Normal successful completion. |
| RTR_STS_INSVIREM | Insufficient virtual memory. |

### Parameters

## pRTREvent

Pointer to an RTREvent object that describes the message which is being processed.

### Description

Create an RTREvent data object if the transaction controller determines that Receive call points to a message of type RTREvent.

### Example

```
CreateRTREvent(*&pRTREvent)
```

## CreateRTRMessage()

CreateRTRMessage() — RTRClassFactory::CreateRTRMessage ();

### Prototype

```
virtual rtr_status_t CreateRTRMessage( RTRMessage *&pRTRMessage)
{
  rtr_status_t sStatus = RTR_STS_OK;
  pRTRMessage = new RTRMessage();
  if (NULL == pRTRMessage)
  {
    sStatus = RTR_STS_INSVIRMEM;
  }
  return sStatus;
};
```

### Return Value

**rtr_status_t** Interpret value for the success or failure of this call.

| Status | Message |
|---|---|
| RTR_STS_OK | Normal successful completion. |
| RTR_STS_INSVIREM | |

### Parameters

## pRTRMessage

Pointer to an RTRMessage object that describes the message which is being processed.

### Description

Create an RTRMessage data object if the transaction controller determines that Receive call points to a message of type RTRMessage.

### Example

```
pApplicationMessage = new ApplicationMessage();
```

## 3.15. RTRData

RTRData is the abstract base class for all data classes.

## RTRData Class Members

## Construction

| Method | Description |
|---|---|
| RTRData() | Default constructor |
| RTRData() | Default destructor |

## Operations

| Method | Description |
|---|---|
| Dispatch() | Basic method. |
| GetActualBufferLength() | Return the message buffer length. |
| GetLogicalBufferLength() | Return the logical buffer length. |
| IsApplicationEvent() | Determine if this object contains application-generated data. |

| Method | Description |
|---|---|
| IsApplicationMessage() | Determine if this object contains application-generated message. |
| IsEvent() | Determine if this object contains an RTR or application-generated event. |
| IsMessage() | Determine if this object contains an RTR or application-generated message. |
| IsRTREvent() | Determine if this object contains RTR-generated data. |
| IsRTRMessage() | Determine if this object contains RTR-generated message. |

## Dispatch()

Dispatch() — RTRData::Dispatch();

### Prototype

```
virtual rtr_status_t Dispatch() = 0;
```

### Return Value

**rtr_status_t** Interpret value for the success or failure of this call. RTR_STS_OK is the normal successful completion.

### Parameters

None.

### Description

This is a pure virtual member function. RTRData does not supply an implementation for Dispatch and therefore cannot be instantiated. All classes that derive from RTRData must implement their own version of Dispatch, with the functionality based on their needs.

## GetActualBufferLength()

GetActualBufferLength() — RTRData::GetActualBufferLength();

### Prototype

```
rtr_msglen_t GetActualBufferLength ();
```

### Return Value

**rtr_msglen_t** The message buffer length.

### Parameters

None.

### Description

The method returns the message buffer length.

### Example

```
GetActualBufferLength ();
```

## GetLogicalBufferLength()

GetLogicalBufferLength() — RTRData::GetLogicalBufferLength();

### Prototype

```
rtr_msglen_t GetLogicalBufferLength();
```

### Return Value

**rtr_msglen_t** Return the logical buffer length.

### Parameters

None.

### Description

Call this method for the logical buffer length.

### Example

```
GetLogicalBufferLength();
```

## IsApplicationEvent()

IsApplicationEvent() — RTRData::IsApplicationEvent();

### Prototype

```
bool IsApplicationEvent ();
```

### Return Value

**bool** A true or false return value.

### Parameters

None.

### Description

If the RTRData object contains an event sent by the application, this function returns TRUE. Otherwise it returns FALSE.

### Example

```
sStatus = Receive(&pResult);
print_status_on_failure(sStatus);
if ( true == pResult->IsApplicationEvent();)
```

## IsApplicationMessage()

IsApplicationMessage() — RTRData::IsApplicationMessage();

### Prototype

```
bool IsApplicationMessage();
```

### Return Value

**bool** A true or false return value.

### Parameters

None.

### Description

If the RTRData object contains a message sent by the application, this function returns TRUE. Otherwise it returns FALSE.

### Example

```
sStatus = Receive(&pResult);
print_status_on_failure(sStatus);
if ( true == pResult->IsApplicationMessage();)
```

## IsEvent()

IsEvent() — RTRData::IsEvent();

### Prototype

```
bool IsEvent();
```

### Return Value

**bool** A true or false return value.

### Parameters

None.

### Description

If the RTRData object contains an event, generated by either RTR or an application, this function returns TRUE. Otherwise it returns FALSE.

### Example

```
if (IsEvent();)
{
    rtr_evtnum_t enEvent;
    sStatus = GetEventNumber(enEvent);
}
```

## IsMessage()

IsMessage() — RTRData::IsMessage();

### Prototype

```
bool IsMessage();
```

### Return Value

**bool** A true or false return value.

### Parameters

None.

### Description

If the RTRData object contains a message, generated by either RTR or an application, this function returns TRUE. Otherwise it returns FALSE.

### Example

```
// Look for a status for this transaction.
RTRData *pTransactionData = new RTRData();
sStatus = GetTransaction()->Receive(pTransactionData);
// Determine if we have a message or an event
if (false == pTransactionData->IsMessage();)
    {
    pTransactionData->Dispatch();
    }
```

## IsRTREvent()

IsRTREvent() — RTRData::IsRTREvent();

### Prototype

```
bool IsRTREvent();
```

### Return Value

**bool** A true or false return value.

### Parameters

None.

### Description

If the RTRData object contains an event sent by the application, this function returns TRUE. Otherwise it returns FALSE.

### Example

```
sStatus = Receive(&pResult);
print_status_on_failure(sStatus);
if ( true == pResult->IsRTREvent();)
```

## IsRTRMessage()

IsRTRMessage() — RTRData::IsRTRMessage();

### Prototype

```
bool IsRTRMessage();
```

### Return Value

**bool** A true or false return value.

### Parameters

None.

### Description

If the RTRData object contains a message sent by the application, this function returns TRUE. Otherwise it returns FALSE.

### Example

```
sStatus = Receive(&pResult);
print_status_on_failure(sStatus);
if ( true == pResult->IsRTRMessage();)
```

## RTRData()

RTRData() — RTRData::RTRData();

### Prototype

```
RTRData(); virtual ~RTRData();
```

### Parameters

None.

### Description

This constructor is a pure virtual function and requires an associated higher-level data object (for example, RTRApplicationMessage). The default constructor should be used by applications when receiving data from a call to Receive that does not intend to handle allocation and de-allocation of memory for the call. By using this form of the constructor, the application requests that RTR allocate enough memory to receive the data.

## 3.16. RTREvent Class

### RTREvent Class Members

### Construction

| Method | Description |
|---|---|
| RTREvent() | Default constructor |
| ~RTREvent() | Default destructor |

### Operations

| Method | Description |
|---|---|
| Dispatch() | Basic method. |
| GetEventData( rtr_msgbuf_t ) | Retrieve the RTR data associated with this RTREvent object. |
| GetEventDataLength(); | Retrieve the actual length of the data associated for this RTREvent object. |
| GetEventNumber( rtr_evtnum_t ) | Retreive the RTR event associated with the data in this RTREvent object. |

### Dispatch()

Dispatch() — RTREvent::Dispatch();

#### Prototype

```
rtr_status_t Dispatch();
```

#### Return Value

**rtr_status_t** Interpret value for the success or failure of this call.

| Status | Message |
|---|---|
| RTR_STS_OK | Normal successful completion |
| RTR_STS_TCDELETED | The application has deleted the transaction controller. |
| RTR_STS_EVENT | The data object does not contain an event. |
| RTR_STS_NOEVENTDATA | There is no event data associated with the event. |
| RTR_STS_MESSAGE | The data object does not contain a message. |
| RTR_STS_HANDLERDELETED | The application has deleted the handler. |

| Status | Message |
|---|---|
| RTR_STS_NOHANDLRREGSTRD | The application has not registered a handler |

## Parameters

None.

## Description

This member function must be overridden by the RTR application. When called the data contained within the object is processed. Processing the data may include performing some application specific logic and/or dispatching to a handler.

## Example

```
sStatus = pOrderEvent->Dispatch();
```

## GetEventData()

GetEventData() — RTREvent::GetEventData();

### Prototype

```
rtr_status_t GetEventData( rtr_msgbuf_t &evEventData );
```

### Return Value

**rtr_status_t** Interpret value for the success or failure of this call.

| Status | Message |
|---|---|
| RTR_STS_OK | Normal successful completion |
| RTR_STS_INVARGPTR | |
| RTR_STS_NOEVENTDATA | There is no event data associated with the event. |

### Parameters

## evEventData

Pointer to event data.

### Description

Retrieve the RTR data associated with this RTREvent object.

### Example

```
RTREvent.GetEventData(&evEventData);
```

## GetEventDataLength()

GetEventDataLength() — RTREvent::GetEventDataLength();

### Prototype

```
 rtr_msglen_t GetEventDataLength();
```

### Return Value

**rtr_msglen_t:** Returns the size of the event data length.

### Parameters

None.

### Description

Retrieve the actual length of the data associated for this RTREvent object.

### Example

```
RTREvent.GetEventDataLength();
```

## GetEventNumber()

GetEventNumber() — RTREvent::GetEventNumber();

### Prototype

```
rtr_status_t GetEventNumber( rtr_evtnum_t &evEventNumber);
```

### Return Value

**rtr_status_t** Interpret value for the success or failure of this call.

| **Status** | **Message** |
|---|---|
| RTR_STS_OK | Normal successful completion |
| RTR_STS_NOEVENT | The data object does not contain an event. |

### Parameters

## evEventNumber

An event number.

### Description

Call this member function to retrieve the RTR event associated with the data in this RTREvent object. This function is typically used by only those applications that do not register an event.

### Example

```
RTREvent.GetEventNumber(&evEventNumber);
```

## 3.17. RTRMessage

### RTRMessage

### Construction

| Method | Description |
|---|---|
| RTRMessage() | Default constructor |
| ~RTRMessage() | Default destructor |

### Operations

| Method | Description |
|---|---|
| Dispatch() | Basic method. |
| GetMessageType(rtr_msg_type_t) | Retrieve the RTR message associated with the data in this RTRMessage object. |
| GetReason() | Retrieve the reason associated with the accepting or rejection of the transaction. |
| GetSecondaryStatus() | Retrieve the secondary status associated with the accepting or rejection of the transaction. |

### Dispatch()

Dispatch() — RTRMessage::Dispatch();

#### Prototype

```
rtr_status_t Dispatch();
```

#### Return Value

**rtr_status_t** Interpret value for the success or failure of this call.

| Status | Message |
|---|---|
| RTR_STS_OK | Normal successful completion. |
| RTR_STS_TCDELETED | The application has deleted the transaction controller. |
| RTR_STS_NOMESSAGE | The data object does not contain a message. |
| RTR_STS_HANDLERDELETED | The application has deleted the handler. |
| RTR_STS_NOHANDLRREGSTRD | The application has not registered a handler. |

#### Parameters

None.

### Description

This member function must be overridden by the RTR application. When called, the data contained within the object is processed. Processing the data may include performing some application specific logic and/or dispatching to a handler.

### Example

```
    sStatus = pOrderMessage->Dispatch();
```

## GetMessageType()

GetMessageType() — RTRMessage::GetMessageType();

### Prototype

```
rtr_status_t GetMessageType(rtr_msg_type_t& mtMessageType);
```

### Return Value

**rtr_status_t** Interpret value for the success or failure of this call.

| **Status** | **Message** |
|---|---|
| RTR_STS_OK | Normal successful completion |
| RTR_STS_NOMESSAGE | The data object does not contain a message |

### Parameters

## mtMessageType

An RTR message.

### Description

Retrieve the RTR message associated with the data in this RTRMessage object.

### Example

```
sStatus = ((RTRMessage*)pResult)->GetMessageType(mtMessageType);
print_status_on_failure(sStatus);
```

## GetReason()

GetReason() — RTRMessage::GetReason();

### Prototype

```
rtr_reason_t GetReason();
```

### Return Value

**rtr_status_t** This function either returns RTR_NO_REASON or the value specified by the participants in the transaction. If different participants provide different reason codes, RTR ORs them.

### Parameters

None.

### Description

Retrieve the reason associated with the accepting or rejection of the transaction.

### Example

```
void OnAccepted(RTRMessage* pRTRData,
        RTRClientTransactionController* pController)
{
        rtr_status_t sStatus =
               pRTRData->GetSecondaryStatus();
        rtr_reason_t rcReasonCode = pRTRData->GetReason();
        m_bAcceptReceived = true;
};
```

## GetSecondaryStatus

GetSecondaryStatus — RTRMessage::GetSecondaryStatus();

### Prototype

```
rtr_status_t GetSecondaryStatus();
```

### Return Value

**rtr_status_t** Interpret value for the success or failure of this call.

| Status | Message |
|---|---|
| RTR_STS_OK | Normal successful completion |
| RTR_STS_NOMESSAGE | The data object does not contain a message. |

### Parameters

None.

### Description

Retrieve the secondary status associated with the accepting or rejection of the transaction.

### Example

```
void OnAccepted(RTRMessage* pRTRData,
       RTRClientTransactionController* pController)
{
      rtr_status_t sStatus =
             pRTRData->GetSecondaryStatus();
        rtr_reason_t rcReasonCode = pRTRData->GetReason();
        m_bAcceptReceived = true;
};
```

## 3.18. RTRStream Class

The RTRStream class derives from RTRData and extends the RTRData class by allowing RTR applications to issue multiple read and write requests to the buffer (managed by RTRData) without needing to maintain Pointer to the end of the buffer.

An RTRStream object automatically handles the details of maintaining the offset within the buffer when the application wants to read and write multiple times to a buffer.

When reading from and writing to a stream, a copy of the data is performed.

### RTRStream

### Construction

| Method | Description |
|---|---|
| RTRStream() | Default constructor |
| ~RTRStream | Default destructor |

### Operations

| Method | Description |
|---|---|
| WriteToStream(rtr _msgbuf_t,<br><br>rtr_msglen_t); | Copy data to the end of the buffer managed by RTRData. |
| WriteToStream(const char); | Copy string to the end of the buffer managed by RTRData. |
| WriteToStream(rtr_sgn_32_t); | Copy the signed integer to the end of the buffer managed by RTRData. |
| WriteToStream(rtr_uns_32_t); | Copy the unsigned integer to the end of the buffer managed by RTRData. |
| ReadFromStream(rtr _msgbuf_t, rtr_msglen_t, rtr_msglen_t); | Copy the data from the buffer managed by RTRData to the buffer specified. |
| ReadFromStream(rtr_sgn_32_t); | Copy the signed integer from the buffer managed by RTRData to uiNumber. |
| ReadFromStream(char, size_t); | Copy the data from the buffer managed by RTRData to pString. |
| ReadFromStream(rtr_uns_32_t); | Copy the unsigned integer from the buffer managed by RTRData to uiNumber. |

### Operators

| Operator | Description |
|---|---|
| RTRStream& operator>> (char) | ReadFromStream operator |

| Operator | Description |
|---|---|
| RTRStream& operator>> (rtr_sgn_32_t ) | ReadFromStream operator |
| RTRStream& operator>> (rtr_uns_32_t ) | ReadFromStream operator |
| RTRStream& operator < < (const char ) | WriteToStream operator |
| RTRStream& operator < < (rtr_sgn_32_t ) | WriteToStream operator |
| RTRStream& operator < < (rtr_uns_32_t ) | WriteToStream operator |

## operator>>

operator>> — RTRStream::operator>>

### Prototype

```
RTRStream& operator>> (char *pString)
  {
        ReadFromStream(pString);
        return *this;
  }
RTRStream& operator>> (rtr_sgn_32_t &iNumber)
  {
        ReadFromStream(iNumber);
        return *this;
  }
RTRStream& operator>> (rtr_uns_32_t &uiNumber)
  {
        ReadFromStream(uiNumber);
        return *this;
  }
```

### Return Value

**rtr_status_t** Interpret value for the success or failure of this call.

| Status | Message |
|---|---|
| RTR_STS_OK | Normal successful completion |
| RTR_STS_INVARGPTR | |

### Parameters

## pString

Pointer to a character string.

## iNumber

A signed integer.

## uiNumber

An unsigned integer.

### Description

>> denotes the ReadFromStream operators. These member functions extract data from a buffer by calling ReadFromStream to read the data and return *this. The three types of stream data are string, signed, and unsigned.

### Example

```
// Populate this object with the data
    *this >> m_uiPrice >> m_uiISBN  >> m_szTitle >> m_szAuthor;

// The 1 line call above is equivilant to the 4 lines below.
//        ReadFromStream(m_uiISBN);
//        ReadFromStream(m_uiPrice);
//        ReadFromStream(m_szTitle);
//        ReadFromStream(m_szAuthor);
```

## operator<<

operator<< — RTRStream::operator < <

### Prototype

```
RTRStream& operator<< (char *pString)
  {
        WriteToStream(pString);
        return *this;
  }
RTRStream& operator<< (rtr_sgn_32_t &iNumber)
  {
        WriteToStream(iNumber);
        return *this;
  }
RTRStream& operator<< (rtr_uns_32_t &uiNumber)
  {
        WriteToStream(uiNumber);
        return *this;
  }
```

### Return Value

**rtr_status_t** Interpret value for the success or failure of this call.

| Status | Message |
|---|---|
| RTR_STS_OK | Normal successful completion |

| Status | Message |
|---|---|
| RTR_STS_INVSTRINGPTRARG | The string pointer argument is invalid - string too long. |

### Parameters

## pString

Pointer to a character string.

## iNumber

A signed integer.

## uiNumber

An unsigned integer.

### Description

< < denotes the WriteToStream operators. These member functions write data to a buffer by calling WriteToStream to write the data and return *this. The three types of stream data are string, signed, and unsigned.

### Example

```
// Save the type of object we are. This is used by the class factory
// on the server side to determine which type of class to allocate.
    *this << ABC_BOOK;
    *this << m_uiPrice << m_uiISBN  << m_szTitle << m_szAuthor;
// The 1 line call above is equivalent to the 4 lines below. We
// can use the << and >> operators because we know that the data
// which we store is not > the current RTR maximum  = 65535 bytes.
//        WriteToStream(m_uiISBN);
//        WriteToStream(m_uiPrice);
//        WriteToStream(m_szTitle);
//        WriteToStream(m_szAuthor);
```

## ReadFromStream()

ReadFromStream() — RTRStream::ReadFromStream();

### Prototype

```
rtr_status_t ReadFromStream(rtr_msgbuf_t pvBuffer,
                             rtr_msglen_t &uiBufferSize
                             rtr_msglen_t &uiSizeCopied);

rtr_status_t ReadFromStream(char *pString, size_t uiStringSize);

rtr_status_t ReadFromStream(rtr_sgn_32_t &iNumber);

rtr_status_t ReadFromStream(rtr_uns_32_t &uiNumber);
```

### Return Value

**rtr_status_t** Interpret value for the success or failure of this call.

| Status | Message |
|---|---|
| RTR_STS_OK | Normal successful completion. |
| RTR_STS_APPBUFFTOOSMALL | The application buffer is too small. |
| RTR_STS_ENDOFSTREAM | The end-of-stream has been reached. |

### Parameters

## uiBufferSize

An unsigned integer for length of the buffer.

## pvBuffer

A void pointer to a buffer.

## uiNumber

An unsigned integer.

## pString

Pointer to a character string.

## iNumber

A signed integer.

### Description

Reads the first instance of a data type from a buffer as specified in the ReadFromStream methods. Note that the string buffer is assumed to be large enough (RTR_MAX_MSGLEN).

### Example

```
RTRStream::ReadFromStream(pString);
```

## RTRStream()

RTRStream() — RTRStream::RTRStream();

### Prototype

```
RTRStream();
```

### Return Value

None.

### Parameters

None.

### Description

Constructor method for the RTRStream class.

### Example

```
RTRStream::RTRStream();
```

## WriteToStream()

WriteToStream() — RTRStream::WriteToStream();

### Prototype

```
rtr_status_t WriteToStream(rtr_const_msgbuf_t pvBuffer, rtr_msglen_t
                                                         uiBufferLength);

rtr_status_t WriteToStream(const char *pString);

rtr_status_t WriteToStream(rtr_sgn_32_t iNumber);

rtr_status_t WriteToStream(rtr_uns_32_t uiNumber);
```

### Return Value

**rtr_status_t** Interpret value for the success or failure of this call.

| Status | Message |
|---|---|
| RTR_STS_OK | Normal successful completion. |
| RTR_STS_ENDOFSTREAM | The end-of-stream has been reached. |

### Parameters

## uiBufferLength

An unsigned integer for length of the buffer.

## pvBuffer

A void pointer to a buffer.

## uiNumber

An unsigned integer.

## iNumber

A signed integer.

## pString

Pointer to a character string.

### Description

Write to a data buffer, specifying the data with either buffer and buffer length, as unsigned integer, or string.

### Example

```
RTRStream::WriteToStream(uiNumber);
```

# Chapter 4. Management Classes

Management classes are offered for both new and existing RTR applications. The types of management classes include:

- Setup class:

  RTR class

- Facility classes:
  - RTRFacilityMember class
  - RTRFacilityMemberArray class
  - RTRFacilityProperties class
  - RTRFacilityManager class
- Partition classes:
  - RTRBackendPartitonProperties class
  - RTRKeySegment class
  - RTRKeySegmentArray
  - RTRPartitionManager class
- Counter classes:
  - RTRCounter class
  - RTRSignedCounter class
  - RTRStringCounter class
  - RTRUnsignedCounter class

## 4.1. RTR

The RTR class is a setup class, for RTR system management operations, designed for starting and stopping RTR, and creating and deleting RTR journals.

