VMS Software

VSI OpenVMS

VSI Reliable Transaction Router C Application Programmer's Reference Manual

Document Number: DO-RTRREF-01A

Publication Date: May 2024

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 Application Programmer's Reference Manual

Legal Notice

Preface

This manual explains how to design and code applications for VSI Reliable Transaction Router (RTR) using the C programming language. It contains full descriptions of the RTR C application programming interface (API) calls, and includes a short tutorial.

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. Intended Audience

This manual is the reference source for persons writing application programs using Reliable Transaction Router (RTR) in the C programming language. It completely describes the RTR C application programming interface (API).

3. Document Structure

This manual contains four chapters and two appendices:

- Chapter 1 introduces the RTR C programming interface.
- Chapter 2 provides a guide to writing RTR applications.
- Chapter 3 describes the RTR C Application Programming Interface (API) showing the syntax and data structures for each RTR call.
- Chapter 4 describes how to compile and link your application.
- Appendix A provides two short RTR C API sample applications and their shared and header files.
- Appendix B provides a short tutorial for the application programmer.

4. Related Documents

The table below describes RTR documents and groups them by audience.

Table 1. RTR Documents

Document	**Content**
For all users:	
VSI Reliable Transaction Router Release Notes	Describes new features, corrections, restrictions, and known problems for RTR.
VSI Reliable Transaction Router Getting Started	Provides an overview of RTR technology and solutions, and includes the glossary that defines all RTR terms.

Document	Content
VSI Reliable Transaction Router Software Product Description	Describes product features.
For the system manager:	
VSI Reliable Transaction Router Installation Guide	Describes how to install RTR on all supported platforms.
VSI Reliable Transaction Router System Manager's Manual	Describes how to configure, manage, and monitor RTR.
For the application programmer:	
VSI Reliable Transaction Router Application Design Guide	Describes how to design application programs for use with RTR, with both C++ and C interfaces.
JRTR Getting Started	Provides an overview of the object-oriented JRTR Toolkit including installation, configuration and Java programming concepts, with links to additional online documentation.
VSI Reliable Transaction Router C++ Foundation Classes	Describes the object-oriented C++ interface that can be used to implement RTR object-oriented applications.
VSI Reliable Transaction Router C Application Programmer's Reference Manual	Explains how to design and code RTR applications using the C programming language and the RTR C API. Contains full descriptions of the basic RTR API calls.

5. 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.

6. OpenVMS Documentation

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

7. 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:

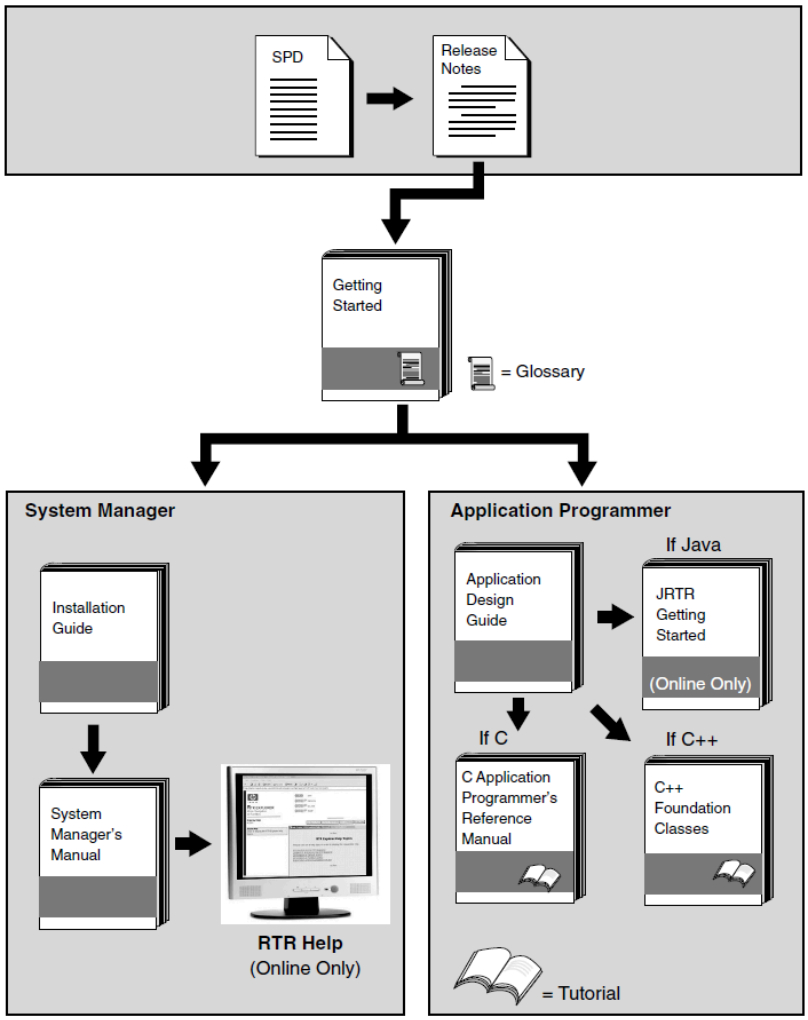
Convention	Meaning
Ctrl/ *x*	A sequence such as **Ctrl/** *x* indicates that you must hold down the key labeled Ctrl while you press another key or a pointing device button.
PF1 *x*	A sequence such as PF1 *x* indicates that you must first press and release the key labeled PF1 and then press and release 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: • Additional optional arguments in a statement have been omitted. • The preceding item or items can be repeated one or more times. • Additional parameters, values, or other information can be entered.
. . .	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. 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.

8. Reading Path

The reading path to follow when using the Reliable Transaction Router information set is shown in the image below.

Figure 1. RTR Reading Path



Chapter 1. Introduction

This chapter introduces the Reliable Transaction Router C programming interface. This interface was formerly called the Portable API. RTR concepts and terms are fully defined in *VSI Reliable Transaction Router Getting Started*.

1.1. RTR Application Programming Interface

The RTR C application programming interface (API) that is provided with Reliable Transaction Router is identical on all hardware and operating system platforms that support RTR. This API is described in the following chapter.

In addition, a web browser and a command line interface (CLI) to the C API are available. The CLI enables you to write simple RTR applications for testing. The RTR CLI is illustrated in *VSI Reliable Transaction Router Getting Started* and fully described in the *VSI Reliable Transaction Router System Manager's Manual*.

1.2. C Programming and RTR APIs

The C-programming RTR API was made available in Reliable Transaction Router Version 3. It superseded the OpenVMS API used in Reliable Transaction Router Version 2 for new applications. The RTR C API is available on all platforms on which RTR is supported.

1.2.1. Compatibility Between RTR Versions

Reliable Transaction Router Version 5 interoperates with RTR Version 4 in a DECnet environment using DECnet Phase IV naming. (The same version of RTR must be installed on all routers and backends. See the section on Network Transports in the *VSI Reliable Transaction Router System Manager's Manual* to find out how to configure your Version 4 nodes.)

Note that the size of an RTR transaction ID was changed in Reliable Transaction Router Version 3 to 28 bytes. (The change ensures that the transaction ID contains a unique node specification.) This remains true for later versions of RTR.

1.2.2. Reasons for a C Programming API

RTR was first developed for use within an OpenVMS environment. Reliable Transaction Router Version 3 extended the applicability of RTR to allow users to create fault-tolerant distributed applications running on networks of heterogeneous machines and platforms.

The OpenVMS API presented some incompatibilities when used on non-OpenVMS platforms as follows:

1. The "$" character contained in all RTR identifiers is not permitted in identifiers in some languages.

2. There was no provision for reformatting user messages passed between machines to account for differing machine representations of particular data types.

3. RTR permits applications to be written to perform multiple concurrent operations, a feature that can be critical for good performance in high-volume transaction processing systems. The notification mechanisms used to indicate completion of such asynchronous operations (event-flag, txsb, completion-AST) were OpenVMS-specific.

1.2.3. Benefits of the C Programming API

The benefits of using the C programming API are:

- Portability over a wide range of language and machine environments
- Simplified handling of concurrency, independent of the type of operating system
- Support for communication between machines with different hardware representations of common data types (little-endian and big-endian, and so forth)
- Interoperability with existing applications using the OpenVMS API
- Features extended above those provided by the OpenVMS API
- Improved performance for commonly-used transaction types
- Support for use within a threaded environment

The C programming API has been designed to:

- Avoid the problem of applications dropping threads.
- Simplify the API.
- Schedule concurrent application operations in the same FIFO manner as is used with AST-driven processing on OpenVMS. This avoids the synchronization worries faced by the application writer when working with ASTs.
- Make it impossible for an application to stall by waiting for one operation to complete and hence being unable to respond to some other event.
- Permit a more efficient implementation. RTR does not have to maintain multiple internal queues.

1.2.4. Comparison of OpenVMS and C Programming API Calls

Table 1.1 compares the OpenVMS and C Programming API calls.

Table 1.1. OpenVMS API (V2) and C Programming API (V3) Compared

OpenVMS API	C Programming API
$dcl_tx_prc()	rtr_open_channel()
$start_tx()	rtr_start_tx() [optional]
$commit_tx()	rtr_accept_tx()
$abort_tx()	rtr_reject_tx()
$vote_tx()	rtr_accept_tx()/rtr_reject_tx()
$deq_tx()	rtr_receive_message()
$enq_tx()	rtr_send_to_server()/ rtr_reply_to_client()/ rtr_broadcast_event()
$dcl_tx_prc() (SHUT)	rtr_close_channel()

OpenVMS API	**C Programming API**
$get_txi()	rtr_request_info()
$set_txi()	rtr_set_info()
ASTPRM (on asynch calls)	rtr_set_user_handle()
–	rtr_error_text()
–	rtr_get_tid()
–	rtr_set_wakeup()

Chapter 2. Overview of the C Programming API

The term C programming API is used to describe the RTR application programming interface (API) adopted in Reliable Transaction Router Version 3. This API is available on all platforms on which Reliable Transaction Router is supported. This API was formerly called the Portable API, when first made available on several operating systems.

2.1. Transactional Messages

RTR allows the client and server applications to communicate by entering into a dialogue consisting of an exchange of messages between a client application (the dialogue initiator) and one or more server applications.

Note

In the context of RTR, client and server are always applications.

Each dialogue forms a transaction in which all participants have the opportunity to either accept or reject the whole transaction. When the transaction is complete, all participants are informed of the transaction's completion status: success (`rtr_mt_accepted`) if all participants accepted it, failure (`rtr_mt_rejected`) if any participant rejected it. (For more information on messages, see Section 2.11, RTR Messages.)

2.2. RTR Channels

With RTR, applications can be engaged in several transactions at a time.

To support many in-progress transactions at the same time, RTR lets applications open multiple channels. An application opens one or more channels to RTR, and any transaction is associated with only one channel. The transaction is said to be active on that channel. For example, a client application opens a channel and then sends the first message of a transaction on that channel. All messages sent and received for that transaction are now associated with that channel.

While waiting for a response from the server, the client application can open a second channel and start a new transaction on it. When the transaction on the first channel has completed, the client application may start the next transaction on it, or simply issue the `rtr_accept_tx` call.

Similarly, a server application may open several channels and, when the first message of a new transaction arrives, RTR delivers it on the first available channel. That channel remains associated with the transaction until it completes.

An application opens a channel before it can send or receive messages; the RTR API call is used to do this. The RTR call specifies whether the channel is a client channel or a server channel; it cannot be both. (This restriction helps to simplify application structure, and to deal with the special properties of each channel type.) A single application can, however, open client channels and server channels.

2.3. Broadcast Messages and Events

In addition to transactional messages, client or server programs may broadcast event messages. These are delivered to some subset of the distributed applications, as specified by the event-number and event-

name parameters. In contrast to transactional dialogues, no completion status is subsequently returned to the initiator. A message can be from 0 to 64K bytes long.

Both client and server channels receive messages from RTR. A client channel receives event messages only from servers, and a server channel receives event messages only from clients. To enable a client application to receive event/broadcast messages from another client application, the application must be both a client and a server application (open a channel with both CLIENT and SERVER flags), and must be in a facility on a node that is both a frontend and a backend. A broadcast event can be sent as long as the server channel is open. Events are more fully described in Section 2.12: RTR Events.

2.4. C Programming API Calls

The C Programming API calls are shown in Table 2.1: C Programming API Calls. Each call is shown with a brief description and whether it can be used on client channels or server channels or both. Calls are listed in alphabetical order.

Table 2.1. C Programming API Calls

RTR Call	Description	Channel Use
rtr_accept_tx	Accepts a transaction	Client and server
rtr_broadcast_event	Broadcasts (sends) an event message	Client and server
rtr_close_channel	Closes a previously opened channel	Client and server
rtr_error_text	Gets the text for an RTR status number	Client and server
rtr_ext_broadcast_event	Broadcasts (sends) an event message with a timeout	Client and server
rtr_get_tid	Gets the current transaction ID	Client and server
rtr_get_user_context	Gets the user-defined context associated with a channel	Client and server
rtr_open_channel	Opens a channel for sending and receiving messages	Client and server
rtr_receive_message	Receives the next message (transaction message, event or completion status)	Client and server
rtr_reject_tx	Rejects a transaction	Client and server
rtr_reply_to_client	Sends a response from a server to a client	Server only
rtr_request_info	Requests information from RTR	Client and server
rtr_send_to_server	Sends a message from a client to the server(s)	Client only
rtr_set_info	Sets an RTR parameter	Client and server
rtr_set_user_context	Sets the value of the user-defined context for a channel	Client and server
rtr_set_user_handle	Associates a user value with a transaction	Client and server

RTR Call	Description	Channel Use
rtr_set_wakeup	Sets a function to be called on message arrival	Client and server
rtr_start_tx	Explicitly starts a transaction	Client only

2.5. Programming Examples

The following pseudocode examples of a client and a server application illustrate the use of the C programming API. Details have been omitted to keep the basic structure clear.

2.5.1. Simple Client

This simple client program issues transactions and receives event messages. It simply issues one transaction, waits for it to be processed, and in the meantime handles any events that arrive. It then issues the next transaction. It does not need to wait until one transaction finishes before starting the next.

The following two examples are single-threaded. They can be made multithreaded by opening more channels. The structure of the main receive loop does not need to be changed to implement this. Note that `rtr_receive_message` receives the next message in the process input queue for *any* of the channels opened by the program (unless preferred channels have been requested in the `rtr_receive_message`).

Example 2.1. Example Client

```
    rtr_open_channel()        ! Open a channel to the required facility
    rtr_receive_message()     ! Get the completion status of the open call
                              ! success returns rtr_mt_opened
send_loop:
    rtr_send_to_server( ...RTR_F_SEN_ACCEPT....)
                              ! Send a tx-message and
                              ! implicitly start a new tx rcv_loop:
    rtr_receive_message()     ! Find out what RTR wants to tell us next
    switch  (message_received_type)
    {
  case    rtr_mt_reply:          Process_Reply_from_Server; break;
        case    rtr_mt_rtr_event:      Process_RTR_Event;  break;
        case    rtr_mt_user_event:     Process_User_Event; break;
        case    rtr_mt_accepted:       Tell_User_It_Worked; break;
        case    rtr_mt_rejected:       Tell_User_About_Failure; break;
     }
     IF ( message_received_type = rtr_mt_accepted )
     OR ( message_received_type = rtr_mt_rejected )
     THEN
         GOTO send_loop ! Last transaction done, issue the next one
     ELSE
         GOTO rcv_loop  ! Get the next incoming message
```

In Example 2.1, note that the switch statement tests on message type. All messages that are received from RTR have a message type; for further information, see Section 2.11.

2.5.2. Simple Server

Example 2.2 is a simple server that receives transactions and events.

Example 2.2. Example Server

```
    rtr_open_channel()         ! open a channel to the desired facility
    rtr_receive_message()      ! get the completion status of the open call
                               ! success returns rtr_mt_opened
rcv_loop:
    rtr_receive_message()      ! Find out what RTR wants to tell us next

    CASE message_received_type
    OF
        rtr_mt_msg1:          Do_Some_SQL_And_Maybe_Send_A_Reply;
        rtr_mt_msgn:          Do_Some_More_SQL_And_Maybe_Send_A_Reply;
        rtr_mt_prepare:       Accept_or_Reject_Tx ;
        rtr_mt_rtr_event:     Process_RTR_Event;
        rtr_mt_user_event:    Process_User_Event;
        rtr_mt_accepted:      Commit_DB ;
        rtr_mt_rejected:      Rollback_DB ;
    END_CASE;
     GOTO rcv_loop
```

2.6. Using the C Programming API

As can be seen from the examples in the previous section, an application first opens one or more channels by calling `rtr_open_channel`.

The application can then process transactions and events on the channels it has opened. When a channel is no longer needed, the application closes it by calling `rtr_close_channel`.

A transaction becomes associated with a channel in one of the following circumstances:

1. When a client issues the first `rtr_send_to_server` call on a previously idle channel
2. When a server receives from a client the first message belonging to a transaction by calling `rtr_receive_message`.
3. When a client issues a `rtr_start_tx` call on a previously idle channel.

From this point on the channel remains associated with the transaction until one of the following occurs:

a. The application rejects the transaction using `rtr_reject_tx`.

   - The transaction is over, no more messages will be received on behalf of this transaction.
   - The channel becomes idle, ready for initiation/reception of another transaction.

b. The application accepts the transaction using `rtr_accept_tx`.

   - After calling `rtr_accept_tx` the application may continue receiving messages belonging to the transaction. However, it cannot subsequently either reverse its decision to accept by calling `rtr_reject_tx`, or (in the case of a client application) make additional calls to `rtr_send_to_server`.
   - The final message received for a transaction will always be a transaction completion status; either `rtr_mt_rejected` or `rtr_mt_accepted`.
   - The channel becomes idle, ready for initiation or reception of another transaction.

c. The application receives, by a call to `rtr_receive_message`, a completion status indicating that the transaction has been rejected by some other participant.

   - The transaction is over. No more messages will be received, and no more calls may be made on behalf of this transaction.
   - The channel becomes idle, ready for initiation or reception of another transaction.

Note that RTR considers a transaction to have been committed to the database (so that it does not need to replay it in case of failure) when the server indicates willingness to receive a new transaction by calling `rtr_receive_message` on the channel, after having received the transaction completion status.

Calling `rtr_close_channel` also indicates to RTR that the last transaction has been committed.

2.7. Concurrency

The routine `rtr_receive_message` is used by an application to receive *all* incoming messages, responses and events. This provides a single consistent method of information delivery.

All RTR routines other than `rtr_receive_message` complete immediately, and any responses are queued for later reception by `rtr_receive_message`.

The application calling `rtr_receive_message` may choose whether (and how long) it should wait for an incoming message to arrive (if there is no message available for immediate reception).

In addition, the application may optionally specify a “wakeup routine” to be called by RTR when a message becomes available for reception.

2.8. Exit Handlers in Applications

Making RTR calls from within an application exit handler does not work, because the channel is usually closed by the time the application exits. If an exit handler contains a call to RTR, then the exit handler must be declared after the first call to RTR. If an exit handler is declared before the first call to RTR, then any call to RTR made within the exit handler will return an error.

The error status returned is `RTR_STS_INV_CHANNEL`.

2.9. Using the RTR Set Wakeup Routine

An application program may typically wish to respond to input from more than one source. An example of this is an application program that prompts for user input in a window and at the same time displays information received asynchronously via broadcast events.

To avoid the application polling its various input sources, RTR provides the `rtr_set_wakeup` routine. This allows the application to specify a routine to be called when there is data to be received from RTR. The application program can then be coded as shown in the example provided with the `rtr_set_wakeup` routine.

The processing context of the application wakeup handler depends upon the platform and RTR library variant employed.

Core RTR functionality and the C API are delivered in a single sharable library. This library is named `rtrdll` on Windows, and `librtr` on other platforms. The latter is supplied in two variants:

`librtr_r` which is targeted at developers of threaded applications, and `librtr` which provides a platform-specific wakeup handler implementation.

Wakeup handlers under `rtrdll` and `librtr_r` are called in a dedicated thread created by RTR for this purpose.

Wakeup handlers under `librtr` on UNIX are called from a signal handler established by RTR to handle SIGIO. If the application also wishes to use this signal, it should establish its handler prior to the first call to the RTR API. In this case the signal handler should be aware that the SIGIO signal may have been generated by RTR, not necessarily by the event for which the signal handler was written.

Wakeup handlers under `librtr` on OpenVMS are called from an AST handler. In the presence of multiple competing ASTs, calling `rtr_set_wakeup()` from the wakeup handler can be used to limit RTR processing and serialize the execution of RTR events with other asynchronous activity in the program.

`Rtrdll` and `librtr_r` provide thread synchronization and are safe to use in a multithreaded environment. `Librtr` offers no such protection.

It is not anticipated that applications on OpenVMS will want to use both threads and ASTs. For this reason the RTR V2 API is functional in `librtr` on OpenVMS only.

Summarizing:

Sharable Name	**Thread-safe**	**Wakeup Mechanism**	**V2 API**
`rtrdll`	Yes	RTR thread	No
`librtr_r`	Yes	RTR thread	No
`librtr` /UNIX	No	signal handler	No
`librtr` /OpenVMS	No	AST	Yes

2.9.1. Restrictions on the RTR Wakeup Handler

The wakeup handler itself cannot call any function that might have to wait such as `rtr_reply_to_client`, `rtr_send_to_server` or `rtr_broadcast_event`; the only RTR call allowed in the wakeup handler is `rtr_receive_message` called with a zero timeout. Other RTR calls may block or halt processing when they need transaction IDs or flow control, which will cause unexpected behavior. This restriction applies to both threaded and unthreaded applications.

A threaded application does not need to use a wakeup handler; its functionality can be provided by a dedicated thread that receives and dispatches RTR messages.

Functions permitted in an `rtr_set_wakeup()` handler:

- While wakeups are unnecessary in threaded application, they may be used in common code in applications that run on OpenVMS. Because mainline code continues to run while the wakeup is executing, extra synchronization may be required. If the wakeup does block then it does not generally hang the whole application.
- For an RTR wakeup handler in a signal handler within an unthreaded UNIX application, no RTR API functions and only the very few asynch-safe system and library functions may be called, because

the wakeup is performed in a signal handler context. An application can write to a pipe or access a volatile `sig_atomic_t` variable, but using `malloc()` and `printf()`, for example, will cause unexpected failures. Alternatively, on most UNIX platforms, you can compile and link the application as a threaded application with the reentrant RTR shared library `-lrtr_r`.

- For maximum portability, the wakeup handler should do the minimum necessary to wake up the mainline event loop. You should assume that mainline code and other threads might continue to run in parallel with the wakeup, especially on machines with more than one CPU.
- The `rtr_set_wakeup()` call may return the errors ACPNOTVIA and NOACP if the RTRACP process is not running. However, these errors will only be returned once before an application succeeds in opening a channel. Subsequent calls will succeed and install the specified handler. Applications wishing to poll for the availability of the ACP should use the `rtr_open_channel()` call.

2.10. API Optimizations

Reliable Transaction Router provides client and server optimizations for greater performance and programming ease.

2.10.1. Client Optimization

Reliable Transaction Router introduces greater flexibility and efficiency in how transactions are packaged at the client.

The total sequence of events that a client application has to execute are as follows:

1. Start a transaction.
2. Send one or more transaction messages, optionally receive one or more transaction messages.
3. Either accept or reject the transaction.
4. Wait for the transaction accept or reject message and process accordingly.
5. Return to Step 1.

In Reliable Transaction Router, all these steps can be followed if required, but optimizations allow some of the steps to be handled implicitly.

- The call to `rtr_start_tx` (Step 1) may be omitted if, for example, no timeout is required for the transaction. A call to `rtr_send_to_server` on a channel that does not have an active transaction automatically implies a call to `rtr_start_tx`.
- Step 3 may be handled implicitly **if the client wishes to accept the transaction**. This is done by setting the `RTR_F_SEN_ACCEPT` flag on the last (or only) call to `rtr_send_to_server`.

2.10.2. Voting Optimization and Server Flags

Reliable Transaction Router introduces greater flexibility and efficiency in how transaction voting (acceptance by servers) is handled; RTR allows implicit voting.

In detail, the sequence of events that a server executes is as follows:

1. Get one or more transaction messages from RTR and process them.

2. Get the vote request message from RTR.

3. Issue the accept (commit).

4. Get the final transaction state.

5. Return to Step 1.

This scheme is not efficient in some cases. For example, a callout (authentication) server may only need to receive the first message of a multiple message transaction, whereupon it can vote immediately.

In Reliable Transaction Router, all these steps can be enforced if required, but optimizations allow some of the steps to be handled implicitly.

An implicit accept allows Step 3 to be omitted; the transaction is accepted by the server when it does the next call to `rtr_receive_message`.

These optimizations are controlled by flags (`RTR_F_OPE_EXPLICIT_PREPARE` and `RTR_F_OPE_EXPLICIT_ACCEPT`) on the call used to open a server channel.

2.10.2.1. The RTR_F_OPE_EXPLICIT_PREPARE Flag

A server channel may be opened with the `RTR_F_OPE_EXPLICIT_PREPARE` flag; this specifies that it will receive prepare messages (messages of type `rtr_mt_prepare`). The server is then expected to accept or reject a transaction on receipt of this message (or earlier). The server may accept the transaction before the prepare message is sent: in this case, the prepare message is not delivered to the server.

The default behaviour of RTR (for example, when this flag is **not** set in the call to `rtr_open_channel`) is to not send prepare messages to the server application. In this case, RTR expects the server to accept or reject transactions without RTR triggering it into voting activity by sending prepare messages.

2.10.2.2. The RTR_F_OPE_EXPLICIT_ACCEPT Flag

A server channel may be opened with the `RTR_F_OPE_EXPLICIT_ACCEPT` flag; this specifies that it will accept transactions **only** by making an explicit call to `rtr_accept_tx`.

The default behaviour of RTR (that is, when this flag is **not** set) is to treat a server's call to `rtr_receive_message` (after the last transaction message has been received) as an implicit acceptance of the transaction.

If a transaction has been accepted before the last message has been received, the setting of the `RTR_F_OPE_EXPLICIT_ACCEPT` is irrelevant.

However, if a transaction has not been prematurely accepted, when the server's vote is required by RTR, the setting of the flags have the following effects:

1. When both `RTR_F_OPE_EXPLICIT_PREPARE` and `RTR_F_OPE_EXPLICIT_ACCEPT` are set, the **`rtr_mt_prepare message`** is returned to the server, and the server must accept or reject the transaction.