### RTR Class Members

### Construction

| **Method** | **Description** |
|---|---|
| RTR() | Constructor |
| ~RTR() | Destructor |

## Operations

| Method | Description |
|---|---|
| CreateJournal(bool) | Create a journal for RTR. |
| DeleteJournal() | Delete the journal for RTR. |
| GetErrorText(rtr_status_t) | Get the error text associated with the rtr_status_t return value. |
| IsRunning() | Determine if RTR is running. |
| Start() | Start RTR. |
| StartWebServer(bool, bool) | Start RTR on a web server. |
| Stop() | Stop RTR. |
| StopWebServer() | Stop RTR on a web server. |

## CreateJournal()

CreateJournal() — RTR::CreateJournal();

### Prototype

```
rtr_status_t CreateJournal( bool bSupersede = false);
```

### Return Value

**rtr_status_t** Interpret value for the success or failure of this call.

| Status | Message |
|---|---|
| RTR_STS_OK | Normal successful completion |
| RTR_STS_ ILLDEVTYP | RTR can only create its journal files on directory structured devices. |

### Parameters

## bSupersede

A boolean attribute that specifies how to handle cases where a journal already exists. Set bSupersede to true to overwrite an existing journal. If set to false, a journal is created only if no journal previously existed.

### Description

Call this method to create an RTR journal file. A journal is required for all facility members with a backend role, and any frontends that participate in nested transactions.

For more information on RTR journals, see the *VSI Reliable Transaction Router Application Design Guide* and the *VSI Reliable Transaction Router System Manager's Manual*.

### Example

```
// Declare an RTR object.
```

```
RTR *myRTR = new RTR();
 rtr_status_t sStatus;
 bool  bSupersede = false;  // false -> no supersede
 sStatus = myRTR->CreateJournal(bSupersede);
```

## DeleteJournal()

DeleteJournal() — RTR::DeleteJournal();

### Prototype

```
rtr_status_t DeleteJournal();
```

### Return Value

**rtr_status_t** The RTR status message return value. RTR_STS_OK is the normal successful completion.

### Parameters

None.

### Description

Call this method for deleting a journal.

For more information on RTR journals, see the *VSI Reliable Transaction Router System Manager's Manual*.

### Example

```
 // declare an RTR object
 RTR *myRTR = new RTR();
 rtr_status_t sStatus;
 sStatus = myRTR->DeleteJournal();
```

## GetErrorText()

GetErrorText() — RTR::GetErrorText();

### Prototype

```
static const char *GetErrorText(rtr_status_t sStatus);
```

### Return Value

Returns a pointer to the error message text associated with a known RTR message.

### Parameters

## rtr_status_t

The RTR status message return value. RTR_STS_OK is the normal successful completion.

## Description

Call this method to retrieve the error message text associated with an RTR status.

## Example

```
// start rtr
RTRmyRTR;
sStatus = myRTR.Start();
cout << myRTR.GetErrorText(sStatus) << endl;
// create journal
sStatus = myRTR.CreateJournal(true);
cout << myRTR.GetErrorText(sStatus) << endl;

//An example from the Sample application in the Examples directory
inline void print_status_on_failure(rtr_status_t sStatus)
{
    switch (sStatus)
    {
case ABCSuccess :
case ABCOrderSucceeded :
case ABCOrderFailed :{
break;
}
default: {
                    cout << RTR::GetErrorText(sStatus);
                    break;
};
  }
return;
}
```

## IsRunning()

IsRunning() — RTR::IsRunning();

## Prototype

```
bool IsRunning();
```

## Return Value

| Status | Message |
|---|---|
| TRUE | RTR is running. |
| FALSE | RTR is not running. |

## Parameters

None.

## Description

Call this method to find out if RTR is running on this node. If RTR is running, it will return a true, otherwise an error code.

## Example

```
 RTR *myRTR = new RTR();
 rtr_status_t sStatus;
 sStatus = myRTR->IsRunning();
```

## RTR()

RTR() — RTR::RTR();

### Prototype

```
RTR();
```

### Return Value

None.

### Parameters

None.

### Description

Call this method to declare an RTR object.

### Example

```
aRTR *myRTR = new RTR();
```

## Start()

Start() — RTR::Start();

### Prototype

```
rtr_status_t Start();
```

### Return Value

**rtr_status_t** Interpret value for the success or failure of this call.

| Status | Message |
|---|---|
| RTR_STS_ ACPDIED | The RTR ACP is no longer running, restart RTR. |
| RTR_STS_ ACPNOTVIA | ACP is no longer a viable entity. |
| RTR_STS_ BYTLMNSUFF | Insufficient process quota bytlm, required 100000. |
| RTR_STS_ ERRSTAACP | Unable to start ACP. |
| RTR_STS_ EXWSMAX | Requested memory quotas exceed the system limit WSMAX. |
| RTR_STS_OK | Normal successful completion. |

## Parameters

None.

## Description

Call this method to start RTR on a node.

## Example

```
    // declare RTR object.
    RTR *myRTR = new RTR();
    rtr_status_t sStatus;
    sStatus = myRTR->Start();
```

## Stop()

Stop() — RTR::Stop();

### Prototype

```
rtr_status_t Stop();
```

### Return Value

**rtr_status_t** Interpret value for the success or failure of this call.

| **Status** | **Description** |
|---|---|
| RTR_STS_ ACPDIED | The RTR ACP is no longer running, restart RTR. |
| RTR_STS_OK | Normal successful completion. |
| RTR_STS_ RTRNOTRUN | RTR not running. |

### Parameters

None.

### Description

Call this method to stop RTR on a node. Calling this method stops all RTR activity on the computer where it is called. Any running applications receive the error indication RTR_STS_NOACP. All facilities, links, and partitions are destroyed.

### Example

```
    // declare RTR object
    RTR *myRTR = new RTR();
    rtr_status_t sStatus;
    sStatus = myRTR->Stop();
```

## StartWebServer()

StartWebServer() — RTR::StartWebServer();

### Prototype

```
rtr_status_t StartWebServer(bool bAuthentication = true,
                            bool bReadOnlyAccess = false);
```

### Return Value

**rtr_status_t** Interpret value for the success or failure of this call. RTR_STS_OK is the normal successful completion.

### Parameters

## bAuthentication

A boolean attribute for specifying and controlling the web server user authentication. The default setting is for the server to perform user authentication using the username and password. This may be disabled, allowing anyone with a browser to access the management component.

## bReadOnlyAccess

A boolean attribute for specifying read-only access to the web browser RTR management component. A server started with the StartWebServer method servers status and monitor pages but does not permit any changes to be made to the configuration. By specifying read-only access for server operation, no shadowing or journaling is required. The message is still written to the journal but is not played to a shadow and is purged after the transaction is completed on the primary server. The message is still needed in the journal to allow recovery of in-flight transactions.

### Description

Call this method to start RTR on a web server. This starts a user's HTTP server component, thus enabling usage of the web browser RTR management component for the calling user.

### Example

```
bool RTR::StartWebServer()
{
    bool bOverallResult = true;
    RTR MyRTR;
rtr_status_t stsStartWebServer;
stsStartWebServer = MyRTR.StartWebServer();
if (IsFailure(stsStartWebServer == RTR_STS_OK))
{
    bOverallResult = false;
    OutputStatus(stsStartWebServer);
}
```

## StopWebServer()

StopWebServer() — RTR::StopWebServer();

### Prototype

```
rtr_status_t StopWebServer();
```

### Return Value

**rtr_status_t** Interpret value for the success or failure of this call.

| **Status** | **Message** |
|---|---|
| RTR_STS_ NFW | Operation requires SETPRV privilege. |
| RTR_STS_OK | Normal successful completion. |
| RTR_STS_SRVDISCON | Server disconnected on node 'nodename.' |

### Parameters

None

### Description

Call this method to stop RTR on a web server.

### Example

```
rtr_status_t stsStopWebServer;
stsStopWebServer = MyRTR.StopWebServer();
if (IsFailure(stsStopWebServer == RTR_STS_OK || stsStopWebServer ==
RTR_STS_SRVDISCON))
{   bOverallResult = false;
    OutputStatus(stsStopWebServer);
}
```

## 4.2. RTRBackendPartitionProperties

This class holds and makes available the properties of its associated RTRPartition object. This allows the RTR application to Get and Set various attributes of an RTR partition. This class may be called by both new C++ API and legacy applications.

## RTRBackendPartitionProperties Class Members

## Construction

| **Method** | **Description** |
|---|---|
| RTRPartitionProperties(const char) | Constructor |
| ~RTRPartitionProperties(const char) | Destructor |

## Operations()

| **Method** | **Description** |
|---|---|
| GetFacilityName(rtr_facnam_t,<br><br>const size_t) | Gets the facility name associated with the RTRPartition object this RTRPartitionProperties object describes. |
| GetNumberOfRecoveredTransactions | Gets the number of recovered transactions associated with the RTRPartition object this RTRPartitionProperties object describes. |

| Method | Description |
|---|---|
| GetPartitionName(rtr_parnam_t, const size_t) | Gets the partition name associated with the RTRPartition object this RTRPartitionProperties object describes. |
| GetRetryCount(rtr_uns_32_t) | Gets the number of retrys associated with the RTRPartition object this RTRPartitionProperties object describes. |
| SetFailoverPolicy(const eRTRFailoverPolicy) | Defines the policy that RTR should take when a primary partition fails. |
| SetPriorityList(const char) | Sets a relative priority used by RTR when selecting a backend member to make active. |
| SetRecoveryRetryCount | Indicates the maximum number of times that a transaction should be presented for recovery before being written to the journal as an exception. |

## GetFacilityName()

GetFacilityName() — RTRBackendPartitionProperties::GetFacilityName();

### Prototype

```
rtr_status_t GetFacilityName(rtr_facnam_t pszFacilityName,
                             const size_t uiFacilityNameSize);
```

### Return Value

**rtr_status_t** Interpret value for the success or failure of this call.

| Status | Message |
|---|---|
| RTR_STS_ APPBUFFTOOSMALL | The application buffer is too small. |
| RTR_STS_INVARGPTR | Invalid parameter address specified on last call. |
| RTR_STS_OK | Normal successful completion. |

### Parameters

## pszFacilityName

Pointer to a zero-terminated string receiving the facility name for the RTRPartition this RTRPartitionProperties object describes.

## uiFacilityNameSize

An unsigned integer for the size of the specified facility name. The maximum string length is RTR_MAX_FACNAM_LEN.

### Description

Gets the facility name associated with the RTRPartition object this RTRPartitionProperties object describes.

### Example

```
// declare a partition properties object.
rtr_status_t sStatus;
RTRBackendPartitionProperties *pPartProperties =
    PartitionManager.GetBackendPartitionProperties("MyPartition");
char *pszFacilityName = new char[RTR_MAX_FACNAM_LEN+1];
sStatus = pPartProperties->GetFacilityName(pszFacilityName,
                                            RTR_MAX_FACNAM_LEN+1);
```

## GetNumberOfRecoveredTransactions()

GetNumberOfRecoveredTransactions() —
RTRBackendPartitionProperties::GetNumberOfRecoveredTransactions()

### Prototype

```
rtr_status_t GetNumberOfRecoveredTransactions(rtr_uns_32_t
                                              &uiNumberRecoveredTxns);
```

### Return Value

Interpret value for the success or failure of this call. RTR_STS_OK is the normal successful completion.

### Parameters

## uiNumberRecoveredTxns

A referenced value of type rtr_uns_32_t which receives the number of recovered transactions.

### Description

Gets the number of recovered transactions associated with the RTRPartition object this RTRPartitionProperties object describes.

For more information, see the *VSI Reliable Transaction Router System Manager's Manual*.

### Example

```
// declare a partition properties object.
rtr_status_t sStatus;
RTRBackendPartitionProperties *pPartProperties =
    PartitionManager.GetBackendPartitionProperties()"MyPartition");
rtr_uns_32_t iNumberRecoveredTxns
sStatus = pPartProperties->
            GetNumberOfRecoveredTransactions(iNumberRecoveredTxns);
```

## GetPartitionName()

GetPartitionName() — RTRBackendPartitionProperties::GetPartitionName();

### Prototype

```
rtr_status_t GetPartitionName(rtr_parnam_t pszPartitionName
                              const size_t uiPartitionNameSize);
```

## Return Value

**rtr_status_t** Interpret value for the success or failure of this call.

| **Status** | **Message** |
|---|---|
| RTR_STS_APPBUFFTOOSMALL | The application buffer is too small. |
| RTR_STS_INVARGPTR | Invalid parameter address specified on last call. |
| RTR_STS_OK | Normal successful completion. |

## Parameters

## pszPartitionName

Pointer to a null-terminated string receiving the partition name for the RTRPartition this RTRPartitionProperties object describes.

## uiPartitionNameSize

An unsigned integer for the size of the specified partition name.

## Description

Gets the partition name associated with the RTRPartition object this RTRPartitionProperties object describes.

## Example

```
// declare a partition properties object.
rtr_status_t sStatus;
RTRBackendPartitionProperties *pPartProperties =
     PartitionManager.GetBackendPartitionProperties("MyPartition");
char *pszPartitionName = new char[RTR_MAX_PARNAM_LEN+1];
sStatus = pPartProperties->GetPartitionName(pszPartitionName,
                                            RTR_MAX_PARNAM_LEN+1);
```

## GetRetryCount()

GetRetryCount() — RTRBackendPartitionProperties::GetRetryCount();

## Prototype

```
rtr_status_t GetRetryCount(rtr_uns_32_t &uiRetryCount);
```

## Return Value

**rtr_status_t** Interpret value for the success or failure of this call. RTR_STS_OK is the normal successful completion.

## Parameters

## uiRetryCount

A referenced value of type rtr_uns_32_t which receives the number of retries.

## Description

Gets the number of times a transaction has been retried after a failure.

## Example

```
// declare a partition properties object.
rtr_status_t sStatus;
RTRBackendPartitionProperties *pPartProperties =
PartitionManager.GetBackendPartitionProperties("MyPartition");
rtr_uns_32_t iRetryCount;
sStatus = pPartProperties->GetRetryCount(iRetryCount);
```

## RTRBackendPartitionProperties()

RTRBackendPartitionProperties() —
RTRBackendPartitionProperties::RTRBackendPartitionProperties();

### Prototype

```
RTRBackendPartitionProperties( rtr_const_parnam_t pszPartitionName );
virtual ~RTRBackendPartitionProperties();
```

### Return Value

None.

### Parameters

## pszPartitionName

Pointer to a zero-terminated string containing the partition name for which this RTRPartitionProperties object is being created.

### Description

Call this constructor to create an RTRPartitionProperties object for the partition named.

### Example

```
// Create BackendPartitionProperties object
RTRBackendPartitionProperties *pBEPartitionProperties;
pBEPartitionProperties = pPartitionManager->
           GetBackendPartitionProperties(GetDefaultPartitionName());
if (IsFailure(pBEPartitionProperties != NULL))
{
  bOverallResult = false;
  cout << endl << "        In Test_GetFacilityName(),
    pPartitionManager->GetBackendPartitionProperties()
        call failed." << endl;
  delete pPartitionManager;
  return bOverallResult;
}
```

## SetFailoverPolicy()

SetFailoverPolicy() — RTRBackendPartitionProperties::SetFailoverPolicy();

### Prototype

```
rtr_status_t SetFailoverPolicy(const eRTRFailoverPolicy eFailoverPolicy);
```

### Return Value

**rtr_status_t** Interpret value for the success or failure of this call.

| **Status** | **Message** |
|---|---|
| RTR_STS_OK | Normal successful completion. |
| RTR_STS_PRTBADCMD | Partition command invalid or not implemented in this version of RTR. |

### Parameters

## eRTRFailoverPolicy

An attribute for specifying an RTR failover policy:

1 = RTRFailOverToShadow

2 = RTRFailOverToStandBy

### Description

Determines the action to take when the primary partition fails. The default action is to allow a standby of the primary to become the new primary. Optionally, RTR can be set to change state so that the secondary becomes primary, and a standby of the old primary (if any) becomes the new secondary.

### Example

```
// declare a partition properties object.
rtr_status_t sStatus;
RTRBackendPartitionProperties *pPartProperties =
      PartitionManager.GetBackendPartitionProperties("MyPartition");
const RTRFailoverPolicy eFailoverPolicy = RTRFailOverToShadow;
sStatus = pPartProperties->SetFailoverPolicy(eFailoverPolicy);
```

## SetPriorityList()

SetPriorityList() — RTRBackendPartitionProperties::SetPriorityList();

### Prototype

```
rtr_status_t SetPriorityList(const char *pszPriorityList);
```

### Return Value

**rtr_status_t** Interpret value for the success or failure of this call.

| Status | Message |
|---|---|
| RTR_STS_OK | Normal successful completion. |
| RTR_STS_PRTBADCMD | Partition command invalid or not implemented in this version of RTR. |

## Parameters

## pszPriorityList

A null-terminated string pointer to a priority list.

## Description

Sets a relative priority used by RTR when selecting a backend member to make active. List the backends in your configuration in decreasing order of priority; the order of the list is taken into consideration when RTR is decides where to make a partition active.

Suspend partitions before changing the priority list. It is not an error to enter different versions of the priority list at different backends, but this is not recommended. If calling SetPriorityList, it is recommended to call SetPriorityList programmatically before you register the partition with the server transaction controller.

## Example

```
// declare a partition properties object.
rtr_status_t sStatus;
RTRBackendPartitionProperties *pPartProperties =
     PartitionManager.GetBackendPartitionProperties("MyPartition");
char *pszPriorityList = "depth,length"; // list of BE for prioirty
sStatus = pPartProperties->SetPriorityList(pszPriorityList);
```

## SetRecoveryRetryCount()

SetRecoveryRetryCount() — RTRBackendPartitionProperties::SetRecoveryRetryCount();

## Prototype

```
rtr_status_t SetRecoveryRetryCount(rtr_uns_32_t & uiRetryCount);
```

## Return Value

**rtr_status_t** Interpret value for the success or failure of this call.

| Status | Message |
|---|---|
| RTR_STS_OK | Normal successful completion |
| RTR_STS_APPBUFFTOOSMALL | The application buffer is too small. |

## Parameters

## uiRetryCount

A referenced value of type rtr_uns_32_t that receives the number of retries.

### Description

Call this method to set the recovery retry count. The recovery retry count indicates the maximum number of times that a transaction should be presented for recovery before being written to the journal as an exception. Once a transaction has been recorded as an exception, it is no longer considered eligible for recovery and will require manual processing by a qualified individual.

### Example

```
// declare a partition properties object.
rtr_status_t sStatus;
RTRBackendPartitionProperties *pPartProperties =
    PartitionManager.GetBackendPartitionProperties("MyPartition");
rtr_uns_32_t iRetryCount=10; // #of times to retry before giveup.
sStatus = pPartProperties->SetRecoveryRetryCount(iRetryCount);
```

## 4.3. RTRFacilityManager

### RTRFacilityManager Class Members

### Construction

| **Method** | **Description** |
|---|---|
| RTRFacilityManager | Constructor |
| ~RTRFacilityManager() | Destructor |

### Operations

| **Method** | **Description** |
|---|---|
| AddBackend(rtr_const_facnam_t, rtr_const_nodnam_t) | Add a backend role to an existing facility. |
| AddFrontend(rtr_const_facnam_t, rtr_const_nodnam_t) | Add a fronted role to an existing facility. |
| AddRouter(rtr_const_facnam_t, rtr_const_nodnam_t) | Add a router role to an existing facility. |
| CreateFacility(rtr_const_facnam_t, rtr_const_nodnam_t, rtr_const_nodnam_t, bool) | Create a facility, designating router and frontend. |
| CreateFacility(rtr_const_facnam_t, rtr_const_nodnam_t, rtr_const_nodnam_t, bool, bool) | Create a facility, designating router and backend. |
| CreateFacility(rtr_const_facnam_t, rtr_const_nodnam_t, rtr_const_nodnam_t, rtr_const_nodnam_t, bool, bool) | Create a facility, designating router, frontend, and backend. |
| DeleteFacility(rtr_const_facnam_t) | Delete a facility. |
| GetFacilityProperties(rtr_const_facnam_t, RTRFacilityProperties) | Retrieve properties for an existing facility. |
| RemoveBackend(rtr_const_facnam_t, rtr_const_nodnam_t) | Remove a backend role from an existing facility. |

| Method | Description |
|---|---|
| RemoveFrontend(rtr_const_facnam_t, rtr_const_nodnam_t) | Remove a fronted role from an existing facility. |
| RemoveRouter(rtr_const_facnam_t, rtr_const_nodnam_t) | Remove a router role from an existing facility. |

## AddBackend()

AddBackend() — RTRFacilityManager::AddBackend();

### Prototype

```
rtr_status_t AddBackend( rtr_const_facnam_t pszFacilityName,
                                          rtr_const_nodnam_t pszBackend );
```

### Return Value

**rtr_status_t** Interpret value for the success or failure of this call.

| Status | Message |
|---|---|
| RTR_STS_ENOIPNAM | Entered node name does not exist. |
| RTR_STS_INVBCKENDNAMARG | The backend name argument is invalid. |
| RTR_STS_INVFACNAMEARG | The facility name argument is invalid. |
| RTR_STS_NOSUCHFAC | No such facility. |
| RTR_STS_OK | Normal successful completion. |
| RTR_STS_RTRNOTSTA | RTR not started. |

### Parameters

## pszFacilityName

A null-terminated pointer to a facility name.

## pszBackend

A pointer to a null-terminated string containing the nodename to add as a backend (BE).

### Description

Call this method to extend a backend to a facility. Facility name and backend node names should not be null values. A node does not have to be reachable but must be valid or RTR returns RTR_STS_ENOIPNAM.

The Backend parameter can be a comma-separated list of nodenames.

### Example

```
// Add a Backend
rtr_status_t stsAddBackend;
```

```
stsAddBackend = pFacilityManager->AddBackend("AddBackend",
                                      m_psTest_ExtraNodeName);
if ( IsFailure( stsAddBackend == RTR_STS_OK ) )
{
bOverallResult = false;
OutputStatus( stsAddBackend );
}
```

## AddFrontend()

AddFrontend() — RTRFacilityManager::AddFrontend();

### Prototype

```
rtr_status_t AddFrontend( rtr_const_facnam_t pszFacilityName,
                                          rtr_const_nodnam_t pszFrontend );
```

### Return Value

**rtr_status_t** Interpret value for the success or failure of this call.

| **Status** | **Message** |
|---|---|
| RTR_STS_ENOIPNAM | Entered node name does not exist. |
| RTR_STS_INVFACNAMEARG | The facility name argument is invalid. |
| RTR_STS_INVFRNTENDNMARG | The frontend name argument is invalid. |
| RTR_STS_NOROUTERS | No routers. |
| RTR_STS_NOSUCHFAC | No such facility. |
| RTR_STS_OK | Normal successful completion. |
| RTR_STS_RTRNOTSTA | RTR not started. |

### Parameters

## pszFacilityName

A null-terminated pointer to a facility name.

## pszFrontend

A pointer to a null-terminated string containing the nodename to add as a frontend (FE).

### Description

Call this method to extend a frontend node for a facility. Facility names and node names should not be null values.

The Frontend parameter can be a comma-separated list of nodenames.

### Example

```
char *pszFacilityName = "Myfacility";
char *pszNodeName = "FENodeNamesSeparatedbyComma";
sStatus = myFac-> AddFrontend (pszFacilityName,l_ pszNodeName);
```

## AddRouter()

AddRouter() — RTRFacilityManager::AddRouter();

### Prototype

```
rtr_status_t AddRouter( rtr_const_facnam_t pszFacilityName,
                                       rtr_const_nodnam_t pszRouter );
```

### Return Value

**rtr_status_t** Interpret value for the success or failure of this call.

| **Status** | **Message** |
|---|---|
| RTR_STS_FACEXTENDED | Router added successfully. |
| RTR_STS_INVFACNAMEARG | The facility name argument is invalid. |
| RTR_STS_INVROUTRNAMEARG | The router name argument is invalid. |
| RTR_STS_NOFRONTEN | No frontends specified. |
| RTR_STS_NOSUCHFAC | No such facility. |
| RTR_STS_OK | Normal successful completion. |
| RTR_STS_RTRNOTSTA | RTR not started. |

### Parameters

## pszFacilityName

A null-terminated pointer to a facility name.

## pszRouter

A null-terminated pointer to a facility member with a router (TR) role.

### Description

Call this method to extend a router for a facility. Facility name and node names should not be null values.

### Example

```
char *pszFacilityName = "Myfacility";
char *pszNodeName = "FENodeNamesSeparatedbyComma";
sStatus = myFac-> AddRouter (pszFacilityName,l_ pszNodeName);
```

## CreateFacility()

CreateFacility() — RTRFacilityManager::CreateFacility();

## Prototype

```
rtr_status_t CreateFacility( rtr_const_facnam_t pszFacilityName,
                             rtr_const_nodnam_t pszRouter,
                             rtr_const_nodnam_t pszFrontend,
                             bool bEnableRouterCallout);

rtr_status_t CreateFacility(  rtr_const_facnam_t pszFacilityName,
                              rtr_const_nodnam_t pszRouter,
                              rtr_const_nodnam_t pszBackend,
                              bool bEnableRouterCallout,
                              bool bEnableBackendCallout );

rtr_status_t CreateFacility( rtr_const_facnam_t pszFacilityName,
                             rtr_const_nodnam_t pszRouter,
                             rtr_const_nodnam_t pszFrontend,
                             rtr_const_nodnam_t pszBackend,
                             bool bEnableRouterCallout,
                             bool bEnableBackendCallout);
```

## Return Value

**rtr_status_t** Interpret value for the success or failure of this call.

| Status | Message |
|---|---|
| RTR_STS_DUPNODNAM | Duplicate node names in list. |
| RTR_STS_INVBCKENDNAMARG | The backend name argument is invalid. |
| RTR_STS_INVFACNAMEARG | The facility name argument is invalid. |
| RTR_STS_INVFRNTENDNMARG | The frontend name argument is invalid. |
| RTR_STS_INVROUTRNAMEARG | The router name argument is invalid. |
| RTR_STS_OWNNODMIS | Executing node is not specified as frontend, router, or backend. |
| RTR_STS_NOBACKEND | No backend specified in facility. |
| RTR_STS_NOFRONTEN | No frontend specified in facility. |
| RTR_STS_NOROUTERS | No routers specified in facility. |
| RTR_STS_OK | Normal successful completion. |
| RTR_STS_OWNNODMIS | Executing node is not specified as Frontend ,router or backend. |
| RTR_STS_JOUNOTFOU | Journal file not found. |

## Parameters

## pszFacilityName

A null-terminated pointer to a facility name.

## pszRouter

A null-terminated pointer to a facility member with a router (TR) role.

## pszFrontend

A null-terminated pointer to a facility member with a frontend (FE) role.

## pszBackend

A null-terminated pointer to a facility member with a backend (BE) role.

## bEnableRouterCallout

A boolean attribute for specifying a callout router.

## bEnableBackendCallout

A boolean attribute for specifying a callout backend.

### Description

Call this method to create a facility. There are three versions of the CreateFacility method. One version designates router, frontend, and backend nodes. One version designates router and frontend nodes. One version designates router and backend nodes. For these last two versions, the CreateFacility method requires the router name to be non-local.

For example, the following two calls would succeed:

```
stsCreateFacility
pFacilityManager->CreateFacility("FacilityWithoutBackend",
                                 "router_nonlocal_nodename",
                                 "frontend_local_nodename",
                                  true);
stsCreateFacility
pFacilityManager->CreateFacility("FacilityWithoutFrontend",
                                 "router_nonlocal_nodename",
                                 "backend_local_nodename",
                                  true);
```

These two calls would return the RTR_STS_xxx errors indicated:

```
stsCreateFacility
pFacilityManager->CreateFacility("FacilityWithoutBackend",
                                 "router_local_nodename",
                                 "frontend_local_nodename",
                                  true);
NOBACKEND
No backends specified
```

Explanation: No backends were specified on a CREATE FACILITY command and the node where the command was executed was specified as being a router. This error message is displayed by the RTR utility.

```
stsCreateFacility
pFacilityManager->CreateFacility("FacilityWithoutFrontend",
                                 "router_local_nodename",
                                 "backend_local_nodename",
                                  true);
NOFRONTEN
No frontends specified
```

Explanation: No frontends were specified on a CREATE FACILITY command and the node where the command was executed was specified as being a router. This error message is displayed by the RTR utility.

## Example

```
RTRFacilityManager::CreateFacilityWithAllRoles_3()
{
bool bOverallResult = true;
//Create facility manager, abort if fails
    RTRFacilityManager * pFacilityManager;
    pFacilityManager = new RTRFacilityManager;
if ( IsFailure(pFacilityManager != NULL) )
    {
return false;
    }
// Create the facility
    rtr_status_t stsCreateFacility;
stsCreateFacility =
    pFacilityManager->CreateFacility("FacilityWithAllRoles_3",
                                  GetDefaultRouterName(),
                                  GetDefaultFrontendName(),
                                  GetDefaultBackendName(),
                                  true,
                                  false);
    // If facility creation is not successful, report it
    if ( IsFailure( stsCreateFacility == RTR_STS_OK ) )
{
bOverallResult = false;
OutputStatus( stsCreateFacility );
}
    else // Delete a successfully created facility
    {
    rtr_status_t stsDeleteFacility;
stsDeleteFacility =
        pFacilityManager->DeleteFacility("FacilityWithAllRoles_3");
    if ( IsFailure( stsDeleteFacility == RTR_STS_OK ) )
    {
    bOverallResult = false;
    OutputStatus( stsDeleteFacility );
    }
    }
// Cleanup and return
    delete pFacilityManager;
return bOverallResult;
}
```

An example from the Sample application in the Examples directory:

```
inline rtr_status_t CreateFacility()
{
// Create a Facility
    rtr_status_t sStatus;
    RTRFacilityManager FacilityManager;
// Get the local node name to create the facility.
    char nodename[ABCMAX_STRING_LEN];
    gethostname(&nodename[0],ABCMAX_STRING_LEN);
// Create the facility specifying that the local node has all roles.
    sStatus =
FacilityManager.CreateFacility(ABCFacility,nodename,nodename,
                               nodename,true,false);
print_status_on_failure(sStatus);
return sStatus;
}
```

For more information on creating a facility, see the *VSI Reliable Transaction Router System Manager's Manual*.

## DeleteFacility()

DeleteFacility() — RTRFacilityManager::DeleteFacility();

### Prototype

```
rtr_status_t DeleteFacility( rtr_const_facnam_t pszFacilityName );
```

### Return Value

**rtr_status_t** Interpret value for the success or failure of this call.

| Status | Message |
|---|---|
| RTR_STS_FACDELETE | Facility deleted successfully |
| RTR_STS_INVFACNAMEARG | The facility name argument is invalid. |
| RTR_STS_NOSUCHFAC | No such facility available |
| RTR_STS_OK | Normal successful completion |
| RTR_STS_RTRNOTSTA | RTR not started |

### Parameters

## pszFacilityName

A null-terminated pointer to a facility name.

### Description

Call this method to delete a facility. This does not clean out the journal; transactions that are to be processed stay in the journal. However, the facility must be recreated before you can process the transactions stored in the journal.

For more information on creating a facility, see the *VSI Reliable Transaction Router System Manager's Manual*.

### Example

```
 rtr_status_t sStatus;
```

```
char *pszFacilityName = "Myfacility";
sStatus = myFac->DeleteFacility(pszFacilityName);
```

## GetFacilityProperties()

GetFacilityProperties() — RTRFacilityManager::GetFacilityProperties();

### Prototype

```
rtr_status_t GetFacilityProperties( rtr_const_facnam_t pszFacilityName,
                                    RTRFacilityProperties *&pFacProp);
```

### Return Value

**rtr_status_t** Interpret value for the success or failure of this call.

| Status | Message |
|---|---|
| RTR_STS_INVFACNAMEARG | The facility name argument is invalid. |
| RTR_STS_INVFACPROPPTARG | Invalid facility properties object pointer argument. |
| RTR_STS_INVSTRINGPTRARG | The string pointer argument is invalid - string too long. |
| RTR_STS_OK | Normal successful completion |

### Parameters

## pszFacilityName

A null-terminated pointer to a facility name.

## pFacProp

Pointer to properties for a given facility.

### Description

Retrieve properties for an existing facility. Caller must delete pFacProp later.

### Example

```
    // Create a FacilityProperties object to get the properties from.
    RTRFacilityProperties *pFacilityProperties =
                    new RTRFacilityProperties("GetFacilityProperties");
    if ( IsFailure(pFacilityProperties != NULL) )
      {
        //Can't continue, so cleanup and return
        delete pFacilityManager;
        return false;
    }
    rtr_status_t stsGetFacilityProperties;
    stsGetFacilityProperties =
        pFacilityManager->GetFacilityProperties("GetFacilityProperties",
                                                pFacilityProperties);
    if ( IsFailure( stsGetFacilityProperties == RTR_STS_OK ) )
      {
        bOverallResult = false;
        OutputStatus( stsGetFacilityProperties );
    }
```

## RemoveBackend()

RemoveBackend() — RTRFacilityManager::RemoveBackend();

### Prototype

```
rtr_status_t RemoveBackend( rtr_const_facnam_t pszFacilityName,
                            rtr_const_nodnam_t pszBackend );
```

### Return Value

**rtr_status_t** Interpret value for the success or failure of this call.

| **Status** | **Message** |
|---|---|
| RTR_STS_INVBCKENDNAMARG | The backend name argument is invalid. |
| RTR_STS_INVFACNAMEARG | The facility name argument is invalid. |
| RTR_STS_INVFACPROPPTARG | Invalid facility properties object pointer argument. |
| RTR_STS_NOBACKEND | No more backends are available in this facility. |
| RTR_STS_ NOSUCHFAC | No such facility available. |
| RTR_STS_OK | Normal successful completion. |
| RTR_STS_RTRNOTSTA | RTR is not started. |

### Parameters

## pszFacilityName

A null-terminated pointer to a facility name.

## pszBackend

A null-terminated pointer to a facility member with a backend (BE) role.

### Description

Call this method to remove backend node from a facility.

### Example

```
rtr_status_t sStatus;
char *pszFacilityName = "MyFacilityName";
char *pszNodeName = "BENodeNamesSeparatedbyComma";
```

```
sStatus = myFac->RemoveBackend(pszFacilityName,pszNodeName);
```

## RemoveFrontend()

RemoveFrontend() — RTRFacilityManager::RemoveFrontend();

### Prototype

```
rtr_status_t RemoveFrontend( rtr_const_facnam_t pszFacilityName,
                                          rtr_const_nodnam_t pszFrontend );
```

### Return Value

**rtr_status_t** Interpret value for the success or failure of this call.

| **Status** | **Message** |
|---|---|
| RTR_STS_INVFACNAMEARG | The facility name argument is invalid. |
| RTR_STS_INVFRNTENDNMARG | The frontend name argument is invalid. |
| RTR_STS_ NOSUCHFAC | No such facility available. |
| RTR_STS_OK | Normal successful completion. |
| RTR_STS_RTRNOTSTA | RTR is not started. |

### Parameters

## pszFacilityName

A null-terminated pointer to a facility name.

## pszFrontend

A null-terminated pointer to a facility member with a frontend (FE) role.

### Description

Call this method to remove frontend nodes from a facility.

### Example

```
rtr_status_t sStatus;
char *pszFacilityName = "MyFacilityName";
char *pszNodeName  = "FENodeNamesSeparatedbyComma";
sStatus = myFac-> RemoveFrontend (pszFacilityName,pszNodeName);
```

## RemoveRouter()

RemoveRouter() — RTRFacilityManager::RemoveRouter();

### Prototype

```
rtr_status_t RemoveRouter( rtr_const_facnam_t pszFacilityName,
                                                rtr_const_nodnam_t
 pszRouter );
```

## Return Value

**rtr_status_t** Interpret value for the success or failure of this call.

| Status | Message |
|---|---|
| RTR_STS_INVFACNAMEARG | The facility name argument is invalid. |
| RTR_STS_INVROUTRNAMEARG | The router name argument is invalid. |
| RTR_STS_NOROUTERS | No more routers are available in this facility. |
| RTR_STS_ NOSUCHFAC | No such facility available. |
| RTR_STS_OK | Normal successful completion. |
| RTR_STS_RTRNOTSTA | RTR is not started. |

## Parameters

## pszFacilityName

A null-terminated pointer to a facility name.

## pszRouter

A null-terminated pointer to a facility member with a router (TR) role.

## Description

Call this method to remove router nodes from a facility.

## Example

```
rtr_status_t sStatus;
char *pszFacilityName = "MyFacilityName";
char *pszNodeName  = "TRNodeNamesSeparatedbyComma";
sStatus = myFac-> RemoveRouter (pszFacilityName,pszNodeName);
```

## RTRFacilityManager()

RTRFacilityManager() — RTRFacilityManager::RTRFacilityManager();

## Prototype

```
RTRFacilityManager(); virtual ~RTRFacilityManager();
```

## Return Value

None.

## Parameters

None.

### Description

Use this method to declare a facility manager object. Facility manager object should be declared for accessing any properties of a facility.

### Example

```
RTRFacilityManager *myFac = new RTRFacilityManager();
```

## 4.4. RTRFacilityMember

RTRFacilityMember provides members that can retrieve information about facilities including member name, role of the member, connectivity to or property of being the local node.

## RTRFacilityMember Class Members

## Construction

| **Method** | **Description** |
|---|---|
| RTRFacilityMember(rtr_const_facnam_t, rtr_const_nodnam_t, const eRTRMemberRoleType) | Constructor |
| ~RTRFacilityMember() | Destructor |

## Operations

| **Method** | **Description** |
|---|---|
| GetName(rtr_facnam_t, const size_t) | Retrieve the name of the facility member. |
| HasBackendRole(bool) | Determine if this facility member has a backend role. |
| HasFrontendRole(bool) | Determine if this facility member has a frontend role. |
| HasRouterRole(bool) | Determine if this facility member has a router role. |
| IsConnectedToLocalNode(bool) | Determine if this facility member has connectivity to the local node. |
| IsLocalNode(bool) | Determine if this facility is the local node. |

## GetName()

GetName() — RTRFacilityMember::GetName();

### Prototype

```
rtr_status_t GetName( rtr_facnam_t pszNodeName, const size_t
                                               uiNodeNameSize);
```

### Return Value

**rtr_status_t**

### Parameters

## pszFacilityName

A pointer to a facility name.

## uiFacilityNameSize

An unsigned integer for the facility name size.

### Description

Retrieve the name of the facility member.

### Example

```
#define MAX_NODNAME 256
char szNodName[MAX_NODNAME];
rtr_status_t stsGetName = FacMember.GetName(szNodName, MAX_NODNAME);
if (IsFailure(stsGetName == RTR_STS_OK))
{
cout << "      RTRFacilityMember::GetName failed\n";
                OutputStatus(stsGetName);
}
else
{...
```

## HasBackendRole()

HasBackendRole() — RTRFacilityMember::HasBackendRole();

### Prototype

```
rtr_status_t HasBackendRole(bool &bHasRole);
```

### Return Value

**rtr_status_t** Interpret value for the success or failure of this call. RTR_STS_OK is the normal successful completion.

### Parameters

## bHasRole

A boolean that is true or false for HasBackendRole.

### Description

Call this method to find out a node is configured as backend.

### Example

```
rtr_status_t stsHasRole;
bool bHasRole;
```

```
stsHasRole = FacMember.HasBackendRole(bHasRole);
if ( IsFailure( stsHasRole == RTR_STS_OK ) )
 {
    bOverallResult = false;
    OutputStatus( stsHasRole );
 }
bOverallResult = (bHasRole == true);
```

## HasFrontendRole()

HasFrontendRole() — RTRFacilityMember::HasFrontendRole();

### Prototype

```
rtr_status_t HasFrontendRole(bool &bHasRole);
```

### Return Value

**rtr_status_t** Interpret value for the success or failure of this call. RTR_STS_OK is the normal successful completion.

### Parameters

## bHasRole

A boolean that is true or false for HasFrontendRole.

### Description

Call this method to find out if a node is configured as frontend.

### Example

```
rtr_status_t stsHasRole;
bool bHasRole;
stsHasRole = FacMember.HasFrontendRole(bHasRole);
if ( IsFailure( stsHasRole == RTR_STS_OK ) )
 {
    bOverallResult = false;
    OutputStatus( stsHasRole );
 }
bOverallResult = (bHasRole == true);
```

## HasRouterRole()

HasRouterRole() — RTRFacilityMember::HasRouterRole();

### Prototype

```
rtr_status_t HasRouterRole(bool &bHasRole);
```

### Return Value

**rtr_status_t** Interpret value for the success or failure of this call. RTR_STS_OK is the normal successful completion.

## Parameters

## bHasRole

A boolean that is true or false for HasRouterRole.

## Description

Call this method to find out if a node is configured as router.

## Example

```
rtr_status_t stsHasRole;
bool bHasRole;
stsHasRole = FacMember.HasRouterRole(bHasRole);
if ( IsFailure( stsHasRole == RTR_STS_OK ) )
 {
    bOverallResult = false;
    OutputStatus( stsHasRole );
 }
bOverallResult = (bHasRole == true);
```

## IsConnectedToLocalNode()

IsConnectedToLocalNode() — RTRFacilityMember::IsConnectedToLocalNode();

## Prototype

```
rtr_status_t IsConnectedToLocalNode(bool &bIsConnected);
```

## Return Value

**rtr_status_t** Interpret value for the success or failure of this call. RTR_STS_OK is the normal successful completion.

## Parameters

## bIsConnected

A boolean that is true if the node is connected to the local node.

## Description

Call this method to find out if this node is connected to the local node.

## Example

```
bool bIsConnected;
rtr_status_t stsIsConnected;
stsIsConnected = FacMember.IsConnectedToLocalNode(bIsConnected);
if (IsFailure(stsIsConnected == RTR_STS_OK))
 {
    OutputStatus(stsIsConnected);
    bOverallResult = false;
 }
if (IsFailure(bIsConnected == true))
 {
 cout << "      RTRFacilityMember::IsConnectedToLocalNode failed\n";
 bOverallResult = false;
 }
```

## IsLocalNode()

IsLocalNode() — RTRFacilityMember::IsLocalNode();

### Prototype

```
rtr_status_t IsLocalNode(bool &bIsLocal);
```

### Return Value

**rtr_status_t** Interpret value for the success or failure of this call. RTR_STS_OK is the normal successful completion.

### Parameters

## bIsLocal

A boolean that is true if the node is a local node.

### Description

Call this method to find out if the node is a local node.

### Example

```
rtr_status_t stsIsLocal;
bool bIsLocalNode;
stsIsLocal = FacMember.IsLocalNode(bIsLocalNode);
if (IsFailure(stsIsLocal == RTR_STS_OK))
 {
    OutputStatus(stsIsLocal);
    bOverallResult = false;
 }
if (IsFailure(bIsLocalNode == true))
 {
    cout << "      RTRFacilityMember::IsLocalNode failed\n";
    bOverallResult = false;
 }
```

## RTRFacilityMember()

RTRFacilityMember() — RTRFacilityMember::RTRFacilityMember();

### Prototype

```
RTRFacilityMember( rtr_const_facnam_t pszFacilityName,
                                  rtr_const_nodnam_t pszMemberName);
virtual ~RTRFacilityMember();
```

### Return Value

None.

### Parameters

## pszFacilityName

A null-terminated pointer to a facility name.

## pszMemberName

A null-terminated string pointer to the name of a facility member.

### Description

Call this method to declare a facility member object. The member role type can be:

- 1 (RTRFacilityBackend)
- 2 (RTRFacilityRouter)
- 3 (RTRFacilityFrontend)

### Example

```
Char *pszFac = "Myfacility";
Char *pszNode ="NodeName";
RTRFacilityMember *FacilityMember =
                    new RTRFacilityMember(pszFac,pszNode);
```

## 4.5. RTRFacilityMemberArray

An RTRFacilityMemberArray object contains pointers to array elements.