2. When RTR_F_OPE_EXPLICIT_PREPARE is set but RTR_F_OPE_EXPLICIT_ACCEPT is not set, the **rtr_mt_prepare** message is also returned to the server, but if the server does not perform an explicit accept or reject, then a subsequent call to rtr_receive_message implies an accept of the transaction.

3. When RTR_F_OPE_EXPLICIT_PREPARE is not set but RTR_F_OPE_EXPLICIT_ACCEPT is set, no **rtr_mt_prepare** message is returned to the server, and no implicit accept of the transaction will be performed: It is assumed that some other event will trigger the application into voting.

4. With neither RTR_F_OPE_EXPLICIT_PREPARE nor RTR_F_OPE_EXPLICIT_ACCEPT set, no **rtr_mt_prepare** message is returned to the server. An implicit transaction accept is performed.

2.11. RTR Messages

All RTR calls return a completion status immediately *except* rtr_receive_message. If the immediate status is successful, many calls will also result in a further message or messages being delivered on the channel.

All RTR received messages are of a defined message type. The message type is given in the message status block. (See *pmsgsb* on rtr_receive_message in Chapter 3).

The message type allows your application to handle the message appropriately; the message type indicates whether this message contains information that is part of a transaction, or a broadcast, or RTR informational, and so on.

The use of rtr_receive_message for both RTR status messages *and* application data messages requires the application designer to consider how to respond to the different message types. Message types for server and client applications are listed in Table 2.2 and Table 2.3.

All received messages cause the message status block (*pmsgsb* on rtr_receive_message) to be filled; most message types also put data into the user buffer (*pmsg* on rtr_receive_message). Only the **rtr_mt_prepare** message type does not fill the user buffer.

Table 2.4 provides information put in the user buffer for each message type. Table 2.2 and Table 2.3 list all the message types that server channels or client channels can receive, together with a description of their meaning and the recommended application behavior. Order is alphabetical.

Table 2.2. RTR Received Message Types for Server Applications

Message Type	Description	Recommended Action
rtr_mt_accepted	The specified transaction has been accepted by all participants.	Commit the transaction in the database and release database locks.
rtr_mt_closed	Channel has been closed. Sent by RTR if an rtr_open_channel fails (that is, no such facility) or as a result of an operator command such as *DELETE FACILITY*, or the last message from a	Examine reason status. Roll back database for any active transaction.

Message Type	Description	Recommended Action
	`rtr_request_info` or `rtr_set_info` call.	
rtr_mt_msg1	This is the first application message of a transaction, sent by a client.	Process the message.
rtr_mt_msg1_uncertain	This is the first application message of a replayed transaction, that is, a previous incarnation of the server failed during the voting phase.	Check in database to see if the transaction has been processed. If not processed, redo the transaction; else forget the transaction.
rtr_mt_msgn	This is the *n*th application message (that is, not the first) of a transaction, sent by a client.	Process the message.
rtr_mt_opened	Channel has been opened.	Use the channel.
rtr_mt_prepare	The specified transaction is complete (that is, all messages from the client have been received). This message type is only received by a server that specified that it requires a prepare. (Servers specify this by using the `RTR_F_OPE_EXPLICIT_` `_PREPARE` flag on the `rtr_open_channel` call.)	Call either `rtr_reject_tx` to reject the transaction, or have all required database records locked before calling `rtr_accept_tx` to accept the transaction.
rtr_mt_rejected	The specified transaction has been rejected by a participant.	Roll back the transaction.
rtr_mt_request_info	Message from a previous call to `rtr_request_info`.	Use information as required.
rtr_mt_rtr_event	An RTR event with an associated message.	*evtnum* describes which RTR event occurred. See Table 2.5.
rtr_mt_set_info	Message from a previous call to `rtr_set_info`.	Use information as required.
rtr_mt_user_event	A user event with an associated message.	*evtnum* has an application-specific meaning.

Table 2.3. RTR Received Message Types for Client Applications

Message Type	Description	Recommended Action
rtr_mt_accepted	The specified transaction has been accepted by all participants.	Inform user of successful completion.
rtr_mt_closed	Channel has been closed. Sent by RTR if an `rtr_open_channel` fails (for example, no such facility) or as a result of an operator command such as *DELETE*	Examine reason status.

Message Type	Description	Recommended Action
	FACILITY, or the last message from an `rtr_request_info` or `rtr_set_info` call.	
rtr_mt_opened	Channel has been opened.	Use the channel.
rtr_mt_rejected	The specified transaction has been rejected by a participant.	Inform user of reason for failure.
rtr_mt_reply	This is an application reply message sent by a server.	Process message.
rtr_mt_request_info	Message from a previous call to `rtr_request_info`.	Use information as required.
rtr_mt_rettosend	This message (which had been sent with the `RTR_F_SEN_RETURN_TO_ _SENDER` flag) could not be delivered and has been returned.	Take appropriate action for the transaction as required by your application.
rtr_mt_rtr_event	An RTR event with an associated message.	*evtnum* describes which RTR event occurred. See Table 2.5.
rtr_mt_set_info	Message from a previous call to `rtr_set_info`.	Use information as required.
rtr_mt_user_event	A user event with an associated message.	*evtnum* has an application-specific meaning.

Table 2.4. Contents of the User Buffer for Different Message Types

Message Type	Buffer Contents
rtr_mt_accepted	**`rtr_status_data_t`**, see Example 2.3.
rtr_mt_closed	**`rtr_status_data_t`**, see Example 2.3.
rtr_mt_msg1	The first application message of a transaction, sent by a client.
rtr_mt_msg1_uncertain	The first application message of a replayed transaction.
rtr_mt_msgn	The *n*th application message (that is, not the first) of a transaction, sent by a client.
rtr_mt_opened	**`rtr_status_data_t`**, see Example 2.3.
rtr_mt_prepare	None.
rtr_mt_rejected	**`rtr_status_data_t`**, see Example 2.3.
rtr_mt_reply	An application reply message sent by a server.
rtr_mt_request_info	Requested information from `rtr_request_info`.
rtr_mt_rettosend	Returned message.
rtr_mt_rtr_event	RTR event message.
rtr_mt_set_info	Set information from `rtr_set_info`.
rtr_mt_user_event	The user broadcast message.

Example 2.3 shows the data type that is returned in the user buffer with message types **`rtr_mt_accepted`**, **`rtr_mt_rejected`**, **`rtr_mt_opened`** and **`rtr_mt_closed`**. You can find the meaning of `rtr_status_t` using the call `rtr_error_text`.

Example 2.3. Type rtr_status_data_t

```
                                /*  Type returned with rtr_mt_rejected,*/
typedef struct                  /*  rtr_mt_accepted, rtr_mt_opened     */
{                               /*  and rtr_mt_closed messages.        */
    rtr_status_t    status;     /*  RTR status                         */
    rtr_reason_t    reason;     /*  User-supplied reason               */
} rtr_status_data_t;
```

2.12. RTR Events

What are events?

An event in RTR is a trigger that causes a notification (also called a “broadcast”) to be sent to the application that subscribed to the event. **RTR Events** are created only by RTR and are used internally by RTR to help manage activities such as site failover. Application developers may subscribe to RTR Events to activate certain processing in their application. **User Events** are also available to enable application developers to send event notification or broadcast messages to other RTR applications. RTR provides the call `rtr_broadcast_event` to enable an application developer to trigger a User Event.

Events have special characteristics and restrictions:

- Event notification is delivered on a subscription basis using information supplied on the `rtr_open_channel` call.
- Events are not transactional and should not be used to transmit information that is, or will be, part of an RTR transaction.
- A user application can turn on or off the reception of any events, both RTR and user events.
- Events can only be transmitted within the RTR facility in which they are defined. Events cannot be sent between facilities or outside RTR.
- Event notification may include an optional message, which has a size limit of 64K.
- User Events can only be transmitted from frontend-to-backend or from backend-to-frontend. User Events cannot be used for peer-to-peer communication such as from frontend-to-frontend or from backend-to-backend.
- RTR Events are transmitted from RTR-to-frontend or RTR-to-backend.

The list below shows the RTR Events that are available for subscription. These events can be grouped in four basic categories:

- Shadow node activity (failover, failback, recovery complete)
- Standby node activity (become active, become standby, recovery complete)
- Changes in facility state and participants (clients/routers/servers entering or exiting the facility)

- Changes in configuration of partition key ranges (server available, server not available)

2.12.1. RTR Event Names and Numbers

RTR sends events to the server either inside or outside a transactional boundary. A transaction is considered to start on receipt of an `rtr_mt_msg1` or `rtr_mt_msg1_uncertain` message, and to end when the transaction is accepted or rejected (receipt of an `rtr_mt_accepted` or `rtr_mt_rejected` message). Events containing information about primary, secondary, or standby servers could arrive outside a transactional boundary. Gain and loss events arrive inside transactional boundaries.

Table 2.5 lists the RTR events that can be received on a channel (associated with the **`rtr_mt_rtr_event`** message type). Events are listed in order of event number. See the description for `rtr_open_channel` in Chapter 3: *RTR Call Reference*, for further information.

Table 2.5. RTR Event Names and Numbers

Event Name	Event Number	Description
RTR_EVTNUM_FACREADY	96	The facility has become operational.
RTR_EVTNUM_FACDEAD	97	The facility is no longer operational.
RTR_EVTNUM_FERTRGAIN	98	Frontend link to current router established.
RTR_EVTNUM_FERTRLOSS	99	Frontend link to current router lost.
RTR_EVTNUM_RTRBEGAIN	100	Current router established link to a backend.
RTR_EVTNUM_RTRBELOSS	101	Current router lost link to a backend.
RTR_EVTNUM_KEYRANGEGAIN	102	Server(s) for new routing key range are now available.
RTR_EVTNUM_KEYRANGELOSS	103	No more servers remain for a particular routing key range.
RTR_EVTNUM_BERTRGAIN	104	Backend established link to a router.
RTR_EVTNUM_BERTRLOSS	105	Backend lost link to a router.
RTR_EVTNUM_RTRFEGAIN	106	Router established link to a frontend.
RTR_EVTNUM_RTRFELOSS	107	Router lost link to a frontend.
RTR_EVTNUM_SRPRIMARY	108	Server has become primary. [1]
RTR_EVTNUM_SRSTANDBY	109	Server has become standby.
RTR_EVTNUM_SRSECONDARY	110	Server in a shadow pair has become secondary. [1]

Event Name	Event Number	Description
RTR_EVTNUM_SRSHADOWLOST	111	Server in a shadow pair lost its shadow partner. [2]
RTR_EVTNUM_SRSHADOWGAIN	112	Server in a shadow pair acquired a shadow partner.
RTR_EVTNUM_SRRECOVERCMPL	113	Server completed recovery processing.

[1]RTR will generate this event between transactional boundaries. This event can be useful to signal the application to begin activities that should only be performed by the primary system, such as processing credit card debits.

[2]This event signals that this system is entering remember mode for future catchup of the shadow partner.

2.12.2. Developing Applications to Use Events

Subscribing to Events

RTR Events can be used for triggering special application processing based on a change in RTR system status, or for sending notification to the system operator after certain application or RTR conditions that require intervention.

User Events can be used for actions such as broadcasting stock prices to update a price table, or triggering special application processing such as handling a failed transaction. User events can be used to send a message in a one-to-one or a one-to-many method.

Event subscription is established when the `rtr_open_channel` call is executed. See the RTR `rtr_open_channel` call description for details on this call. The `rtr_open_channel` call is as follows:

```
rtr_open_channel (channel,
                  flags,
                  facnam,
                  rcpnam,
                  pevtnum,
                  access,
                  numseg,
                  pkeyseg)
```

Two parameters on the call are used to establish event subscription: *rcpnam* and *pevtnum*.

rcpnam is a pointer to an optional channel name for receiving event messages. If a User Event is sent to a particular channel name, only those subscribers that match both name AND event number are notified. For example, a client channel named “New York” and a client channel named “Hong Kong” could both subscribe to receive User Event number 999. If event 999 was triggered by the server using the channel named “Hong Kong,” the event would be received only by the “Hong Kong” client. Specify *RTR_NO_RCPNAM* for this parameter if a name is not used. This parameter is case sensitive.

pevtnum is a pointer to lists of RTR and User event numbers to which the channel wants to subscribe. These lists use the numeric values of the events shown in Table 2.5. Use the special symbols in Table 2.6 to construct the event list.

Table 2.6. Symbols for Event Lists

Symbol	Description
RTR_NO_PEVTNUM	No events selected.

Symbol	Description
RTR_EVTNUM_USERDEF	Begin User Event list.
RTR_EVTNUM_RTRDEF	Begin RTR Event list.
RTR_EVTNUM_ENDLIST	End of entire list.
RTR_EVTNUM_UP_TO	Specifies an event range in the form *x* RTR_EVTNUM_UP_TO *y*.
RTR_EVTNUM_USERBASE	Smallest User Event number (0).
RTR_EVTNUM_USERMAX	Largest User Event number (250).
RTR_EVTNUM_RTRBASE	Smallest RTR Event number.
RTR_EVTNUM_RTRMAX	Largest RTR Event number.

Example 2.4 illustrates how to set up a list of all User Event numbers for the `rtr_open_channel` call.

Example 2.4. User Event Example

```
rtr_evtnum_t all_user_events[]={
   RTR_EVTNUM_USERDEF,
       RTR_EVTNUM_USERBASE,
       RTR_EVTNUM_UP_TO,
       RTR_EVTNUM_USERMAX,
   RTR_EVTNUM_ENDLIST
   } ;
```

Example 2.5 illustrates how to set up a list of all RTR and User Event numbers for the `rtr_open_channel` call.

Example 2.5. RTR and User Event Example

```
rtr_evtnum_t all_events[]={
   RTR_EVTNUM_USERDEF,
       RTR_EVTNUM_USERBASE,
       RTR_EVTNUM_UP_TO,
       RTR_EVTNUM_USERMAX,
   RTR_EVTNUM_RTRDEF,
       RTR_EVTNUM_RTRBASE,
       RTR_EVTNUM_UP_TO,
       RTR_EVTNUM_RTRMAX,
   RTR_EVTNUM_ENDLIST
   } ;
```

Sending Events

A broadcast event is triggered when the `rtr_broadcast_event` call is executed. See the `rtr_broadcast_event` call description for details on this call. The `rtr_broadcast_event` call syntax is as follows:

```
rtr_broadcast_event (channel,
                     flags,
                     pmsg,
                     msglen,
                     evtnum,
```

```
                      rcpspc,
                      msgfmt)
```

The significant parameters on this call are:

channel is the channel identifier returned from the `rtr_open_channel` call.
pmsg is a pointer to the message to be broadcast.
msglen is the length in bytes of the message.
evtnum is the User Event number that the application developer has assigned to this event.
rcpspc is the optional recipient channel name that can be specified with the *rcpnam* parameter on the `rtr_open_channel` call.

Example 2.6 shows an example of an `rtr_broadcast_event` call.

Example 2.6. Broadcast Event Example

```
if ( bServerShutdown )
{ sts = rtr_broadcast_event (
     /* channel */ BY_CHAN_CLIENT(cCurrentChannel,client)->chan,
     /* flags   */ RTR_NO_FLAGS,
     /* pmsg    */ &msgbuf,
     /* msglen  */ cbTotalSize,
     /* evtnum  */ USER_EVT_SHUTDOWN,
     /* rcpnam  */ "*",
     /* msgfmt  */ szMsgFmt);
exit_if_error ( "rtr_broadcast_event", sts );
}
```

Receiving Events

Any RTR transaction, RTR Event, or User Event can be received when the application executes the `rtr_receive_message` call. See the RTR `rtr_receive_message` call description for details on this call. The `rtr_receive_message` call syntax is as follows:

```
 rtr_receive_message (channel,
                      flags,
                      prcvchan,
                      pmsg,
                      maxlen,
                      timoutms,
                      pmsgsb)
```

The significant parameters on this call are:

channel is the channel on which the message is received.
pmsg is a pointer to an application buffer where the message is written.
maxlen is the maximum length of the application buffer in bytes.
pmsgsb is a pointer to a *message status block* describing the received message.

Notification of Events

If the application has subscribed to events, any call to `rtr_receive_message` can return an event notification, either an RTR Event notification or a User Event notification. The results are described in Table 2.7.

Table 2.7. Event Notifications

If this notification is delivered:	the `rtr_receive_message` call returns a message of type:	and the user/application buffer contains the associated:
RTR Event	`rtr_mt_rtr_event`	event message
User Event	`rtr_mt_user_event`	user broadcast message

When RTR receives a role-gain or role-loss event, it provides both the facility name and the nodename of the node (FE, TR, or BE) that sent the event notification. Only events for roles (FE, TR, BE) provide this additional information. For a definition of roles in RTR, see the *VSI Reliable Transaction Router Getting Started* manual and the RTR Glossary. In RTR, only facilities have roles. Example 2.7 shows the results of a frontend gain event (FEGAIN, event 106) and a frontend loss event (FELOSS, event 107).

Example 2.7. Frontend Gain and Loss Examples

```
RTR> call rece
%RTR-S-OK, normal successful completion
 channel name:  RTR$DEFAULT_CHANNEL
 msgsb
   msgtype:     rtr_mt_rtr_event
   msglen:      34
   evtnum:      106     (RTR_EVTNUM_RTRFEGAIN)
 message
   facility:    RTR$DEFAULT_FACILITY
       link:    nodename
RTR> call rece
%RTR-S-OK, normal successful completion
 channel name:  RTR$DEFAULT_CHANNEL
 msgsb
   msgtype:     rtr_mt_rtr_event
   msglen:      34
   evtnum:      107     (RTR_EVTNUM_RTRFELOSS)
 message
   facility:    RTR$DEFAULT_FACILITY
       link:    nodename
```

Returned Event Data

Two RTR Events return key range data to the application:

Table 2.8. Events that Return Key Range Data

Event Name	Event Number
RTR_EVTNUM_KEYRANGEGAIN	102
RTR_EVTNUM_KEYRANGELOSS	103

The key range data are received in the message returned to the application, with the length of the message specified in the message status block (`msgsb`). For example, the following illustrates `rtr_receive_message` usage.

```
rtr_status_t
rtr_receive_message (
  rtr_channel_t      *pchannel,
```

```
  rtr_rcv_flag_t     flags,
  rtr_channel_t      *p_rcvchan,
  rtr_msgbuf_t       pmsg,
  rtr_msglen_t       maxlen,
  rtr_timout_t       timoutms,
  rtr_msgsb_t        *p_msgsb
  )
```

The message status block pointed to by `*p_msgsb` has the following structure:

```
   typedef struct {
     rtr_msg_type_t   msgtype;
     rtr_usrhdl_t     usrhdl;
     rtr_msglen_t     msglen;
     rtr_tid_t        tid;
     rtr_evtnum_t     evtnum;
   }rtr_msgsb_t;
```

When an event number is 102 or 103, RTR returns key range data (the low and high bounds) in the message, padded as required for data marshalling and interoperability. The key range data can be examined by the application. For more detail on data marshalling and formatting, see Section 2.14.

Bounds data are treated as if defined as a structure. For example, if there are two key segments defined as `rtr_uns_8_t` and `rtr_uns_32_t`, then the bounds data are copied to outbuf as if they were contained in the structure; that is, the 32-bit *int*s are correctly aligned in the structure and the structure size is a multiple of four. For example,

```
       struct{
         rtr_uns_8_t  low_bound_1;
         rtr_uns_32_t low_bound_2;
         rtr_uns_8_t  hi_bound_1;
         rtr_uns_32_t hi_bound_2;
       }
```

The “four-byte-alignment-fits-all” requirement is enforced for interoperability; no padding is allowed.

Example 2.8, which can be run manually from the RTR CLI, illustrates the return of key range data with the RTR Event `RTR_EVTNUM_KEYRANGELOSS`. The RTR CLI interprets the format of this message as appropriate. In Example 2.8, the format is string or ASCII data, the default.

Example 2.8. Returned Event Key Range Data Example

```
RTR> crea fac jws/all=sucre
%RTR-S-FACCREATED, facility jws created
RTR> crea part ab/fac=jws/noshadow/nostandby-
 /key1=(type=string,length=2,offset=0,low="AB",high="CD")
%RTR-I-PRTCREATE, partition created
RTR> rtr_open/chan=s/server/noshadow/nostandby/part=ab/fac=jws
%RTR-S-OK, normal successful completion
RTR> rtr_rec/chan=s/time=10
%RTR-S-OK, normal successful completion
 channel name:  S
 msgsb
   msgtype:     rtr_mt_opened
   msglen:      8
 message
   status:      normal successful completion
   reason:      0x00000000
```

```
RTR> rtr_open/chan=c/client/event=(102,103)/fac=jws
%RTR-S-OK, normal successful completion
RTR> rtr_rec/chan=c/time=10
%RTR-S-OK, normal successful completion
 channel name:  C
 msgsb
   msgtype:     rtr_mt_opened
   msglen:      8
 message
   status:      normal successful completion
   reason:      0x00000000
RTR> rtr_close/chan=s
%RTR-S-OK, normal successful completion
RTR> rtr_rec/chan=c/time=10
%RTR-S-OK, normal successful completion
 channel name:  C
 msgsb
   msgtype:     rtr_mt_rtr_event
   msglen:      4
   evtnum:      103     (RTR_EVTNUM_KEYRANGELOSS)
 message
   ks_lo_bound: AB
   ks_hi_bound: CD
RTR> reca
RTR> rtr_rec/chan=c/time=10
%RTR-E-TIMOUT, call to rtr_receive_message timed out
```

Design consideration: When an RTR application executes an `rtr_receive_message` call, the programmer could incorrectly anticipate that a particular message type may be received and only write instructions to respond to the expected message. However, an RTR or User Event could be received on any instance of the `rtr_receive_message` call (as could other unanticipated RTR messages). Therefore, as a general application design guideline, the application developer should always program the application so that it can properly handle any type of message that could be received by the `rtr_receive_message` call.

Events are delivered in the order in which they are broadcast; therefore event serialization will be preserved for a particular user. However, RTR does not enforce any particular serialization across different subscribers, so different subscribers could receive event notifications in any order.

Example 2.9 shows an `rtr_receive_message` call in use.

Example 2.9. Receive Message Example

```
status = rtr_receive_message( &channel,
                              RTR_NO_FLAGS,
                              RTR_ANYCHAN,
                              &receive_msg,
                              RTR_ANYCHAN,
                              &receive_msg,
                              sizeof(receive_msg),
                              receive_time_out,
                              &msgsb);
 check_status( "rtr_receive_message", status );
```

2.12.3. Event Management by RTR

RTR manages both event routing and event delivery.

Event Routing

When an event subscription is created with the `rtr_open_channel` call, the event details are stored in a subscriber database on all routers. When an event is triggered, notification is delivered to all routers connected to that system in that facility. The routers then check their subscriber database for any systems that have subscribed to that event. If one or more subscribers are located, and the subscribers are currently attached to this router, then the router broadcasts the message to the subscribers. If no subscriber is located, then the message is discarded.

Event Delivery

RTR reliably delivers RTR transactions and RTR events. The delivery of User Events on a properly configured system is reliable, but RTR Flow Control manages delivery of User Events if the subscriber cannot process events as quickly as they are delivered. Flow Control is RTR's message traffic governor that helps affected systems to manage spikes in message traffic. For more detail on RTR Flow Control, refer to the *VSI Reliable Transaction Router System Manager's Manual*.

When a User Event is triggered, a broadcast that includes message data is routed to the subscriber system. User Events, along with RTR Events and transactions, are placed into an incoming message queue on the destination system until the subscriber application executes an `rtr_receive_message` call to receive the message into the application. If too many messages are sent to the destination system, then the RTR Flow Control feature will be activated.

Flow Control may then force the sending application to wait awhile in the next RTR call that sends data, or it may discard broadcasts from the message queue, until the message queue length reduces and Flow Control allows new broadcasts to be sent to the destination system. Because User Event broadcasts are usually used for streaming information such as the periodic update of a price table, RTR does not store event messages that are impacted by Flow Control for later processing. This technique would cause the application to spend time viewing stale data. Instead, RTR Flow Control may discard the message to help relieve the messaging backlog, and will rely on a future message delivery to supply the updated information.

Design issue: Because of the possibility that a User Event message could become delayed or discarded due to Flow Control, User Events should not be used for delivering information that is of a business critical nature, including information that previously was, or later will be, used in a transaction. To compensate for the possibility of a discarded message, the application developer may consider adding a sequence number to the event message and providing a read-only transaction in the application to detect and request retransmission of any discarded broadcast data from the sender.

Overhead of Using Events

Delivery of User Events is based upon the registration databases that are kept on the routers. The event is delivered from the sender to all connected routers, which means each event triggers a message traffic load of 1 (for a FE sender) or the number of routers (for a BE sender). The event is then propagated by the routers to all subscribers, creating message traffic of 0 or the number of systems with subscribers to the event.

Design Issue: Processing event messages does consume some system resources and could impact overall performance. If system resources become constrained, RTR Flow Control may become active, thus reducing the RTR throughput on the affected systems. Care should be exercised to provide enough system resources to handle the message load.

2.12.4. Event Troubleshooting

Several RTR MONITOR screens can be helpful in troubleshooting events, as described below. Sample screens are available in the *VSI Reliable Transaction Router System Manager's Manual*.

Monitoring Events

User Event traffic (broadcasts) may be monitored specifically for each node using the MONITOR BROADCAST screen in RTR. This screen shows the total event throughput, along with a count of any discarded broadcasts.

The MONITOR FACILITY screen in RTR provides a combined summary of all RTR Events and User Events processed for each facility.

The SHOW CLIENT/FULL and SHOW SERVER/FULL commands in RTR are helpful for viewing the current event subscription list for a particular client or server, along with any channel name specified in the *rcpnam* parameter on the `rtr_open_channel` call.

Execution of `rtr_broadcast_event` calls and event message traffic in RTR can be monitored using the MONITOR CALLS screen in RTR. This screen shows the frequency of use of the `rtr_broadcast_event` call, and the number of RTR Events and User Events processed. If an event is in pending ("pend") status, it indicates that the event is waiting for an `rtr_receive_message` call to be performed.

The MONITOR ROUTING screen shows the transaction and broadcast throughput on the system. This display shows the number of events and also the rate over time during the monitoring interval.

The MONITOR STALLS screen is helpful to determine if RTR Flow Control is affecting a particular system. Flow Control stalls that have occurred are categorized by duration. Any stall that lasts more than 60 seconds results in a Link Drop entry. A Stall ("stll") entry in the far-right column indicates that a Flow Control stall is currently in progress on the link indicated. For the purposes of User Event broadcast delivery, any stall could indicate that a broadcast message could have been discarded.

It is possible to monitor additional details of RTR Flow Control by using the MONITOR CONGEST, MONITOR FLOW, and MONITOR TRAFFIC monitor screens in RTR.

2.13. Use of XA Support

Users need to register a resource manager first, to invoke RTR XA support when creating a facility. Please see the *VSI Reliable Transaction Router System Manager's Manual* for more details about how to register and unregister resource managers.

In the server application, specify the flag `RTR_F_OPE_XA_MANAGED` and the underlying resource manager information when issuing the `rtr_open_channel` call. Once this flag is specified for a given RTR partition, all transactions running in that RTR partition are committed using the XA interface between RTR and the resource manager. When the partition is deleted or the resource manager is unregistered, RTR commits transactions running in this partition in a conventional manner.

2.14. RTR Applications in a Multiplatform Environment

Applications using RTR in a multiplatform (that is, mixed endian) environment with nonstring application data have to tell RTR how to marshall the data for the destination architecture. The sender

of a message must supply both a description of the application data being sent and the application data itself. This description is supplied as the *msgfmt* argument to `rtr_send_to_server`, `rtr_reply_to_client`, and `rtr_broadcast_event`.

The default (that is, when no *msgfmt* is supplied) is to assume the application message is string data.

2.14.1. Defining a Message Format

The *msgfmt* 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:	
	Code	**Meaning**
	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 the following data structure:

```
    typedef struct {
    rtr_uns_32_t first ;
    rtr_sgn_32_t second ;
    char  str[12] ;
} example_t ;
```

The *msgfmt* for this structure could be "%UL%SL%12C".

The transparent data type conversion of RTR does not support certain conversions (for example, floating point).

2.14.1.1. Data Types

Data types supported by RTR are:

- Unsigned
- Signed
- Char
- Boolean

2.14.1.2. Alignment

Alignment of data on byte boundaries depends on several factors, including the compiler used in creating an application. RTR's data marshalling software manages these alignments.

2.15. Application Design and Tuning Issues

This section addresses some considerations for design and tuning, including:

- Transactions that can cause server failure
- Transaction grouping and database applications
- Transaction sequence and shadow servers
- Transaction independence

2.15.1. Transactions That Can Cause Server Failure

It is possible for a “rogue” client transaction, due to a user application bug, to “kill” the server process. If RTR were to reapply this transaction indefinitely, all available servers would be destroyed. To avoid a transaction killing all server processes, the following mechanism is implemented:

- A transaction for which no `rtr_accept_tx` has been called by a server is aborted after it has caused the death of three concurrent servers to which it has been presented. The transaction abort status reported to the client is *RTR_STS_SRVDIED*. Retry count for transactions that have not been voted on is three; for transactions that have been voted on, retry count can be limited with the RTR command *SET PARTITION/RECOVERY_RETRY_COUNT* (default: unlimited).
- An RTR error log message with the same status is also written on the backend where the server deaths occurred.

The limitation of this feature to transactions that have not yet been accepted prevents possible transaction inconsistencies that could otherwise arise between client and server(s), and on shadow secondary sites. Thus a server application should complete any necessary validation of client transaction messages *before* accepting the transaction, to take advantage of this feature.

2.15.2. Transaction Grouping and Database Applications

RTR generates commit sequence numbers (CSN) for each transaction committed on the primary site. Concurrent servers can have several transactions assigned to a single CSN value. Transactions with the same CSN are understood by RTR to be independent, and hence their relative commit ordering to the database does not violate the serializability requirements of transactions.

For purposes of throughput, RTR attempts to group as many transactions as possible into a single CSN during a given vote cycle. (Grouped transactions are only those that explicitly vote (that is, call `rtr_accept_tx` on the server.)

The vote cycle completes as soon as RTR is ready to ask a server to commit the next transaction. For this mechanism to work correctly with the application, RTR places the following restriction on the server design:

A server must obtain an exclusive lock on any resource that another concurrent server may be accessing for a different transaction **before** it issues the call to `rtr_accept_tx`.

Database applications, in general, comply with this requirement. If the database management software allows “dirty reads,” the application should apply this rule explicitly, so that RTR can correctly serialize transactions during shadowing or other recovery. *Failure to comply with this rule can cause unsynchronised copies of shadow databases.*

2.15.3. Transaction Sequence and Shadow Servers

When using a facility having a shadow site and two or more partitions, the transaction sequence is the same at both shadow sites **within a single partition only. Sequences across partitions are not preserved.** For example, suppose the following transactions are executed on half of a shadow site in the following chronological order:

tx1_for_partition1
tx2_for_partition1
tx3_for_partition1
tx1_for_partition2
tx4_for_partition1

When replayed on the secondary, the order could be:

tx1_for_partition1
tx2_for_partition1
tx3_for_partition1
tx4_for_partition1
tx1_for_partition2

Do not write your application to expect preservation of transaction serialization across partitions.

2.15.4. Transaction Independence

RTR normally assumes that each transaction processed by a given server depends on the transactions that particular server has previously accepted.

To keep the shadowed database identical to that on the primary, RTR controls the order in which the secondary executes transactions. The secondary is constrained to execute transactions in the same order as the primary. Under some circumstances, this can lead to the secondary sitting idle, waiting to be given a transaction to execute.

RTR provides a performance enhancement that may help some applications decrease idle time on the secondary, reducing the corresponding backlog. If the application knows that particular transactions are independent of the transactions previously received, then the application can set one of two flags listed in Table 2.9.

Table 2.9. Independent Transaction Flags

Flag	Meaning
RTR_F_ACC_INDEPENDENT	Set on an `rtr_accept_tx` call to indicate this transaction is independent.
RTR_F_REP_INDEPENDENT	Set on an `rtr_reply_to_client` call along with `RTR_F_REP_ACCEPT` to indicate this transaction is independent.

A transaction accepted with one of these flags can be started on the secondary while other transactions are still running. **All transactions flagged with one of these flags must truly be independent of the transactions that have previously executed. They will execute in an arbitrary sequence on the secondary site.**

If the server channel has been opened with `RTR_F_OPE_EXPLICIT` (explicit accept), then the `RTR_F_REP_INDEPENDENT` flag can only be used together with `RTR_F_REP_ACCEPT`. If the server channel has been opened with implicit accept, then using `RTR_F_REP_INDEPENDENT` implies using `RTR_F_REP_ACCEPT`.

An application can be written to create CSN boundaries to ensure independence. A transaction always receives a CSN, and the `INDEPENDENT` flag could be used to prevent the CSN from being incremented, so an application could be coded to force dependence between sets of transactions. This could be important in certain cases where transactions coming in at a particular time of day are independent of each other, but other transactions executed, say, at the end of the day, need to ensure that the day's transactions have been processed, and the following day's transactions need to ensure that the previous end-of-day processing has completed. For more details on user of independent transactions, refer to the discussion of CSNs in the *VSI Reliable Transaction Router Application Design Guide*.

Chapter 3. RTR Call Reference

This chapter contains the environmental limits, field length maxima, and syntax definitions for all RTR C programming API calls.

3.1. RTR Environmental Limits

RTR deals with several environmental entities that have architectural limits as shown in Table 3.1. Actual limits in a specific configuration are determined by performance.

Table 3.1. Environmental Limits

Component	Limit
BE or TR nodes	512
Bytes per message	64000
Channels per application process	1024
Facilities	100 to 1000, depending on operating system
FE nodes	1000
Journal files	16
Memory per process	OpenVMS: 4GB; UNIX: unlimited; NT: 2GB
Messages per transaction - server to client	unlimited
Messages per transaction - client to server	65534
Partitions	65536 (dynamic; default:500)
Processes per node	1000
Size of journal file	256MB
Threads per application process (where supported by operating system)	4096

3.2. RTR Maximum Field Lengths

Table 3.2 contains definitions of RTR field length maxima. The file `rtr.h` contains values for these field lengths.

Table 3.2. RTR Maximum Field-Length Definitions

Field Name	Description
RTR_MAX_ACCESS_LEN	Maximum length of access string.
RTR_MAX_BLOB_LEN	Maximum length of data that can be passed in a prepare call.
RTR_MAX_FACNAM_LEN	Maximum length of facility name.
RTR_MAX_FE_NAM_LEN	Maximum length of frontend name.
RTR_MAX_MSGFMT_LEN	Maximum length of message format.
RTR_MAX_MSGLEN	Maximum length of an RTR message.
RTR_MAX_NUMSEG	Maximum number of segments in key.

Field Name	Description
RTR_MAX_PARNAM_LEN	Maximum length of partition name.
RTR_MAX_RCPNAM_LEN	Maximum length of broadcast recipient name.
RTR_MAX_RCPSPC_LEN	Maximum length for broadcast recipient specification.
RTR_MAX_SELVAL_LEN	Maximum length for selector value.

3.3. RTR C API Calls

The calls are presented in alphabetical order.

rtr_accept_tx

rtr_accept_tx — Accept the transaction currently active on the specified channel.

Format

```
status = rtr_accept_tx (channel, flags, reason)
```

Argument	Data Type	Access
status	rtr_status_t	write
channel	rtr_channel_t	read
flags	rtr_acc_flag_t	read
reason	rtr_reason_t	read

C Binding

```
rtr_status_t rtr_accept_tx (
        rtr_channel_t channel ,
        rtr_acc_flag_t flags ,
        rtr_reason_t reason
        )
```

Arguments

channel

The channel identifier (returned earlier by `rtr_open_channel( )`).

flags

Flags that specify options for the call.

Table 3.3 shows the flags that are defined.

Table 3.3. Accept Transaction Flags

Flag name	Description
RTR_F_ACC_FORGET	Set to prevent receipt of any more messages (or completion status) associated with the transaction.

Flag name	Description
	Any such messages are discarded. This flag is valid only on server channels; it has no effect on client channels.
RTR_F_ACC_INDEPENDENT	Set to indicate this transaction is independent. (See Section 2.15.4 for further information.)

If you do not require any flags, specify `RTR_NO_FLAGS` for this parameter.

reason

Optional reason for accepting the transaction. This reason is **OR**ed together with the reasons of the other participants in the transaction and returned in the *reason* field of the **`rtr_status_data_t`** structure returned with the **`rtr_mt_accepted`** message to all participants of the transaction. Specify `RTR_NO_REASON` if no reason is required.

Description

The `rtr_accept_tx()` call accepts the transaction currently active on the specified channel. After `rtr_accept_tx()` has been called, the caller may no longer actively participate in the fate of the transaction; that is, messages and the final completion status can still be received, but no further messages may be sent for the transaction. An attempt to send a further message yields an *RTR_STS_TXALRACC* return status.

Note

RTR_STS_PRTSTACHGTXRES – When the RTR environment has changed, such as backend lost quorum (for example, router not available), or a concurrent server is crashed etc., a transaction currently being processed would be aborted. The server would receive a `rtr_mt_rejected` message with this status. RTR would reschedule the tx to get processed in a different server, if available. This error message is passed back to the application and is returned in the `RTR$L_TXSB_STATUS` field of the TXSB when an RTR system service call completes.

Return Value

A value indicating the status of the routine. Possible status values are:

RTR_STS_AMBROUNAM	Ambiguous API routine name for call - supply more characters
RTR_STS_CHANOTOPE	Channel not opened
RTR_STS_INVCHANNEL	Invalid channel argument
RTR_STS_INVFLAGS	Invalid flags argument
RTR_STS_OK	Normal successful completion
RTR_STS_TXALRACC	Transaction already accepted
RTR_STS_TXNOTACT	No transaction currently active on this channel

Example

```
/*
* Client is done with the txn; if the server accepts the
```

```
  * transaction, there is no reason for us to reject it.
  * Accept it, then go on to a new transaction.
  */
 if (msgsb.msgtype == rtr_mt_accepted)
       {
             status = rtr_accept_tx(
                             channel,
                             RTR_NO_FLAGS,
                             RTR_NO_REASON );
             check_status( status );
         }
         else
         .
         .       Issue the error message returned by the
         .       server, and recover from there.
         .
```

See Also

rtr_open_channel( )

rtr_reject_tx( )

rtr_reply_to_client( )

rtr_broadcast_event

rtr_broadcast_event — Broadcast (send) a user event message.

Format

status = rtr_broadcast_event (channel, flags, pmsg, msglen, evtnum, rcpspc, msgfmt

Argument	Data Type	Access
status	rtr_status_t	write
channel	rtr_channel_t	read
flags	rtr_bro_flag_t	read
pmsg	rtr_msgbuf_t	read
msglen	rtr_msglen_t	read
evtnum	rtr_evtnum_t	read
rcpspc	rtr_rcpspc_t	read
msgfmt	rtr_msgfmt_t	read

C Binding

```
rtr_status_t rtr_broadcast_event (
        rtr_channel_t channel ,
        rtr_bro_flag_t flags ,
        rtr_msgbuf_t pmsg ,
        rtr_msglen_t msglen ,
        rtr_evtnum_t evtnum ,
        rtr_rcpspc_t rcpspc ,
```

```
rtr_msgfmt_t msgfmt
)
```

Arguments

channel

The channel identifier (returned earlier by `rtr_open_channel( )`).

flags

No flags are currently defined. Specify `RTR_NO_FLAGS` for this parameter.

pmsg

Pointer to the message to broadcast.

msglen

Length in bytes of the message broadcast.

evtnum

User event number associated with this broadcast. (Recipients must specify this to receive it.) For more information on user event numbers, see Section 2.12.

rcpspc

Name of the recipient(s). This null-terminated character string contains the name of the recipient(s), specified with the *rcpnam* parameter on the call to `rtr_open_channel( )`.

Wildcards ("*" for any sequence of characters, and "%" for any one character) can be used in this string to address more than one recipient. *rcpspc* is an optional parameter. Specify `RTR_NO_RCPSPC` for this parameter if no *rcpspc* is required.

Note

Named Events.

- To receive named events, the correct event number must also be specified. The event number (*evtnum*) must be specified by both the sender (*rcpspc*) and the recipient (*rcpnam*).
- Both *rcpnam* and *rcpspc* are case sensitive.
- Both *rcpnam* and *rcpspc* default to the case-insensitive channel name if no explicit *rcpnam* or *rcpspc* is provided.

---

msgfmt

Message format description. This null-terminated character string contains 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. See Section 2.14 for information on defining a message format description.

This parameter is optional. Specify `RTR_NO_MSGFMT` if message content is platform independent, or there is no intent to use other hardware platforms.

Description

The `rtr_broadcast_event( )` call broadcasts a user event message. The caller must first open a channel (using `rtr_open_channel( )`), before it can send user event messages.

A client channel can be used to send user event messages to servers.

A server channel can be used to send user event messages to clients.

Return Value

RTR_STS_CHANOTOPE	Channel not opened
RTR_STS_INSVIRMEM	Insufficient virtual memory
RTR_STS_INVCHANNEL	Invalid channel argument
RTR_STS_INVEVTNUM	Invalid evtnum argument
RTR_STS_INVFLAGS	Invalid flags argument
RTR_STS_INVMSGFMT	Invalid msgfmt argument
RTR_STS_INVMSGLEN	Invalid msglen argument
RTR_STS_INVRCPSPC	Invalid rcpspc argument
RTR_STS_NOACP	No RTRACP process available
RTR_STS_OK	Normal successful completion
RTR_STS_WOULDBLOCK	Operation would block. Try again later.

Example

```
#define reunion_announcement 678     // In user .h file.

rtr_msg_buf_t reunion_msg = "Jones family reunion today!";
rtr_rcpspc_t recipients = "*Jones";
/*
 * If today is the date of the Jones family reunion, tell
 * any client whose last name is Jones that they need to
 * be there!
 */
        if (strcmp(today, reunion_date) == 0)
        {
                status = rtr_broadcast_event(
                        &channel,
                        RTR_NO_FLAGS,
                        reunion_msg,
                        strlen(reunion_msg),
                        reunion_announcement,
                        recipients,
                        RTR_NO_MSGFMT );
                check_status( status );
}
```

See Also

`rtr_receive_message( )`

`rtr_open_channel( )`

rtr_close_channel

rtr_close_channel — Close a previously opened channel.

Format

`status = rtr_close_channel (channel, flags)`

Argument	Data Type	Access
status	rtr_status_t	write
channel	rtr_channel_t	read
flags	rtr_clo_flag_t	read

C Binding

```
rtr_status_t rtr_close_channel (
        rtr_channel_t channel ,
        rtr_clo_flag_t flags
        )
```

Arguments

channel

The channel identifier (returned earlier by `rtr_open_channel( )`, or `rtr_request_info( )` or `rtr_set_info( )`).

flags

Flags that specify options for the call.

The flag RTR_F_CLO_IMMEDIATE is defined for this call.

Normally `rtr_close_channel( )` processes a pending transaction that was in a commit state by forgetting the transaction (removing it from the journal). To close the channel but leave transactions in the journal, use the flag RTR_F_ CLO_IMMEDIATE to `rtr_close_channel( )`.

In some situations, an accepted transaction cannot be completed and replay is required. For example, a transaction may be accepted but the database becomes unavailable before the transaction is committed to the database. To deal with such a situation, an application can use the close-immediate flag RTR_F_CLO_IMMEDIATE. This closes the channel but leaves the transactions in the journal for use on replay when database access is restored. If you do not need any flags, specify RTR_NO_FLAGS for this argument.

Description

The `rtr_close_channel( )` call closes a previously opened channel. A channel may be closed at any time after it has been opened via `rtr_open_channel( )` or `rtr_request_info( )`. If the channel is a server channel, an implicit acknowledgment is sent, if you have a current transaction.

If you call the CLOSE_CHANNEL API from the CLI, after receiving the rtr_ mt_accepted message, then it leaves the transaction in the journal, however if you call it from the program after receiving ther rtr_mt_accepted message, then it is cleared from the journal, irrespective of the flag RTR_F_CLO_ IMMEDIATE being used or not.

Return Value

A value indicating the status of the routine. Possible status values are:

RTR_STS_ACPNOTVIA	RTR ACP no longer a viable entity, restart RTR or application
RTR_STS_BYTLMNSUFF	Insufficient process quota bytlm, required 100000
RTR_STS_INVCHANNEL	Invalid channel argument
RTR_STS_NOACP	No RTRACP process available
RTR_STS_OK	Normal successful completion

Example

```
/* If the status returned by the previous call is not success,
 * close now, and exit the program.
 */
if (status != RTR_STS_OK)
        {
        printf(fpLog, "Unexpected error, must close immediately!");
        status = rtr_close_channel( channel, RTR_CLO_IMMEDIATE);
        exit(status);
}
/*
 * Normal processing complete, close the channel.
 */
        printf(fpLog, "Closing channel");
status = rtr_close_channel ( channel, RTR_NO_FLAGS );
```

See Also

`rtr_open_channel( )`

rtr_error_text

rtr_error_text — Return the text associated with an RTR status value.

Format

`retval = rtr_error_text (sts)`

Argument	Data Type	Access
retval	char*	write
sts	rtr_status_t	read

C Binding

```
char *rtr_error_text (
        rtr_status_t sts
        )
```

Arguments

sts

The RTR error number for which the text is required.

Description

The `rtr_error_text( )` call returns a pointer to the text associated with an RTR error number.

The text string is a constant. If an invalid value for *sts* is supplied, a pointer is also returned to an error text, indicating an invalid value.

Example

```
/* If the status returned by the previous call is not success,
 * print the message text to the error log, and exit.
 */
if (status != RTR_STS_OK)
{
    printf(errLog, rtr_error_text(status));
        exit(status);
      }
```

rtr_ext_broadcast_event

rtr_ext_broadcast_event — Broadcast (send) a user event message or an RTR_STS_TIMOUT status if RTR is unable to issue to broadcast message within the specified timeout period. The call is the same as rtr_broadcast_event with the addition of the timeout period, given in milliseconds.

Format

`status = rtr_ext_broadcast_event (channel, flags, pmsg, msglen, evtnum, rcpsp`

Argument	Data Type	Access
status	rtr_status_t	write
channel	rtr_channel_t	read
flags	rtr_bro_flag_t	read
pmsg	rtr_msgbuf_t	read
msglen	rtr_msglen_t	read
evtnum	rtr_evtnum_t	read
rcpspc	rtr_rcpspc_t	read
msgfmt	rtr_msgfmt_t	read
timoutms	rtr_timout_t	read

C Binding

```
rtr_status_t rtr_ext_broadcast_event (
        rtr_channel_t channel ,
        rtr_bro_flag_t flags ,
        rtr_msgbuf_t pmsg ,
        rtr_msglen_t msglen ,
        rtr_evtnum_t evtnum ,
        rtr_rcpspc_t rcpspc ,
        rtr_msgfmt_t msgfmt ,
        rtr_timout_t timoutms
```

)

Arguments

channel

The channel identifier (returned earlier by `rtr_open_channel( )`).

flags

No flags are currently defined. Specify `RTR_NO_FLAGS` for this parameter.

pmsg

Pointer to the message to broadcast.

msglen

Length in bytes of the message to be broadcast.

evtnum

User event number associated with this broadcast. (Recipients must specify this to receive it.) For more information on user event numbers, see Section 2.12.

rcpspc

Name of the recipient(s). This null-terminated character string contains the name of the recipient(s), specified with the *rcpnam* parameter on the call to `rtr_open_channel( )`.

Wildcards ("*" for any sequence of characters, and "%" for any one character) can be used in this string to address more than one recipient. *rcpspc* is an optional parameter. Specify `RTR_NO_RCPSPC` for this parameter if no *rcpspc* is required.

Note

Named Events

- To receive named events, the correct event number must also be specified. The event number (*evtnum*) must be specified by both the sender (*rcpspc*) and the recipient (*rcpnam*).
- Both *rcpnam* and *rcpspc* are case sensitive.
- Both *rcpnam* and *rcpspc* default to the case-insensitive channel name if no explicit *rcpnam* or *rcpspc* is provided.

---

msgfmt

Message format description. This null-terminated character string contains 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. See Section 2.14 for information on defining a message format description.

This parameter is optional. Specify `RTR_NO_MSGFMT` if message content is platform independent, or there is no intent to use other hardware platforms.

timoutms

Timeout value in milliseconds that the call will wait before timing out. Returns status RTR_STS_TIMOUT if RTR is unable to process the call. If no timeout is needed, specify RTR_NO_TIMOUTMS.

Description

The `rtr_ext_broadcast_event( )` call broadcasts a user event message. The caller must first open a channel (`using rtr_open_channel( )`), before it can send user event messages.

A client channel can be used to send user event messages to servers.

A server channel can be used to send user event messages to clients.

In some circumstances, a broadcast event can wait a long time if RTR runs out of channel credits; it may seem that the application is hanging. To eliminate such a wait, the application can specify a timeout value from which the call returns an RTR_STS_TIMOUT status if RTR is unable to issue the broadcast message within the specified timeout period.

Return Value

A value indicating the status of the routine. Possible status values are:

RTR_STS_CHANOTOPE	Channel not opened
RTR_STS_INSVIRMEM	Insufficient virtual memory
RTR_STS_INVCHANNEL	Invalid channel argument
RTR_STS_INVEVTNUM	Invalid evtnum argument
RTR_STS_INVFLAGS	Invalid flags argument
RTR_STS_INVMSGFMT	Invalid msgfmt argument
RTR_STS_INVMSGLEN	Invalid msglen argument
RTR_STS_INVRCPSPC	Invalid rcpspc argument
RTR_STS_NOACP	No RTRACP process available
RTR_STS_WOULDBLOCK	Operation would block. Try again later
RTR_STS_OK	Normal successful completion

Example

```
#define reunion_announcement 10     /* In user .h file. */

rtr_msg_buf_t reunion_msg = "Jones family reunion today!";
rtr_rcpspc_t recipients = "*Jones";
rtr_timout_t = 1000 /* 1 second to time out */
/*
 * If today is the date of the Jones family reunion, tell
 * any client whose last name is Jones that they need to
 * be there!
 */
        if (strcmp(today, reunion_date) == 0)
        {
                status = rtr_ext_broadcast_event(
                        &channel,
                        RTR_NO_FLAGS,
                        reunion_msg,
```

```
                        strlen(reunion_msg),
                        reunion_announcement,
                        recipients,
                        RTR_NO_MSGFMT,
                        timoutms );

                check_status( status );
}
```

See Also

`rtr_broadcast_event( )`

`rtr_receive_message( )`

`rtr_open_channel( )`

rtr_get_tid

rtr_get_tid — Return the transaction ID for the current transaction.

Format

`status = rtr_get_tid (channel, flags, ptid)`

Argument	Data Type	Access
status	rtr_status_t	write
channel	rtr_channel_t	read
flags	rtr_tid_flag_t	read
ptid	void*	write

C Binding

```
rtr_status_t rtr_get_tid (
        rtr_channel_t channel ,
        rtr_tid_flag_t flags ,
        void *ptid
        )
```

Arguments

channel

The channel identifier (returned previously by `rtr_open_channel( )`).

flags

Flags that specify options for the call.

Table 3.4 shows the flags that are defined.

Table 3.4. Get TID Flags

Flag	Pointer Data Types	Returns
RTR_NO_FLAGS	rtr_tid_t	RTR transaction ID

Flag	Pointer Data Types	Returns
RTR_F_TID_RTR	rtr_tid_t	RTR transaction ID
RTR_F_TID_XA	rtr_xid_t	XA transaction ID
RTR_F_TID_DDTM	rtr_ddtmid_t	DECdtm transaction ID

If you do not require any flags, specify `RTR_NO_FLAGS` for this argument. Specifying `RTR_NO_FLAGS` is equivalent to specifying `RTR_F_TID_RTR`; this capability is maintained for compatibility with RTR versions earlier than RTR Version 3.2.

The structure `rtr_xid_t` is based on the X/Open XA specification and is defined as follows:

```
typedef struct rtr_xid_t {
    long formatID; /* format identifier */
    long gtrid_length; /* value from 1 through 64 */
    long bqual_length; /* value from 1 through 64 */
    char data[RTR_XIDDATASIZE];
} rtr_xid_t;
```

The XID structure contains a format identifier, two length fields and a data field. The data field comprises at most two contiguous components: a global transaction ID (*gtrid*) and a branch qualifier (*bqual*).

The *gtrid_length* field specifies the number of bytes (1-64) that constitute *gtrid*, starting at the first byte in data (that is, data[0]). The *bqual_length* field specifies the number of bytes (1-64) that constitute *bqual*, starting at the first byte after *gtrid* (that is, *data[gtrid_length]*). Neither component in data is null terminated. Any unused bytes in data are undefined.