### Note

The RTRFacilityMemberArray class requires the holder of the array to clean up the objects pointed to by the elements of the array. The array does not clean up these objects.

## RTRFacilityMemberArray Class Members

## Construction

| Method | Description |
|---|---|
| RTRFacilityMemberArray() | Constructor |

| Method | Description |
|---|---|
| ~RTRFacilityMemberArray() | Destructor |

## Operations

| Method | Description |
|---|---|
| Add(RTRFacilityMember) | Adds a pointer to an RTRFacility member to the array. |
| Clear() | Clears elements of the array. |
| Insert(size_t, RTRFacilityMember) | Inserts a pointer to an RTRFacility member. |
| operator[] (size_t) | Returns an element of the array which is a pointer to an RTRFacility member. |
| Remove(const size_t) | Removes an element of the array. |
| Size(const) | Returns the number of elements in the array. |

## Add()

Add() — RTRFacilityMemberArray::Add();

### Prototype

```
bool Add(RTRFacilityMember* pFacMember);
```

### Return Value

True or False.

### Parameters

## pFacMember

A pointer to a facility member.

### Description

Add a member to a facility member array by adding a pointer to an RTRFacility member. The caller is responsible for creating and destroying the actual object. The array destructor does not destroy the objects pointed to.

### Example

```
bool RTRFacilityMemberArray::Add()
{
bool bArrayAddStatus;
RTRFacilityMemberArray ar;
RTRFacilityMember* pFacMember;

pFacMember = new RTRFacilityMember(GetDefaultFacilityName(),
                                   GetDefaultRouterName());
if (IsFailure(pFacMember != NULL))
{
cout << "      new RTRFacilityMember failed.\n";
return false;
}
bArrayAddStatus = ar.Add(pFacMember);
if (IsFailure(bArrayAddStatus))
{
cout << "      RTRFacilityMemberArray::Add failed\n";
}
return bArrayAddStatus;
}
```

## Clear

Clear — RTRFacilityMemberArray::Clear();

### Prototype

bool Clear();

### Return Value

True or False.

### Parameters

None.

### Description

This method clears the elements of the array, resulting in the array having a size of zero. This method does not destroy the objects pointed to; the caller must delete the contents.

### Example

```
bool bArrayClearStatus = ar.Clear();
 if (IsFailure(bArrayClearStatus))
 {
  cout << "      RTRFacilityMemberArray::Clear failed\n";
 }
 return bArrayClearStatus;
```

## Insert

Insert — RTRFacilityMemberArray::Insert();

### Prototype

bool Insert(size_t n, RTRFacilityMember* pFacMember);

### Return Value

True or False.

### Parameters

#### n

The element in the array ( ar[0] is the first element). The element is a pointer to an object.

#### pFacMember

A pointer to a facility member.

### Description

This method inserts a pointer to an RTRFacility member into the Nth position, moving the remainder of the array to make room.

### Example

```
    bool bArrayInsertStatus;
    bArrayInsertStatus = ar.Insert(1, pFacMember);
    if (IsFailure(bArrayInsertStatus))
    {
    cout << "       RTRFacilityMemberArray::Insert failed\n";
    }
    return bArrayInsertStatus;
```

## operator[]

operator[] — RTRFacilityMemberArray::GetMemberList();

### Prototype

RTRFacilityMember*& operator[] (size_t n);

### Return Value

Pointer to the Nth element of the array.

### Parameters

#### n

The element in the array (ar[0] is the first element). The element is a pointer to an object.

### Description

This operator returns the Nth element of the array which is a pointer to an RTRFacility member. You can also use this operator to set the Nth element of the array.

The existing element pointed to is not destroyed; the caller must delete the contents.

### Example

```
RTRFacilityMemberArray array;
```

```
RTRFacilityMember* pFacMember;
pFacMember = array;
if (IsFailure(pFacMember != NULL))
{
cout << "      RTRFacilityMemberArray operator[] failed\n";
}
return pFacMember != NULL;
```

## Remove

Remove — RTRFacilityMemberArray::Remove();

### Prototype

```
bool Remove (const size_t n);
```

### Return Value

True or False.

### Parameters

### n

The element in the array (ar[0] is the first element). The element is a pointer to an object.

### Description

This method removes the Nth element of the array. This does not destroy the object pointed to; the caller must delete the contents.

### Example

```
    bool bArrayRemoveStatus;
    bArrayRemoveStatus = ar.Remove(1);
    if (IsFailure(bArrayRemoveStatus))
    {
    cout << "      RTRFacilityMemberArray::Remove failed\n";
    }
    return bArrayRemoveStatus;
```

## RTRFacilityMemberArray

RTRFacilityMemberArray — RTRFacilityMemberArray:: RTRFacilityMemberArray();

### Prototype

```
RTRFacilityMemberArray(); virtual ~RTRFacilityMemberArray;
```

### Return Value

None.

### Parameters

None.

### Description

Construct an RTRFacilityMemberArray object.

### Example

```
RTRFacilityMemberArray::RTRFacilityMemberArray()
{
}
```

## Size

Size — RTRFacilityMemberArray::Size();

### Prototype

```
size_t Size() const;
```

### Return Value

**size_t**

### Parameters

None.

### Description

The method returns the number of elements in the array.

### Example

```
size_t nArraySize = ar.Size();
if (IsFailure(nArraySize == 1))
{
cout << "      RTRFacilityMemberArray::Size failed\n";
}
return nArraySize == 1;
```

## 4.6. RTRFacilityProperties

### RTRFacilityProperties Class Members

### Construction

| Method | Description |
|---|---|
| RTRFacilityProperties(rtr_const_facnam_t) | Constructor |

| Method | Description |
|---|---|
| ~RTRFacilityProperties | Destructor |

## Operations

| Method | Description |
|---|---|
| GetMemberList(RTRFacilityMemberArray) | Retrieves a list of nodes and their roles for an existing facility. |
| SetBalance(bool) | Allows intelligent reconnection of frontend to routers according to the number of connections on each active router. |

## GetMemberList()

GetMemberList() — ww

### Prototype

```
rtr_status_t GetMemberList(RTRFacilityMemberArray &aFacilityMembers);
```

### Return Value

**rtr_status_t** Interpret value for the success or failure of this call.

| Status | Message |
|---|---|
| RTR_STS_OK | Normal successful completion. |
| RTR_STS_DATANOTAVAILABL | Member list data not available. |

### Parameters

## aFacilityMembers

An array listing a facility's members.

### Description

Retrieve a list of nodes and their roles for an existing facility.

### Example

```
rtr_status_t stsGetMemberList;
RTRFacilityMemberArray arFacMembers;
stsGetMemberList = FacProps.GetMemberList(arFacMembers);
if (IsFailure(stsGetMemberList == RTR_STS_OK))
{
bOverallResult = false;
OutputStatus(stsGetMemberList);
}
int nNbrFacMembers = arFacMembers.Size();
for (int i=0; i<nNbrFacMembers; i++)
{
```

```
delete arFacMembers[i];
}
CleanupRTR();
return bOverallResult;
```

## RTRFacilityProperties()

RTRFacilityProperties() — RTRFacilityProperties::RTRFacilityProperties();

### Prototype

```
RTRFacilityProperties( rtr_const_facnam_t pszFacilityName);
virtual ~RTRFacilityProperties();
```

### Return Value

None.

### Parameters

## pszFacility

A null-terminated pointer to a facility name.

### Description

This method retrieves the properties associated with the facility object.

### Example

```
char *pszFacility = "Myfacilityname";
RTRFacilityProperties *FacilityPropterties = new
                               RTRFacilityProperties(pszFacility);
```

## SetBalance()

SetBalance() — RTRFacilityProperties::SetBalance();

### Prototype

```
rtr_status_t SetBalance( bool bBalancingOn );
```

### Return Value

**rtr_status_t** Interpret value for the success or failure of this call. RTR_STS_OK is the normal successful completion.

### Parameters

## bBalancingOn

A boolean attribute for specifying RTR balancing of client requests for server processing.

### Description

Specifies whether router balancing is to be performed.

### Example

```
rtr_status_t  sStatus;
char *pszFacilityName= "MyfacilityName";
bool bBalanceON = true;
sStatus = MyFacilityProperties->SetBalance(bBalanceOn);
```

## 4.7. RTRKeySegment

A key segment describes the data that the RTR application is sending. RTR uses this description for routing the data to an appropriate server. Key segments are of no value unless they are associated with a partition. When creating a partition, the caller is allowed to specify one or more key segments.

The RTRKeySegment class defines key segments (ranges) used in defining RTRPartition objects. RTRPartition objects are used to enable the partitioning of data across multiple servers.

### RTRKeySegment Class Members

### Construction

| **Method** | **Description** |
|---|---|
| RTRKeySegment(rtr_keyseg_type_t, rtr_keylen_t, rtr_keylen_t, rtr_const_pointer_t, rtr_const_pointer_t) | Constructor |
| ~ RTRKeySegment() | Destructor |

### Operations

| **Method** | **Description** |
|---|---|
| GetKeySegmentHighValue() | Gets the upper bound of the key range for the key segment. |
| GetKeySegmentLength() | Gets the length of the key segment key. |
| GetKeySegmentLowValue() | Gets the lower bound of the key range for the key segment. |
| GetKeySegmentOffset() | Gets the offset of the key segment key. |
| GetKeySegmentType() | Gets the type of the key segment. |
| SetKeySegmentHighValue(rtr_const_pointer_t) | Sets the upper bound of the key range for the key segment. |
| SetKeySegmentLength(const rtr_keylen_t) | Sets the length of the key segment key. |
| SetKeySegmentLowValue(rtr_const_pointer_t) | Sets the lower bound of the key range for the key segment. |
| SetKeySegmentOffset(const rtr_keylen_t) | Sets the offset of the key segment key. |

| **Method** | **Description** |
|---|---|
| SetKeySegmentType(const rtr_keyseg_type_t) | Sets the type of the key segment. |

## GetKeySegmentHighValue()

GetKeySegmentHighValue() — RTRKeySegment::GetKeySegmentHighValue();

### Prototype

```
rtr_pointer_t GetKeySegmentHighValue();
```

### Return Value

**rtr_pointer_t** Pointer to the returned upper-bound key value.

### Parameters

None.

### Description

This method returns the upper-bound key value of the key segment.

### Example

```
RTRKeySegment CharacterStringSegment.GetKeySegmentHighValue();
```

## GetKeySegmentLength()

GetKeySegmentLength() — RTRKeySegment::GetKeySegmentLength();

### Prototype

```
rtr_keylen_t GetKeySegmentLength();
```

### Return Value

**rtr_keylen_t** The returned value is the length of the key segment.

### Parameters

None.

### Description

This method gets the length of the key segment key.

### Example

```
rtr_keylen_t keylength =
                  CharacterStringSegment.GetKeySegmentLength();
```

## GetKeySegmentLowValue()

GetKeySegmentLowValue() — RTRKeySegment::GetKeySegmentLowValue();

### Prototype

```
rtr_pointer_t GetKeySegmentLowValue();
```

### Return Value

**rtr_pointer_t** Pointer to the returned lower-bound key value.

### Parameters

None.

### Description

This method returns the lower-bound key value of the key segment.

### Example

```
rtr_keylen_t keylength =
                   CharacterStringSegment.GetKeySegmentLowValue();
```

## GetKeySegmentOffset()

GetKeySegmentOffset() — RTRKeySegment::GetKeySegmentOffset();

### Prototype

```
rtr_keylen_t GetKeySegmentOffset();
```

### Return Value

**rtr_keylen_t** The returned value is the offset of the key value.

### Parameters

None.

### Description

This method gets the offset of the key segment key within the message stream.

### Example

```
rtr_keylen_t keylength =
                    CharacterStringSegment.GetKeySegmentOffset();
```

## GetKeySegmentType()

GetKeySegmentType() — RTRKeySegment::GetKeySegmentType();

## Prototype

```
rtr_keyseg_type_t GetKeySegmentType();
```

## Return Value

**rtr_keyseg_type_t**

One of the values of type rtr_keyseg_type_t, that can be:

- rtr_keyseg_signed
- rtr_keyseg_unsigned
- rtr_keyseg_string

## Parameters

None.

## Description

This method gets the data type of the key segment.

## Example

```
rtr_keylen_t keylength = CharacterStringSegment.GetKeySegmentType();
```

## RTRKeySegment()

RTRKeySegment() — RTRKeySegment::RTRKeySegment();

## Prototype

```
RTRKeySegment(rtr_keyseg_type_t keySegmentType,
              rtr_keylen_t keySegmentLength,
              rtr_keylen_t keySegmentOffset,
              rtr_const_pointer_t pKeySegmentLowValue,
              rtr_const_pointer_t pKeySegmentHighValue );
virtual ~RTRKeySegment();
```

## Return Value

None.

## Parameters

## keySegmentType

One of the values of type rtr_keyseg_type_t, that can be one of the following:

- rtr_keyseg_signed
- rtr_keyseg_unsigned

- rtr_keyseg_string

## keySegmentLength

A numerical length value in bytes of type rtr_keylen_t.

Default = 4

## keySegmentOffset

A numerical offset value in bytes of type rtr_keylen_t.

Default = 0.

## pKeySegmentLowValue

A pointer of type rtr_pointer_t to a lower-bound key value of type rtr_keyseg_type_t.

Default = NULL.

## PKeySegmentHighValue

A pointer of type rtr_pointer_t to an upper-bound key value of type rtr_keyseg_type_t.

Default = NULL.

### Description

Call this constructor to create an RTRKeySegment object.

### Example

```
void ClassDerivedFromHandler::StartProcessingOrdersAtoL( )
{
// This function defines a key segment and calls
StartProcessingOrders to process all orders that have a ticker
symbol beginning with the letters A-L.
// Create a KeyRange
m_pkeyRange = new RTRKeySegment( rtr_keyseg_string,
                                 1,
                                 OffsetIntoApplicationProtocol,
                                "A",
                                "L" );
StartProcessingOrders(PARTITION_NAMEAToL,m_pkeyRange);
}
```

## SetKeySegmentHighValue()

SetKeySegmentHighValue() — RTRKeySegment::SetKeySegmentHighValue();

### Prototype

```
rtr_status_t SetKeySegmentHighValue(rtr_const_pointer_t
                                pKeySegmentHighValue );
```

### Return Value

**rtr_status_t** Interpret value for the success or failure of this call.

| Status | Message |
|---|---|
| RTR_STS_OK | Normal successful completion. |
| RTR_STS_INVKYSGVLPTARG | Invalid key segment value pointer argument. |

### Parameters

## pKeySegmentHighValue

Pointer to the upper-bound key value to be set.

### Description

This method sets the upper bound of the key range for the key segment.

### Example

```
rtr_keyseg_type_t PKeySegmentHighValue = L;
CharacterStringSegment.SetKeySegmentHighValue(KeySegmentHighValue);
```

## SetKeySegmentLength()

SetKeySegmentLength() — RTRKeySegment::SetKeySegmentLength();

### Prototype

```
rtr_status_t SetKeySegmentLength(const rtr_keylen_t keySegmentLength );
```

### Return Value

**rtr_status_t** Interpret value for the success or failure of this call. RTR_STS_OK is the normal successful completion.

### Parameters

## keySegmentLength

This parameter holds the key length value of type rtr_keylen_t to be set.

### Description

This method sets the length of the key segment key.

### Example

```
rtr_keylen_t keylength = 1;
```

```
CharacterStringSegment.SetLength(keylength);
```

## SetKeySegmentLowValue()

SetKeySegmentLowValue() — RTRKeySegment::SetKeySegmentLowValue();

### Prototype

```
rtr_status_t SetKeySegmentLowValue(rtr_const_pointer_t
 pKeySegmentLowValue);
```

### Return Value

**rtr_status_t** Interpret value for the success or failure of this call.

| Status | Message |
|---|---|
| RTR_STS_OK | Normal successful completion. |
| RTR_STS_INVKYSGVLPTARG | Invalid key segment value pointer argument. |

### Parameters

## pKeySegmentLowValue

Pointer to the lower-bound key value to be set.

### Description

This method sets lower bound of the key range for the key segment.

### Example

```
rtr_keyseg_type_t PKeySegmentLowValue = A;
CharacterStringSegment.SetKeySegmentLowValue(KeySegmentLowValue);
```

## SetKeySegmentOffset()

SetKeySegmentOffset() — RTRKeySegment::SetKeySegmentOffset();

### Prototype

```
rtr_status_t SetKeySegmentOffset( const rtr_keylen_t keySegmentOffset );
```

### Return Value

**rtr_status_t** Interpret value for the success or failure of this call. RTR_STS_OK is the normal successful completion.

### Parameters

## keySegmentOffset

This parameter holds the key segment key offset of type rtr_keylen_t.

## Description

This method sets the offset of the key segment key within the message stream.

## Example

```
rtr_keylen_type_t PKeySegmentOffset = ;
CharacterStringSegment.SetKeySegmentOffset(KeySegmentOffset);
```

## SetKeySegmentType()

SetKeySegmentType() — RTRKeySegment::SetKeySegmentType();

### Prototype

```
rtr_status_t SetKeySegmentType( const rtr_keyseg_type_t keySegmentType);
```

### Return Value

**rtr_status_t** Interpret value for the success or failure of this call. RTR_STS_OK is the normal successful completion.

### Parameters

## keySegmentType

One of the values of type rtr_keyseg_type_t, that can be one of the following:

- rtr_keyseg_signed
- rtr_keyseg_unsigned
- rtr_keyseg_string

### Description

This method sets the data type of the key segment.

### Example

```
rtr_status_t stsSetKeySegmentType;
rtr_keyseg_type_t NewType = rtr_keyseg_string;
stsSetKeySegmentType = KeySeg.SetKeySegmentType(NewType);
if (IsFailure(stsSetKeySegmentType == RTR_STS_OK))
{
bOverallResult = false;
}
rtr_keyseg_type_t CurrType;
CurrType = KeySeg.GetKeySegmentType();
if (IsFailure(CurrType == NewType))
{
bOverallResult = false;
cout << )      RTRKeySegment::Set/GetKeySegmentType failed.\n);
}
return bOverallResult;
```

## 4.8. RTRKeySegmentArray

An RTRKeySegmentArray object contains pointers to array elements.

**Note**

The RTRKeySegmentArray class requires the holder of the array to clean up (delete) the objects pointed to by the elements of the array; the array does not clean up these objects (does not delete the contents of the array).

### RTRKeySegment Class Members

### Construction

| **Method** | **Description** |
|---|---|
| RTRKeySegmentArray() | Constructor |
| ~ RTRKeySegment() | Destructor |

### Operations()

| **Method** | **Description** |
|---|---|
| Add(RTRKeySegment) | Add a pointer to the array. |
| Clear() | Clear the elements of the array. |
| Insert(size_t, RTRKeySegment) | Insert a pointer to an RTRKeySegment member. |
| Remove() | Remove an element of the array. |
| RTRKeySegment operator( size_t) | Return an element of the array which will be a pointer to an RTRKeySegment member. |
| Size() | Return the number of elements in the array. |

### Add()

Add() — RTRKeySegmentArray::Add();

#### Prototype

```
bool Add(RTRKeySegment* pFacMember);
```

#### Return Value

True or False.

#### Parameters

### pFacMember

Pointer to a facility member.

### Description

Add a pointer to an RTRKeySegment member to the array. The caller is responsible for creating and destroying the actual object. The array destructor does not destruct the objects pointed to.

### Example

```
bool RTRKeySegmentArray::Add ()
{
bool bArrayAddStatus;
RTRKeySegmentArray ar;
RTRKeySegment* pKeySeg;
unsigned low=0;
unsigned high=10000;
pKeySeg = new RTRKeySegment(rtr_keyseg_unsigned, sizeof(unsigned),
                             0, &low, &high);
if (IsFailure(pKeySeg != NULL))
return false;
bool bAddOk = ar.Add(pKeySeg);
if (IsFailure(bAddOk == true))
{
delete pKeySeg;
return false;
}
delete pKeySeg;
return true;
}
```

## Clear()

Clear() — RTRKeySegmentArray::Clear();

### Prototype

```
bool Clear();
```

### Return Value

True or False.

### Parameters

None.

### Description

This method clears the elements of the array, resulting in the array having a size of zero. The Clear method does not destroy the objects pointed to.

### Example

```
bool RTRKeySegmentArray::Clear ()
{
bool bArrayAddStatus;
RTRKeySegmentArray ar;
RTRKeySegment* pKeySeg0 = NULL;
RTRKeySegment* pKeySeg1 = NULL;
unsigned low0=0;
unsigned high0=10000;
unsigned low1=10001;
unsigned high1=20000;
pKeySeg0 = new RTRKeySegment(rtr_keyseg_unsigned, sizeof(unsigned),
                             0, &low0, &high0);
if (IsFailure(pKeySeg0 != NULL))
return false;
bool bAddOk = ar.Add(pKeySeg0);
if (IsFailure(bAddOk == true))
{
delete pKeySeg0;
return false;
}
if (ar.Size() != 1)
{
delete pKeySeg0;
return false;
}
pKeySeg1 = new RTRKeySegment(rtr_keyseg_unsigned, sizeof(unsigned),
                             0, &low1, &high1);
if (IsFailure(pKeySeg1 != NULL))
{
delete pKeySeg0;
return false;
}
bool bInsertOk = ar.Insert(0, pKeySeg1);
if (IsFailure(bInsertOk == true))
{
delete pKeySeg0;
delete pKeySeg1;
return false;
}
bool bClearOk = ar.Clear();
if (IsFailure(bClearOk == true))
{
delete pKeySeg0;
delete pKeySeg1;
return false;
}
if (IsFailure(ar.Size() == 0))
{
delete pKeySeg0;
delete pKeySeg1;
return false;
}
delete pKeySeg0;
delete pKeySeg1;
return true;
}
```

## Insert()

Insert() — RTRKeySegmentArray::Insert();

## Prototype

```
bool Insert(size_t n, RTRKeySegment* pFacMember);
```

## Return Value

True or False.