The contents of data depend on the format of the transaction ID (TX ID), which is specified by the format identification field. Some valid format ID values are shown in Table 3.5.

Table 3.5. Format Identification and Data Content

Format Identification	Data Content
RTR_XID_FORMATID_NONE	Null XID. No XID has been returned. This will be the value if the call to `rtr_get_xid / rtr_get_tid` returns an error, for example.
RTR_XID_FORMATID_OSI_CCR	The XID is specified using the naming rules specified for OSI CCR atomic action identifiers. RTR does not use this convention directly, but such a transaction ID format can be returned if some other associated transaction or resource manager uses this convention. If OSI CCR (ISO standard) naming is used, then the XID's *formatID* element should be set to 0 (zero); if another format is used, then the *formatID* element should be greater than 0. A value of -1 in *formatID* means that the XID is null.
RTR_XID_FORMATID_RTR	Identifies an RTR transaction ID. In this case, the *gtrid_length* is 28 and *bqual_length* is zero. The contents of data can be interpreted using the format defined by `rtr_tid_t`. Note that one should still use the `rtr_get_tid` call to get the

Format Identification	Data Content
	RTR transaction ID for a transaction active on a channel. The `rtr_get_xid` call could be used, for example, if a nested transaction is started where the foreign transaction manager is also RTR.
RTR_XID_FORMATID_DDTM	Identifies a transaction ID for a transaction that uses a resource managed by DECdtm. The *gtrid_length* field is 16, and *bqual_length* is 0.
RTR_XID_FORMATID_RTR_XA	Identifies a transaction ID for a transaction started using an XA resource manager.

ptid

A pointer to where the unique transaction ID for the current transaction is returned. Data type depends on any flag that has been set; see Table 3.4.

Description

`rtr_get_tid( )` returns the RTR transaction ID for the current transaction.

The RTR transaction ID is a unique number generated by RTR for each transaction in the RTR virtual network.

In addition, `rtr_get_tid( )` is capable of returning transaction identifiers associated with XA and DECdtm managed transactions when RTR is operating with either of these transaction managers.

Return Value

A value indicating the status of the routine. Possible status values are:

RTR_STS_CHANOTOPE	Channel not opened
RTR_STS_DTXNOSUCHXID	No distributed transaction ID found for this channel
RTR_STS_INVARGPTR	Invalid parameter address specified on last call
RTR_STS_INVCHANNEL	Invalid channel argument
RTR_STS_INVFLAGS	Invalid flags argument
RTR_STS_NOXACHAN	No XA channel available
RTR_STS_OK	Normal successful completion
RTR_STS_TXNOTACT	No transaction currently active on this channel

Example

```
        rtr_xid_tid xa_tid;
        char global_id_buff[64];
        char branch_qual_buff[64];
        int i, j;

/* The server executed an rtr_receive_message. In the
 * rtr_msgsb_t structure, the msgtype field equals
 * rtr_mt_msg1_uncertain. This indicates that a recovery
 * is in process, and RTR did not get a confirmation
 * that the current transaction had been
```

```
 * completed. RTR is now 'replaying' the transaction,
 * and this is the first message in that transaction.
 *
 * Get the transaction id.
 */
status = rtr_get_tid(
                &channel,
                RTR_F_TID_XA,
                &xa_tid );

check_status(status);

/*
 * Isolate the information in the xa_tid structure.
 */
        if (xa_tid.formatID != RTR_XID_FORMATID_RTR_XA)
{
    printf(errLog, "This channel only for X/Open transactions");
    exit(BAD_TXTYPE_CHAN);
)
for (i = 0; i < xa_tid.gtrid_length; i++)
        global_id_buff[i] = xa_tid.data[i];
global_id_buff[i] = 0;
for
(j = i; j < (xa_tid.gtrid_length + xa_tid.bqual_length); j++)
        branch_qual_buff[j - i] = xa_tid.data[j];
branch_qual_buff[j] = 0;

/* Query the database to see if the transaction whose global_id
 * and branch qualifier match these had been committed. If so,
 * ignore; otherwise, continue as though this were the first
 * time the message was received.
 */
```

rtr_get_user_context

rtr_get_user_context — Retrieve the optional user-defined context associated with the specified RTR channel.

Format

`user_context = rtr_get_user_context (channel )`

Argument	Data Type	Access
user_context	rtr_usrctx_t	write
channel	rtr_channel_t	read

C Binding

```
rtr_usrctx_t rtr_get_user_context (
        rtr_channel_t channel
        )
```

Arguments

channel

The channel whose context is to be returned.

Usage example:

```
struct { rtr_channel_t chan; int state, ... } context[10]; *ctx;
rtr_channel_t chan;
rtr_open_channel(&ctx[4].chan, ...);
rtr_receive_message(&chan, ...);
ctx = rtr_get_user_context(chan);
if (ctx->state) { ... }
```

Description

The `rtr_get_user_context( )` call retrieves the user context for a channel. The default value of the user context is the value of the pchannel argument passed to RTR at the time the channel was opened using one of the following calls or routines:

`rtr_open_channel( )`

`rtr_request_info( )`

`rtr_set_info( )`

The context value may optionally be changed at any later time using `rtr_set_user_context( )`, provided the channel is still open.

The routine returns the user context of the specified channel.

Return Value

A value indicating the status of the routine. Possible status values are:

RTR_NO_USER_CONTEXT	The specified channel was not declared or has closed

rtr_open_channel

rtr_open_channel — Open a channel to allow for communication with other applications.

Format

`status = rtr_open_channel (pchannel, flags, facnam, rcpnam, pevtnum, access, numse`

Argument	Data Type	Access
status	rtr_status_t	write
pchannel	rtr_channel_t	write
flags	rtr_ope_flag_t	read
facnam	rtr_facnam_t	read
rcpnam	rtr_rcpnam_t	read
pevtnum	rtr_evtnum_t	read
access	rtr_access_t	read

Argument	Data Type	Access
numseg	rtr_numseg_t	read
pkeyseg	rtr_keyseg_t	read

C Binding

```
rtr_status_t rtr_open_channel (
        rtr_channel_t *pchannel ,
        rtr_ope_flag_t flags ,
        rtr_facnam_t facnam ,
        rtr_rcpnam_t rcpnam ,
        rtr_evtnum_t *pevtnum ,
        rtr_access_t access ,
        rtr_numseg_t numseg ,
        rtr_keyseg_t *pkeyseg
        )
```

Arguments

Flags that specify options for the call.

Defined flags are shown in Table 3.6, Table 3.7, and Table 3.8.

Table 3.6. Open Channel Flags (One Required)

Flag	Description
RTR_F_OPE_CLIENT	Indicates that the channel will be used as a client.
RTR_F_OPE_CREATE_ PARTITION	Requests that a partition be created. Specify partition key segments and name with the *pkeyseg* argument. The name is passed using an `rtr_keyseg_t` descriptor where *ks_type* = *rtr_keyseg_partition* and *ks_lo_bound* point to the name string. On a successful call, a channel is opened on which the completion status can be read from the ensuing message of type `rtr_mt_closed`. The completion status is found in the status field of the message data of `rtr_status_data_t`.
RTR_F_OPE_DELETE_ PARTITION	Requests that a partition be deleted. Specify partition or name key segments with the *pkeyseg* argument. The name is passed using an `rtr_keyseg_t` descriptor where *ks_type* = *rtr_keyseg_partition* and *ks_lo_bound* points to the name string. On a successful call, a channel is opened on which the completion status can be read from the ensuing message of type `rtr_mt_closed`. The completion status is found in the status field of the message data of `rtr_status_data_t`.
RTR_F_OPE_SERVER	Indicates that the channel will be used as a server. *numseg* and *pkeyseg* must be specified for all servers except call-out servers.

Table 3.7. Open Channel Client Flags

Flags	Description
RTR_F_OPE_EXPLICIT_START	Valid for client channels only. Use of this flag requires that an explicit `rtr_start_tx( )` be called on this channel. The procedure is in effect until the channel is closed. The EXPLICIT_START flag ensures that the `rtr_send_to_server( )` will not generate new transactions should the `rtr_start_tx( )` time out. If the user calls `rtr_send_to_server( )` without first calling `rtr_start_tx( )`, the error message `RTR-F-INVIMPLCTSTRT` is returned informing the caller that they must call `rtr_start_tx( )` first on this channel.
RTR_F_OPE_FOREIGN_TM	Valid for client channels only. This indicates that the global coordinating transaction manager is a foreign transaction manager (non-RTR), and that all transactions on this channel will be coordinated by the foreign transaction manager. If this flag is set, then calls to `rtr_start_tx` on this channel must supply a value for the *jointxid* parameter, which is the ID of the parent transaction.

Note

Calling `rtr_open_channel( )` with the `RTR_F_OPE_FOREIGN_TM` flag set causes a local RTR journal scan to occur, if a journal has not already been opened on that node.

The flags in Table 3.8 apply only if RTR_F_OPE_SERVER is set.

Note

Server attributes such as key range definition, shadow and standby flags, can be defined and modified outside the application program by the system manager. A server should preferably use specific flags.

Table 3.8. Open Channel Server Flags

Flag	Description
RTR_F_OPE_BE_CALL_OUT	The server is a backend callout server. By default a server is not a backend callout server.
RTR_F_OPE_DECDTM_MANAGED	Indicates that DECdtm manages the channel. Valid only for server channels.
RTR_F_OPE_EXPLICIT_ACCEPT	A call to `rtr_receive_message( )` is not to be interpreted as an implicit call of `rtr_accept_tx( )`.
RTR_F_OPE_EXPLICIT_PREPARE	The server needs to receive an explicit prepare message from RTR when each transaction has

Flag	Description
	been completely received. By default, no prepare message is generated.
RTR_F_OPE_NOCONCURRENT	The server may not be concurrent with other servers. By default a server may have other concurrent servers.
RTR_F_OPE_NORECOVERY	Valid for a server channel, this flag specifies partition operation without the services of the RTR recovery journal. This option may be useful for applications whose focus is on the timely delivery of messages with limited lifetimes, where the recovery of possibly stale data is not of interest. Since no IO operations to the RTR journal are performed, resource consumption per transaction will be lower, particularly for applications where the number of concurrently active servers is small. Partitions operating in this mode will perform the usual recovery operations, but no recovery transactions will be found. Further, shadowed partitions in remember mode using this option are also not using the journal, so shadow recovery of such partitions will find no shadow recovery transactions. Since consistency between shadowed sites can thus no longer be maintained, server channels attached to such partitions will automatically be closed should such a non-journalled partition transition from an active to an inactive state.
RTR_F_OPE_NOSTANDBY	The server may not be (or have) standby(s). By default, servers may have standby(s).
RTR_F_OPE_RECEIVE_REPLIES	The server, a backend callout server, can receive server-to-client messages.
RTR_F_OPE_SHADOW	The server is part of a shadow pair. By default a server is not part of a shadow pair.
RTR_F_OPE _STRICT_SHAD_ ORDER	See the Usage Restriction below.
RTR_F_OPE_TR_CALL_OUT	The server is a router callout server. By default a server is not a router callout server.
RTR_F_OPE_XA_MANAGED	Associates the channel with the XA protocol.

Usage Restriction

Ordinarily RTR determines groups of independently voting concurrent transactions on the primary site from server behavior. Transactions within a group can then be presented on the secondary in any order. The Shadow Order flag modifies this behavior so that transactions are presented on the secondary site strictly in the order in which they are accepted by the application on the primary.

Allowed Settings

For consistent operation, the shadow order flag depends on the journal-less flag. That is, only certain combinations are allowed:

This Shadow Order Flag Setting	With this No_ Recovery Flag Setting	Is:
STRICT_SHD_ ORDER	NO_RECOVERY	
0	0	Allowed
0	1	Allowed
1	1	Allowed
1	0	Not allowed

facnam

A null-terminated string containing the facility name. A facility name is required.

rcpnam

An optional null-terminated string containing the name of the recipient. This name is used to receive named event messages. Specify `RTR_NO_RCPNAM` when named event recipients are not used.

These names are additional qualifiers on the event delivery, are matched to the sender name, and are ANDed to the event number for delivery. For example, a client "New York" and a client "Hong Kong" could be set up to both receive event number 100. If the event 100 was generated by the server with the name "Hong Kong," the event would not be received by the "New York" client.

Named Events

- To receive named events, the correct event number must also be specified. The event number (*evtnum*) must be specified by both the sender (*rcpspc*) and the recipient (*rcpnam*).
- Both *rcpnam* and *rcpspc* are case sensitive.
- Both *rcpnam* and *rcpspc* default to the case-insensitive channel name if no explicit *rcpnam* or *rcpspc* is provided.

pevtnum

Optional pointer to a list of event numbers to which the channel wishes to subscribe. There are two types of event: user events and RTR events. This parameter is used to specify all user and RTR events that the channel is to receive.

Start the list of user event numbers with `RTR_EVTNUM_USERDEF`, and the list of RTR event numbers with `RTR_EVTNUM_RTRDEF`. End the entire list with `RTR_EVTNUM_ENDLIST`. Specify a range of event numbers using the constant `RTR_EVTNUM_UP_TO` between the lower and upper (inclusive) bounds. For example, to specify the list of all user event numbers, use:

```
rtr_evtnum_t all_user_events[]={
    RTR_EVTNUM_USERDEF,
        RTR_EVTNUM_USERBASE,
        RTR_EVTNUM_UP_TO,
        RTR_EVTNUM_USERMAX,
    RTR_EVTNUM_ENDLIST
    } ;
```

For example, to specify the list of all event numbers, use:

```
rtr_evtnum_t all_events[]={
    RTR_EVTNUM_USERDEF,
        RTR_EVTNUM_USERBASE,
```

```
        RTR_EVTNUM_UP_TO,
        RTR_EVTNUM_USERMAX,
    RTR_EVTNUM_RTRDEF,
        RTR_EVTNUM_RTRBASE,
        RTR_EVTNUM_UP_TO,
        RTR_EVTNUM_RTRMAX,
    RTR_EVTNUM_ENDLIST
    } ;
```

Specify RTR_NO_PEVTNUM when the channel is to receive no events. Event names and numbers are listed in Table 2–5, RTR Event Names and Numbers.

access

An optional null-terminated string containing the access parameter. The access parameter is a security key used to authorize access to a facility by clients and servers. Specify RTR_NO_ACCESS when there is no access string.

numseg

The number of key segments defined. The *numseg* parameter is not required for client channels or callout server channels. (Callout servers always receive all messages.) Specify RTR_NO_NUMSEG when defining client channels.

A key can consist of up to RTR_MAX_NUMSEG segments.

pkeyseg

Pointer to the first block of key segment information. Only the first *numseg* elements are used. The structure of rtr_keyseg_t is:

```
typedef struct                      /* RTR Key Segment Type */
{
    rtr_keyseg_type_t ks_type ;     /* Key segment data type */
    rtr_uns_32_t     ks_length ;    /* Key segment length (bytes) */
    rtr_uns_32_t     ks_offset ;    /* Key segment offset (bytes) */
    void             *ks_lo_bound ; /* Ptr to key segment low bound */
    void             *ks_hi_bound ; /* Ptr to key segment high bound */
} rtr_keyseg_t ;
```

The data type of a key segment can be one of the following:

Table 3.9. Key Segment Data Type

Data Type	Description
rtr_keyseg_foreign_tm_id	Foreign transaction manager identifier.
rtr_keyseg_partition	Partition name, the name of the partition assigned.
rtr_keyseg_rmname	Resource manager name, the name of the foreign resource manager.
rtr_keyseg_signed	Signed
rtr_keyseg_string	ASCII string
rtr_keyseg_unsigned	Unsigned

The *pkeyseg* parameter is not required for client channels or callout server channels. (Callout servers always receive all messages.) Specify RTR_NO_PKEYSEG when defining client channels. The *ks_type*

field can be one of the data types shown in Table 3–9. The value of the offset *ks_offset* must be different for different key segments or key ranges.

If an `rtr_keyseg_t` of *rtr_keyseg_string* is specified, then it is up to the application programmer to ensure that the key value is valid for the complete range of the key length.

For example, if the key length is 4, and server code includes a statement like:

```
strcpy(keyvalue, "k");
```

with *keyvalue* passed as one of the bounds values, then potentially the bound value can differ from one open channel call to the next, because the two bytes following the "k" will contain uninitialized values but still form part of the key-bound definition. (In this case, one should clear the *keyvalue* buffer before copying the bounds values.)

A call to `rtr_open_channel( )` may be used to create a named partition or to open a server channel associated with an existing named partition. To do this, supply a partition name when opening a server channel. The *pkeyseg* argument specifies an additional item of type `rtr_keyseg_t`, assigning the following values:

- `ks_type = rtr_keyseg_partition`, indicating that a partition name is being passed
- `ks_lo_bound` should point to the null-terminated string to use for the partition name

---

Note

When using the RTR CLI, if a key-bound value length is less than the key length, the key bound is automatically null-padded to the required length. For example,

```
RTR> call rtr_open_channel/server/type=string/low=1/high=2
```

Because no key length is specified, the length defaults to four. The low and high bound values are automatically null-padded to four bytes by RTR.

The key segment array may not contain more than RTR_MAX_NUMSEG elements.

XA Usage

Specify `RTR_F_OPE_XA_MANAGED` only for a server channel. With this flag, use *ks_type* = *rtr_keyseg_rmname* to indicate that the server application provides resource manager information when a channel is open. *ks_lo_bound* should point to the null-terminated string to use for the resource manager (RM) name, which cannot contain more than 31 letters. *ks_hi_bound* should point to the null-terminated string to use for the RM-specific open string used to connect to the underlying RM. The open string cannot contain more than 255 letters. Neither *ks_length* nor *ks_offset* apply when using the flag `RTR_F_OPE_XA_MANAGED`.

Description

The `rtr_open_channel( )` call opens a channel for communication with other applications on a particular facility.

The caller of `rtr_open_channel( )` specifies the role (client or server) for which the channel is used.

For use with XA:

1. Change the `rtr_open_channel( )` call as described in the call description.
2. Remove unnecessary SQL calls from server code such as commit or rollback in a two-phase commit environment. If these calls remain in your application code, they may cause vendor-specific warnings.
3. RTR allows only one RM instance to be registered for each RTR partition.
4. Only one transaction is processed on an RTR channel at any given time. This implies that a server process or a thread of control can only open one channel to handle a single XA request.
5. Using a multithreaded server application is strongly recommended for better throughput.

Return Value

A value indicating the status of the routine. Possible status values are:

RTR_STS_ACPNOTVIA	RTR ACP no longer a viable entity, restart RTR or application
RTR_STS_ALLSRVSTRCT	All partition instances must agree on the setting of STRCT_SHD_ORDER
RTR_STS_BYTLMNSUFF	Insufficient process quota bytlm, required 100000
RTR_STS_DTXOPENFAIL	Distributed transaction request to open a session to the RM has failed
RTR_STS_DUPLRMNAME	Duplicate RM partition name
RTR_STS_ERROPEJOU	Error opening journal file
RTR_STS_INSVIRMEM	Insufficient virtual memory
RTR_STS_INVACCESS	Invalid access argument
RTR_STS_INVCHANNEL	Invalid channel argument
RTR_STS_INVEVTNUM	Invalid evtnum argument
RTR_STS_INVFACNAM	Invalid facnam argument
RTR_STS_INVFLAGS	Invalid flags argument
RTR_STS_INVIMPLCTSTRT	Implicit start transaction disallowed by channel properties
RTR_STS_INVKSLENGTH	Invalid ks_length argument
RTR_STS_INVKSTYPE	Invalid ks_type argument
RTR_STS_INVNUMSEG	Invalid numseg argument
RTR_STS_INVPKEYSEG	Invalid pkeyseg argument
RTR_STS_INVRCPNAM	Invalid rcpnam argument
RTR_STS_INVRMNAME	Invalid resource manager name
RTR_STS_INVSVRCLIFLG	Either both /Client and /Server flags were supplied or they were missing
RTR_STS_JOUACCDEN	No access to journal for attempted operation: permission denied
RTR_STS_JOUNOTFOU	Journal not found
RTR_STS_NOACP	No RTRACP process available

RTR_STS_NONRECPAR	Non-recoverable partition is no longer active - channel closed automatically
RTR_STS_OK	Normal successful completion
RTR_STS_RMSTRINGLONG	Resource manager open or close string too long
RTR_STS_TOOMANCHA	Too many channels already opened
RTR_STS_MISSINGREQFLG	A required flag is missing

Example

Examples show the following:

- A simple client application
- A simple server application
- An application using XA
- An application using partition names

Example 3.1. Client Application

```
        rtr_channel_t channel;
        rtr_status_t status;
        rtr_string_t user_name;
/* Get the user's name through login or other user interface
 */
        user_name = <input data>

/* Open client application's channel to the router;
 * use the facility named 'CCardPurchases', and the user's
 * name to identify this client.
 *
 * This client will receive messages only, no events,
 * and is going to use a foreign transaction manager
 * that implements the X/Open standard transaction
 * formats.
 */
status = rtr_open_channel(
                    &channel,
                    RTR_F_OPE_CLIENT | RTR_F_OPE_FOREIGN_TM,
                    "CCardPurchases",
                    user_name,
                    RTR_NO_PEVTNUM,
                    RTR_NO_ACCESS,
                    RTR_NO_NUMSEG ,
                    RTR_NO_PKEYSEG );

check_status(status);
```

Example 3.2. Server Application

```
*******************************************************
/* Open a channel in a server application. This server will
 * handle only records where the last name begins with A.
 * It also wants an explicit message sent when it is time
 * to prepare the transaction, and one when it is time to
```

```
 * vote whether to accept or reject the transaction.
 */
rtr_channel_t    channel;
rtr_status_t     status;
rtr_keyseg_t     p_keyseg[1];
rtr_string_t     last = "A";

/*
 * Use this rtr_keyseg_t structure to define this server as
 * handling only those records whose last name begins
 * with 'A'.
 */

p_keyseg[0].ks_type = rtr_keyseg_string;
p_keyseg[0].ks_length = 1;
p_keyseg[0].ks_offset = 0;
p_keyseg[0].ks_lo_bound = last;
p_keyseg[0].ks_hi_bound = last;

/* Open the channel as a server that wants explicit ACCEPT and
 * PREPARE messages. It is a member of the CcardPurchases
 * facility, accepts no events (only messages) and we are
 * sending 1 rtr_keyseg_t structure that defines those
 * messages to be handled by this server.
 *
 * Note also that we are specifying that this channel
 * will be 'XA managed'; that is, the transaction manager
 * will be one that implements the X/Open standard.
 */
status = rtr_open_channel(
                    &channel,
    RTR_F_OPE_SERVER | RTR_F_OPE_EXPLICIT_ACCEPT |
    RTR_F_OPE_EXPLICIT_PREPARE | RTR_OPE_XA_MANAGED,
                "CCardPurchases",
                NULL,
                RTR_NO_PEVTNUM,
                RTR_NO_ACCESS,
                1,
                p_keyseg);
check_status(status);
```

Using RTR with XA

The snippets from the sample server applications show use of the RM information, the XA flag, and commenting out RM commits and rollbacks.

New XA example, for V4.1 and later

Starting with RTR Version 4.1, when a server application opens a new channel it does not have to specify the `RTR_F_OPE_XA_MANAGED` flag and RM name along with the RM's attributes such as `open_string` in order to invoke RTR XA service. The server application just has to specify the name of a partition that is associated with a specific RM, provided that the user specifies an RM name when creating the partition. All transactions processed through this channel will be managed by the RTR XA service.

Impact on Server Application

Using an RTR XA service has some impact on existing server applications, as follows:

- RTR will not present messages of type `mt_uncertain` to server applications. The server application does not have to replay transactions during the recovery. All transactions will be recovered by RTR when the facility is created.
- The server application does not need to explicitly commit or roll back the transactions with the underlying resource manager because transactions are managed directly by RTR using the XA protocol.

Example 3.3 shows how to open a new channel using RTR V4.1:

Example 3.3. Sample XA Server Application, Version 4.1 and Later

```
srv_key[0].ks_type = rtr_keyseg_partition;
    srv_key[0].ks_length = 0; /* N/A */
    srv_key[0].ks_offset = 0; /* N/A */
    srv_key[0].ks_lo_bound = &partition_name[0]; /* null terminated */

  flag = RTR_F_OPE_SERVER |
         RTR_F_OPE_EXPLICIT_PREPARE |
         RTR_F_OPE_EXPLICIT_ACCEPT;

status = rtr_open_channel(&s_chan,
                          flag,
                          reply_msg.fac_name,
                          NULL,            /* rcpnam */
                          &pevtnum,
                          RTR_NO_ACCESS,
                          num_seg,         /* numseg */
                          srv_key);        /* key range */
```

However, if the server application is running a version of RTR prior to RTR V4.0, the server application must specify the RTR_F_OPE_XA_MANAGED flag, the RM's name, and other RM attributes such as `open_string`. You must overload the `rtr_keyset_t` data structure with the RM-specific information and then pass it when creating an RTR channel, as shown in Example 3.4.

Example 3.4. Sample XA Server Application Prior to Version 4.1

```
        srv_key[0].ks_type = rtr_keyseg_unsigned;
        srv_key[0].ks_length = sizeof(rtr_uns_8_t);
        srv_key[0].ks_offset = 0;
        srv_key[0].ks_lo_bound = &low;
        srv_key[0].ks_hi_bound = &high;
        srv_key[1].ks_type = rtr_keyseg_rmname;
        srv_key[1].ks_length = 0; /* N/A */
        srv_key[1].ks_offset = 0; /* N/A */
        srv_key[1].ks_lo_bound = &rm_name[0];        /* null terminated */
        srv_key[1].ks_hi_bound = &xa_open_string[0]; /* null terminated */
flag = RTR_F_OPE_SERVER |
                    RTR_F_OPE_EXPLICIT_PREPARE |
                    RTR_F_OPE_EXPLICIT_ACCEPT |
                    RTR_F_OPE_XA_MANAGED;

                status = rtr_open_channel(&s_chan,
                                  flag,
                                  reply_msg.fac_name,
                                  NULL,            /* rcpnam */
                                  &pevtnum,
                                  RTR_NO_ACCESS,
```

```
                                    num_seg,          /* numseg */
                                    srv_key);         * key range */
/* Demonstrate use of partition names                */
/*                                                   */
/*                                                   */

#include     "rtr.h"
#include     <stdio.h>

main()
{

/* This program will open a server channel. Servers
 * need to identify the partition they will be operating
 * on by passing information coded in the pkeyseg argument.
 * If the partition already exists and its name is known,
 * it suffices to specify the partition name. If this is
 * not the case, then the partition must be specified by
 * describing the key segments. In the latter case, name
 * information is optional. If present, the new partition
 * will receive the specified name, otherwise a default
 * name will be generated.                           */
/*                                                   */
 * This program assumes the presence of a partition named
 * par_test in the facility fac_test and opens a server
 * channel to it. Create the partition prior to running
 * the program, e.g., */
/*                                                   */
/* RTR> create partition par_test/facility=fac_test  */
/*                                                   */
rtr_channel_t      AChannel;
const char         *pszFacilityName = "fac_test";
const char         *pszPartitionName = "par_test";
rtr_status_t       status;
rtr_ope_flag_t     flags = RTR_F_OPE_SERVER;
rtr_keyseg_t       partition_info;

partition_info.ks_type     = rtr_keyseg_partition;
partition_info.ks_lo_bound = (rtr_pointer_t)pszPartitionName;
partition_info.ks_hi_bound = NULL;
                         /* Must be NULL             */

status = rtr_open_channel(
                         &AChannel,
                         flags,
                         pszFacilityName,
                         RTR_NO_RCPNAM,
                         RTR_NO_PEVTNUM,
                         RTR_NO_ACCESS,
                         1,
                         &partition_info);

/* Call rtr_receive_message() to receive completion status  */
}
```

See Also

rtr_close_channel( )

rtr_receive_message

rtr_receive_message — Receive a message from RTR or the application.

Format

```
status = rtr_receive_message (pchannel, flags, prcvchan, pmsg, maxlen, timoutms, p
```

Argument	Data Type	Access
status	rtr_status_t	write
pchannel	rtr_channel_t	write
flags	rtr_rcv_flag_t	read
prcvchan	rtr_channel_t	read
pmsg	rtr_msgbuf_t	write
maxlen	rtr_msglen_t	read
timoutms	rtr_timout_t	read
pmsgsb	rtr_msgsb_t	write

C Binding

```
rtr_status_t rtr_receive_message (
        rtr_channel_t *pchannel ,
        rtr_rcv_flag_t flags ,
        rtr_channel_t *prcvchan ,
        rtr_msgbuf_t pmsg ,
        rtr_msglen_t maxlen ,
        rtr_timout_t timoutms ,
        rtr_msgsb_t *pmsgsb
        )
```

Arguments

pchannel

The channel identifier on which the message was received.

flags

No flags are currently defined. Specify RTR_NO_FLAGS for this parameter.

prcvchan

A pointer to a list of channels on which a receive is required. This parameter can be used to select a subset of channels on which messages can be received. Terminate the list with RTR_CHAN_ENDLIST.

If no selection is required, that is, a receive from any open channel is acceptable, specify RTR_ANYCHAN for this parameter.

Note

See the restriction on using RTR_ANYCHAN with RTR V2 applications in the *VSI Reliable Transaction Router System Manager's Manual*

pmsg

Required pointer to the user buffer where the received message is written.

maxlen

Size allocated in the user buffer for received messages, in bytes.

timeoutms

Receive timeout specified in milliseconds. If the timeout expires, the call completes with status `RTR_STS_TIMOUT`.

If no timeout is required, specify `RTR_NO_TIMOUTMS`.

pmsgsb

Pointer to a message status block describing the received message. The message status block is shown in Example 3.5.

Example 3.5. RTR Message Status Block

```
typedef struct         /* RTR message status block */
{
    rtr_msg_type_t msgtype;
    rtr_usrhdl_t usrhdl;
    rtr_msglen_t msglen;
    rtr_tid_t tid;
    rtr_evtnum_t evtnum;
} rtr_msgsb_t ;
```

The *msgtype* field can assume one of the values listed in Table 2.2, RTR Received Message Types for Server Applications and Table 2.3, RTR Received Message Types for Client Applications.

The *usrhdl* field contains the value supplied with a call to `rtr_set_user_handle( )`.

The *msglen* field contains the length of the data stored in the user buffer after the call has been executed.

The *tid* field contains the RTR unique ID for the transaction to which this received message belongs.

The *evtnum* field contains the event number if the *msgtype* field is `rtr_mt_rtr_event` or `rtr_mt_user_event`.

Description

The `rtr_receive_message( )` call is used to receive a message.

The caller must have previously opened at least one channel (via `rtr_open_channel( )` or `rtr_request_info( )`).

By default, this function waits for a message to become available if no message is currently ready to be received.

Upon successful return (RTR_STS_OK), the message status block pointed to by *pmsgsb* contains the description of the message received.

When a client application calls `rtr_send_to_server`, RTR sends the message from frontend to router. It is the router's job to find out which key range the message belongs to (by looking at the key field in the application message), and then to forward the message to the backend node where the server application for this key range is running. If the router does not know of a backend that has a server running for this key range, then the router aborts the transaction. In this case, the client application receives an `rtr_mt_rejected` message for this transaction with status RTR_STS_NODSTFND.

If a client application receives an RTR_STS_NODSTFND error, then the client can try to resend the transaction, as the cause may have been only temporary. Note that the reasons the router cannot find a backend node with an appropriate server include:

1. The application server for this key range has not been started.

2. The link between the router and backend has gone down.

3. In unusual circumstances, a transaction can be rejected with RTR_STS_ NODSTFND status after the client calls `rtr_accept_tx`. This can occur for transactions with multiple participants and no timeout specified where the link between the router (which is quorate) and one of the backend participants has gone down for a period greater than the router's transaction replay timeout period. (This can occur even if the messages in the transaction had all been sent with the `RTR_F_SEN_EXPENDABLE` flag set.)

Return Value

A value indicating the status of the routine. Possible status values are:

RTR_STS_ACPNOTVIA	RTR ACP no longer a viable entity, restart RTR or application
RTR_STS_BYTLMNSUFF	Insufficient process quota bytlm, required 100000
RTR_STS_INVCHANNEL	Invalid channel 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_NOXACHAN	No XA channel available
RTR_STS_OK	Normal successful completion
RTR_STS_SRVDCLSBY	Successful server declaration, but as standby
RTR_STS_TIMOUT	Call to rtr_receive_message timed out
RTR_STS_TRUNCATED	Buffer too short for msg. Message has been truncated

Example

```
status = rtr_receive_message(
                        &channel,
                        RTR_NO_FLAGS,
                        RTR_ANYCHAN,
                        &receive_msg,
                        sizeof(receive_msg),
```

```
                        receive_time_out,
                        &msgsb);
        check_status( "rtr_receive_message", status );

/* The rtr_msgsb_t tells us what type of
 * message we are receiving. This server has asked to
 * be notified when it is time to prepare the transaction.
 * It should also handle other message types, as well.
 */
        if (msgsb.msgtype == rtr_mt_prepare)
        {
                // Do the work necessary to prepare the transaction
                // before committing.
```

See Also

`rtr_broadcast_event( )`

`rtr_accept_tx( )`

`rtr_open_channel( )`

`rtr_reject_tx( )`

`rtr_send_to_server( )`

rtr_reject_tx

rtr_reject_tx — Reject the transaction currently active on a channel.

Format

`status = rtr_reject_tx (channel, flags, reason)`

Argument	Data Type	Access
status	rtr_status_t	write
channel	rtr_channel_t	read
flags	rtr_rej_flag_t	read
reason	rtr_reason_t	read

C Binding

```
rtr_status_t rtr_reject_tx (
        rtr_channel_t channel ,
        rtr_rej_flag_t flags ,
        rtr_reason_t reason
        )
```

Arguments

channel

The channel identifier (returned earlier by `rtr_open_channel( )`).

flags

No flags are currently defined. Specify RTR_NO_FLAGS for this parameter.

reason

The reason for the rejection. This rejection reason is returned to the other participants in the transaction. It is returned in the reason field of the structure rtr_status_data_t with the rtr_mt_rejected message. Specify RTR_NO_REASON if no reason is to be supplied.

Description

The rtr_reject_tx( ) call rejects the transaction that is active on the specified channel.

When rtr_reject_tx( ) returns, the channel is no longer associated with the transaction.

Once an rtr_accept( ) has been called by the server application, the rtr_reject_tx( ) call is not allowed until the first message of the next transaction is received. An attempt to call rtr_reject_tx( ) yields an RTR_STS_TXALRACC return status.

Return Value

A value indicating the status of the routine. Possible status values are:

RTR_STS_CHANOTOPE	Channel not opened
RTR_STS_DLKTXRES	Deadlock detected, transaction rescheduled. This status is returned to a client in the status field of a message of type rtr_mt_rejected if a transaction currently being processed has been aborted because of a deadlock with other transactions using the same servers. RTR replays the transaction after the deadlock has been cleared. This condition can be caused by either a classic database deadlock or a potential deadlock that RTR tries to avoid in cases such as concurrent server death or server role change. For more details, see the section in the *VSI Reliable Transaction Router Application Design Guide*, Handling Error Conditions.
RTR_STS_INVCHANNEL	Invalid channel argument
RTR_STS_INVFLAGS	Invalid flags argument
RTR_STS_OK	Normal successful completion
RTR_STS_TXALRACC	Transaction already accepted
RTR_STS_TXNOTACT	No transaction currently active on this channel

Example

```
rtr_uns_32_t MT_LAST_NAME = 678;
          /* Defined in user's .h file. */
if (last_name == null)
{
/* Missing last name! Not everything is ready for
```

```
 * committing the current transaction (e.g., through
 * validations), and so wishes to reject it, rather than
 * to commit it.
 */
          status = rtr_reject_tx(
                  channel,
                  // Same channel it came in on.
                  RTR_F_REJ_RETRY,
                  // Retry from msg1 of txn.
          MT_LAST_NAME );
                  // User-defined error code.
        check_status(status);
        return;
        }
```

See Also

`rtr_open_channel( )`

`rtr_accept_tx( )`

rtr_reply_to_client

rtr_reply_to_client — Send a server's reply to a client's transactional message.

Format

`status = rtr_reply_to_client (channel, flags, pmsg, msglen, msgfmt)`

Argument	Data Type	Access
status	rtr_status_t	write
channel	rtr_channel_t	read
flags	rtr_rep_flag_t	read
pmsg	rtr_msgbuf_t	read
msglen	rtr_msglen_t	read
msgfmt	rtr_msgfmt_t	read

C Binding

```
rtr_status_t rtr_reply_to_client (
        rtr_channel_t channel ,
        rtr_rep_flag_t flags ,
        rtr_msgbuf_t pmsg ,
        rtr_msglen_t msglen ,
        rtr_msgfmt_t msgfmt
        )
```

Arguments

channel

The channel identifier (returned earlier by `rtr_open_channel( )`).

flags

Table 3.10 shows the flags defined for this call

Table 3.10. Reply To Client Flag

Flag	Description
RTR_F_REP_ACCEPT	The transaction is accepted by this server. This is equivalent to sending a reply to the server and immediately following it with a call to `rtr_accept_tx( )`. This is useful in those cases where the sender knows that the transaction is definitely acceptable.
RTR_F_REP_FORGET	Set to prevent receipt of any more messages or completion status associated with the transaction after it has been accepted. Using this flag requires that the `RTR_F_ACC_ FORGET` flag be set in the `rtr_accept_tx` call, indicating that the transaction is to be accepted.
RTR_F_REP_INDEPENDENT	Set to indicate that this transaction is independent; can only be used with `RTR_F_REP_ACCEPT`. (See Section 2.15.4, Transaction Independence, for further information.)

Specify `RTR_NO_FLAGS` for this parameter if no flags are required.

pmsg

Pointer to the reply message to be sent.

msglen

Length of the message to be sent, in bytes.

msgfmt

Message format description. *msgfmt* 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. See Section 2.14, RTR Applications in a Multiplatform Environment, for information on defining a message format description.

This parameter is optional. Specify `RTR_NO_MSGFMT` if the message content is platform independent, or other hardware platforms will not be used.

Description

The `rtr_reply_to_client( )` call sends a transactional message back to the client that started the transaction.

The caller must first obtain a server channel (using the `rtr_open_channel( )` call) and must have received a message from a client using the `rtr_receive_message( )` call.

Once an `rtr_accept_tx( )` has been called by the server application, the `rtr_reply_to_client( )` call is not allowed until the first message of the next transaction is

received. An attempt to call `rtr_reply_to_client( )` yields an RTR_STS_TXALRACC return status.

Return Value

A value indicating the status of the routine. Possible status values are:

RTR_STS_CHANOTOPE	Channel not opened
RTR_STS_INSVIRMEM	Insufficient virtual memory
RTR_STS_INVCHANNEL	Invalid channel argument
RTR_STS_INVFLAGS	Invalid flags argument
RTR_STS_INVMSGFMT	Invalid msgfmt argument
RTR_STS_INVMSGLEN	Invalid msglen argument
RTR_STS_NOACP	No RTRACP process available
RTR_STS_OK	Normal successful completion
RTR_STS_TXALRACC	Transaction already accepted
RTR_STS_TXNOTACT	No transaction currently active on this channel

Example

```
/* The purchase_msg structure is defined in the user's
 * application header file.
 */
typedef struct {
                rtr_uns_8_t my_msg_type;
                string31 last_name;
                rtr_uns_32_t order_total;
                rtr_uns_32_t shipping_amt;
                string7 user_id;
} purchase_msg;

purchase_msg purch_msg;
/* The client has made a request on the server; the server
 * has fulfilled this request, and now needs to let the
 * client know the result.
 *
 * In this case, the client has asked the server to total
 * the purchases in the user's shopping cart. The server
 * is accepting the transaction at this time as well, without
 * being explicitly asked to.
 */
purch_msg.my_msg_type = MY_TOTAL_PURCHASES;
purch_msg.last_name = cust_last_name;
.
.       Fill the struct based on database query or calculations.
.
status = rtr_reply_to_client (
                            channel,
                            RTR_F_REP_ACCEPT,
                            &purch_msg,
                            sizeof(purch_msg),
                            RTR_NO_MSGFMT);
        check_status(status);
```

See Also

`rtr_receive_message( )`

`rtr_open_channel( )`

`rtr_accept_tx( )`

rtr_request_info

rtr_request_info — Request information about the RTR environment.

Format

`status = rtr_request_info (pchannel, flags, infcla, selitm, selval, getitms)`

Argument	Data Type	Access
status	rtr_status_t	write
pchannel	rtr_channel_t	write
flags	rtr_req_flag_t	read
infcla	rtr_infoclass_t	read
selitm	rtr_itemcode_t	read
selval	rtr_selval_t	read
getitms	rtr_itemcode_t	read

C Binding

```
rtr_status_t rtr_request_info (
        rtr_channel_t *pchannel ,
        rtr_req_flag_t flags ,
        rtr_infoclass_t infcla ,
        rtr_itemcode_t selitm ,
        rtr_selval_t selval ,
        rtr_itemcode_t getitms
        )
```

Arguments

pchannel

Pointer to the channel opened by a successful call to `rtr_request_info( )`.

flags

No flags are defined for this call. Use `RTR_NO_FLAGS` for this parameter.

infcla

A null-terminated text string that specifies the type of information for which data are requested. The table below lists information types and their specifying information class strings. Within an information class, you retrieve a specific datum with *selitm*, *selval*, and *getitms* parameters specified as strings. Data

returned by `rtr_request_info` are valid only under certain conditions as listed in the table below. For example, to obtain information about a node, use the "rtr" string; RTRACP must be running for data to be valid.

When the gcs information class is used with `rtr_request_info( )`, the first message returned is unique; it includes attributes of the requested information, and overhead information present because the backend requests information from many routers.

The first message returned with the gcs information class contains at least three fields separated by the null character (ASCII 0).

- The first field indicates if the gcs data is complete or incomplete. If the backend gathering the information cannot gather all the information from the routers, then the data may be incomplete; otherwise, the data is complete. The first field thus contains one of two literal strings: either "complete data" or "incomplete data."

- The second field indicates how many seconds remain until the backend's cache is updated. Since the backend must request information from all routers, it caches the information to avoid extra overhead for every request. (See the SET NODE /INFO_CACHE_LIFETIME qualifier for more information on the cache.) The field has the form "n seconds until update," where *n* can be a value from 0 to UINT_MAX. Note that the word "seconds" remains plural for all cases of *n*.

- The third field indicates the age, in seconds, of the backend's cache. The age is the number of seconds since the cache was refreshed with current information. The field has the form: "*n* seconds old," where *n* may take a value between 0 to UINT_MAX. Note that the word "seconds" remains plural for all cases of *n*.

- The fourth and following fields may be present to explain why information is incomplete. If the information is complete, these fields are not present. If the information is incomplete, one or more of these fields may contain strings meant to be read by humans. These strings can, for example, be logged in a log file.

  Applications using the gcs information class may choose to parse the first message to store the information's attributes, or ignore the first message and acquire the information found in the second and subsequent messages. The application must not assume that the first message contains the requested information. For an example, see the last example for this C API call.

Table 3.11. Information Classes

For this type of information:	Use this Information Class string:	To obtain valid data:	For available items and strings, see:
Application process	prc	An application process must have been started (`rtr_open_channel` called).	Table 3.12
Client process	cli	A client channel must have been opened.	Table 3.13
Facility	fac	A facility must be defined.	Table 3.14
Global Configuration and Status	gcs	The gcs information class can only be accessed from a backend and VSI recommends that the backend be connected	Table 3.15

For this type of information:	Use this Information Class string:	To obtain valid data:	For available items and strings, see:
		to all routers. If the backend is disconnected from one or more routers, gcs information will still be available but may be incomplete. This incomplete status is indicated in the first message returned by `rtr_request_info`. For additional information, see the description earlier on the gcs information class and the gcs example.	
Key segment	ksg	A server channel must have been opened.	Table 3.16
Link to a node	lnk	A facility must be defined.	Table 3.17
Node or RTRACP	rtr	RTRACP must be running.	Table 3.18
Partition on a backend	bpt	A server channel must have been opened.	Table 3.19
Partition on a router	rpt	A server channel must have been opened.	Table 3.20
Partition history	hpt	A server channel must have been opened.	Table 3.21
Server process	srv	A server channel must have been opened.	Table 3.22
Transaction on a backend	btx	A transaction must be in progress on the backend.	Table 3.23
Transaction on a frontend	ftx	A client application must have a transaction in progress.	Table 3.24
Transaction on a router	rtx	A transaction must be in progress on the router.	Table 3.25

selitm

Null-terminated text string giving the strings used to select information such as facility name or transaction ID. Use this argument to reduce the amount of information returned. If you specify a null string (""), all available information for the class is returned. A string containing multiple items should be a comma-separated list. Some SHOW commands display the same data. For example, to obtain the RTR version number (displayed by SHOW RTR/VERSION), use the string `rtr_version_string` from the "rtr" information class.

The tables are in alphabetical order by Information Class, and grouped by function within each table.

Table 3.12. Application Process ("prc") Strings

For this selitm:	Use this string:
Process-id	process_id

For this selitm:	Use this string:
Process Name	process_name

Table 3.13. Client Process ("cli") Strings

For this selitm:	Use this string:
Process-id	dpb_pid
Facility	fdb_f_name
Channel	dpb_chan
Flags	dpb_dclflg
State	dpb_req_sts
rcpnam	dpb_evtnam
User Events	dpb_user_evtnum
RTR Events	dpb_rtr_evtnum

Table 3.14. Facility ("fac") Strings

For this selitm:	Use this string:
Facility	fdb_f_name
Frontend	fdb_attr.fdb_attr_bits.is_fe
Router	fdb_attr.fdb_attr_bits.is_rtr
Backend	fdb_attr.fdb_attr_bits.is_be
Reply Checksum	fdb_attr.fdb_attr_bits.reply_enabled
Router call-out	fdb_attr.fdb_attr_bits.tr_call_out
Backend call-out	fdb_attr.fdb_attr_bits.be_call_out
Load balance	fdb_attr.fdb_attr_bits.feshare
Quorum-check off	fdb_attr.fdb_attr_bits.qrt_chk
FE -> TR	fdb_trsrch
Router quorate	fdb_state.fdb_state_bits.tr_qrt
Backend quorate	fdb_state.fdb_state_bits.be_qrt
Quorum threshold	fdb_iqt_cnt
Min bcst rate	fdb_cn_fct_min_brd_out_rate
Frontends connected	fdb_fecnt
Frontends allowed	fdb_fecdt
Load coordinator	fdb_status.fdb_status_bits.qm_be
Quorate routers	fdb_trtot
Total Frontends	fdb_fetot
Current Credit	fdb_curcdt
FE -> TR	fdb_trsrch
Link to	fac_ndb
Frontend	fac_fe.rol_bits.rol_cfg
Router	fac_tr.rol_bits.rol_cfg

For this selitm:	Use this string:
Backend	fac_be.rol_bits.rol_cfg
Router -> Frontend	fac_reasons.fac_reason_bits.trfelnk
Frontend -> Router	fac_reasons.fac_reason_bits.fetrlnk
Backend -> Router	fac_reasons.fac_reason_bits.betrlnk
Router -> Backend	fac_reasons.fac_reason_bits.trbelnk
Router quorate	fac_tr.rol_bits.rol_quorum
Backend -> Router	fac_reasons.fac_reason_bits.betrlnk
Router -> Backend	fac_reasons.fac_reason_bits.trbelnk
Router quorate	fac_tr.rol_bits.rol_quorum
Backend quorate	fac_be.rol_bits.rol_quorum
Router current	fac_tr.rol_bits.rol_cur
Backend coordinator	fac_be.rol_bits.rol_qmaster

Table 3.15. Global Configuration and Status ("gcs") Strings

For this selitm:	Use this string:
Node name	gsc_node
Facility Name	gsc_fac
Role	gsc_role
Cluster	gsc_clust
Operating System	gsc_os
RTR version	gsc_version
Connection State	gsc_connected
Detected Problem Name	gsc_name
Detected Problem Message	gsc_mesg
Detected Problem Severity	severity
Partition Name	gpt_ptn
Partition State	gpt_state

Table 3.16. Key Segment ("ksg") Strings

For this selitm:	Use this string:
Facility	fdb_f_name
Data Type	ksd_dtyp
Length	ksd_length
Offset	ksd_offset

Table 3.17. Node Links ("lnk") Strings

For this selitm:	Use this string:
To Node	ndb_name
Address	ndb_idp

For this selitm:	Use this string:
Outgoing message sequence nr	ndb_xcnt
Incoming message sequence nr	ndb_rcnt
Current receive buffer size	ndb_credit
Current transmit buffer size	ndb_cdt_out
Current number of link users	ndb_reasons
Write buffer timed out	ndb_status.wbuftmo
Write buffer full, may be sent	ndb_status.wbufrdy
Write buffer allocated	ndb_status.wbufalc
I/O error detected in write	ndb_status.wrerror
I/O error detected in read	ndb_status.rderror
Pipe temporarily blocked	ndb_status.blocked
Connection broken	ndb_status.aborted
Write issued, not completed	ndb_status.writing
Read is pending	ndb_status.reading
Node initiated the connection	ndb_status.initiator
Connection established	ndb_status.connected
Connection in progress	ndb_status.connecting
Node is configured	ndb_status.configured
Autoisolation enabled	ndb_attr.attr_bits.isol_ebld
Link disabled	ndb_attr.attr_bits.disabled
Link isolated	ndb_attr.attr_bits.isolated
In facility	fac_ifn
Frontend	fac_fe.rol_bits.rol_cfg
Router	fac_tr.rol_bits.rol_cfg
Backend	fac_be.rol_bits.rol_cfg
Router -> Frontend	fac_reasons.fac_reason_bits.trfelnk
Frontend -> Router	fac_reasons.fac_reason_bits.fetrlnk
Backend -> Router	fac_reasons.fac_reason_bits.betrlnk
Router -> Backend	fac_reasons.fac_reason_bits.trbelnk
Router quorate	fac_tr.rol_bits.rol_quorum
Backend quorate	fac_be.rol_bits.rol_quorum
Router current	fac_tr.rol_bits.rol_cur
Backend coordinator	fac_be.rol_bits.rol_qmaster

Table 3.18. Node and ACP ("rtr") Strings

For this selitm:	Use this string:
Network state	ncf_isolated
Auto isolation	ncf_isol_ebld

For this selitm:	Use this string:
Inactivity timer/s	ncf_lw_inact
RTR Version Number	rtr_version_string

Table 3.19. Partition on a Backend ("bpt") Strings

For this selitm:	Use this string:
Partition name	$name
Facility	ppb_fdbptr
State	ppb_pst.prt_ps
Low Bound	ppb_krd.krd_low_bound
High Bound	ppb_krd.krd_high_bound
Active Servers	srb_active_q.#crm_server_block
Free Servers	srb_free_q.#crm_server_block
Transaction presentation	tx_presentation_state
Last Rcvy BE	last_lcl_rec_be
Txns Active	tkb_q.#crm_tx_kr_block
Txns Rcvrd	rec_be_txs
Failover policy	ppb_failover_policy
Key range ID	ppb_krid

Table 3.20. Partition on a Router ("rpt") Strings

For this selitm:	Use this string:
Facility	fdb_f_name
State	krb_sts
Low Bound	krb_low_bound
High Bound	krb_high_bound
Failover policy	krb_failover_policy
Backends	bpsb_ndbptr
States	bpsb_pst.prt_ps
Primary Main	krb_pri_act_bpsbptr.bpsb_ndbptr
Shadow Main	krb_sec_act_bpsbptr.bpsb_ndbptr

Table 3.21. Partition History ("hpt") Strings

For this selitm:	Use this string:
Partition name	$name
Facility	phr_fdb
Low Bound	phr_krd.krd_low_bound
High Bound	phr_krd.krd_high_bound
Creation time	phr_creation_time

Table 3.22. Server Process ("srv") Strings

For this selitm:	Use this string:
Process-id	dpb_pid
Facility	fdb_f_name
Channel	dpb_chan
Flags	dpb_dclflg
State	ppb_pst.prt_ps
Low Bound	ppb_krd.krd_low_bound
High Bound	ppb_krd.krd_high_bound
rcpnam	dpb_evtnam
User Events	dpb_user_evtnum
RTR Events	dpb_rtr_evtnum
Partition-Id	dpb_krid

Table 3.23. Transaction on a Backend ("btx") Strings

For this selitm:	Use this string:
Tid	tb_txdx.tx_id
Facility	fac_id
FE-User	tb_txdx.fe_user
State	state
Start time	tb_txdx.tx_start_time
Router	tr_ndbptr
Invocation	invocation
Active-Key-Ranges	#crm_tx_kr_block
Recovering-Key-Ranges	#crm_tr_block
Total-Tx-Enqs	nr_tx_enqs
Key-Range-Id	kr_id
Server-Pid	pid
Server-State	sr_state
Journal-Node	jnl_node_id
Journal-State	jnl_state
First-Enq	first_enq_nr
Nr-Enqs	nr_enqs
Nr-Replies	nr_replys

Table 3.24. Transaction on a Frontend ("ftx") Strings

For this selitm:	Use this string:
Tid	tb_txdx.tx_id
Facility	fac_id
FE-User	tb_txdx.fe_user

For this selitm:	Use this string:
State	state
Start time	tb_txdx.tx_start_time
Router	tr_ndbptr
Nr-Enqs	enqs_from_rq
Nr-Replies	replys_rcvd

Table 3.25. Transaction on a Router ("rtx") Strings

For this selitm:	Use this string:
Tid	tb_txdx.tx_id
Facility	fac_id
FE-User	tb_txdx.fe_user
State	state
Start time	tb_txdx.tx_start_time
FE-Connected	fe_ndbptr
Total-Tx-Enqs	nr_tx_enqs
First-Enq	first_enq_nr
Nr-Enqs	nr_enqs
Backend	be_ndbptr
Key-Range-State	kr_state
Key-Range-Id	kr_id
Journal-State	be_state

selval

Null-terminated text string; contains a value for the item named in *selitm*. For example, if *selitm* specifies *fac_id* indicating that a facility name is used for the selection, and *selval* contains the string "TESTFAC", then only information for facility TESTFAC is returned. Wildcards can be used in this specification.

getitms

Null-terminated text string containing a comma-separated list of items whose values are returned. For each instance that matches the selection criterion, the values of the items specified by *getitms* are returned in a message of type `rtr_mt_request_info`.