## Parameters

## n

The element in the array (ar[0] is the first element). The element is a pointer to an object.

## pFacMember

Pointer to a facility member.

## Description

Insert a pointer to an RTRKeySegment member into the Nth position, moving the remainder of the array to make room.

## Example

```
bool RTRKeySegmentArray::Insert ()
{
bool bArrayAddStatus;
RTRKeySegmentArray ar;
RTRKeySegment* pKeySeg0;
RTRKeySegment* pKeySeg1;
unsigned low0=0;
unsigned low1=10001;
unsigned high0=10000;
unsigned high1=20000;
pKeySeg0 = new RTRKeySegment(rtr_keyseg_unsigned, sizeof(unsigned),
                             0, &low0, &high0);
if (IsFailure(pKeySeg0 != NULL))
return false;
bool bAddOk = ar.Add(pKeySeg0);
if (IsFailure(bAddOk == true))
{
delete pKeySeg0;
return false;
}
if (ar.Size() != 1)
{
delete pKeySeg0;
return false;
}
pKeySeg1 = new RTRKeySegment(rtr_keyseg_unsigned, sizeof(unsigned),
                             0, &low1, &high1);
if (IsFailure(pKeySeg1 != NULL))
{
delete pKeySeg0;
return false;
}
bool bInsertOk = ar.Insert(0, pKeySeg1);
if (IsFailure(bInsertOk == true))
{
delete pKeySeg0;
delete pKeySeg1;
return false;
}
delete pKeySeg0;
delete pKeySeg1;
return true;
}
```

## Remove()

Remove() — RTRKeySegmentArray::Remove(const size_t n);

### Prototype

```
size_t Remove(const size_t n);
```

### Return Value

**size_t** The amount of allocated space.

### Parameters

### n

The element in the array (ar[0] is the first element). The element is a pointer to an object.

### Description

This method removes the Nth element of the array. Calling this method does not destroy the object pointed to; the caller needs to delete the contents.

### Example

```
bool RTRKeySegmentArray::Remove ()
{
bool bArrayAddStatus;
RTRKeySegmentArray ar;
RTRKeySegment* pKeySeg;
unsigned low=0;
unsigned high=10000;
pKeySeg = new RTRKeySegment(rtr_keyseg_unsigned, sizeof(unsigned),
                            0, &low, &high);
if (IsFailure(pKeySeg != NULL))
return false;
bool bAddOk = ar.Add(pKeySeg);
if (IsFailure(bAddOk == true))
{
delete pKeySeg;
return false;
}
bool bRemoveOk = ar.Remove(0);
if (IsFailure(bRemoveOk == true))
{
delete pKeySeg;
return false;
}
delete pKeySeg;
return true;
}
```

## RTRKeySegmentArray()

RTRKeySegmentArray() — RTRKeySegmentArray::RTRKeySegmentArray();

### Prototype

```
RTRKeySegmentArray(); virtual ~RTRKeySegmentArray();
```

### Return Value

None.

### Parameters

None.

### Description

Call this method to construct new RTRKeySegmentArray object.

### Example

```
Test_RTRKeySegmentArray::Test_RTRKeySegmentArray ()
{
}
```

## Operator()

Operator() — RTRKeySegmentArray::operator();

### Prototype

```
RTRKeySegment*& operator[ ] (size_t n);
```

### Return Value

Returns the Nth element of the array.

### Parameters

## n

The element in the array (ar[0] is the first element). The element is a pointer to an object.

### Description

This operator returns the Nth element of the array which will be a pointer to an RTRKeySegment member. This operator can also be used to set the Nth element of the array. The existing element pointed to is not destroyed; the caller must delete this.

### Example

```
bool Test_RTRKeySegmentArray::arrayoper()
{
bool bArrayAddStatus;
RTRKeySegmentArray ar;
RTRKeySegment* pKeySeg;
unsigned low=0;
unsigned high=10000;
pKeySeg = new RTRKeySegment(rtr_keyseg_unsigned, sizeof(unsigned),
                            0, &low, &high);
if (IsFailure(pKeySeg != NULL))
return false;
bool bAddOk = ar.Add(pKeySeg);
if (IsFailure(bAddOk == true))
{
delete pKeySeg;
return false;
}
if (ar.Size() != 1)
{
delete pKeySeg;
return false;
}
RTRKeySegment* pSeg0 = ar[0];
if (IsFailure(pSeg0 != NULL))
{
delete pKeySeg;
return false;
}
delete pKeySeg;
return true;
}
```

## Size()

Size() — RTRKeySegmentArray::Size();

### Prototype

```
size_t Size() const;
```

### Return Value

**size_t** The amount of space to be allocated.

### Parameters

None.

### Description

The method returns the number of elements in the array.

### Example

```
bool RTRKeySegmentArray::Size ()
{
bool bArrayAddStatus;
RTRKeySegmentArray ar;
RTRKeySegment* pKeySeg;
unsigned low=0;
unsigned high=10000;
pKeySeg = new RTRKeySegment(rtr_keyseg_unsigned, sizeof(unsigned),
                            0, &low, &high);
if (IsFailure(pKeySeg != NULL))
return false;
bool bAddOk = ar.Add(pKeySeg);
if (IsFailure(bAddOk == true))
{
delete pKeySeg;
return false;
}
if (ar.Size() != 1)
{
delete pKeySeg;
return false;
}
delete pKeySeg;
return true;
}
```

## 4.9. RTRPartitionManager

The RTRPartitionManager allows the RTR applictaion to create, delete and obtain properties for a partition.

A partition is composed of one or more key segments. These key segments define the location of data within the applications message, the type of the data and a range of values for the data. RTR uses this information to perform its data routing. One or more partitions can be registered with a server transaction controller.

The key segments are associated with a partition when the partition is created. A partition exists within one facility. A facility can have many partitions.

### RTRPartitionManager Class Members

### Construction

| Method | Description |
|---|---|
| RTRPartitionManager() | Constructor |
| ~RTRPartitionManager() | Destructor |

## Operations

| Method | Description |
|---|---|
| CreateBackendPartition(rtr_const_parnam_t, rtr_const_facnam_t, RTRKeySegment, const bool, const bool, const bool) | Creates a partition on a backend within an existing facility. |
| CreateBackendPartition(rtr_const_parnam_t, rtr_const_facnam_t, RTRKeySegmentArray, const bool, const bool, const bool) | Creates a partition on a backend within an existing facility.using an RTRKeySegmentArray. |
| DeletePartition(rtr_const_parnam_t, rtr_const_facnam_t) | Deletes a partition. |
| GetBackendPartitionProperties(rtr_const_parnam_t) | Retrieves properties for a partition on a backend. |

## CreateBackendPartition()

CreateBackendPartition() — RTRPartitionManager::CreateBackendPartition();

### Prototype

```
virtual rtr_status_t CreateBackendPartition(
                                  rtr_const_parnam_t pszPartitionName,
                                  rtr_const_facnam_t pszFacilityName,
                                  RTRKeySegment &KeySegment,
                                  const bool bShadow = false,
                                  const bool bConcurrent = true,
                                  const bool bStandby = true);
virtual rtr_status_t CreateBackendPartition(
                                  rtr_const_parnam_t pszPartitionName,
                                  rtr_const_facnam_t pszFacilityName,
                                  RTRKeySegment &KeySegmentArray,
                                  const bool bShadow = false,
                                  const bool bConcurrent = true,
                                  const bool bStandby = true);
```

### Return Value

**rtr_status_t** Interpret value for the success or failure of this call.

| Status | Message |
|---|---|
| RTR_STS_INVFACNAMEARG | The facility name argument is invalid. |
| RTR_STS_INVKEYSEGPTARG | Invalid key segment object pointer argument. |
| RTR_STS_INVPARTNAMEARG | The partition name argument is invalid. |
| RTR_STS_MAXPARTREG | Maximum partition limit. |
| RTR_STS_OK | Normal successful completion. |

### Parameters

## pszPartitionName

A null-terminated pointer to a partition name.

## pszFacilityName

A null-terminated pointer to a facility name.

## KeySegment

A key segment for the specified partition name.

## KeySegmentArray

An array of key segments for the specified partition name.

## bShadow

A boolean attribute for specifying a shadow server.

## bConcurrent

A boolean attribute for specifying a concurrent server.

## bStandby

A boolean attribute for specifying a standby server.

### Description

CreateBackendPartition method creates an RTR backend partition. The partition characteristics that may be defined include key range or ranges and whether attached server process can be shadows or standbys. The command must be issued before any server application programs using the partition are started.

### Example

```
RTRKeySegment *pCharacterStringSegment = new RTRKeySegment(
                               rtr_keyseg_string, 1,0,"y","z");

RTRPartitionManager PartitionManager;

sStatus = PartitionManager.CreateBackendPartition(

                                           "MyPartition",
                                           "myfac",
                                           &pCharacterStringSegment,
                                           false,true,true);
// boolean parameters are for specifying shadow, concurrent, standby
```

From the Sample application in the Examples directory:

```
RTRPartitionManager PartitionManager;
sStatus = PartitionManager.CreateBackEndPartition(  ABCPartition1,
                                                    ABCFacility,
                                      KeyZeroTo99,false,true,false);
```

## DeletePartition()

DeletePartition() — RTRPartitionManager::DeletePartition();

### Prototype

```
virtual rtr_status_t DeletePartition( rtr_const_parnam_t pszPartitionName,
                                      rtr_const_facnam_t pszFacility );
```

### Return Value

**rtr_status_t** Interpret value for the success or failure of this call.

| Status | Message |
|---|---|
| RTR_STS_INVFACNAMEARG | The facility name argument is invalid. |
| RTR_STS_INVPARTNAMEARG | The partition name argument is invalid. |
| RTR_STS_OK | Normal successful completion. |
| RTR_STS_PRTNDELETED | Deletion of partition failed with error. |

### Parameters

## pszPartitionName

A null-terminated pointer to a partition name.

## pszFacility

A null-terminated pointer to a facility name.

### Description

Call this method to delete a partition from a facility.

### Example

```
Char *pszFac = "MyFacility";
Char *pszPartition = "MyPartitionName";
sStatus = PartitionManager.DeletePartition(pszFac,pszPartition);
```

## GetBackendPartitionProperties()

GetBackendPartitionProperties() — RTRPartitionManager::GetBackendPartitionProperties();

### Prototype

```
virtual RTRBackendPartitionProperties*
     GetBackendPartitionProperties(rtr_const_parnam_t pszPartitionName);
```

### Return Value

**RTRBackendPartitionProperties*** Pointer to the RTRBackendPartitionProperties object associated with this RTRPartitionManager object.

| Status | Message |
|---|---|
| RTR_STS_OK | Normal successful completion. |
| RTR_STS_INVPARTNAMEARG | The partition name argument is invalid. |

### Parameters

### pszPartitionName

A null-terminated pointer to a partition name.

### Description

This method retrieves the properties associated with the RTRPartitionManager object. These properties are contained within an associated RTRBackendPartitionProperties object.

### Example

```
RTRBackendPartitionProperties *pPartProperties =
PartitionManager.GetBackendPartitionProperties("MyPartition");
```

### RTRPartitionManager()

RTRPartitionManager() — RTRPartitionManager::RTRPartitionManager();

### Prototype

```
RTRPartitionManager();
virtual ~RTRPartitionManager();
```

### Return Value

None.

### Parameters

None.

### Description

This method defines an RTRPartitionManager object.

### Example

```
     RTRPartitionManager PartitionManager;
```

## 4.10. RTRSignedCounter

To use a counter, perform the following steps:

- Declare the names for counter name and group name.
- Instantiate the counter using the counter name and group.
- Set the counter value and test for success.
- Increment the counter.
- Get the incremented value.

## RTRSignedCounter Class Members

## Construction

| Method | Description |
|---|---|
| RTRSignedCounter( rtr_const_countername_t, rtr_const_countergroupname_t) | Constructor |
| ~RTRSignedCounter() | Destructor |

## Operations

| Method | Description |
|---|---|
| Decrement() | Decrement the value managed by the counter class. |
| GetValue(rtr_sgn_32_t) | Retrieve the value managed by the counter class. |
| Increment() | Increment the value managed by the counter class. |
| SetValue(rtr_sgn_32_t) | Set the value managed by the counter class. |

## Decrement()

Decrement() — RTRSignedCounter::Decrement();

### Prototype

```
rtr_status_t Decrement();
```

### Return Value

**rtr_status_t** Interpret value for the success or failure of this call.

| Status | Message |
|---|---|
| RTR_STS_CTRBADOPER | The supplied argument specifies an illegal counter operation. |
| RTR_STS_CTRBADREF | The supplied argument does not reference a valid counter. |
| RTR_STS_INVOBJFAILCNSTR | Invalid object due to failure during object construction. |
| RTR_STS_OK | Normal successful completion. |
| RTR_STS_PRTBADCMD | Partition command invalid or not implemented in this version of RTR. |

The more specific counter class error status descriptions for RTR_STSCTRBADREF and RTR_STS_INVOBJFAILCNSTR are:

- RTR_STSCTRBADREF

  The object has not been initialized by the application. All counters must be given a default value by calling SetValue() after object construction.

- RTR_STS_INVOBJFAILCNSTR

  The object is invalid because the values passed in the constructor were invalid.

## Parameters

None.

## Description

Call this method to decrement a numeric counter. Decrement method can be called only after setting value (RTRSignedCounter::SetValue(CounterVal)).

## Example

```
rtr_const_countername_t kCounter = "test-counter-signed-decrement";
rtr_const_countergroupname_t kGroup = "test-counter-group";
RTRSignedCounter c(kCounter, kGroup);
rtr_sgn_32_t v = 0;
const rtr_sgn_32_t kValue = 669;
bool bOverallResult = true;

rtr_status_t stsSetValue;
stsSetValue = c.SetValue(kValue);
if (IsFailure(stsSetValue == RTR_STS_OK))
{
bOverallResult = false;
OutputStatus(stsSetValue);
}

rtr_status_t stsDecrement;
stsDecrement = c.Decrement();
if (IsFailure(stsDecrement == RTR_STS_OK))
{
bOverallResult = false;
OutputStatus(stsDecrement);
}
```

## GetValue()

GetValue() — RTRSignedCounter::GetValue();

### Prototype

```
rtr_status_t GetValue(rtr_sgn_32_t &CounterVal);
```

### Return Value

**rtr_status_t** Interpret value for the success or failure of this call.

| Status | Message |
|---|---|
| RTR_STS_CTRBADREF | The supplied argument does not reference a valid counter. |
| RTR_STS_INVOBJFAILCNSTR | Invalid object due to failure during object construction. |

| Status | Message |
|---|---|
| RTR_STS_OK | Normal successful completion |

The more specific counter class error status descriptions for RTR_STSCTRBADREF and RTR_STS_INVOBJFAILCNSTR are:

RTR_STSCTRBADREF

The object has not been initialized by the application. All counters must be given a default value by calling SetValue() after object construction.

RTR_STS_INVOBJFAILCNSTR

The object is invalid because the values passed in the constructor were invalid.

## Parameters

# CounterVal

A counter value for a specified RTR counter.

## Description

Call this method to get a counter value. GetValue can be called only after setting value (SetValue).

## Example

```
rtr_status_t stsGetValue;
stsGetValue = c.GetValue(v);
if (IsFailure(stsGetValue == RTR_STS_OK))
{
bOverallResult = false;
OutputStatus(stsGetValue);
}
```

# Increment()

Increment() — RTRSignedCounter::Increment();

## Prototype

```
rtr_status_t Increment();
```

## Return Value

**rtr_status_t** Interpret value for the success or failure of this call.

| Status | Message |
|---|---|
| RTR_STS_CTRBADOPER | The supplied argument specifies an illegal counter operation. |
| RTR_STS_CTRBADREF | The supplied argument does not reference a valid counter. |

| Status | Message |
|---|---|
| RTR_STS_INVOBJFAILCNSTR | Invalid object due to failure during object construction. |
| RTR_STS_OK | Normal successful completion |

The more specific counter class error status descriptions for RTR_STSCTRBADREF and RTR_STS_INVOBJFAILCNSTR are:

- RTR_STSCTRBADREF

  The object has not been initialized by the application. All counters must be given a default value by calling SetValue() after object construction.

- RTR_STS_INVOBJFAILCNSTR

  The object is invalid because the values passed in the constructor were invalid.

### Parameters

None.

### Description

Call this method to increment a numeric counter. This method can be called only after setting value.

### Example

```
rtr_const_countername_t kCounter = "test-counter-signed-increment";
rtr_const_countergroupname_t kGroup = "test-counter-group";

RTRSignedCounter c(kCounter, kGroup);
rtr_sgn_32_t v = 0;
const rtr_sgn_32_t kValue = 668;
bool bOverallResult = true;

rtr_status_t stsSetValue;
stsSetValue = c.SetValue(kValue);
if (IsFailure(stsSetValue == RTR_STS_OK))
{
bOverallResult = false;
OutputStatus(stsSetValue);
}

rtr_status_t stsIncrement;
stsIncrement = c.Increment();
if (IsFailure(stsIncrement == RTR_STS_OK))
{
bOverallResult = false;
OutputStatus(stsIncrement);
}
```

## SetValue()

SetValue() — RTRSignedCounter::SetValue();

## Prototype

```
rtr_status_t SetValue( rtr_sgn_32_t CounterVal );
```

## Return Value

**rtr_status_t** Interpret value for the success or failure of this call.

| Status | Message |
|---|---|
| RTR_STS_OK | Normal successful completion |
| RTR_STS_INVOBJFAILCNSTR | Invalid object due to failure during object construction. |

The more specific counter class error status description for RTR_STS_INVOBJFAILCNSTR is:

The object is invalid because the values passed in the constructor were invalid.

## Parameters

## CounterVal

A counter value for a specified RTR counter.

## Description

Call this method to set value for a counter. Object should be declared before setting value.

## Example

```
rtr_status_t sStatus;
int iSetValue = 100;
sStatus = IIntCounter->SetValue(iSetValue);
```

## RTRSignedCounter()

RTRSignedCounter() — RTRSignedCounter::RTRSignedCounter();

## Prototype

```
RTRSignedCounter( rtr_const_countername_t pszCounterName ,
                  rtr_const_countergroupname_t pszCounterGroupName);
```

## Return Value

None.

## Parameters

## pszCounterName

A null-terminated string pointer to the name of an RTR counter.

### pszCounterGroupName

A null-terminated string pointer to the name of an RTR counter group.

#### Description

This method used to declare an RTRSignedCounter object. The constructor creates an instance of the RTRSignedCounter class. The application must call SetValue() to initialize the counter.

All counters are process-specific. All counter names must be unique within the entire process without regard to the group name. For example, it is invalid to have an RTRSignedCounter name "MyCounter" and another RTRStringCounter name "MyCounter."

#### Example

```
rtr_counter_data_type eCtrtype  = rtr_counter_int;
RTRSignedCounter *iIntCounter = new
             RTRSignedCounter("MyCounter3","GroupName",eCtrtype);
```

## 4.11. RTRStringCounter

To use a counter, perform the following steps:

- Declare the names for counter name and group name.
- Instantiate the counter using the counter name and group.
- Set the counter value and test for success.
- Increment the counter.
- Get the incremented value.

### RTRStringCounter Class Members

### Construction

| **Method** | **Description** |
|---|---|
| RTRStringCounter( const char, const char, rtr_counter_data_type) | Constructor |
| ~RTRStringCounter() | Destructor |

### Operations

| **Method** | **Description** |
|---|---|
| GetValue(char) | Retrieve the value managed by the counter class. |
| SetValue(const char) | Set the value managed by the counter class. |

### GetValue()

GetValue() — RTRStringCounter::GetValue();

### Prototype

```
rtr_status_t GetValue(char * pszCounterVal);
```

### Return Value

**rtr_status_t** Interpret value for the success or failure of this call.

| **Status** | **Message** |
|---|---|
| RTR_STS_CTRBADREF | The supplied argument does not reference a valid counter. |
| RTR_STS_INVOBJFAILCNSTR | Invalid object due to failure during object construction. |
| RTR_STS_OK | Normal successful completion |

The more specific counter class error status descriptions for RTR_STSCTRBADREF and RTR_STS_INVOBJFAILCNSTR are:

- RTR_STSCTRBADREF

  The object has not been initialized by the application. All counters must be given a default value by calling SetValue() after object construction.

- RTR_STS_INVOBJFAILCNSTR

  The object is invalid because the values passed in the constructor were invalid.

### Parameters

## pszCounterVal

A null-terminated string pointer to the name of an RTR counter.

### Description

Call this method to get a counter value. GetValue can be called only after setting a value (SetValue).

### Example

```
int IIntCounter;
rtr_status_t sStatus;
sStatus = cMyCounter.GetValue(IIntCounter);
if (sStatus!= RTR_STS_OK) cerr<<"Error while getting counter value";
```

## SetValue()

SetValue() — RTRStringCounter::SetValue();

### Prototype

```
rtr_status_t SetValue(const char * pszCounterVal);
```

### Return Value

**rtr_status_t** Interpret value for the success or failure of this call.

| Status | Message |
|---|---|
| RTR_STS_OK | Normal successful completion |
| RTR_STS_INVOBJFAILCNSTR | Invalid object due to failure during object construction. |

The more specific counter class error status description for RTR_STS_INVOBJFAILCNSTR is:

The object is invalid because the values passed in the constructor were invalid.

### Parameters

## pszCounterVal

A null-terminated string pointer to the value of an RTR counter.

### Description

Call this method to set a counter value.

### Example

```
rtr_status_t sStatus;
int iSetValue = 100;
sStatus = IIntCounter->SetValue(iSetValue);
```

## RTRStringCounter()

RTRStringCounter() — RTRStringCounter::RTRStringCounter();

### Prototype

```
RTRStringCounter(const char *pszCounterName ,
                 const char *pszCounterGroupName);
```

### Return Value

None.

### Parameters

## pszCounterName

A null-terminated string pointer to the name of an RTR counter.

## pszCounterGroupName

A null-terminated string pointer to the name of an RTR counter group.