Description

An application program can use the `rtr_request_info()` call to interrogate the RTR environment and retrieve information about facilities, transactions, key ranges, and so on. The call accesses data maintained by RTR on behalf of application programs, and data maintained by the RTR ACP itself.

The way to obtain data is to specify the requirement as parameters to `rtr_request_info()`. RTR then opens a channel on which the requested information can be received by calling `rtr_receive_message()` on the channel. The channel is automatically closed when the requested data (if any) has been completely delivered (that is, an `rtr_mt_closed` message is received on the channel.) You may close the channel earlier, if no more information is needed, by calling `rtr_close_channel()`.

The selection criteria specify an information class, a select item and a value. This is like doing a table lookup, where the class represents the specific table, and the select item and value represent the row and column in the table. For example, the following statement: `rtr_request_info/channel=I/infcla=rtr/selitm="", selval="*" getitms=rtr_version_string` requests information from the RTR (rtr) information class.

The `rtr_request_info()` call accesses the RTR tables in memory as follows:

1. The *infcla* parameter selects the class to be accessed, for example "rtr".
2. The *selitm* parameter names the row of the RTR table in memory to be accessed. This can be a null string, for example `selitms=""` to retrieve all data for the class.
3. The *selval* parameter defines what to search for in the row. For example, in a table containing information about backend transactions, if *selitm* specifies *fac_id* indicating that a facility name is the selection criterion, and if *selval* contains the value "TESTFAC", RTR selects only transactions for the facility TESTFAC.
4. The *getitms* parameter specifies the items to be returned from the selected row(s). In the example of a table containing information about backend transactions, `rtr_request_info` can specify transaction ID and transaction start time. The data for these items are returned for all transactions matching the selection criteria.

The results of the selection are returned as none, one, or more messages of type `rtr_mt_request_info`, one message being returned for each selected row in the table (in a btx example, one message for each backend transaction).

The contents of these messages are defined by the *getitms* parameter. For example, if three item names specified for *getitms* are "item_1,item_2,item_3", then the corresponding **`rtr_mt_request_info`** message or messages contain three concatenated and null-terminated strings that are the values of those fields, "value1\0value2\0value3\0".

Casing of Text when Using the GCS Infoclass

- Backend and router node names will be in the case as entered on the backend where the rtr_request_info( ) call was issued
- Facility names will be in the case as entered on the backend where the rtr_request_info( ) call was issued
- If the frontend node names are all entered in the same case on all the routers, then their names will be in that case
- If the frontend node names are entered in different cases on different routers, then the frontend node names can have the case as entered on any of the routers. There is no guarantee of which router(s) the case will come from.
- Status problem names and messages will be in the case as they came from the node reporting the problem
- Generated keywords including role (frontend,router,backend) and connection status (ncf_conn, ncf_disconn) will be in lowercase

Return Value

A value indicating the status of the routine. Possible status values are:

RTR_STS_CLASSREQ	At least one data-class definition required

RTR_STS_INVCHANNEL	Invalid pchannel argument
RTR_STS_INVFLAGS	Invalid flags argument
RTR_STS_INVGETITMS	Invalid getitms argument
RTR_STS_INVINFCLA	Invalid information class
RTR_STS_INVSELITM	Invalid selitm argument
RTR_STS_INVSELVAL	Invalid selval argument
RTR_STS_NOACP	No RTRACP process available
RTR_STS_OK	Normal successful completion
RTR_STS_TOOMANCHA	Too many channels already opened

Example

Programming Example:

```
/*
   This routine retrieves the facility names of all facilities
   that have been defined.
*/

#include <string.h>
#include <stdio.h>
#include "rtr.h"

void GetFacilityName()
{
  char* itemlist[10];   /* Set the elements in this array to point to
                        each item in getitembuf for later output. */
  char* cp = 0;
  char getitembuf[1024];
  rtr_status_t status;
  rtr_channel_t channel;
  char msg[1024];                     /* Receive message buffer. */

  unsigned int getitemcnt = 0;
  char infcla_buf[4] = "fac";   /* Set info class to Facility class.*/
   rtr_msgsb_t txsb;

getitembuf[0] = '\0';
/* Set up the request's get-item buffer for
   requesting the facility name. */
 itemlist[getitemcnt] = &getitembuf[strlen(getitembuf)];
 strcat(getitembuf, "fdb_f_name");
/* Increment counters. */
 getitemcnt++;
 /* Add second item FE -> TR. ** Code commented out ** */
 /* (Demonstrates multi-item request. Uncomment code to use.)
 strcat(getitembuf, ",");           //Add comma separator.
 itemlist[getitemcnt] = &getitembuf[strlen(getitembuf)];
 strcat(getitembuf, "fdb_trsrch");
 getitemcnt++;
 */

 /* Call rtr_request_info. */
 status = rtr_request_info (
```

```
                      /* *pchannel     */ &channel,
                       /* flags                 */ RTR_NO_FLAGS,
                       /* infcla                */ infcla_buf,
                       /* selitm                */ "",
                       /* selval                */ "*",
                       /* getitms               */ getitembuf);
if (status != RTR_STS_OK) return;

/* Do a receive message to get the information that RTR returns
 * in response to this request.
 */
do
{
        status = rtr_receive_message(
                      /* See 'rtr_receive_message'. */
                              &channel,
                              RTR_NO_FLAGS,
                              RTR_ANYCHAN,
                              msg,
                              sizeof(msg),
                              RTR_NO_TIMOUTMS,

                              &txsb);
/* Check for bad return status from rtr_receive_message(). */
          if (status != RTR_STS_OK) return;

/* Caller expects either an rtr_mt_closed
   or an rtr_mt_request_info message. */
          if (txsb.msgtype == rtr_mt_closed) break;
                         /* End of data, exit loop.
                            Channel closed by RTR. */
          if (txsb.msgtype != rtr_mt_request_info)
          {
                  printf("Unexpected msgtype returned. \n");
                  break;
          }
          else
          {
                  /* Receive the requested information.
                     Scan through item list, output item and value.
                  */
            unsigned int i;
            for (i=0, cp = msg; i < getitemcnt; i++, cp += strlen(cp)+1)
                  {
                      *(itemlist[i+1]-1) = '\0'; /* Overwrite comma. */
                            printf("%-8s:%40s\t= '%^s'\n",
                                       infcla_buf,
                                           itemlist[i],
                                           cp);
                  }
          }
} while (1 == 1);
return;
```

Command Line Example:

```
RTR> call rtr_request_info/infcla=rtr/selitm=""
         /selval="*"/getitms=rtr_version_string/chann=D
```

```
%RTR-S-OK, normal successful completion

RTR> call rtr_receive_message/chann=D/tim
%RTR-S-OK, normal successful completion
 channel name:   D
 msgsb
   msgtype:      rtr_mt_request_info
   msglen:       18
 message
   offset   bytes                                            text
000000 52 54 52 20 56 33 2E 32 28 32 33 30 29 20 46 54 RTR V3.2(230) FT
000010 33 00                                           3.
```

First Message Example:

The following example illustrates the contents of a first message. In this example, the information is incomplete, 20 seconds remain until the cache is updated, the cache is 0 seconds old, and an explanation is given regarding why the information is incomplete. The example illustrates how the fields are formed.

```
%RTR-S-OK, normal successful completion
 channel name:    RTR$DEFAULT_CHANNEL
 msgsb
   msgtype:       rtr_mt_request_info
   msglen:        197
message
 offset  bytes                                             text
 000000  69 6E 63 6F 6D 70 6C 65 74 65 20 64 61 74 61 00 incomplete data.
 000010  32 30 20 73 65 63 6F 6E 64 73 20 75 6E 74 69 6C 20 seconds until
 000020  20 75 70 64 61 74 65 00 30 20 73 65 63 6F 6E 64 update.0 second
 000030  73 20 6F 6C 64 00 54 68 65 20 72 65 71 75 65 73 s old.The reques
 000040  74 69 6E 67 20 6E 6F 64 65 20 66 6F 78 20 68 61 ting node wlm ha
 000050  73 20 61 20 64 69 73 63 6F 6E 6E 65 63 74 65 64 s a disconnected
 000060  20 6C 69 6E 6B 20 77 69 74 68 20 20 72 6F 75 74 link with rout
 000070  65 72 20 68 65 78 2E 20 48 65 6E 63 65 20 74 68 er hex. Hence th
 000080  65 20 72 65 71 75 65 73 74 69 6E 67 20 6E 6F 64 e requesting nod
 000090  65 20 63 6F 75 6C 64 20 6E 6F 74 20 67 61 74 68 e could not gath
 0000A0  65 72 20 61 6C 6C 20 6F 66 20 74 68 65 20 72 65 er all of the re
 0000B0  71 75 65 73 74 65 64 20 69 6E 66 6F 72 6D 61 74 quested informat
 0000C0  69 6F 6E 2E 00                                    ion..
```

See Also

`rtr_close_channel( )`

`rtr_receive_message( )`

rtr_send_to_server

rtr_send_to_server — Send a transactional message to a server.

Format

`status = rtr_send_to_server (channel, flags, pmsg, msglen, msgfmt)`

Argument	Data Type	Access
status	rtr_status_t	write

Argument	Data Type	Access
channel	rtr_channel_t	read
flags	rtr_sen_flag_t	read
pmsg	rtr_msgbuf_t	read
msglen	rtr_msglen_t	read
msgfmt	rtr_msgfmt_t	read

C Binding

```
rtr_status_t rtr_send_to_server (
        rtr_channel_t channel ,
        rtr_sen_flag_t flags ,
        rtr_msgbuf_t pmsg ,
        rtr_msglen_t msglen ,
        rtr_msgfmt_t msgfmt ,
        )
```

Arguments

channel

The channel identifier (returned earlier by `rtr_open_channel( )`).

flags

Table below shows the flags that specify options for the call.

Table 3.26. Send to Server Flags

Flag name	Description
RTR_F_SEN_ACCEPT	This is the last message of the transaction, and the tx is accepted. This optimization avoids the need for a separate call to `rtr_accept_tx( )` in those cases where the sender knows this is the last (or only) message in the transaction.
RTR_F_SEN_READONLY	Specifies a read-only server operation. Hence no shadowing or journalling 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. The message is still needed in the journal to allow recovery of in-flight transactions.)
RTR_F_SEN_RETURN_TO_SENDER	The message is to be returned to the sender if undeliverable.
RTR_F_SEN_EXPENDABLE	The whole transaction is not aborted if this send fails.

Specify RTR_NO_FLAGS for this parameter if no flags are required.

pmsg

Pointer to the message to be sent.

msglen

Length in bytes of the message to be sent, up to RTR_MAX_MSGLEN bytes. The value of RTR_MAX_MSGLEN is defined in rtr.h.

msgfmt

Message format description. *msgfmt* 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. See Section 2.14, RTR Applications in a Multiplatform Environment, for information on defining a message format description.

This parameter is optional. Specify RTR_NO_MSGFMT if the message content is platform independent, or it is not intended to be used on other hardware platforms.

Description

The rtr_send_to_server( ) call sends a client's transactional message to a server.

The caller must first open a client channel (using the rtr_open_channel( ) call), before it can send transactional messages.

If no transaction is currently active on the channel, a new transaction is started.

Return Value

A value indicating the status of the routine. Possible status values are:

RTR_STS_CHANOTOPE	Channel not opened
RTR_STS_INSVIRMEM	Insufficient virtual memory
RTR_STS_INVCHANNEL	Invalid channel argument
RTR_STS_INVFLAGS	Invalid flags argument
RTR_STS_INVJOINTXID	Invalid join transaction argument
RTR_STS_INVMSGFMT	Invalid msgfmt argument
RTR_STS_INVMSGLEN	Invalid msglen argument
RTR_STS_NOACP	No RTRACP process available
RTR_STS_NOXACHAN	No XA channel available
RTR_STS_OK	Normal successful completion
RTR_STS_REPLYDIFF	Reply from new server did not match earlier reply

Example

```
/* The my_msg structure is defined in the user's
 * application header file.
 */
typedef struct {
                rtr_uns_8_t routing_key;
                rtr_uns_32_t sequence_number;
                rtr_uns_8_t my_msg_type;
                string31 last_name;
                rtr_uns_32_t order_total;
```

```
                rtr_uns_32_t shipping_amt;
                string16 cc_number;
                string7 cc_expire;
        } my_msg;
        my_msg send_msg;
        .
        .       Load purchase data into send_msg.
        .
/*
 * Tell the server to validate the credit card for the
 * amount of this order.
 */
my_msg.my_msg_type = VALIDATE_CC;
status = rtr_send_to_server(
                channel,
                RTR_NO_FLAGS ,
                &send_msg,
                sizeof(send_msg),
                RTR_NO_MSGFMT );
```

See Also

rtr_receive_message( )

rtr_open_channel( )

rtr_set_info

rtr_set_info — Sets or changes a managed object in the RTR environment.

Format

status = rtr_set_info (*pchannel, flags, verb, object, *select_qualifiers, *s

Argument	Data Type	Access
status	rtr_status_t	write
*pchannel	rtr_channel_t	write
flags	rtr_set_flag_t	read
verb	rtr_verb_t	read
object	rtr_managed_object_t	read
*select_qualifiers	rtr_qualifier_value_t	read
*set_qualifiers	rtr_qualifier_value_t	read

C Binding

```
rtr_status_t rtr_set_info (
        rtr_channel_t *pchannel ,
        rtr_set_flag_t flags ,
        rtr_verb_t verb ,
        rtr_managed_object_t object ,
        const rtr_qualifier_value_t *select_qualifiers ,
        const rtr_qualifier_value_t *set_qualifiers
```

)

Arguments

pchannel

Pointer to the channel opened by a successful call to `rtr_set_info( )`.

flags

No flags are currently defined. Specify `RTR_NO_FLAGS` for this argument.

verb

Always `rtr_verb_set`.

object

Establishes the type of object to which the call is directed. Values are:

- `rtr_partition_object`: the target object is a partition
- `rtr_transaction_object`: the target object is a transaction

select_qualifiers

Pointer to array containing selection qualifiers. Values depend on object type:

For:	See the values in:
Set Partition	Table 3.27
Set Transaction	Table 3.28

For example:

```
typedef struct rtr_qualifier_value_t {
    rtr_qualifier_t qv_qualifier ;    /* Which qualifier this is */
    void *qv_value ;                  /* What value it has */
    } rtr_qualifier_value_t ;
```

The last value in the array must be `rtr_qualifiers_end` (see the example). Specify sufficient descriptors to identify the target object.

Table 3.27. Select Qualifiers for the Set Partition Object

Qualifier	Value Type	Description	Example
rtr_facility_name	const char*	Facility name string	"facility_name"
rtr_partition_name	const char*	Partition name string	"partition_name"

Table 3.28. Select Qualifiers for the Set Transaction Object

Qualifier	Value Type	Description	Example
rtr_facility_name	facname	Facility name string	"facility_name"
rtr_partition_name	partname	Partition name string	"partition_name"

Qualifier	Value Type	Description	Example
rtr_txn_state	rtr_txn_jnl_commit	Current transaction state	See Table 3.29 for valid changes from one state to another.
rtr_txn_tid	tid	Transaction ID	63b01d10,0,0,0,0,2e59, 43ea2002

When using the Set Transaction Object, the qualifier `rtr_txn_state` is required. In addition, when using `rtr_txn_state` without `rtr_facility_name` or `rtr_partition_name`, `rtr_txn_tid` is required. The qualifiers `rtr_facility_name` and `rtr_partition_name` must be used together. You must always provide the current state when making a state change.

Table 3.29. Valid Set Transaction State Changes

From (current state):	To (new state):					
	COMMIT	ABORT	EXCEPTION	DEFER	PRI_DONE	DONE
SENDING		YES				
VOTED	YES	YES				
COMMIT			YES			YES
EXCEPTION	YES					YES
PRI_DONE				YES		YES
DEFER					YES	

set_qualifiers

Pointer to an array containing values of type `rtr_qualifier_value_t` (see Select Qualifiers above) that describe the desired change to be effected. Table 3.30 and Table 3.31 list qualifiers and value types for the managed object types.

Table 3.30. Qualifiers for Set Partition

Qualifier	Value Type	Value	Desired Action
rtr_partition_state	rtr_partition_state_t	rtr_partition_state_ _suspend	Suspend transaction presentation.
rtr_partition_state	rtr_partition_state_t	rtr_partition_state_ _resume	Resume transaction presentation.
rtr_partition_state	rtr_partition_state_t	rtr_partition_state_ _recover	(Re)start partition recovery.
rtr_partition_state	rtr_partition_state_t	rtr_partition_state_ _exitwait	Exit partition recovery wait/fail state.
rtr_partition_state	rtr_partition_state_t	rtr_partition_state_ _shadow	Enable shadowing.
rtr_partition_state	rtr_partition_state_t	rtr_partition_state_	Disable shadowing.

Qualifier	Value Type	Value	Desired Action
		_noshadow	
rtr_partition_cmd_ _timeout_secs	rtr_uns_32_t	unsigned int	Optional partition suspend timeout period (in seconds).
rtr_partition_rcvy_ _retry_count	rtr_uns_32_t	unsigned int	Limit number of recovery replays for a transaction.
rtr_partition_failover_ _policy	rtr_partition_failover_ _policy_t	rtr_partition_fail_to_ _standby	Set failover policy to standby.
rtr_partition_failover_ _policy	rtr_partition_failover_ _policy_t	rtr_partition_fail_to_ _shadow	Set failover policy to shadow.
rtr_partition_failover_ _policy	rtr_partition_failover_ _policy_t	rtr_partition_pre32_ _compatible	Set failover policy as pre-V3.2 compatible.

For both managed object types, a message of type `rtr_mt_closed` is returned. See Table 3.31 for the value that can be set for the transaction type.

Completion status is read from message data, which is of type `rtr_status_data_t`. In addition, a number (type integer) indicating the number of transactions processed is returned. This number can be read from the message following the `rtr_status_data_t` data item. The last value in the array must be `rtr_qualifiers_end`.

Table 3.31. Qualifiers for Set Transaction

Set Qualifier	Set Qualifier Value	Value	Description
rtr_txn_state	rtr_transaction_state_t	rtr_tx_jnl_commit	Set a transaction's state to COMMIT to commit the transaction.
rtr_txn_state	rtr_transaction_state_t	rtr_tx_jnl_abort	Set a transaction state to ABORT to abort the transaction.
rtr_txn_state	rtr_transaction_state_t	rtr_tx_jnl_exception	Mark this as an exception transaction.
rtr_txn_state	rtr_transaction_state_t	rtr_tx_jnl_done	Remove this transaction from the RTR journal; that is, forget this transaction completely.

Description

The `rtr_set_info( )` call requests a change in a characteristic of the RTR environment. If the call is successful, a channel is opened for asynchronous completion notification. Applications should use the `rtr_receive_message( )` call to retrieve informational messages on the opened channel.

The `rtr_set_info( )` call can manipulate two managed object types:

- Partition type

- Transaction type

See Table 3.30 for values that can be set for the partition object and Table 3.31 for values that can be set for the transaction object. Completion status is read from message data, which is of type `rtr_status_data_t`.

Return Value

A value indicating the status of the routine, normally returned as function completion status. Possible status values are:

RTR_STS_ALRDYINSTATE DAG	Partition is already in the desired state
RTR_STS_BADPRTSTATE DAG	Disallowed attempt to make an illegal or undefined partition state transition
RTR_STS_FACNAMLON DAG	Facility name *facility_name* is larger than 30 characters
RTR_STS_FENAMELONG	Frontend name string length greater than permitted maximum
RTR_STS_INSUFPRIV	Insufficient privileges to run RTR
RTR_STS_INSVIRMEM	Insufficient virtual memory
RTR_STS_INVCHANNEL	Invalid channel argument
RTR_STS_INVFACNAM	Invalid FACNAM argument
RTR_STS_INVFLAGS	Invalid flags argument
RTR_STS_INVOBJCT	Specified object type invalid for managed object request
RTR_STS_INVSTATCHANGE	Invalid to change from the current state to the specified state
RTR_STS_IVQUAL	Unrecognized qualifier - check validity, spelling, and placement
RTR_STS_IVVERB	Unrecognized command verb - check validity and spelling
RTR_STS_NOACTION	No object management action specified - check argument set qualifier
RTR_STS_NODNOTBAC DAG	Node not defined as a backend
RTR_STS_NOSUCHPRTN DAG	No such partition in the system
RTR_STS_OK	Normal successful completion
RTR_STS_PARTNAMELONG	Partition name too long
RTR_STS_PRTBADCMD DAG	Partition command invalid or not implemented in this version of RTR
RTR_STS_PRTBADFPOL DAG	Unrecognized partition failover policy code
RTR_STS_PRTLCLRECEXT DAG	Partition local recovery terminated by operator
RTR_STS_PRTMODRMBR DAG	Partition must be in remember mode on the active member
RTR_STS_PRTNOSRVRS DAG	Partition has no servers – please start servers and retry

RTR_STS_PRTNOTBACKEND [DAG]	Partition command must be entered on a backend node
RTR_STS_PRTNOTSUSP [DAG]	Unable to resume partition that is not suspended
RTR_STS_PRTNOTWAIT [DAG]	Partition not in a wait state – no action taken
RTR_STS_PRTRECSTATE [DAG]	Partition must be in remember or active (non-recovery) state
RTR_STS_PRTRESUMED [DAG]	Partition *partition_name* resumed by operator *operator*
RTR_STS_PRTRUNDOWN [DAG]	Partition is in a rundown prior to deletion – no action taken
RTR_STS_PRTSHDRECEXT [DAG]	Partition shadow recovery terminated by operator
RTR_STS_SETTRANDONE [DAG]	*n* transaction(s) updated in partition *partition_name* of facility *facility_name*
RTR_STS_SETTRANROUTER [DAG]	Cannot process this command, coordinator router is still available
RTR_STS_TOOMANCHA	Too many channels already opened
RTR_STS_TRNOTALL032 [DAG]	Not all routers are at the minimum required version of V3.2
RTR_STS_VALREQ	Missing qualifier or keyword value – supply all required values
RTR_STS_WTTR	Not in contact with sufficient router nodes – please retry later

[DAG] Returned in status field of rtr_status_data_t data returned with the **rtr_mt_closed** message. Indicates outcome of request.

Example

```
/*
 * This might follow a call to commit the transaction to the database.
 * If the SQL commit returns an error that is beyond the control of
 * this application: for example, database disk full, network to
 * database not responding, or timeout exceeded, it executes.
 *
 * Declarations:
 */

rtr_tid_t             tid;
rtr_uns_32_t          select_idx;
rtr_uns_32_t          set_idx;
rtr_qualifier_value_t select_qualifiers[8];
rtr_qualifier_value_t set_qualifiers[3];

/* Everyone has voted to accept the transaction, and RTR has told the
 * server to commit it. The client has moved on to performing the next
 * transaction. This transaction will be changed from 'commit' status
 * to 'exception' status for a later attempt at committing.
 *
 * Get the transaction id. The channel has previously been
 * declared in an rtr_open_channel call.
 */

rtr_get_tid(channel, RTR_F_TID_RTR, &tid);
```

```
/* Load the rtr_qualifier_value_t structures that contain the
 * selection criteria for the transaction: 'the transaction whose tid
 * is pointed at by 'tid', whose facility name is in 'facname', whose
 * partition name is in 'partname', and whose transaction state is
 * 'rtr_txn_jnl_commit' (logged to the journal as committed).
 */

select_idx = 0;
select_qualifiers[select_idx].qv_qualifier = rtr_txn_tid;
select_qualifiers[select_idx].qv_value = &tid;
select_idx++;

/* Facility name
 */

select_qualifiers[select_idx].qv_qualifier =
rtr_facility_name;
select_qualifiers[select_idx].qv_value = facname;
select_idx++;

/* Partition name
 */

select_qualifiers[select_idx].qv_qualifier = rtr_partition_name;
select_qualifiers[select_idx].qv_value = partname;
select_idx++;

/* Transaction state in journal
 */

select_qualifiers[select_idx].qv_qualifier = rtr_txn_state;
select_qualifiers[select_idx].qv_value = &rtr_txn_jnl_commit;
select_idx++;

/* Last one on array must be 'rtr_qualifiers_end'
 */

select_qualifiers[ select_idx].qv_qualifier =
                           rtr_qualifiers_end,
select_qualifiers[ select_idx].qv_value = NULL;
select_idx++;

/* Load the
 * rtr_qualifier_t structs that we will use to set the
 * new property for the transaction: in this case, only the
 * state of the transaction. We will change it to
 * rtr_txn_jnl_exception, or 'exception'.
 */

set_idx = 0;
set_qualifiers[set_idx].qv_qualifier = rtr_txn_state;
set_qualifiers[set_idx].qv_value = &rtr_txn_jnl_exception;
set_idx++;

/* Terminate the array with an rtr_qualifiers_end.
 */
set_qualifiers[set_idx].qv_qualifier =
```

```
                    rtr_qualifiers_end;
set_qualifiers[set_idx].qv_value = NULL;
set_idx++;

/* Tell RTR to change the transaction's state.
 */
status = rtr_set_info( &pchannel,
                       RTR_NO_FLAGS,
                       rtr_verb_set,
                       rtr_transaction_object,
                       select_qualifiers,
                       set_qualifiers);

check_status(status);

/* The server should now look for an
 * RTR_STS_SETTRADONE message
 * from RTR, which confirms that it has changed the status.
 */
```

See Also

rtr_close_channel( )

rtr_receive_message( )

rtr_request_info( )

rtr_set_user_context

rtr_set_user_context — Sets the current value of the optional user-defined context associated with the specified RTR channel.

Format

status = rtr_set_user_context (channel, usrctx)

Argument	Data Type	Access
status	rtr_status_t	write
channel	rtr_channel_t	read
usrctx	rtr_usrctx_t	read

C Binding

```
rtr_status_t rtr_set_user_context (
        rtr_channel_t channel ,
        rtr_usrctx_t usrctx
        )
```

Arguments

channel

The channel whose context is to be set.

usrctx

User-supplied context value.

Description

Sets the current value of the optional user-defined context associated with the specified RTR channel. The user context value may be subsequently retrieved using rtr_get_user_context( ). The context value `RTR_NO_USER_CONTEXT` is reserved.

Return Value

A value indicating the status of the routine. Possible values are:

RTR_STS_INVCHANNEL	Invalid channel argument
RTR_STS_OK	Normal successful completion

See Also

`rtr_get_user_context( )`

rtr_set_user_handle

rtr_set_user_handle — Associate a user-defined value (handle) with a transaction.

Format

`status = rtr_set_user_handle (channel, usrhdl)`

Argument	Data Type	Access
status	rtr_status_t	write
channel	rtr_channel_t	read
usrhdl	rtr_usrhdl_t	read

C Binding

```
rtr_status_t rtr_set_user_handle (
        rtr_channel_t channel ,
        rtr_usrhdl_t usrhdl
        )
```

Arguments

channel

The channel identifier, returned earlier by the `rtr_open_channel( )` call.

usrhdl

Value to associate with the channel. This value is returned in the *usrhdl* field of the *msgsb* message status block when subsequent calls to `rtr_receive_message( )` return messages associated with this channel.

The *usrhdl* argument can be used to hold a pointer.

Description

The `rtr_set_user_handle( )` call associates a user-defined value (handle) with a channel. An application can either use a handle, or client and servers can act independently.

The current value of a handle is associated with a channel; the current handle value is associated with each operation on the channel. The message status block supplied with a message delivered on the channel contains the user handle value that was current at the time of the associated operation. For example, an `rtr_mt_accepted` message has the user handle that was current when the corresponding call to `rtr_accept_tx( )` was made, and the `rtr_mt_rettosend` message has the user handle that was current when the corresponding call to `rtr_send_to_server( )` was made.

Note that the value of a handle is process local, and a different handle would be associated for the same transaction by the client and server. The scope for the user handle is within the process in which the user handle is set.

Return Value

A value indicating the status of the routine. Possible values are:

RTR_STS_CHANOTOPE	Channel not opened
RTR_STS_INVCHANNEL	Invalid channel argument
RTR_STS_OK	Normal successful completion
RTR_STS_TXACTIVE	Transaction is active

Example

```
/* This client does not use nested transactions, and it does
 * not wait for the mt_accepted message before sending
 * the next transaction. Instead, it matches each 'accepted'
 * message it receives with a transaction.
 */
        typedef struct {
                rtr_uns_32_t txn_number;
                rtr_uns_32_t message_id_sent;
                char my_record[255];
} txn_handle;

/* Allocate and load the txn_handle data structure that
 * you create.
 */
txn_handle txn_ident = (txn_handle*)calloc(1, sizeof(txn));
txn_ident->txn_number = ++count;
txn_ident->message_id_sent = my_message_id;
strcpy(txn_ident->record, my_record);

/* Attach this struct to the channel on which we're sending the
 * transaction.
 */
        status = rtr_set_user_handle( channel, txn_ident );
```

See Also

`rtr_receive_message( )`

rtr_set_wakeup

rtr_set_wakeup — Register a function to be called when a message arrives.

Format

`status = rtr_set_wakeup (void (*wu_rou)(void))`

Argument	Data Type	Access
status	rtr_status_t	write
wu_rou	procedure	read

C Binding

```
rtr_status_t rtr_set_wakeup (
        procedure void (*wu_rou) (void)
        )
```

Arguments

void (*wu_rou) (void)

The routine to be called by RTR when a message is waiting to be delivered.

Description

The `rtr_set_wakeup( )` call sets the address of a function to be called when a message is waiting to be delivered. To cancel wakeups, call the routine with an argument of NULL.

Execution of the specified wakeup indicates that you may have messages.

At the time of the execution of the wakeup there may be 0, 1 or more messages available. Each incoming application message does not generate a separate wakeup callback, so following a wakeup callback a program should call `rtr_receive_message( ..., timoutms=0, ...)` in a loop at some point to ensure that no message is left uncollected.

See CALL `rtr_receive_message` in the *VSI Reliable Transaction Router System Manager's Manual* for restrictions on using V2 and later RTR version calls in the same application.

If a wakeup routine has been set using this call, subsequent calls to `rtr_set_wakeup( )` should either disable the wakeup feature (with an argument of `NULL`), or replace the current wakeup routine with another.

For details and restrictions on using the RTR wakeup handler `rtr_set_wakeup`, see the discussion in Section 2.9.

Return Value

A value indicating the status of the routine. Possible values are:

RTR_STS_ACPNOTVIA	RTR ACP no longer a viable entity, restart RTR or application
RTR_STS_BYTLMNSUFF	Insufficient process quota bytlm, required 100000

RTR_STS_INVCHANNEL	Invalid channel argument
RTR_STS_NOACP	No RTRACP process available
RTR_STS_OK	Normal successful completion

Example

```
#include <stdlib.h>

void app_wakeup_routine (void)
{

    /* NB This is called from an AST, ALRM or IO signal handler,
     * or another thread depending on the platform.
     * Although RTR blocks signals, ASTs and the wakeup thread
     * until it is safe and convenient,
     * you may prefer to just set a flag or generate an event and
     * perform the receive_message in your main thread instead.
     */
    /* Get all outstanding rtr messages */
    do
    {
        sts = rtr_receive_message(..., /* timoutms */ 0 ) ;
        check ( sts ) ;
        process_message () ;
    } while ( sts != RTR_STS_TIMOUT ) ;
}

static void app_cancel_wakeup (void)
{
    rtr_set_wakeup( NULL );
}

main ()
{
    sts = rtr_set_wakeup( app_wakeup_routine );
    atexit(app_cancel_wakeup);
    .
    .
}
```

If RTR data is available when `rtr_set_wakeup` is called, the application's wakeup routine is called immediately.

See Also

`rtr_receive_message( )`

rtr_start_tx

rtr_start_tx — Explicitly start a transaction on the specified channel.

Format

`status = rtr_start_tx (channel, flags, timoutms, pjointxid)`

Argument	Data Type	Access
status	rtr_status_t	write
channel	rtr_channel_t	read
flags	rtr_sta_flag_t	read
timoutms	rtr_timout_t	read
pjointxid	rtr_pointer_t	read

C Binding

```
rtr_status_t rtr_start_tx (
        rtr_channel_t channel ,
        rtr_sta_flag_t flags ,
        rtr_timout_t timoutms ,
        rtr_pointer_t pjointxid
        )
```

Arguments

channel

The channel identifier returned earlier by the `rtr_open_channel( )` call.

flags

Flags that specify options for the call. Normally specify `RTR_NO_FLAGS` for this parameter.

timoutms

Transaction timeout specified in milliseconds. If the transaction is not accepted by all participants within the specified timeout period, RTR aborts the transaction and reports a status of `RTR_STS_TIMOUT`.

The granularity of the underlying timer is 1 second. Fractional values of the *timoutms* argument are rounded up to the next whole second. A value of 0 causes an immediate transaction abort. If no timeout is required, specify `RTR_NO_TIMOUTMS`.

pjointxid

Pointer to the transaction identifier of the parent transaction.

Description

The `rtr_start_tx( )` call is used to start a transaction explicitly.

An explicit transaction start is only necessary if one of the following conditions exists:

- a join to an existing transaction is to be done
- a transaction timeout is to be specified

Transactions are implicitly started when a message is sent on a currently inactive channel. Implicitly started transactions have no timeout and are not joined to other RTR transactions.

Return Value

A value indicating the status of the routine. Possible status values are:

RTR_STS_ACPNOTVIA	RTR is no longer a viable entity, restart RTR or application
RTR_STS_INVCHANNEL	Invalid channel argument
RTR_STS_INVFLAGS	Invalid flags argument
RTR_STS_INVJOINTXID	Invalid join transaction argument. The flag `RTR_F_OPE_FOREIGN_TM` was defined in the call to `rtr_open_channel( )`, but *pjointxid* is equal to `RTR_NO_JOINTXID`, or the *formatID* field of an XA transaction in the *pjointxid* parameter is equal to `RTR_XID_FORMATID_NONE`.
RTR_STS_INVOP4SRV	Invalid operation for server channel
RTR_STS_INVTIMOUTMS	Invalid timoutms argument
RTR_STS_NOACP	No RTRACP process available
RTR_STS_NOXACHAN	No XA channel available
RTR_STS_OK	Normal successful completion
RTR_STS_TRAALRSTA	Transaction already started
RTR_STS_VERMISMAT	RTR version mismatch. The RTR router is running an older version of RTR that does not support nested transactions.

Example

```
rtr_xid_t xa_txn;

/* This client/server pair handles transactions that contain
 * multiple messages within each one. Transactions are explicitly
 * started and prepared, as directed by this client.
 *
 * Fill in the information in the XA transaction id struct.
 * The information will be sent to the server to tag the transaction.
 */
        xa_txn.formatID = RTR_XID_FORMATID_RTR_XA;
        xa_txn.gtrid_length = 4;
        xa_txn.bqual_length = 4;
        strcpy(xa_txn.data, "6789.0003");
/* Start the transaction; specify a timeout so we don't get
 * stuck waiting forever. May not complete immediately.
 */
        status = rtr_start_tx(
                                channel,
                                RTR_F_STA_TID_XA,
                                1000,
                                &xa_txn );

check_status(status);     /* May be RTR_STS_TIMEOUT. */
```

See Also

`rtr_open_channel( )`

`rtr_send_to_server( )`

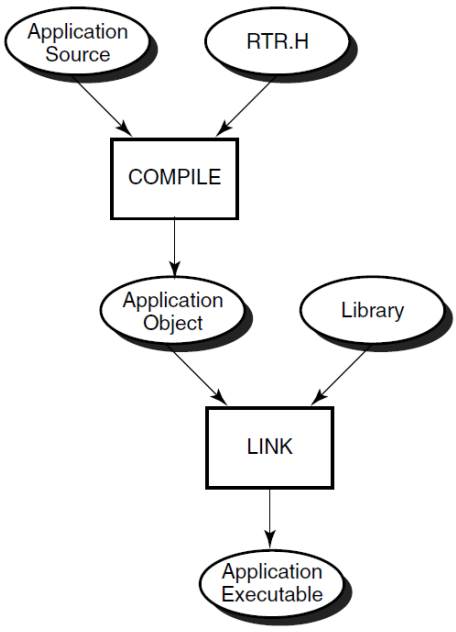
Chapter 4. Compiling and Linking Your Application

All client and application programs must be written using C, C++, or a language that can use RTR API calls. Include the RTR data types and error messages file rtr.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 rtr.h using ANSI C include rules. For example, if rtr.h is in the same directory as your C code, compile with the following statement: #include "rtr.h".
4. Link your object code with the RTR library to produce your application executable.

This process is illustrated in Figure 4.1. In this figure, Library represents the RTR C API shareable images (OpenVMS), DLLs (Win32), and shared libraries (UNIX).

Figure 4.1. Compile Sequence



4.1. Compilers

Compilers commonly used in developing RTR applications include those in Table 4.1. For additional information, see the *Reliable Transaction Router Software Product Description*.

Table 4.1. Minimum Compiler Versions for Developing RTR Applications

Operating System	Compiler	Compiler Version
Microsoft Windows	Microsoft Visual C++ (Microsoft Visual Studio 6.0)	Version 6.0 SP6

Operating System	Compiler	Compiler Version
OpenVMS Alpha	Compaq C	Version 6.2-006
OpenVMS VAX	Compaq C	Version 6.2-003
Sun	Forte Compilers	Version 6.0
Tru64 UNIX	Compaq C	Version 6.3-126

4.2. Linking Libraries

To compile and link a C RTR application, use command lines as shown below. Separate examples are shown for use of RTR with threaded or unthreaded libraries. 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.

Windows

```
> cl /c /MT  yourapp.c
> link yourapp.obj /out:yourapp.exe rtrdll.lib
```

Linux I32 (Frontend)

Single-threaded:

```
# cc -o yourapp -lrtr yourapp.c
```

Multi-threaded:

```
# cc -o yourapp -pthread -lrtr_r yourapp.c
```

Linux I64

Single-threaded:

```
# cc -o yourapp -lrtr yourapp.c
```

Multi-threaded:

```
# cc -o yourapp -pthread -lrtr_r yourapp.c
```

OpenVMS Alpha

Single-threaded:

```
$ cc yourapp.c
$ link yourapp,sys$input/opt
SYS$SHARE:librtr/share
^Z
```

Multi-threaded:

```
$ cc yourapp.c
```

```
$ link yourapp,sys$input/opt
SYS$SHARE:librtr_r/share
^Z
```

OpenVMS I64

Single-threaded:

```
$ cc yourapp.c
$ link yourapp,sys$input/opt
SYS$SHARE:librtr/share
^Z
```

Multi-threaded:

```
$ cc yourapp.c
$ link yourapp,sys$input/opt
SYS$SHARE:librtr_r/share
^Z
```

HP-UX I64

Single-threaded:

```
# cc -o yourapp +DD64 +DSitanium2 -lrtr yourapp.c
```

Multi-threaded:

```
# cc -o yourapp +DD64 +DSitanium2 -lpthread -lrtr_r yourapp.c
```

Appendix A. RTR C API Sample Applications

A.1. Overview

The software kit contains a short sample application that is unsupported and not part of the RTR product. Code for the sample application is in the [EXAMPLES] directory on the software kit. This sample application contains four components:

adg_client.c
adg_server.c
adg_shared.c
adg_header.h

The code is shown on the next few pages. Note the following:

- Return value checking after `fprintf() fclose()` and so on, is omitted for clarity.
- `time()` and `ctime()` are used instead of higher resolution reentrant alternatives that are less portable.

A.2. Client Application

```
/* Client Application */

/*****************************************************************************
* Copyright Compaq Computer Corporation 1998. All rights reserved.
* Restricted Rights: Use, duplication, or disclosure by the U.S.
* Government is subject to restrictions as set forth in subparagraph (c)
* (1) (ii) of DFARS 252.227-7013, or in FAR 52.227-19, or in FAR 52.227-14
* Alt. III, as applicable.
* This software is proprietary to and embodies the confidential technology
* of Compaq Computer Corporation. Possession, use, of copying of this
* software and media is authorized only pursuant to a valid written
* license from Compaq, Digital or an authorized sublicensor.
*****************************************************************************/
/
*****************************************************************************
* APPLICATION: RTR Sample Client Application
* MODULE NAME: adg_client.c
* AUTHOR:      Compaq Computer Corporation
* DESCRIPTION: This client application initiates transactions and requests
*              transaction status asynchronously. It is to be used with
               adg_server.c, adg_header.h, and adg_shared.c.
* DATE       : Oct 22, 1998
*****************************************************************************/
/*
    adg_client.c

    Goes with adg_server.c

    To build on Unix:
        cc -o adg_client adg_client.c adg_shared.c -lrtr
*/
```

```
#include "adg_header.h"

void declare_client ( rtr_channel_t *pchannel );
FILE *fpLog;

int main ( int argc, char *argv[] )
{
    /*
     * This program expects 3 parameters :
     *   1: client number (1 or 2)
     *   2: partition range
     *   3: messages to send
     */

    rtr_status_t        status;
    rtr_channel_t       channel ;
    time_t time_val = { 0 };

    message_data_t send_msg = {0};
    receive_msg_t receive_msg = {0};
    int txn_cnt;
    rtr_timout_t receive_time_out = RTR_NO_TIMOUTMS;
    rtr_msgsb_t msgsb;
    char CliLog[80];

    send_msg.sequence_number = 1 ;
    strcpy( send_msg.text , "from Client");

    get_client_parameters( argc , argv, &send_msg, &txn_cnt);

    sprintf( CliLog, "CLIENT_%c_%d.LOG", send_msg.routing_key,
             send_msg.client_number );
    fpLog = fopen( CliLog, "w");

    if ( fpLog == NULL )
    {
        perror("adg_client: fopen failed");
        fprintf(stderr, " Error opening client log %s\n", CliLog );
        exit(EXIT_FAILURE);
    }

    printf( "\n Client log = %s\n", CliLog );

    fprintf(fpLog, " txn count = %d\n", txn_cnt );
    fprintf(fpLog, " client number = %d\n", send_msg.client_number );
    fprintf(fpLog, " routing key = %c\n\n", send_msg.routing_key);

    declare_client ( &channel );

    /* Send the requested number of txns */

    for ( ; txn_cnt > 0; txn_cnt--, send_msg.sequence_number++ )
    {
        status = rtr_send_to_server(
                channel,
                RTR_NO_FLAGS ,
                &send_msg,
```

```
        sizeof(send_msg),
        msgfmt );

check_status( "rtr_send_to_server", status );

fprintf(fpLog, "\n ********** sequence %10d **********\n",
        send_msg.sequence_number);
time(&time_val);
fprintf(fpLog, "   send_to_server at:         %s",
        ctime( &time_val));
fflush(fpLog);

/*
 *  Get the server's reply OR
 *  an rtr_mt_rejected
 */

status = rtr_receive_message(
                &channel,
                RTR_NO_FLAGS,
                RTR_ANYCHAN,
                &receive_msg,
                sizeof(receive_msg),
                receive_time_out,
                &msgsb);

check_status( "rtr_receive_message", status );

time(&time_val);
switch (msgsb.msgtype)
{
case rtr_mt_reply:
    fprintf(fpLog, "   reply from server at:      %s",
            ctime( &time_val));
    fprintf(fpLog, "     sequence %10d from server %d\n",
            receive_msg.receive_data_msg.sequence_number,
            receive_msg.receive_data_msg.server_number);
    fflush(fpLog);
    break;

case rtr_mt_rejected:
    fprintf(fpLog, "  txn rejected at:           %s",
            ctime( &time_val));
    fprint_tid(fpLog, &msgsb.tid );
    fprintf(fpLog, "     status is : %d\n", status);
    fprintf(fpLog, "     %s\n", rtr_error_text(status));
    fflush(fpLog);

    /* Resend same sequence_number after reject */
    send_msg.sequence_number--;
    txn_cnt++;
    break;

default:
    fprintf(fpLog,
            "  unexpected msg at: %s", ctime( &time_val));
    fprint_tid(fpLog, &msgsb.tid );
    fflush(fpLog);
```

```
            exit(EXIT_FAILURE);
        }


        if (msgsb.msgtype == rtr_mt_reply)
        {
            status = rtr_accept_tx(
                            channel,
                            RTR_NO_FLAGS,
                            RTR_NO_REASON );

            check_status( "rtr_accept_tx", status );

            status = rtr_receive_message(
                            &channel,
                            RTR_NO_FLAGS,
                            RTR_ANYCHAN,
                            &receive_msg,
                            sizeof(receive_msg),
                            receive_time_out,
                            &msgsb);

            check_status( "rtr_receive_message", status );

            time(&time_val);

            switch ( msgsb.msgtype )
            {
            case rtr_mt_accepted:
                fprintf(fpLog, "   txn accepted at :
                        %s", ctime( &time_val));
                fprint_tid(fpLog, &msgsb.tid );
                fflush(fpLog);
                break;

            case rtr_mt_rejected:
                fprintf(fpLog, "   txn rejected at :
                       %s", ctime( &time_val));
                fprint_tid(fpLog, &msgsb.tid );
                fprintf(fpLog, "      status is : %d\n",
                       receive_msg.receive_status_msg.status);
                fprintf(fpLog, " %s\n",

rtr_error_text(receive_msg.receive_status_msg.status));
                fflush(fpLog);

                /* Resend same sequence_number after reject */

                send_msg.sequence_number--;
                txn_cnt++;
                break;

            default:
                fprintf(fpLog,
                        " unexpected status on rtr_mt_accepted message\n");
                fprint_tid(fpLog, &msgsb.tid );
                fprintf(fpLog, "      status is : %d\n",
                        receive_msg.receive_status_msg.status);
```

```
                fprintf(fpLog,
                " %s\n",
 rtr_error_text(receive_msg.receive_status_msg.status));
                fflush(fpLog);
                break;
            }
        }


    }

    close_channel ( channel );
}

void
declare_client ( rtr_channel_t *pchannel )
{
    rtr_status_t        status;
    receive_msg_t       receive_msg;
    rtr_timout_t receive_time_out = RTR_NO_TIMOUTMS;  /* forever */
    rtr_msgsb_t msgsb;  /* Structure into which receive puts msgtype */

    status = rtr_open_channel(
                    pchannel,
                    RTR_F_OPE_CLIENT ,
                    FACILITY_NAME,
                    NULL,                       /* rpcnam */
                    RTR_NO_PEVTNUM,
                    RTR_NO_ACCESS               /* access */
                    RTR_NO_NUMSEG ,
                    RTR_NO_PKEYSEG );

    check_status( "rtr_open_channel", status);

    status = rtr_receive_message(
                    pchannel,
                    RTR_NO_FLAGS,
                    RTR_ANYCHAN,
                    &receive_msg,
                    sizeof(receive_msg),
                    receive_time_out,
                    &msgsb);

    check_status( "rtr_receive_message", status );

    if ( msgsb.msgtype != rtr_mt_opened )
    {
       fprintf(fpLog, " Error opening rtr channel %s : \n", FACILITY_NAME);

        fprintf(fpLog, "%s\n",
                rtr_error_text(receive_msg.receive_status_msg.status));
        exit(EXIT_FAILURE);
    }

    fprintf(fpLog, " Client channel successfully opened\n");
    return;
}
```

A.3. Server Application

```
/* Server Application */

/
*************************************************************************
* Copyright Compaq Computer Corporation 1998. All rights reserved.
* Restricted Rights: Use, duplication, or disclosure by the U.S.
* Government is subject to restrictions as set forth in subparagraph (c)
* (1) (ii) of DFARS 252.227-7013, or in FAR 52.227-19, or in FAR 52.227-14
* Alt. III, as applicable.
* This software is proprietary to and embodies the confidential technology
* of Compaq Computer Corporation. Possession, use, of copying of this
* software and media is authorized only pursuant to a valid written
* license from Compaq, Digital or an authorized sublicensor.
*************************************************************************/
/
*************************************************************************
* APPLICATION: RTR Sample Server Application
* MODULE NAME: adg_server.c
* AUTHOR     : Compaq Computer Corporation
* DESCRIPTION: This server application receives transactions and receives
*              transaction status. It is to be used with adg_client.c,
*              adg_header.h, and adg_shared.c.
* DATE       : Oct 22, 1998
*************************************************************************/
/*
    adg_server.c
      Goes with adg_client.c

    To build on Unix:
        cc -o adg_server adg_server.c adg_shared.c -lrtr
*/

#include "adg_header.h"

void declare_server (rtr_channel_t *channel, const message_data_t *outmsg);

FILE *fpLog;

int main( int argc, char *argv[] )
{
    /*
     * This program expects 2 parameters :
     *   1: server number (1 or 2)
     *   2: partition range
     */

    rtr_msgsb_t msgsb;
    receive_msg_t receive_msg;
    message_data_t reply_msg;
    rtr_timout_t receive_time_out = RTR_NO_TIMOUTMS;
    char SvrLog[80];
    time_t time_val = { 0 };

    rtr_channel_t   channel;
```

```
rtr_status_t    status      = (rtr_status_t)0;
rtr_bool_t replay;

strcpy( reply_msg.text , "from Server");

get_server_parameters ( argc, argv, &reply_msg );

sprintf( SvrLog, "SERVER_%c_%d.LOG", reply_msg.routing_key,
         reply_msg.server_number );
fpLog = fopen( SvrLog, "w");

if ( fpLog == NULL )
{
    perror("adg_server: fopen() failed");
    printf( " Error opening server log %s\n", SvrLog );
    exit(EXIT_FAILURE);
}

printf( " Server log = %s\n", SvrLog );

fprintf(fpLog, " server number = %d\n", reply_msg.server_number );
fprintf(fpLog, " routing key = %c\n", reply_msg.routing_key);

declare_server(&channel, &reply_msg);

while ( RTR_TRUE )
{
    status = rtr_receive_message(
                                  &channel,
                                  RTR_NO_FLAGS,
                                  RTR_ANYCHAN,
                                  &receive_msg,
                                  sizeof(receive_msg),
                                  receive_time_out,
                                  &msgsb);
    check_status( "rtr_receive_message", status);

    time(&time_val);

    switch (msgsb.msgtype)
    {
    case rtr_mt_msg1_uncertain:
    case rtr_mt_msg1:
        if (msgsb.msgtype ==  rtr_mt_msg1_uncertain)
            replay = RTR_TRUE;
        else
            replay = RTR_FALSE;

       fprintf(fpLog, "\n ********** sequence %10d **********\n",
                receive_msg.receive_data_msg.sequence_number);

        if ( replay == RTR_TRUE )
            fprintf(fpLog, "   uncertain txn started at :%s",
                    ctime( &time_val));
        else
            fprintf(fpLog, "   normal txn started at :%s",
                    ctime( &time_val));
```

```
            fprintf(fpLog, "     sequence %10d from client %d\n",
                    receive_msg.receive_data_msg.sequence_number,
                    receive_msg.receive_data_msg.client_number);
            fflush(fpLog);

            reply_msg.sequence_number =
                    receive_msg.receive_data_msg.sequence_number;

            status = rtr_reply_to_client (
                                        channel,
                                        RTR_NO_FLAGS,
                                        &reply_msg,
                                        sizeof(reply_msg),
                                        msgfmt);

            check_status( "rtr_reply_to_client", status);
            break;

        case rtr_mt_prepare:
            fprintf(fpLog, "   txn prepared at :          %s",
                    ctime( &time_val));
            fflush(fpLog);

            status = rtr_accept_tx (
                                    channel,
                                    RTR_NO_FLAGS,
                                    RTR_NO_REASON);
            check_status( "rtr_accept_tx", status);
            break;

        case rtr_mt_rejected:
            fprintf(fpLog, "   txn rejected at :          %s",
                    ctime( &time_val));
            fprint_tid(fpLog, &msgsb.tid );
            fprintf(fpLog, "      status is : %d\n", status);
            fprintf(fpLog, "      %s\n", rtr_error_text(status));
            fflush(fpLog);
            break;

        case rtr_mt_accepted:
            fprintf(fpLog, "   txn accepted at :          %s",
                    ctime( &time_val));
            fprint_tid(fpLog, &msgsb.tid );
            fflush(fpLog);
            break;

        } /* End of switch */
    } /* While loop */
}

void
declare_server (rtr_channel_t *channel, const message_data_t *outmsg)
{
    rtr_status_t    status;
    rtr_uns_32_t    numseg = 1;
    rtr_keyseg_t    p_keyseg[1];
    receive_msg_t  receive_msg;
    rtr_timout_t receive_time_out = RTR_NO_TIMOUTMS;  /* forever */
```

```
    rtr_msgsb_t msgsb;   /* Structure into which receive puts msgtype */
    const char *facility = FACILITY_NAME;

    p_keyseg[0].ks_type = rtr_keyseg_string;
    p_keyseg[0].ks_length  = 1;
    p_keyseg[0].ks_offset  =  0;
    p_keyseg[0].ks_lo_bound =
             /* const_cast */ (rtr_uns_8_t *)(&outmsg->routing_key);
    p_keyseg[0].ks_hi_bound =
             /* const_cast */ (rtr_uns_8_t *)(&outmsg->routing_key);

    status = rtr_open_channel(
                    &channel,
                    RTR_F_OPE_SERVER,/* | RTR_F_OPE_EXPLICIT_ACCEPT | */
                                     /*   RTR_F_OPE_EXPLICIT_PREPARE, */
                    facility,
                    NULL,            /* rpcnam */
                    RTR_NO_PEVTNUM,
                    RTR_NO_ACCESS,   /* access */
                    numseg,
                    p_keyseg);

    check_status( "rtr_open_channel", status);

    status = rtr_receive_message(
                    &channel,
                    RTR_NO_FLAGS,
                    RTR_ANYCHAN,
                    &receive_msg,
                    sizeof(receive_msg),
                    receive_time_out,
                    &msgsb);

    check_status( "rtr_receive_message", status);

    if ( msgsb.msgtype != rtr_mt_opened )
    {
        fprintf(fpLog, " Error opening rtr channel %s: \n", facility);

        fprintf(fpLog, "%s\n",
                rtr_error_text(receive_msg.receive_status_msg.status));
        fclose (fpLog);
        exit(EXIT_FAILURE);
    }

    fprintf(fpLog, "  Server channel successfully opened \n");
    return;
}
```