### Description

This method used to declare an RTRStringCounter object. The constructor creates an instance of the RTRStringCounter class. The application must call SetValue() to initialize the counter.

All counters are process-specific. All counter names must be unique within the entire process without regard to the group name. For example, it is invalid to have an RTRStringCounter name "MyCounter" and another RTRSignedCounter name "MyCounter."

### Example

```
rtr_counter_data_type eCtrtype  = rtr_counter_int;

RTRStringCounter *iIntCounter = new
      RTRStringCounter("MyCounter2","GroupName",eCtrtype);
```

## 4.12. RTRUnsignedCounter

To use a counter, perform the following steps:

- Declare the names for counter name and group name.
- Instantiate the counter using the counter name and group.
- Set the counter value and test for success.
- Increment the counter.
- Get the incremented value.

## RTRUnsignedCounter Class Members

## Construction

| Method | Description |
|---|---|
| RTRUnsignedCounter( rtr_const_countername_t, rtr_const_countergroupname_t) | Constructor |
| ~RTRUnsignedCounter() | Destructor |

## Operations

| Method | Description |
|---|---|
| Decrement() | Decrement the value managed by the counter class. |
| GetValue(rtr_uns_32_t) | Retrieve the value managed by the counter class. |
| Increment() | Increment the value managed by the counter class. |
| SetValue(rtr_uns_32_t) | Set the value managed by the counter class. |

## Decrement()

Decrement() — RTRUnsignedCounter::Decrement();

## Prototype

```
rtr_status_t Decrement();
```

## Return Value

**rtr_status_t** Interpret value for the success or failure of this call.

| Status | Message |
|---|---|
| RTR_STS_CTRBADOPER | The supplied argument specifies an illegal counter operation. |
| RTR_STS_CTRBADREF | The supplied argument does not reference a valid counter. |
| RTR_STS_INVOBJFAILCNSTR | Invalid object due to failure during object construction. |
| RTR_STS_OK | Normal successful completion |

The more specific counter class error status descriptions for RTR_STSCTRBADREF and RTR_STS_INVOBJFAILCNSTR are:

- RTR_STSCTRBADREF

  The object has not been initialized by the application. All counters must be given a default value by calling SetValue() after object construction.

- RTR_STS_INVOBJFAILCNSTR

  The object is invalid because the values passed in the constructor were invalid.

## Parameters

None.

## Description

Call this method to decrement a numeric counter. Decrement method can be called only after setting value.

## Example

```
RTRUnsignedCounter c(kCounter, kGroup);
rtr_uns_32_t v = 0;
const rtr_uns_32_t kValue = 669;
bool bOverallResult = true;

rtr_status_t stsSetValue;
stsSetValue = c.SetValue(kValue);
if (IsFailure(stsSetValue == RTR_STS_OK))
{
bOverallResult = false;
OutputStatus(stsSetValue);
}
```

```
rtr_status_t stsDecrement;
stsDecrement = c.Decrement();
if (IsFailure(stsDecrement == RTR_STS_OK))
{
bOverallResult = false;
OutputStatus(stsDecrement);
}
```

## GetValue()

GetValue() — RTRUnsignedCounter::GetValue();

### Prototype

```
rtr_status_t GetValue(rtr_uns_32_t &CounterVal);
```

### Return Value

**rtr_status_t** Interpret value for the success or failure of this call.

| Status | Message |
|---|---|
| RTR_STS_CTRBADREF | The supplied argument does not reference a valid counter. |
| RTR_STS_INVOBJFAILCNSTR | Invalid object due to failure during object construction. |
| RTR_STS_OK | Normal successful completion. |

The more specific counter class error status descriptions for RTR_STSCTRBADREF and RTR_STS_INVOBJFAILCNSTR are:

- RTR_STSCTRBADREF

  The object has not been initialized by the application. All counters must be given a default value by calling SetValue() after object construction.

- RTR_STS_INVOBJFAILCNSTR

  The object is invalid because the values passed in the constructor were invalid.

### Parameters

## CounterVal

A counter value for a specified RTR counter.

### Description

Call this method to get a counter value. GetValue can be called only after setting value (SetValue).

### Example

```
rtr_status_t sStatus;
```

```
sStatus = IIntCounter->GetValue(iReturnValue);
```

## Increment()

Increment() — RTRUnsignedCounter::Increment();

### Prototype

```
rtr_status_t Increment();
```

### Return Value

**rtr_status_t** Interpret value for the success or failure of this call.

| Status | Message |
|---|---|
| RTR_STS_CTRBADOPER | The supplied argument specifies an illegal counter operation for the counter type. |
| RTR_STS_CTRBADREF | The supplied argument does not reference a valid counter. |
| RTR_STS_INVOBJFAILCNSTR | Invalid object due to failure during object construction. |
| RTR_STS_OK | Normal successful completion |

The more specific counter class error status descriptions for RTR_STSCTRBADREF and RTR_STS_INVOBJFAILCNSTR are:

RTR_STSCTRBADREF

The object has not been initialized by the application. All counters must be given a default value by calling SetValue() after object construction.

RTR_STS_INVOBJFAILCNSTR

The object is invalid because the values passed in the constructor were invalid.

### Parameters

None.

### Description

Call this method to increment a numeric counter. This method can be called only after setting value.

### Example

```
rtr_status_t stsIncrement;
stsIncrement = c.Increment();
if (IsFailure(stsIncrement == RTR_STS_OK))
{
bOverallResult = false;
OutputStatus(stsIncrement);
}
```

## SetValue()

SetValue() — RTRUnsignedCounter::SetValue();

### Prototype

```
rtr_status_t SetValue( rtr_uns_32_t CounterVal );
```

### Return Value

**rtr_status_t** Interpret value for the success or failure of this call.

| Status | Message |
|---|---|
| RTR_STS_OK | Normal successful completion |
| RTR_STS_INVOBJFAILCNSTR | Invalid object due to failure during object construction. |

The more specific counter class error status description for RTR_STS_INVOBJFAILCNSTR is:

The object is invalid because the values passed in the constructor were invalid.

### Parameters

## CounterVal

A counter value for a specified RTR counter.

### Description

Call this method to set value for a counter. Object should be declared before setting value.

### Example

```
rtr_status_t sStatus;
int iSetValue = 100;
sStatus = IIntCounter->SetValue(iSetValue);
```

## RTRUnsignedCounter()

RTRUnsignedCounter() — RTRUnsignedCounter::RTRUnsignedCounter();

### Prototype

```
RTRUnsignedCounter( rtr_const_countername_t pszCounterName ,
                    rtr_const_countergroupname_t pszCounterGroupName);
```

### Return Value

None.

## Parameters

## pszCounterName

A null-terminated string pointer to the name of an RTR counter.

## pszCounterGroupName

A null-terminated string pointer to the name of an RTR counter group.

## Description

This method used to declare an RTRUnsignedCounter object. The constructor creates an instance of the RTRUnsignedCounter class. The application must call SetValue() to initialize the counter.

All counters are process-specific. All counter names must be unique within the entire process without regard to the group name. For example, it is invalid to have an RTRUnsignedCounter name "MyCounter" and another RTRStringCounter name "MyCounter."

## Example

```
rtr_counter_data_type eCtrtype  = rtr_counter_int;
RTRUnsignedCounter *iIntCounter = new
            RTRUnsignedCounter("MyCounter1","GroupName",eCtrtype)
```

# Chapter 5. Sample Application Tutorial

## 5.1. Purpose

This tutorial goes through all of the steps needed to set up a simple RTR C++ API-based application for a new developer. The intent is to provide a starting point for learning about RTR, and to simplify the main concepts of RTR; you will be able to cruise through this at a more rapid pace than you normally would with the RTR reference information. At the end of this tutorial, you'll find brief descriptions of some of the more complex features RTR provides, and pointers to the documentation where you can study them in detail.

## 5.2. Summary

This tutorial walks you through designing, coding and setting up a basic RTR- based client-server application. To do this, you'll use RTR to perform two important services for you:

- To act as the communication mechanism between the client and the server applications
- To insure that the server application is always available to its clients

In the system that you are about to develop, the client application interacts with the user to read and display data. The server application handles requests from the client, and sends replies back to it. When we refer to `client' and `server', we will be referring to the applications. When we refer to the computer nodes on which the client or server is executing, we will call them `frontend' and `backend' nodes, respectively.

In most applications, the server would probably talk to a database in order to retrieve or save data according to what a user had entered in the user- interface. In the interest of simplifying this tutorial, however, this server is only going to tell you whether it received your client's request.

What is different in this system from a non-RTR system is that there will be two servers: one of the servers, also known as the `primary server', almost always talks with the client. In a perfect world, nothing would ever happen to this server; clients would always get the information they asked for, and all changes would be made to the database when the user updated information. Every time anyone attempted to access this server, it would always be there, ready and waiting to `serve', and users could feel secure in the knowledge that the data in the database was changed exactly as they had requested.

But we're all well aware that this is not always the case, and when servers do go down, it's usually at the most inopportune time. So you are going to use RTR to designate a second server as a "standby" server. In this way, if a user is attempting to get some real work done, and the primary server is down, the user will never notice. The standby server will spring into action, and replace the original server by handling the user's requests in just the same way as the primary server had been doing. And, this will be done from the same point at which the primary server had crashed!

### Materials List

In order to fully develop this system, you need a client application and frontend node, a server application and two backend nodes, and a router.

## Frontend

The frontend node is the system on which your client application is executing. As in any client-server system, the client application interacts with the user, then conveys the user's requests to the server. When developing an RTR-based client-server system, your client will have the following characteristics:

- Display an interface to the user, allow the user make a request, then communicate with the server to get or set data according to what actions the user has taken
- Execute on a Solaris, Tru64 UNIX, Windows (NT or 95 or 98) or OpenVMS system node, which has RTR installed on it
- Be attached to a TCP/IP or DECnet network and able to "see" the server machines; this means that if you use the `ping' utility to find a computer node by name, the computer responds back to the node you are on

Example code for the client application and the server application can be found in the `examples' subdirectory of your RTR installation directory.

## Backend1

Your first backend node will be running the primary server application. It, too, can be on any of the above operating systems, except the Windows system must be NT. It also must have RTR installed on it, and will contain your server application. Your server application will use RTR to listen for requests from the client, receive and handle those requests, and return the result with a message to the client.

## Backend2

This machine will run the standby server application. It will probably also be doing any one of a number of other things that have nothing to do with this tutorial, or even with RTR. It most probably will be sitting on one of your co-workers' desks, helping him or her to earn their weekly salary and support their family. Hopefully, you get along with this coworker well enough that they will install RTR on their machine, so that you may complete this tutorial.

## Router

Your router is simply RTR software which keeps track of everything that is going on for you when your application is running. The router can execute on a separate machine, on a frontend machine, or on a backend machine. In this tutorial, the router is kept on the same machine as the client.

## Install RTR

Your first step, once you have determined the three computers you are going to use for this tutorial, is to be sure RTR is installed and configured on each machine. The RTR installation is well documented and straightforward, although slightly different for each operating system on which the installation is being run. Refer to the section in the *VSI Reliable Transaction Router Installation Guide* for the system on which you are installing RTR. For the purpose of documenting examples, the machine you have decided to use for the:

- Client application is referred to as FE (frontend)
- Primary server is referred to as as BE1 (backend 1)

- Secondary server is referred to as BE2 (backend 2)

Remember that the router is on the FE machine. The journal must be accessible to both backend servers. (This requires clusters, NFS or Windows share are not supported)

## Start RTR

You need to start RTR on each of the machines on which you have installed it. You may do this from one machine. In order to be able to issue commands to RTR on a remote node, however, you must have an account on that node with the necessary access privileges. The operating system's documentation, or your system manager, will have information on how to set up privileges to enable users to run applications over the network. Use the command interface on your system to interact with RTR. At the command prompt, type in RTR, and press the Return or Enter key. You are then at the RTR> prompt, and can start RTR on all of the nodes. (Start RTR and create facilities independently on separate nodes.) For example, on a UNIX system, it looks like this:

```
% rtr
RTR> start rtr/node=(FE,BE1,BE2)
RTR> exit
```

This command starts `services' or `daemons' on each of the nodes in the list. These are processes that listen for messages being sent by other RTR services or daemons over the network. After executing the command, a `ps', `show process' or Task Manager review of processes executing on your system should now show at least one process named `rtr' or `rtr.exe' on each of the machines. This process is the one that manages the communications between the nodes in the RTR-based application, and handles all transactions and recoveries.

Starting RTR can also be done programmatically.

## Create a Recovery Journal

This step holds the key to letting the second server pick up on the work at exactly the right time through a recovery journal. In the case of a failure, the secondary server ensures that no work is lost, and the hot swap to the standby server is automatic. RTR keeps track of the work being done by writing data to the recovery journal. If a failure occurs, all incomplete transactions are being kept track of here, and can be replayed by the standby server when it comes to the rescue. When transactions have been completed, they are removed from this journal. For this example, only your backend nodes need a recovery journal, and you must create the journal before creating your facility; you'll learn more about facilities in the next section. You'll now need to go to each of the backend nodes that you'll be using and create a journal there. Log into each machine and, using the command prompt interface, run RTR and create the journal. When you specify the location of the journal, it should be the disk name or share name where the journal will be located. The journal must be accessible by both of the backend servers.

This is an example of what the command would look like on an OpenVMS system.

```
$ RTR
RTR> create journal user2
RTR> exit
```

To allow both servers to access the journal, you have a number of options:

- Use a disk in the disk farm on your cluster if you use clusters. This is supported on OpenVMS, Windows NT, and Tru64.
- Use a disk served via NFS with UNIX systems.

- Use a share when using Windows NT systems.

  NFS and Windows shares are not supported for journal disks.

In any case, you should be sure the disk is not on your primary server, since this is the machine that we are protecting, in case of a crash. If the machine goes down, the standby server would not be able to access the disk.

## The Database

While we are having this discussion on sharing resources, we should also mention how a database fits into this system, as well. This tutorial and the example code provided with it does not do database transactions. However, there are likely places in the code where you would probably want to access the database in most applications. Because the standby server steps into place when the primary server crashes, each must have access to your database.

This configuration can be supplied using a number of options:

- Use a database server, such as SQL Server or Oracle's database server
- Use machines in a cluster to run the database as well as the servers
- Use a database API that implements RPC stubs to move data across the network

## Create a Facility

There can be numerous RTR applications running on any of your computers in your network. The systems or nodes that service one RTR application and the role of each must be clearly defined. This makes the RTR daemons and processes aware of who is talking with whom, and why. The description of a configuration of a group of nodes into frontends, backends and routers is called a facility. To create a facility, use your command prompt utility again and type `RTR'; at the RTR> prompt, create the facility for this example with the following command on a Windows system in the DOS command prompt window:

```
C:\>  rtr
RTR>  create facility RTRTutor/node=(FE,BE1,BE2) -
_RTR> /frontend=FE/router=FE/backend=(BE1,BE2)
RTR>  exit
```

(You can also repeat this command separately on all three nodes rather than using remote commands.)

With this command, you have now:

- Created a Facility named `RTRTutor' on all three nodes, and
- Defined the role of each node in that facility to show who participates as the client, the primary server, the secondary server and the router.

You can create a facility programmatically as follows:

```
rtr_sStatus_t sStatus;
RTRFacilityManager FacilityManager;
char nodename[ABCMAX_STRING_LEN] = "kenmare";
// gethostname(&nodename[0],ABCMAX_STRING_LEN);
sStatus = FacilityManager.CreateFacility(ABCFacility,
                                         nodename,
                                         nodename,
                                         nodename,
                                         true,
                                         false);
print_sStatus_on_failure(sStatus);
return sStatus;
```

## Take a Break

At this point you have accomplished a lot; you've configured RTR to protect a multi-tiered application by providing failover capability, and to handle communications between your client and your server. Next, you write the application for your client to talk to RTR, and your server to talk to RTR. RTR delivers the messages between the client and server and, if the server crashes, brings in the standby server to handle your client's requests. The client never knows that the server has been switched, and no data or requests to retrieve or modify data is lost.

## Sample Application Code

The C++ modules and header files for this sample application are located in the `examples' subdirectory of the directory into which you installed RTR. They consist of the following files:

- ABC_clientfilenames.h and cpp: The client application

  These files include:

  - ABCOrderTaker

- ABC_serverfilenames .h and cpp: The server application

  These files include:

  - ABCOrderProcessor
  - ABCSClassFactory
  - ABCSHandlers

- ABC_sharedfilenames.h and cpp: Data object implementation that is common to both the client and server applications

  These files include:

  - ABCBook
  - ABCMagazine
  - ABCOrder

- ABCCommon.h: Header file containing definitions specific to both sample applications

Although you won't have much typing to do, this tutorial explains what the code in each file is doing. Copy all of these files into a working directory of your own. For convenience, you may also wish to copy rtrapi.h from the RTR installation directory into your working directory as well.

The example code you'll run must reference the facility you created earlier, so edit the example file *headerfilename.h* and change the FACILITY value to "RTRTutor".

The sample application code supplied with RTR has a lot going on inside of it, but can be broken down into a few general and very simple concepts that will give you an idea of the power of RTR, and how to make it work for you. As you see, you have code for the client application and the server application. Each application talks only to RTR. RTR moves the messages and data between the client and sample applications. This frees you from the worrying about:

- RPC Stubs
- Time zones
- Endianism
- Network protocols and packets

Aren't you relieved? Maybe you should take another break to celebrate!

## Client Application

The files shipped with the RTR kit used in the client application for this tutorial are ABCOrderTaker.h and ABCOrderTaker.cpp. and all of the common files. All applications that wish to talk to RTR through its C++ API need to include `rtrapi.h' as a header file. This file lives in the directory into which RTR was installed, and contains the definitions for RTR classes and values that you'll need to reference in your application. Please do not modify this file. Always create your own application header file to include, as we did in the sample (ABCCommon.h) whenever you need additional definitions for your application.

```
#include "ABCCommon.h"
#include "rtrapi.h"
```

The client application design follows this outline:

1. Initialize RTR
2. Send a message to the server
3. Send a second message to the server
4. Get a response from the server
5. Decide what to do with the response

The messages the client sends are for book orders and magazine orders. These orders are implemented as ABCBookOrder and ABCMagazineOrder data objects.

## Initialize RTR Client Application

This is the first thing that every RTR client application needs to do: tell RTR that it wants to get a facility up and running, and to talk with the server. You find this happening in the (RegisterFacility method in the RTRClientTransactionController class. In the sample application, the implementation for this is in private methods, Initialize and Register, of the ABCOrderTaker class, which derives from RTRClientTransactionController. You remember from the `Start RTR' step in this tutorial that there are RTR daemons or processes executing on the nodes in a facility, listening for communications from other RTR components and applications. Your client application is going to request that all processes associated with the RTRTutor facility "listen up." To do this, you create a client transaction controller and then register a facility in order to enable communication between the client transaction controller and the RTR router. Remember that the RTR router has been described as "keeping track of everything" that goes on in an RTR application.

Create an RTRClientTransactionController object:

```
ABCOrderTaker::ABCOrderTaker():m_bRegistered(false)
{
}
```

First, register with RTR if the client hasn't already done so:

```
rtr_sStatus_t ABCOrderTaker::Register()
{
rtr_sStatus_t sStatus = RTR_STS_OK;
if(false == m_bRegistered)
  {
    // If RTR is not already started then start it now.
       sStatus = StartRTR();
    // Create a Facility if not already created.
       sStatus = CreateFacility();
```

Register the facility with RTR:

```
sStatus = RegisterFacility(ABCFacility);
print_sStatus_on_failure(sStatus);
if (RTR_STS_OK == sStatus)
  {
    m_bRegistered = true;
```

The transaction controller represents the means of communication from the client to the rest of the components in this system. There is a lot going on here to make the communication work, but it's all being done by RTR so you won't have to worry about all of the problems inherent in communicating over a network.

Let's examine what the RegisterFacility method does. First, the RTRFacilityName parameter we sent to it is ABCFacility. This tells RTR the name of the facility we created earlier. Suddenly, RTR has a whole lot more information about your application: where to find the server, the standby server, and the router. You will see later in this tutorial that the server also declares itself and supplies the same facility name.

The RegisterFacility method tells RTR that this application is acting as a client. So now RTR knows that if the server goes down, it certainly doesn't want to force this application to come to the rescue as the standby server! And there will be other things that RTR will be handling that are appropriate only to clients or only to servers. This information helps it to keep track of all the players.

The second parameter, szRecipientName, designates the facility member that has the backend role. The default value is the wildcard "*", meaning that there is no specific recipient name specified.

The third parameter, *pszAccess, is a pointer to the null-terminated string containing the access parameter. This is a security key for authorizing access to a facility by clients and servers. The default value is RTR_NO_ACCESS, when there is no specified access parameter.

At this point, RTR has all of the information it needs to put the pieces together into one system; you're ready to start sending messages to the server, and to get messages back from it.

## RTR Return Status

Your facility may have more than just one client talking to your server. In fact, your neighbor who so generously allowed you to run your standby server on his or her machine might want to get in on this RTR thing, too. That's all right: just add a machine to the RTRTutor facility definition that will also run a copy of the client. But not yet; we're only telling you this to illustrate the point that there can be more than one client in an RTR-based application. Because of this, after the RTR router hands off your client's request to your server, it must then be able to do the same for other clients.

Servers can also decide they want to talk to your client, and the RTR router may need to handle their requests at any time, as well. If RTR were to wait for the server to do its processing and then return the answer each time, there would be an awful bottleneck.

But RTR doesn't wait. This means that the sStatus that you get back from each call means only, "I passed your message on to the server", not that the server successfully handled it and here is the result. So how does your client actually get the result of the request it made on the server? It will need to explicitly "receive" a message, as you'll see later in this tutorial.

## Checking RTR Status

Throughout this code example, you'll see a line of code that looks like:

```
assert(RTR_STS_OK == sStatus );
```

or

```
if(RTR_STS_OK == sStatus)
```

This is good because, as you know from your Programming 101 course, you should always check your return sStatus. But it's also good that your program knows when something has gone wrong and can tell the user, or behave accordingly. The assert function is not part of RTR, but is something you will probably want to do in your application.

To check RTR's return sStatus, compare it to RTR_STS_OK. If it's the same, everything is fine, and you can go on to the next call. But if it is something else, you'll probably to print a message to the user. To get the text string that goes with this sStatus, call `rtr_error_text' which returns a null terminated ASCII string containing the message in human readable format.

## Receiving Messages

As explained earlier, RTR does not hold your client up while it processes your request, or even a request from another client. You must first wait for the client transaction controller RegisterFacility call to let you know that everything is ready to go for the client to start sending messages to a server application.

With the C++ API, your client application receives messages through data class objects on the Receive method of the transaction controller class. The RTRClassFactory class creates the appropriate data object based on the type of data that the transaction controller is about to receive. All C++ API data objects derive from the RTRData class.

The Receive call waits to receive a message or event from RTR:

```
sStatus = Receive(*pRTRData);
```

The *pRTRData is a pointer to a data object and RTR_NO_TIMEOUTMS is the default for the tTimeout parameter.

Remember Programming 101 - check your sStatus every time!

```
assert(RTR_STS_OK == sStatus);
```

In the client sample application ABCOrderTaker, the client derived receive method is DetermineOutcome. This method receives a message from RTR to determine whether the book or magazine order was processed successfully or not.

```
    eOrderStatus ABCOrderTaker::DetermineOutcome()
    {
        RTRData  *pResult = NULL;
        rtr_sStatus_t sStatus;
        eOrderStatus eTxnResult = OrderBeingProcessed;
        rtr_msg_type_t mtMessageType;
```

This code illustrates how an application can retrieve and use the message from an RTRData derived object.

```
        while (OrderBeingProcessed == eTxnResult)
        {
          sStatus = Receive(&pResult);
          print_sStatus_on_failure(sStatus);
          if ( true == pResult->IsRTRMessage())
        {
```

Check to see if we have a sStatus for the transaction. If the transaction sStatus is:

- rtr_mt_accepted, then the server successfully processed the request.
- rtr_mt_rejected, then the server could not process the request.

```
    sStatus = ((RTRMessage*)pResult)->GetMessageType(mtMessageType);
    print_sStatus_on_failure(sStatus);
    if (rtr_mt_accepted == mtMessageType) return eTxnResult =
 OrderSucceeded;
    if (rtr_mt_rejected == mtMessageType) return eTxnResult = OrderFailed;
    }
    }
    return eTxnResult;
    }
```

Information about whether RTR or your server has successfully handled your client's request is returned in the data object. It is received by the transaction controller from RTR in the RTRData object on the Receive call.

The implementation for the handler methods OnAccepted and OnRejected in ABCSHandlers.cpp is:

```
    void ABCSHandlers::OnAccepted( RTRMessage *pRTRMessage,
    RTRServerTransactionController *pController )
    {
        pController->AcknowledgeTransactionOutcome();
        return;
    }
    void ABCSHandlers::OnRejected( RTRMessage *pRTRMessage,
    RTRServerTransactionController *pController )
    {
        pController->AcknowledgeTransactionOutcome();
        return;
    }
```

## Send Messages

With the C++ API, the start of a transaction is implicit, with the sending of the first message to a server application. Once the client transaction controller has registered a facility and its message and event handlers, the rest of the client application is simply a `send/receive' message loop. It continues to send messages to the server, then listen for the server's response. It is important to remember that, although the client is sending these messages to the server, it is doing so through RTR. Because of this, the client can receive, asynchronously, different types of messages and events, including:

- A notice from the server of failure to process the sent message
- An answer to the sent message from the server
- An "out of band" message from the server regarding server sStatus

With the C++ API, there are four types of data you can receive:

- RTREvent
- RTRMessage
- RTRApplicationEvent
- RTRApplicationMessage

The RTRClassFactory creates these data objects when a Receive method is called for a transaction controller. The class factory takes the incoming RTRData object and creates the appropriate data object based on the type of incoming data.

In addition to clients and servers sending and receiving messages, RTR may send the client messages under certain conditions. So the client application must be prepared to accept any of these messages, and not necessarily in a particular sequence.

That's certainly a tall order! How should you handle this? Well, there are a number of ways, but you typically implement these possibilities in the client message and event handlers. (The implementation details of handling messages and events on a Receive are implemented in the sample server application ABCOrderProcessor.) In this tutorial we will explain how to run a "message loop" that both sends and receives messages.

The client sample application ABCOrderTaker has a derived SendOrder method for sending RTRApplicationMessage objects to the server application. These objects can be either book orders or magazine orders. (ABCOrderTaker derives from RTRClientTransactionController and thus inherits the Register and SendApplicationMessage methods.)

```
bool ABCOrderTaker::SendOrder(ABCOrder *pOrder)
{
    rtr_sStatus_t sStatus;
    eOrderStatus eTxnResult = OrderBeingProcessed;
// Register with RTR if we havn't already done so.
// This will make sure we are ready to start sending data.
    sStatus = Register();
    if (RTR_STS_OK != sStatus) return false;
// If we can't register with RTR then exit
// Send this Book Order object to a server capable
// of processing it.
    sStatus = SendApplicationMessage(pOrder);
    print_sStatus_on_failure(sStatus);
//  Let RTR know that this is the object  being sent and
//  that we are done with our work.
    sStatus = AcceptTransaction();
    print_sStatus_on_failure(sStatus);
//  Determine if the server successfully processed the request
    eTxnResult = DetermineOutcome();
```

```
        return true;
    }
```

## A Word about RTR Data types

You may have noticed that your client, server and router can be on any one of many different operating systems. And you've probably written code for more than one operating system and noticed that each has a number of data types that the other doesn't have. If you send data between a Solaris UNIX machine and an OpenVMS or Windows NT machine, you'll also have to worry about the order different operating system stores bytes in their data types (called "endian" order). And what happens to the data when you send it from a 16 bit Intel 486 Windows machine to a 64 bit Alpha UNIX machine?

Thanks to RTR, you don't need to worry about it. RTR will handle everything for you. Just write standard C++ code that will compile on the machines you choose, and the run-time problems won't complicate your design. When you do this, you need to use RTR data types to describe your data. RTR translates everything necessary when your data gets to a new machine by converting the data to the native data types on the operating system with which it happens to be communicating at the time.

To illustrate this, the example code evaluates your input parameters and places them into an RTRData-derived RTRApplicationMessage object, ABCOrder. One sample application data class is ABCBook, which derives from ABCOrder. This subclass defines the data that is passed from client to server for a book order. This data class is defined in ABCBook.h and implemented in ABCBook.cpp

You'll notice that the data types which make up this object aren't your standard data types - they are RTR data types. And they are generic enough to be able to be used on any operating system: 8 bit unsigned, 32 bit unsigned, and a string.

```
UINT         m_uiPrice;
UINT         m_uiISBN;
CString      m_csTitle;
CString      m_csAuthor;)
unsigned int m_uiISBN;
unsigned int m_uiPrice;
char         m_szTitle[ABCMAX_STRING_LEN];
char         m_szAuthor[ABCMAX_STRING_LEN];
```

## Send/Receive Message Loop

As mentioned earlier, an RTR client application typically contains a message loop that sends messages to the server via the RTR router, and handles messages that come from the server via the router, or from RTR itself.

This code illustrates how an application can retrieve and use the message from an RTRData derived object.

```
while (OrderBeingProcessed == eTxnResult)
{
sStatus = Receive(&pResult);
print_sStatus_on_failure(sStatus);
if ( true == pResult->IsRTRMessage())
{
// Check to see if we have a sStatus for the txn.
// rtr_mt_accepted = Server successfully processed our request.
// rtr_mt_rejected = Server could not process our request.
sStatus = ((RTRMessage*)pResult)->GetMessageType(mtMessageType);
print_sStatus_on_failure(sStatus);
```

```
    if (rtr_mt_accepted == mtMessageType)
        return eTxnResult = OrderSucceeded;
    if (rtr_mt_rejected == mtMessageType)
    return eTxnResult = OrderFailed;
    }
    sStatus = SendApplicationMessage( *pRTRApplicationMessage,
                                      bReadonly = false,
                                      bReturnToSender = false,
                                      mfMessageFormat=RTR_NO_MSGFMT);
    }
    assert(RTR_STS_OK == sStatus);
```

The first message is sent to the server in the first parameter of the SendApplicationMessage call. As you will see, this is part of the flexibility and power of RTR. The parameter pRTRApplicationMessage is a pointer to a block of memory containing your data. RTR doesn't know what it's a pointer to, but it doesn't need to know this. You, as the programmer, are the only one who cares what it is. It's your own data object that carries any and all of the information your server will need in order to do your bidding. We'll see this in detail when we look at the server code.

You do not need to tell RTR how big the piece of memory being pointed to pRTRApplicationMessage is. The data object automatically lets RTR know how many bytes to move from your client machine to your server machine, so that your server application has access to the data being sent by the client.

And now, the client waits for a response from the server.

The client receives the server's reply or an rtr_mt_rejected and calls the client message handler method, OnRejected

```
    sStatus = Receive( *pRTRData);
    assert(RTR_STS_OK == sStatus);
```

Again you see the pRTRData parameter is a pointer to a data object created by you as the programmer, and can carry any information you need your server to be able to communicate back to the your client.

The RTRData object contains a code that tells you what kind of a message you are now receiving on your transaction controller. If the RTR message type contains the value rtr_mt_reply, then you are receiving a reply to a message you already sent, and your receive message object has been written to with information from your server.

```
    sStatus = ((RTRMessage*)pResult)->GetMessageType(mtMessageType);
    print_sStatus_on_failure(sStatus);
    if (rtr_mt_accepted == mtMessageType)
        return eTxnResult = OrderSucceeded;
    if (rtr_mt_rejected == mtMessageType)
        return eTxnResult = OrderFailed;
```

If GetMessageType contains the value rtr_mt_rejected, then something has happened that caused your transaction to fail after you sent it to the router. You can find out what that `something' is by looking at the sStatus returned by the Receive call. You will recall that making the rtr_error_text call and passing the sStatus value will return a human readable null terminated ASCII string containing the error message.

This is where you'll need to make a decision about what to do with this transaction. You can abort and exit the application, issue an error message and go onto the next message, or resend the message to the server. This code re-sends a rejected transaction to the server.

When your client application receives an rtr_mt_reply message, your message has come full circle. The client has made a request of the server on behalf of the user; the server has responded to this request. If you're satisfied that the transaction has completed successfully, you must notify RTR so that it can do its own housekeeping. To this point, this transaction has been considered "in progress", and its sStatus kept track of at all times. If all parties interested in this transaction (this includes the client AND the server) notify RTR that the transaction has been completed, RTR will stop tracking it, and confirm to all parties that it has been completed. This is called `voting'.

```
if (msgsb.msgtype == rtr_mt_reply)
{
sStatus = AcceptTransaction(RTR_NO_REASON)
assert (RTR_STS_OK == sStatus);
```

And now the client waits to find out what the result of the voting is.

```
sStatus = Receive( *pRTRData, RTR_NO_TIMOUTMS);
assert(RTR_STS_OK == sStatus);
```

If everyone voted to accept the transaction, the client can move on to the next one. But if one of the voters rejected the transaction, then another decision must be made regarding what to do about this transaction. This code attempts to send the transaction to the server again.

```
sStatus = ((RTRMessage*)pResult)->GetMessageType(mtMessageType);
print_status_on_failure(sStatus);
if (rtr_mt_accepted == mtMessageType) return eTxnResult =
                                          OrderSucceeded;
if (rtr_mt_rejected == mtMessageType) return eTxnResult =
                                          OrderFailed;
```

All of the requested messages, or transactions, have been sent to the server, and responded to. The only RTR cleanup we need to do before we exit the client is to close the transaction controller. This is similar to signing off, and RTR releases all of the resources it was holding for the client application.

Now, that wasn't so bad, was it? Of course not. And what has happened so far? The client application has sent a message to the server application. The server has responded. RTR has acted as the messenger by carrying the client's message and the server's response between them.

Next, let's see how the server gets these messages, and sends a response back to the client.

## Server Application

The files shipped with the RTR kit used in the server application for this tutorial are the ABCOrderProcessor, ABCSHandlers and ABCSClassFactory files **,** in addition to the common files **.** These common files, including ABCCommon, ABCBook, and ABCMagazine are used in both client and the server applications. This is for a number of reasons, but most importantly that both the client and the server use the same definitions for the data objects they pass back and forth as messages. With the exception of only two items, there will be nothing in this server that you haven't already seen in the client. It's doing much the same things as the client application is doing. It creates a server transaction controller object for connecting to the router, telling the router that it is a server application; and then registers a partition. It waits to hear that the RegisterPartition request has been successfully executed; runs a loop that receives messages from the client; carries out the client's orders; sends the response back to the client. And the server gets to vote, too, on whether each message/response loop is completed.

One of the differences between the client andserver is the types of messages a server can receive from RTR; we'll go through some of them in this section of the tutorial about the server application.

The other difference is the RegisterPartition call which is sent to RTR. We mentioned partitions while discussing the client application, but said we'd discuss them later. Well, it's later...

## Initialize RTR

The server creates a transaction controller and registers a partition. In addition, the server registers message and event handlers and a class factory, causing RTR to initialize a number of resources for use by the server, as well as to gather information about the server. In the Register methods in the server application, ABCOrderProcessor.cpp, you'll find the example server calling RegisterPartition. You see that the RegisterPartition method creates a single RTR data partition for each time it is called. In the server code, there are two partitions, ABCPartition1 and ABCPartition2.

```
sStatus = RegisterPartition(ABCPartition1);
print_sStatus_on_failure(sStatus);

sStatus = RegisterPartition(ABCPartition2);
print_sStatus_on_failure(sStatus);
```

In order to call RegisterPartition, the sample application includes a CreateRTREnvironment method that is first called in the ABCOrderProcessor::Register method.

## Data Partitions

What is data partitioning, and why would you wish to take advantage of it? It is possible to run a server application on each of multiple backend machines, and to run multiple server applications on any backend machine. When a server registers a partition to begin communicating with the RTR router, it uses the KeySegment information to tell RTR that it is available to handle certain key segments. A key segment can be "all last names that start with A to K" and "all last names that start with L to Z", or "all user identification numbers from zero to 1000" and "all user identification numbers from 1001 to 2000".

In the sample application, the implementation is as follows:

```
void ABCOrderProcessor::CreateRTREnvironment()
{
rtr_sStatus_t sStatus;
// If RTR is not already started then start it now.
StartRTR();
// Create a Facility if not already created.
CreateFacility();
// Create a partition that processes ISBN numbers in
// the range 0 - 99
unsigned int low = 0;
unsigned int max = 99;
RTRKeySegment KeyZeroTo99(rtr_keyseg_unsigned,
                          sizeof(int),
                          0,
                          &low,
                          &max );
RTRPartitionManager PartitionManager;
sStatus = PartitionManager.CreateBackendPartition(ABCPartition1,
                                                  ABCFacility,
                                                  KeyZeroTo99,
                                                  false,
                                                  true,
                                                  false);
print_sStatus_on_failure(sStatus);
// Create a partition that processes ISBN numbers in
// the range 100 - 199
low = 100;
max = 199;
RTRKeySegment Key100To199( rtr_keyseg_unsigned,
                           sizeof(int),
                           0,
                           &low,
                           &max );
sStatus = PartitionManager.CreateBackendPartition(ABCPartition2,
ABCFacility,
                                                  Key100To199,
                                                  false,
                                                  true,
                                                  false);
    print_sStatus_on_failure(sStatus);
}
```

Each key segment describes a data partition. Data partitions allow you to use multiple servers to handle the transactions all of your clients are attempting to perform; in this way, they don't all have to wait in line to use the same server. They can get more done in less time. Data partitions can be specified through a command line interface or programmatically through the RTRPartitionManager class.

The *VSI Reliable Transaction Router Application Design Guide* goes into more detail about data partitioning.

Again, we use the RTR data object that RTR will place information in, and the user-defined data object, ABCOrder, that the client's data will be copied into. But at this point, the server is talking with RTR only, not the client, so it is expecting an answer from RTR; all the server really wants to know is that the transaction controller is ready to receive a client request. If it isn't, the server application will write out an error message and exit with a failure sStatus. The implementation of the sample server application's Register function:

- Creates the environment for the server to run in
- Registers a facility
- Registers two partitions
- Registers a class factory
- Registers message and event handlers

The sStatus is checked after each of these calls and if they are all successful, the server is ready to receive incoming requests from the client application.

```
void ABCOrderProcessor::Register()
{
    rtr_sStatus_t sStatus;
// Create an environment that our server can run in.
    CreateRTREnvironment();
// Register with RTR the following objects
    sStatus = RegisterFacility(ABCFacility);
    print_sStatus_on_failure(sStatus);
// ABC Partition
    sStatus = RegisterPartition(ABCPartition1);
    print_sStatus_on_failure(sStatus);
    sStatus = RegisterPartition(ABCPartition2);
    print_sStatus_on_failure(sStatus);
// ABC Class Factory
    sStatus = RegisterClassFactory(&m_ClassFactory);
```

```
        print_sStatus_on_failure(sStatus);
    // ABC Handlers
        sStatus = RegisterHandlers(&m_rtrHandlers,&m_rtrHandlers);
        print_sStatus_on_failure(sStatus);
        return;
```

The RegisterHandlers method takes two parameters; the first parameter is a pointer to an RTRServerMessageHandler object and the second parameter is a pointer to an RTRSeverEventHandler object. ABCHandlers multiply derives from both of these foundation classes.

The server message handler specifies all messages generated by RTR or the RTR application that a server application may receive. The server event handler specifies all events generated by RTR or the RTR application that a server application may receive.

And now that the transaction controller has been established, the server waits to receive messages from the client application and the RTR router.

## Receive/Reply Message Loop

The server sits in a message loop receiving messages from the router, or from the client application via the router. Like the client, it must be prepared to receive various types of messages in any order and then handle and reply to each appropriately. But the list of possible messages the server can receive is different than that of the client. This example includes some of those. First, the server waits to receive a message from RTR.

The implementation in the ProcessIncomingOrders method of the ABCOrderProcessor class:

```
    // Start processing orders
    abc_sStatus sStatus;
    RTRData *pOrder = NULL;
    while (true)
    {
    // Receive an Order
    sStatus = Receive(&pOrder);
    print_sStatus_on_failure(sStatus);
    // If we can't get an Order then stop processing.
    if(ABC_STS_SUCCESS != sStatus) break;
    // Dispatch the Order to be processed
    sStatus = pOrder->Dispatch();
    print_sStatus_on_failure(sStatus);
    // Check to see if there were any problems processing the order.
    // If so, let the handler know to reject this txn when asked to
    // vote.
    CheckOrderStatus(sStatus);
```

Upon receiving the message the server checks the RTRData object's message type field to see what kind of message it is. Some are messages directly from RTR and others are from the client. In any event, the class factory creates the appropriate data object for the server application to handle the incoming data. When the message is from the client, your application will read the data object you constructed to pass between your client and server and, based on what it contains, do the work it was written to do. In many cases, this will involve storing and retrieving information using your database.

But when the message is from RTR, how should you respond? Let's look at some of the types of messages a server gets from RTR, and what should be done about them.

The following implementation from the ABCSHandlers.cpp file is for an rtr_mt_msg1_uncertain RTR message:

```
void ABCSHandlers::OnUncertainTransaction( RTRApplicationMessage
*pRTRData, RTRServerTransactionController *pController )
{
    return;
}
```

The rtr_mt_msg1 and rtr_mt_msg1_uncertain messages identify the beginning of a new transaction. The rtr_mt_msg1 message says that this is a message from the client, and it's the first in a transaction. When you receive this message type, you will find the client data in the object pointed to by pRTRData parameter of this call. The client and server have agreed on a common data object that the client will send to the server whenever it makes a request: this is the ABCOrder object we looked at in the client section of this tutorial. RTR has copied the data from the client's data object into the one whose memory has been supplied by the server. The server's responsibility when receiving this message is to process it. On receiving an rtr_mt_msg1, the server application calls the OnInitialize and OnApplicationMessage server message handler methods.

On receiving an RTR message rtr_mt_msg1, the server application calls the handler methods OnInitialize and OnApplicationMessage by default. Business logic processing can be done within the OnApplicationData method.

The sample server application implements the OnInitialize method and overloads the Dispatch method with an implementation in ABCOrder that deserializes the ABCOrder data object, rather than having the default Dispatch method invoking OnApplicationMessage.

From the ABCSHandlers class:

```
void ABCSHandlers::OnInitialize( RTRApplicationMessage *pRTRData,
RTRServerTransactionController *pController )
{
    m_bVoteToAccept = true;
    return;
}
```

The overloaded Dispatch method in the ABCOrder data object:

```
rtr_sStatus_t ABCOrder::Dispatch()
{
// Populate the derived object
    ReadObject();
// Process the purchase that the derived object represents
    bool bStatus = Process();
    if (true == bStatus)
    {
        return ABC_STS_SUCCESS;
    }
    else
    {
        return ABC_STS_ORDERNOTPROCESSED;
    }
}
```

## Recovered Transactions:

The rtr_mt_msg1_uncertain message type tells the server that this is the first message in a recovered transaction. In this instance, the original server the application was communicating with failed, possibly leaving some of its work incomplete, and now the client is talking to the standby server. What happens to that incomplete work left by the original server? Looking back at the client you will recall that everyone got to vote as to whether the transaction was accepted or rejected, and then the client waited to see what the outcome of the vote was. While the client was waiting for the results of this vote, the original server failed, and the standby server took over. RTR uses the information it kept storing to the recovery journal, which you also created earlier, to replay to the standby server so that it can recover the incomplete work of the original server.

When a server receives the `uncertain' message, it knows that it is stepping in for a defunct server that had, to this point, been processing client requests. But it doesn't know how much of the current transaction has been processed by that server, and how much has not, even though it receives the replayed transactions from RTR. The standby server will need to check in the database or files to see if the work represented by this transaction is there and, if not, then process it. If it has already been done, the server can forget about it .

If the received message contains rtr_mt_msg1_uncertain:

```
replay = RTR_TRUE;
else
replay = RTR_FALSE;
if ( replay == TRUE )
// The server should use this opportunity to
// clean up the original attempt, and prepare
// to process this request again.
else
// Process the request.
```

The server then replies to the client indicating that it has received this message and handled it.

The server typically uses the SendApplicationMessage call to answer the request the client has made. In some cases, this may mean that data needs to be returned. This will be done in the data object that has been agreed upon by both the client and the server.

## Prepare Transaction

The rtr_mt_prepare message tells the server to prepare to commit the transaction. All messages from the client that make up this transaction have been received, and it is now almost time to commit the transaction in the database. This message type will never be sent to a server that has not requested an explicit prepare.

After determining whether it is possible to complete the transaction based on what has occurred to this point, the server can either call RejectTransaction to reject the transaction, or set all of the required locks on the database before calling AcceptTransaction to accept the transaction.

```
void ABCSHandlers::OnPrepareTransaction( RTRMessage *pRTRMessage,
RTRServerTransactionController *pController )
{
// Check to see if anything has gone wrong. If so, reject
//  the transaction, otherwise accept it.
    if (true == m_bVoteToAccept)
    {
        pController->AcceptTransaction();
    }
    else
    {
        pController->RejectTransaction();
    }
```

```
        return;
    }
```

Because this example code is not dealing with a database, nor is it bundling multiple messages into a transaction, the code here immediately votes to accept the transaction.

## Transaction Rejected

The rtr_mt_rejected message is from RTR, telling the server application that a participant in the transaction voted to reject it. If one participant rejects the transaction, it fails for all. The transaction will only be successful if all participants vote to accept it. When it receives this message, the server application should take this opportunity to roll back the current transaction if it is processing database transactions.

The sample server application includes the following code to check the order if it was not processed. If the order was not processed properly, then the handler method OnABCOrderNotProcessed is called and m_bVoteToAccept is set to false. This causes OnPrepareTransaction to reject the transaction.

```
    void ABCOrderProcessor::CheckOrderStatus (abc_sStatus sStatus)
    {
    // Check to see if there were any problems
    // processing the order. If so, let the handler know
    // to reject this txn when asked to vote.
        if (sStatus == ABC_STS_ORDERNOTPROCESSED)
        {
    // Let the handler know that the current txn should be rejected
        GetHandler()->OnABCOrderNotProcessed();
        };
    };
    void ABCSHandlers::OnABCOrderNotProcessed()
    {
        m_bVoteToAccept = false;
        return;
    }
```

Finally, explicitly end the transaction on a reject, the handler method OnRejected is called:

```
    void ABCSHandlers::OnRejected( RTRMessage *pRTRMessage,
    RTRServerTransactionController *pController )
    {
        pController->AcknowledgeTransactionOutcome();
        return;
    }
```

## Transaction Accepted

RTR is telling the server that all participants in this transaction have voted to accept it. If database transactions are being done by the server, this is the place at which the server will want to commit the transaction to the database, and release any locks it may have taken on the database.

```
 void ABCSHandlers::OnAccepted( RTRMessage *pRTRMessage,
 RTRServerTransactionController *pController )
    {
        pController->AcknowledgeTransactionOutcome();
        return;
    }
```

Note the AcknowledgeTransactionOutcome call in the server. This is an explicit method for completing a transaction.

That's it. You now know how to write a client and server application using RTR as your network communications, availability and reliability infrastructure. Congratulations!

## Build and Run the Servers

Compile the ABC server and ABC shared files on the operating system that will run your server applications. If you are using two different operating systems, then compile it on each of them. To build on UNIX, issue the command:

```
cxx -o server server.c shared.c /usr/shlib/librtr.so -DUNIX
```

You should start the servers before you start your clients. They will register with the RTR router so that the router will know where to send client requests. Start your primary server with the appropriate `run' command for its operating system along with the two parameters `1' and `h'. To run on UNIX:

```
% ./server 1 h
```

Start your standby server with the parameters `2' and `h'.

```
% ./server 2 h
```

Build and Run the Client: Compile the ABC CLIENT and ABC SHARED modules on the operating system which will run your client application. To build on UNIX:

```
% cxx -o client client.c shared.c /usr/shlib/librtr.so -DUNIX
```

Run the client with the following command:

```
% ./client 1 h 10
```

or

```
C:\RtrTutor\> client.exe 1 h 10
```

In many ways, this tutorial has only scratched the surface of RTR. There is a great deal more that RTR gives you to make your distributed application reliable, available, and perform better. The following sections of this document highlight some of the capabilities you have at your service. For more details on each item, and information on what additional features will help you to enhance your application, look first through the *VSI Reliable Transaction Router Application Design Guide*. Then, earlier sections of this *VSI Reliable Transaction Router C++ Foundation Classes* manual will tell you in detail how to implement each capability.

Compaq Computer Corporation also offers training classes for RTR, and if you'd like to attend any of them, contact your Compaq representative.

## Callout Server

RTR supports the concept of a "callout server" for authentication. You may designate an additional application on your server machines or your router machine as a callout server with the RTRFacilityManager class methods. Callout servers will be asked to check all requests in a facility, and are asked to vote on every transaction.

The CreateFacility method in the RTRFacilityManager class includes a boolean parameter bEnableBackendCallout for specifying a callout server.

## Events

In addition to messages, RTR can be used to dispatch asynchronous events on servers and clients. A callback function in the user's server and client applications can be designated which RTR will call asynchronously to dispatch events to your application.

## Shadowing

This tutorial only discussed failover to a standby server. But RTR also supports shadowing: while your server is making changes to your database, another "shadow" server can be making changes to an exact copy of your database in real time. If your primary server fails, your shadow server will take over, and record all of the transactions occurring while your primary server is down. Your primary server will be given the opportunity to update the original database and catch up to the correct state when it comes back up. Primary and secondary shadow server can also have standby servers for failover! So as you can see, if your database and transactions are important enough to you, you have the opportunity to double and triple protect them with an RTR configuration including any of

- Multiple standby software servers on a primary hardware backend system
- Shadow backend system replicating all transactions on a duplicate database
- Failover backend systems for each of your primary and shadow backends
- Failover routers

## Transactions

One of RTR's greatest strengths is in supporting transactions. The *VSI Reliable Transaction Router Application Design Guide* goes into more detail regarding transactions and processing of transactions.

## RTR Utility

You've seen how to use the RTR utility (or the command line interface) to start RTR and to create a facility. But the RTR utility contains many more features than this, and in fact can be used to prototype an application. Refer to the *VSI Reliable Transaction Router System Manager's Manual* for details.

# Chapter 6. Sample Application Code

The RTR book ordering sample application shows how the C++ Foundation classes can be used to simulate purchasing merchandise for a fictitious company named ABC.

The client, ABCOrderTaker, has a hard-coded book request which is represented by the ABCBook class. This book request has an ISBN number used for data routing. The server will display a dialog box containing the contents of the newly reconstituted ABCBook object.

The following sample application code comes from the Examples directory and includes:

- Sample1

  This file contains `int main` and provides a sample for which the sample application takes book and magazine orders from the client application (ABCOrderTaker) and processes them in the server application (ABCOrderProcessor).

- ABCOrderTaker

  Located in ABCOrderTaker.h and ABCOrderTaker.cpp, this client-side class supplies ability to pass an object derived from ABCOrder to a server. This class is derived from RTRClientTransactionController and derives from RTRClientMessageHandler and RTRClientEventHandler

- ABCOrderProcessor

  Located in ABCOrderProcessor.h, ABCOrderProcessor.cpp, this server-side class processes the request sent to it by the client. This class is derived from RTRServerTransactionController.

- ABCOrder

  Located in ABCOrder.h, ABCOrder.cpp, this is an abstract base class (for ABCBook and ABCMagazine) which requires all derived classes to implement three member functions.

  ReadObject()
  WriteObject()
  Process()

  This class is derived from RTRApplicationMessage.

- ABCBook

  Located in ABCBook.h, ABCBook.cpp, this class represents a book order. This class is able to write and read its state to the memory managed by its base class RTRData. This class derives from ABCOrder.

There are also class factory, client and server handlers, ABCMagazine, and ABCCommon classes in the Examples directory.

## 6.1. Sample Main Program

```
#include "ABCCommon.h"
```

```
#include "ABCOrderTaker.h"
#include "ABCOrderProcessor.h"
#include "ABCBook.h"
#include "ABCMagazine.h"
void GenerateOrders();

int main(int argc, char* argv[])
{
bool bValidInput = false;
while (false == bValidInput)
{
cout << endl;
cout << "*********************************************" << endl;
cout << endl;
cout << "1 - Start Server to process incoming orders" << endl;
cout << "2 - Start Client to generate predefined orders" << endl;
cout << "0 - Quit" << endl;
cout << endl;
cout << "*********************************************" << endl;
cout << endl << "Which Test should be run? : ";
unsigned int uiAnswer;
cin >> uiAnswer;
switch (uiAnswer)
    {
    case 1 : {  ABCOrderProcessor OrderProcessor;
        // Call ProcessIncomingOrders which will loop
        //forever processing orders from clients.
        OrderProcessor.ProcessIncomingOrders();
        break;
    }
    case 2 : {
        // Send some orders
        GenerateOrders();
        break;
    }
    case 0 : {
        return 0;
    }
    } // switch
      } //while
    return 0;
}

void GenerateOrders()
{
    abc_status sStatus;
    // Create an Order Taker.
    ABCOrderTaker OrderTaker;

// Create a sample book order and populate it with the
// ISBN 49, Price and Title
ABCBook Book;
Book.AddOrder( 49, 12345, "Everything to the Internet",
                                  "Michael Capellas");
// Send this book order to the server for processing.
// note: This will be txn #1
    cout << endl << "Transaction # 1" <<endl;
    sStatus = OrderTaker.SendOrder(&Book);
```

```
        cout << endl;
    // Reset the stream. This way we will reuse the beginning of
    // the buffer that the stream manages.
    Book.ResetStream();

        // Send another order to a server which handles ISBN 99
    Book.AddOrder( 99, 56789, "Java How To Program",
                                    "Deitel & Deitel");
// Send this book order to the server for processing.
        // note: This will be txn #2
        cout << endl << "Transaction # 2" <<endl;
        sStatus = OrderTaker.SendOrder(&Book);
        cout << endl;

        ABCMagazine Magazine;

    Magazine.AddOrder(29,"PC Week","ZIFF-DAVIS", "February 2000");

        // Send this book order to the server for processing.
        // note: This will be txn #3
        cout << endl << "Transaction # 3" <<endl;
        sStatus = OrderTaker.SendOrder(&Magazine);
        cout << endl;
    }
```

## 6.2. Client Application ABCOrderTaker

```
    // ABCOrderTaker.cpp: implementation of the ABCOrderTaker class.
    //
    //////////////////////////////////////////////////////////////////////

    #include "ABCCommon.h"
    #include "ABCOrderTaker.h"

    //////////////////////////////////////////////////////////////////////
    // Construction/Destruction
    //////////////////////////////////////////////////////////////////////

    ABCOrderTaker::ABCOrderTaker() : m_bRegistered(false)
    {

    }

    ABCOrderTaker::~ABCOrderTaker()
    {

    }

    abc_status ABCOrderTaker::SendOrder(ABCOrder *pOrder)
    {
        abc_status sStatus;
    //  Register with RTR if we havn't already done so.
    //  This will make sure we are ready to start sending data.
        sStatus = Register();
        if (ABCSuccess != sStatus) return false;
    // If we can't register with RTR then exit
    // Start the Transaction
```

```
        cout << "StartTransaction..." << endl;
        sStatus = StartTransaction();
        print_status_on_failure(sStatus);
    //   Send this Book Order object to a server capable
    //   of processing it.
        cout << "SendApplicationMessage..." << endl;
sStatus = SendApplicationMessage(pOrder);
print_status_on_failure(sStatus);

    //   Let RTR know that this is the only object being sent
    //   and that we are done with our work.
        cout << "AcceptTransaction..." << endl;
        sStatus = AcceptTransaction();
        print_status_on_failure(sStatus);
//   Determine if the server successfully processed the request
            return DetermineOutcome();
        }
    rtr_status_t ABCOrderTaker::Register()
    {
        rtr_status_t sStatus = RTR_STS_OK;
    if(false == m_bRegistered)
        {
    //   If RTR is not already started then start it now.
            sStatus = StartRTR();
    //   Create a Facility if not already created.
            sStatus = CreateFacility();

    //   Register our facility with RTR.
            sStatus = RegisterFacility(ABCFacility);
            print_status_on_failure(sStatus);
    if(RTR_STS_OK == sStatus)
            {
                m_bRegistered = true;
            }

    // ABC Handlers
        sStatus = RegisterHandlers(&m_rtrHandlers,&m_rtrHandlers);
        print_status_on_failure(sStatus);
        }
        return sStatus;
    }
    abc_status ABCOrderTaker::DetermineOutcome()
    {
        RTRData  *pResult = NULL;
        abc_status sStatus = ABCSuccess;

        // Simply wait for RTR to send us an accepted or rejected.

        // We can dispatch everything we get and let the default
        // handlers process what we don't care about.

        bool bDone = false;
        while (!bDone)
        {
        sStatus = Receive(&pResult);
        print_status_on_failure(sStatus);
        sStatus = pResult->Dispatch();
        if (ABCOrderSucceeded == sStatus)
        {
        cout << "Transaction succeeded..." << endl;
        bDone = true;
        }
        else
            if (ABCOrderFailed == sStatus)
            {
            cout << "Transaction failed..." << endl;
            bDone = true;
            }
        }
    delete pResult;
        return sStatus;
    }
```

## 6.3. Server Application ABCOrderProcessor

```
    // ABCOrderProcessor.cpp: implementation of the ABCOrderProcessor
 class.
    //
    //////////////////////////////////////////////////////////////////////

    #include "ABCCommon.h"
    #include "ABCOrderProcessor.h"
    #include <stdio.h>

    //////////////////////////////////////////////////////////////////////
    // Construction/Destruction
    //////////////////////////////////////////////////////////////////////

    ABCOrderProcessor::ABCOrderProcessor()
    {

    }

    ABCOrderProcessor::~ABCOrderProcessor()
    {

    }

    void ABCOrderProcessor::ProcessIncomingOrders()
    {
    //   Register with RTR. This will make sure we are ready to
    //   start receiving data.
        Register();
    //   Start processing orders
        abc_status sStatus = RTR_STS_OK;
        RTRData *pOrder = NULL;

        while (1)
        {
    //   Receive an Order
        sStatus = Receive(&pOrder);
        print_status_on_failure(sStatus);
        if(ABCSuccess != sStatus) break;
        // if we can't get an Order then stop processing.
    // Dispatch the Order to be processed
    // note: This could be any kind of data. ie. RTRMessage RTREvent,
    // RTRApplicationMessage or RTRApplicationEvent.
    // The class ABCOrder(derived from RTRApplicationMessage) has
    // redefined the Dispatch() method to call the Process() method
    // of its derived class (ABCBook or ABCMagazine). All other
    // data classes use the default implemenation of Dispatch()
    // which will call the appropriate handler.
            sStatus = pOrder->Dispatch();
            print_status_on_failure(sStatus);
    // Check to see if there were any problems processing the order.
    // If so, let the handler know to reject this txn when asked to
    // vote.
    // note : For the ABC company, orders are processed in the
    // Process() method of all ABCOrder derived classed.
            CheckOrderStatus(sStatus);

    //   Delete this order that was allocated by the class factory.
    // note: In this sample the class factory returns a separate
    // instance of an order each time it is called.
            delete pOrder;
        }
    return;
}
void ABCOrderProcessor::Register()
    {
        rtr_status_t sStatus;

        // Create an environment that our server can run in.
        CreateRTREnvironment();

        // Register with RTR the following objects
        sStatus = RegisterFacility(ABCFacility);
        print_status_on_failure(sStatus);

        // ABC Partition
        sStatus = RegisterPartition(ABCPartition1);
        print_status_on_failure(sStatus);

        sStatus = RegisterPartition(ABCPartition2);
        print_status_on_failure(sStatus);

        // ABC Class Factory
        sStatus = RegisterClassFactory(&m_ClassFactory);
        print_status_on_failure(sStatus);

        // ABC Server Handlers
        sStatus = RegisterHandlers(&m_rtrHandlers,&m_rtrHandlers);
        print_status_on_failure(sStatus);

        return;
    }

    void ABCOrderProcessor::CreateRTREnvironment()
    {
        rtr_status_t sStatus;
        // If RTR is not already started then start it now.
        StartRTR();
        // Create a Facility if not already created.
        CreateFacility();
        // Create a partition that processes ISBN numbers in the
        // range 0 - 99
        unsigned int low = 0;
        unsigned int max = 99;
        RTRKeySegment KeyZeroTo99(  rtr_keyseg_unsigned,
                                    sizeof(int),
                                         0,
                                         &low,
                                         &max );
        RTRPartitionManager PartitionManager;
        sStatus = PartitionManager.CreateBackendPartition(
                                                ABCPartition1,
                                                ABCFacility,
                                                KeyZeroTo99,
                                                false,
                                                true,
                                                false);
        print_status_on_failure(sStatus);

        // Create a partition that processes ISBN numbers in the
        // range 100 - 199
        low = 100;
        max = 199;
        RTRKeySegment Key100To199(  rtr_keyseg_unsigned,
                                    sizeof(int),
                                         0,
                                         &low,
                                         &max );
        sStatus = PartitionManager.CreateBackendPartition(
                                                   ABCPartition2,
                                                   ABCFacility,
                                                   Key100To199,
                                                   false,
                                                   true,
                                                   false);
        print_status_on_failure(sStatus);
    }


    void ABCOrderProcessor::CheckOrderStatus (abc_status sStatus)
    {
    // Check to see if there were any problems processing the order.
    // If so, let the handler know to reject this transaction when
    // asked to vote.
        if (sStatus == ABCOrderFailed)
        {
    // Let the handler know that the current txn should be rejected
            GetHandler()->OnABCOrderNotProcessed();
        };

    }
```

## 6.4. Data Class ABCOrder

```
// ABCOrder.cpp: implementation of the ABCOrder class.
```

```
//
//////////////////////////////////////////////////////////////////////

#include "ABCCommon.h"
#include "ABCOrder.h"

//////////////////////////////////////////////////////////////////////
// Construction/Destruction
//////////////////////////////////////////////////////////////////////

ABCOrder::ABCOrder() : m_uiPrice(0)
{
    m_szTitle[0] = '\0';
    m_szAuthor[0] = '\0';
}

ABCOrder::~ABCOrder()
{

}

rtr_status_t ABCOrder::Dispatch()
{
    // Populate the derived object
    ReadObject();
// Since we have overridden Dispatch() in our base class
// (RTRApplictaionMessage), the handler will not be called
// unless we do it ourselves. If we call our base class Dispatch
// method the handler methods OnInitialize() and
// OnApplictionMessage() will be called. This sample uses
// OnInitialize() to print out notification that a new
// transaction
// is starting.
    RTRApplicationMessage::Dispatch();
// Process the purchase which the derived object represents
    abc_status status = ProcessOrder();
return status;
}
```

## 6.5. Data Class ABCBook

```
// ABCBook.cpp: implementation of the ABCBook class.
//
//////////////////////////////////////////////////////////////////////

#include "ABCCommon.h"
#include "ABCBook.h"

//////////////////////////////////////////////////////////////////////
// Construction/Destruction
//////////////////////////////////////////////////////////////////////

ABCBook::ABCBook() : m_uiISBN(0)
{

}
```

```
   ABCBook::~ABCBook()
   {

   }


   bool ABCBook::AddOrder( unsigned int uiPrice,
                           unsigned int uiISBN,
                           char *pszName,
                           char *pszAuthor)
   {
   //  Copy the Book purchase to our Book object.
       m_uiISBN = uiISBN;
       m_uiPrice = uiPrice;
       strcpy(&m_szTitle[0],pszName);
       strcpy(&m_szAuthor[0],pszAuthor);
       WriteObject();
       return true;
   }
void ABCBook::WriteObject()
{
// Save the type of object we are. This is used by the
// class factory on the server side to determine which type
// of class to allocate.
*this << ABC_BOOK;
*this << m_uiPrice << m_uiISBN  << m_szTitle << m_szAuthor;
// The 1 line call above is equivalent to the 4 lines below. We
// can use the << and >> operators because we know that the data
   // which we store is not > the current RTR maximum  = 65535 byes.
   // WriteToStream(m_uiISBN);
   // WriteToStream(m_uiPrice);
   // WriteToStream(m_szTitle);
   // WriteToStream(m_szAuthor);
       char mystring[] = "ABCDEFGHIJKLMNOPQRSTUVWZYZ";
       rtr_msgbuf_t p = &mystring[0];
       rtr_msglen_t length = strlen(mystring)+1;
       WriteToStream(p,length);
   }
   void ABCBook::ReadObject()
   {
   // The first data is the type of class we should be.
   // Validate that everything is fine.
       unsigned int uiClassType = 0;
       *this >> uiClassType;
       assert(uiClassType == ABC_BOOK);
   //  Populate this object with the data
       *this >> m_uiPrice >> m_uiISBN  >> m_szTitle >> m_szAuthor;
   // The 1 line call above is equivilant to the 4 lines below.
   // ReadFromStream(m_uiISBN);
   // ReadFromStream(m_uiPrice);
   // ReadFromStream(m_szTitle,GetLogicalBufferLength());
   // ReadFromStream(m_szAuthor,GetLogicalBufferLength());
   }
   abc_status ABCBook::ProcessOrder()
   {
       // It is here that we would process the request for this book.
       // For this sample simply print out the Book order.
   cout <<"ABCBook::ProcessOrder()" << endl;
```

```
        cout << "     " << "ISBN = " << m_uiISBN << endl;
        cout << "     " << "Price = " << m_uiPrice << endl;
        cout << "     " << "Title = " << m_szTitle << endl;
        cout << "     " << "Author = " << m_szAuthor << endl;

        return ABCOrderSucceeded;
    }
}
```