A.4. Shared Code

```
/* Shared Code */

/
*************************************************************************
* Copyright Compaq Computer Corporation 1998. All rights reserved.
* Restricted Rights: Use, duplication, or disclosure by the U.S.
```

```
* Government is subject to restrictions as set forth in subparagraph (c)
* (1) (ii) of DFARS 252.227-7013, or in FAR 52.227-19, or in FAR 52.227-14
* Alt. III, as applicable.
* This software is proprietary to and embodies the confidential technology
* of Compaq Computer Corporation. Possession, use, of copying of this
* software and media is authorized only pursuant to a valid written license
* from Compaq, Digital or an authorized sublicensor.
*******************************************************************************/
/
********************************************************************************
* APPLICATION: RTR Sample Client Application
* MODULE NAME: adg_shared.c
* AUTHOR         : Compaq Computer Corporation
* DESCRIPTION: This shared code is to be used with adg_server.c,
*                          adg_header.h, and adg_client.c.
* DATE           : Oct 22, 1998
*******************************************************************************/

#include "adg_header.h"

void
check_status( char *call, rtr_status_t status )
{
    time_t time_val = { 0 };
    if (status != RTR_STS_OK)
    {
                time(&time_val);
                fprintf(fpLog, "    Call to %s failed at %s:\n",
                        call, ctime( &time_val));
                fprintf(fpLog, "\n Call status = %s\n",
                        rtr_error_text(status));
                fflush(fpLog);
                exit(status);
        }
}

void
get_server_parameters ( rtr_sgn_32_t argc, char *argv[], message_data_t
*o_msg)
{
    String31 buffer;
  if (argc < 2)
    {
                printf (" Server number : " );
                gets(buffer);
                o_msg->server_number = atoi(buffer);

                printf(" routing key : " );
                gets (buffer);
                o_msg->routing_key = buffer[0];
    }
    else
    {
                sscanf( argv[1], "%1d", &(o_msg->server_number) );
                o_msg->routing_key = *(argv[2]);
    }
} /* End of get_server_parameters */
```

```
void
get_client_parameters ( rtr_sgn_32_t argc, char *argv[], message_data_t
*o_msg, int *txn_cnt)
{
        String31        buffer;

        if (argc < 3)
        {
                printf (" Client number : " );
                gets(buffer);
                o_msg->client_number = atoi(buffer);

                printf(" routing key : " );
                gets (buffer);
                o_msg->routing_key = buffer[0];

                printf(" Message Count : " );
                gets (buffer);
                *txn_cnt = atoi(buffer);
        }
        else
        {
                sscanf( argv[1], "%1d", &(o_msg->client_number) );
                sscanf(argv[2], "%s", buffer );

        }
} /* End of get_client_parameters */

/
**************************************************************************/

void fprint_tid ( FILE *fp , rtr_tid_t  *tid )
{
          fprintf ( fp , "     tid: %x,%x,%x,%x,%x,%x,%x\n", tid->tid32[0],
tid->tid32[1],
                tid->tid32[2], tid->tid32[3], tid->tid32[4], tid->tid32[5],
                tid->tid32[6] );
}

void
close_channel ( rtr_channel_t channel )
{
    rtr_status_t status;

    printf ( " Closing Channel.\n" );

    status = rtr_close_channel (
                    channel ,
                    RTR_NO_FLAGS);

    check_status( "rtr_close_channel", status );

    return;

}
```

A.5. Header Code

```
/* Header Code */

/
*****************************************************************************
* Copyright Compaq Computer Corporation 1998. All rights reserved.
* Restricted Rights: Use, duplication, or disclosure by the U.S.
* Government is subject to restrictions as set forth in subparagraph (c)
* (1) (ii) of DFARS 252.227-7013, or in FAR 52.227-19, or in FAR 52.227-14
* Alt. III, as applicable.
* This software is proprietary to and embodies the confidential technology
* of Compaq Computer Corporation. Possession, use, of copying of this
* software and media is authorized only pursuant to a valid written
* license from Compaq, Digital or an authorized sublicensor.
*****************************************************************************/
/
*****************************************************************************
* APPLICATION: RTR Sample Application
* MODULE NAME: adg_header.h
* AUTHOR         : Compaq Computer Corporation
* DESCRIPTION: This header file is to be used with adg_server.c,
*                          adg_client.c, and adg_shared.c.
* DATE           : Oct 22, 1998
*****************************************************************************/
/*
    Header file for adg_client.c and adg_server.c

*/

#include "rtr.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <math.h>
#include <signal.h>
#include <ctype.h>
#include <fcntl.h>
#include <sys/stat.h>
/* #include <sys/types.h> */

#define PERMS 0666 /* File permissions */
#define FACILITY_NAME "DESIGN"

typedef char String31[31];
typedef char String200[200];

typedef  struct {
    rtr_uns_8_t     routing_key;
    rtr_uns_32_t    server_number;
    rtr_uns_32_t    client_number;
    rtr_uns_32_t    sequence_number;
    String31        text;
    } message_data_t;

typedef union {
```

```
    message_data_t receive_data_msg;
    rtr_status_data_t receive_status_msg;
    } receive_msg_t;

typedef struct  {
    rtr_uns_32_t    low;
    rtr_uns_32_t    high;
    rtr_uns_32_t    expected ;
    rtr_tid_t       prior_txn;
    rtr_uns_32_t    prior_seqno;
    }boundaries_t;

/* Function prototype section */

void
check_status( char *call, rtr_status_t status );

void get_client_parameters ( rtr_sgn_32_t argc,
                        char *argv[],
                        message_data_t *o_msg,
                        int *txn_cnt);
void get_server_parameters ( rtr_sgn_32_t argc,
                        char *argv[],
                        message_data_t *o_msg);

rtr_status_t send_reply (
                        message_data_t *o_msg,
                        rtr_channel_t channel);

void close_channel ( rtr_channel_t channel );

void fprint_tid (FILE *fpLog, rtr_tid_t  *tid );

/* External section */

extern String200            Errormsg;

extern time_t               time_val;
extern boundaries_t         txn_range[10];
extern char                 TxnLog[];
extern char                 SvrLog[];
extern rtr_uns_32_t         msg_cnt;
extern int                  errno;
extern FILE*                fpLog;
```

Appendix B. RTR Application Development Tutorial

Start here!

Purpose:

This tutorial goes through all of the steps needed to set up a simple RTR-based application for a new user. 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. This tutorial uses the implicit start, prepare, and accept transaction capabilities of RTR that are described in the *VSI Reliable Transaction Router Application Design Guide*, a prerequisite for using this manual.

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 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's 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:

To fully develop this system, you will need a client application and frontend node, a server application and two backend nodes, and a router. What are these things?

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 to 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 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 will respond 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 supported as a server. 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 confirm the result 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 coworkers' desks, helping him or her to earn their 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, we will keep our router 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 will be referred to as FE (frontend), primary server as BE1 (backend 1), secondary server as BE2 (backend 2). Remember that the router will be on the FE machine. The journal must be accessible to both backend servers.

Start RTR:

You will need to start RTR on each of the machines on which you have installed it. You may do this from one machine. 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 will then be at the RTR> prompt, and can start RTR on all of the nodes. For example, on a UNIX system, it will look 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, rtr.exe, or RTRACP. 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.

Create a Recovery Journal:

This step holds the key to letting the second server pick up on the work at exactly the right time; no work is lost, and the hot swap to the standby server is “automagic.” RTR keeps track of the work being done by writing data to this 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 a VMS system.

```
 $  RTR RTR> create journal
user2 RTR> exit
```

Be sure to do this on both machines or you can use the /NODE qualifier to do it on each machine from one node.

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.
- Use a disk served via NFS with UNIX systems.
- Use a share when using Windows systems.

In any case, you should be sure the disk is not on your primary server, because 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 primary server and the standby server must be physically separate machines.

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 command prompt window:

```
C:\> rtr
RTR> create facility RTRTutor/node=(FE,BE1,BE2) -
_RTR> /frontend=FE/router=FE/backend=(BE1,BE2)
RTR> exit
```

With this command, you have now:

- Created a Facility named RTRTutor on all three nodes.
- 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.

Take a Break:

At this point you have accomplished a lot; you've configured RTR to protect a multitiered application by providing failover capability, and to handle communications between your client and your server. Next, you will write the application: your client will talk to RTR, and your server will talk to RTR. RTR will deliver the messages between them and, if the server crashes, bring in the standby server to handle your client's requests. The client will never know that the server has been switched, and no data or requests to retrieve or modify data will be lost!

Application:

The C modules and header files for this application are located in the examples subdirectory of the directory into which you installed RTR. They consist of the following files:

adg_client.c	The client application
adg_server.c	The server application
adg_shared.c	C code common to both the client and server applications
adg_header.h	Header file containing definitions specific to both sample applications

Although you won't have much typing to do, this tutorial will explain what the code in each module is doing. Copy all four of these files into a working directory of your own. For convenience, you may also wish to copy rtr.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 adg_header.h and change the FACILITY value to “RTRTutor”.

The application example 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 will talk only to RTR, who will move the messages and data between them. And you are free not to worry 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 adg_client.c, adg_shared.c and adg_header.h.

All applications that wish to talk to RTR through its API need to include rtr.h as a header file. This file lives in the directory into which RTR was installed, and contains the definitions for RTR structures 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 (adg_header.h), whenever you need additional definitions for your application.

```
#include "adg_header.h"
#include "rtr.h"
```

The client application design follows this outline:

1. Initialize RTR
2. Send a message to the server
3. Get a response from the server
4. Decide what to do with the response

Pretty straightforward, don't you think? Let's look at how it's done.

Initialize RTR:

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 will find this happening in the declare_client function in adg_client.c, and somewhat more simplified here.

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'll open a channel that enables communication between the client and the RTR router. Remember that the RTR router has been described as “keeping track of everything” that goes on in an RTR application.

Declare the items needed for the open channel call:

```
rtr_status_t    status;   /* will be returned by RTR */
rtr_channel_t   channel;  /* a channel */
```

Open the channel:

```
status = rtr_open_channel(
            &channel,           /* channel of communication */
            RTR_F_OPE_CLIENT ,  /* I am a client */
            "RTRTutor",         /* the facility we created */
            NULL,               /* recipient name */
            RTR_NO_PEVTNUM,     /* don’t send events, just messages */
            RTR_NO_ACCESS       /* access key */
            RTR_NO_NUMSEG ,     /* number of key segments */
            RTR_NO_PKEYSEG );   /* first key segment */
```

Let's examine what this open channel call does. First, the channel parameter we sent to it is only a pointer to a block of memory; we've done nothing to set any values in it. RTR will use this block of memory to store the information it needs to assign and keep track of this channel. The channel 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.

The second parameter 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.

And now [trumpets are heard in the distance!] the third parameter 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.

At this point, RTR has all of the information it needs to put the pieces together into one system; youre ready to start sending messages to the server, and to get messages back from it.

A Word About RTR's API Parameters:

You may have noticed that although we've looked at only three of the parameters in the open channel call, there are a number more. It's a quirk of RTR that you'll often need to tell it to default. Rather than defaulting on its own when you do not provide a parameter (or provide a null parameter), it needs the “default” parameter. So you'll see things like RTR_NO_PEVTNUM to tell it “I don't want to be notified of any events” which is actually a default, and RTR_NO_NUMSEG to tell it “I have defined no key segments” which is also a default. Whenever we skip the discussion on non-null parameters, you'll know they are default parameters.

The parameter RTR_NO_FLAGS tells RTR that there are no flags.

A Word About RTR's 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 status 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 this (with a different string in the first parameter each time):

```
check_status( "rtr_open_channel", status );
```

This is good because, as you know from your Programming 101 course, you should always check your return status. But it's also good that your program knows when something has gone wrong and can tell the user, or behave accordingly. The check_status function is not part of RTR, but is something you will probably want to do in your application.

To check RTR's return status, 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 want to print a message to the user. To get the text string that goes with this status, call rtr_error_text which returns a null terminated ASCII string containing the message in human readable format.

```
if (status != RTR_STS_OK)
{
printf("    Call failed: %s", rtr_error_text(status));
}
```

Receiving Messages:

As explained earlier, RTR doesn't hold your client up while it processes your request, or even a request from another client. And because nothing can continue until the system has been set up, you now need to wait for the open channel call to let you know that everything is started up and ready to go. This is what the rest of the code in the “declare_client” function does. These statements declare the memory for a “receive” message and a message status block:

```
receive_msg_t receive_msg = {0}; /* message received */
rtr_msgsb_t msg_status;     /* message status block */
```

And now the rtr_receive_message waits to receive a message from RTR.

```
status = rtr_receive_message(
            &channel,           /* channel on which message received */
            RTR_NO_FLAGS,       /* sending no flags (default) */
            RTR_ANYCHAN,        /* default channel */
            &receive_msg,       /* location to place return info */
            sizeof(receive_msg),/* size of last */
            RTR_NO_TIMEOUTMS,   /* do not timeout */
            &msg_status);       /* location to return status */
```

The channel parameter and the RTR_NO_FLAGS parameter should now be familiar to you; we discussed them in the sections of this document on Initialize and Parameters. RTR_ANYCHAN and RTR_NO_TIMEOUTMS are defaults for this API.

Remember Programming 101 — check your status every time!

Information about whether RTR or your server has successfully handled your client's request is returned in an rtr_msgsb_t message status block structure. It is received from RTR as the last parameter in the rtr_receive_message call. For rtr_open_channel, we are looking for the “rtr_mt_opened” message type in the status block to confirm that the channel has been opened, and that we are now prepared to do all of the rest of the messaging on it for our application. If we don't have the “opened” message, then we can expect there to be an error status in the receive message block.

```
if ( msg_status.msgtype != rtr_mt_opened )
{
printf(" Error opening rtr channel : ");
    printf( rtr_error_text(receive_msg.receive_status_msg.status));
```

```
    }
```

The rtr.h header file provided with the RTR installation kit describes the rtr_msgsb_t structure in detail.

Send Messages:

The rest of the client application is simply a send/receive message loop. It continues to send messages to the server, then listens 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 the RTR router. Because of this, the client can receive, asynchronously, different types of messages:

- 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 status

In addition, 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 in this tutorial we will explain how to run a “message loop” that both sends and receives messages.

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 a VMS 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 Tru64 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 converts the data to the native data types on the operating system with which it happens to be communicating at the time.

Think of RTR as your very own “Babel fish,” if you've read the “Hitchhiker's Guide to the Galaxy” series. It will translate everything necessary when your data gets to a new machine. The little fish you put in your ear is actually made up of the RTR application programming interface and the RTR data types.

To illustrate this, the example code evaluates your input parameters and places them into a message_data_t structure called send_msg. Message_data_t is not an RTR structure, but created by the programmer who wrote this client. The message_data_t structure is defined in adg_header.h.

```
typedef  struct {
    rtr_uns_8_t     routing_key;
    rtr_uns_32_t    server_number;
    rtr_uns_32_t    client_number;
```

```
    rtr_uns_32_t    sequence_number;
    String31        text;
    } message_data_t;
```

You'll notice that the data types that make up message_data_t 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.

Earlier, we looked at the receive message code when the client opened a channel. The structure it used to obtain information, receive_message_t, is also one created by the programmer, and not a standard RTR structure. If you look at its definition in the adg_header.h file, you'll see that it's the same as the message_data_t structure, plus it contains a location for RTR status. There will be more on this in the next section.

Send/Receive Message Loop:

As mentioned earlier, the sample code for the RTR client application 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.

The following discussion will reference a simplified version of that loop; code in the sample to add time stamps and print to a log file has been removed here for clarity.

When you run the sample client, the client expects three parameters: a client number, a partition range, and the number of messages to send, in that order. We will talk more about partition ranges later when we look at the server application, but for now it is enough to know that we'll use one character, the letter h.

The input command parameters are evaluated and placed in the message_data_t structure named send_msg. The number of messages parameter which you'll input on the command line is placed in the txn_cnt variable. The for loop which sends and receives messages will execute this number of times.

The message_data_t structure also holds a sequence number value that is incremented each time the loop is executed; so now our loop begins:

```
for ( ; txn_cnt > 0; txn_cnt--, send_msg.sequence_number++ )
{
      status = rtr_send_to_server(
                channel,
                RTR_NO_FLAGS ,
                &send_msg,
                sizeof(send_msg),
                RTR_NO_MSGFMT );

      check_status( "rtr_send_to_server", status );
```

Note

The check_status function is not part of RTR; you must define it in the application.

The first message has been sent to the server in the third parameter of the rtr_send_to_server call. As you will see, this is part of the flexibility and power of RTR. This third parameter is no more than 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 structure that carries any and all of the information your server will need to do your bidding. We'll see this in detail when we look at the server code.

In the fourth parameter, you must tell RTR how big the piece of memory being pointed to by the third parameter is. RTR needs to 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.

The rest of the parameters bear some looking at, as well: there's the channel again. You'll see the channel parameter in almost every RTR call. You may be becoming suspicious about the channel, and think that it's really more than just a line for communicating. And you'd be right. RTR uses the channel much like you use that third parameter in this call. The RTR developers are the only ones who know what's in it, and it contains much of the information they need to make RTR work.

You'll recognize two more default parameters, RTR_NO_FLAGS and RTR_NO_MSGFMT.

And now, the client waits for a response from the server.

```
/*
 *  Get the server's reply OR an rtr_mt_rejected
 */
 status = rtr_receive_message(
           &channel,
           RTR_NO_FLAGS,
           RTR_ANYCHAN,
          &receive_msg,
          sizeof(receive_msg),
          RTR_NO_TIMOUTMS,
           &msgsb);

 check_status( "rtr_receive_message", status );
```

Again you see the channel and the default flags; the receive_msg parameter is a pointer to another data structure created by you as the programmer, and can carry any information you need your server to be able to communicate back to your client. In your own application, you would actually create this data structure in your application's header file. You can see what the example receive message looks like by checking out the receive_msg_t in the adg_header.h file. RTR picks it up from your server and writes it here for your client to read.

The *msgsb* parameter is an RTR data structure: you saw this message status block earlier when we looked at the open channel code. Its msgtype field contains a code that tells you what kind of a message you are now receiving. If msgsb.msgtype contains the value rtr_mt_reply, then you are receiving a reply to a message you already sent, and your receive message data structure has been written to with information from your server.

```
switch (msgsb.msgtype)
{
case rtr_mt_reply:
     fprintf(fpLog, "     sequence %10d from server %d\n ",
                    receive_msg.receive_data_msg.sequence_number,
                    receive_msg.receive_data_msg.server_number);
     break;
```

If msgsb.msgtype 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 status returned by the rtr_receive_message call. You will recall that making the rtr_error_text call

and passing the status value will return a human readable null terminated ASCII string containing the error message.

```
case rtr_mt_rejected:
       fprintf(fpLog, "  txn rejected at: %s",
                    ctime( &time_val));
       fprint_tid(fpLog, &msgsb.tid );
```

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 resends a rejected transaction to the server.

```
      /* Resend message with same sequence_number after reject */
      send_msg.sequence_number--;
      txn_cnt++;
      break;

default:
      fprintf(fpLog, "  unexpected msg”);
      fprint_tid(fpLog, &msgsb.tid );
      fflush(fpLog);
      exit((int)-1);
}
```

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 status 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)
      {
            status = rtr_accept_tx(
                           channel,
                           RTR_NO_FLAGS,
                           RTR_NO_REASON );

            check_status( "rtr_accept_tx", status );
```

And now the client waits to find out the result of the voting.

```
            status = rtr_receive_message(
                           &channel,
                           RTR_NO_FLAGS,
                           RTR_ANYCHAN,
                           &receive_msg,
                           sizeof(receive_msg),
                           receive_time_out,
                           &msgsb);

            check_status( "rtr_receive_message", status );
            time(&time_val);
```

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.

```
            switch ( msgsb.msgtype )
            {
            case rtr_mt_accepted:
                  fprintf(fpLog, "   txn accepted at :  %s",
                              ctime( &time_val));
                  break;

            case rtr_mt_rejected:
                  fprintf(fpLog, "   txn rejected at :  %s",
                              ctime( &time_val));

                   /* Resend same sequence_number after reject */
                   send_msg.sequence_number--;
                   txn_cnt++;
                   break;

            default:
                   fprintf(fpLog,
                   " unexpected status on rtr_mt_accepted message\n");

                   fprintf(fpLog,
                   " %s\n",

 rtr_error_text(receive_msg.receive_status_msg.status);

                  break;
            }
         }
    } /* end of for loop */
```

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 channel. This is similar to signing off, and RTR releases all of the resources it was holding for the client application.

```
     close_channel ( channel );
```

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 adg_server.c, adg_shared.c and adg_header.h. You'll notice that adg_shared.c and adg_header.h 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 will use the same definition for the data structures 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 opens a channel to the router, telling the router that it is a *server* application; waits to hear that the open channel 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 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 declaration of a partition that is sent to RTR in the open channel call. We mentioned partitions while discussing the client application, but said we'd discuss them later. Well, it's later...

Initialize RTR:

Just like the client, the server opens a channel to the router, causing RTR to initialize a number of resources for use by the server, as well as to gather information about the server. In the declare_server function in the server example application, adg_server.c, you'll find the example server calling rtr_open_channel.

Immediately, you see that the code initializes an RTR data structure called rtr_keyseg_t. In the example server code, the variable name of the structure is p_keyseg. This structure is a required parameter in the server open channel call to implement *data partitioning*.

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 opens a channel to begin communicating with the RTR router, it uses the rtr_keyseg_t information in its last two parameters 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”.

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.

VSI Reliable Transaction Router Application Design Guide and API reference manual go into much more detail about data partitioning.

This is how the example server application defines the key segment that it will handle:

```
p_keyseg[0].ks_type = rtr_keyseg_string;
p_keyseg[0].ks_length  = 1;
p_keyseg[0].ks_offset  =  0;
p_keyseg[0].ks_lo_bound = &outmsg->routing_key;
p_keyseg[0].ks_hi_bound = &outmsg->routing_key;
```

It tells RTR that this server is interested only in records containing a *string* of *1 byte* at the beginning of the record; this actually makes it a single character. The value of that character is from and including the value of the character in the routing_key field of outmsg, to and including the value of the character in the routing_key field of outmsg. As you can see, this too describes only one character.

The structure outmsg is actually a msg_data_t structure, which is the structure you saw the client application using to pass data to the server application. The value of this character is input when you start the server. Because we decided to use the letter h when we start the client, it would be really nice if the

server we start identifies itself as one that can handle the client's request. So we'll start the server using h as well; in this way, the h gets into outmsg->routing_key. The complete server command line for both the client and the server is documented later in this tutorial.

```
status = rtr_open_channel(
              &channel,
              RTR_F_OPE_SERVER,
              "RTRTutor",
              NULL,
              RTR_NO_PEVTNUM,
              RTR_NO_ACCESS,
              1,
              p_keyseg);

check_status( "rtr_open_channel", status);
```

Note

The check_status function is not part of RTR; you must define it in the application.

The server has requested a channel on which to communicate with RTR, and advertised itself as handling all requests from clients in the RTRTutor facility that have a key segment value of *h*. The remaining parameters contain defaults.

Now the server waits for a message confirming that RTR opened the channel successfully. If it did, the server can then begin receiving requests from the client, via RTR.

```
     status = rtr_receive_message(
                    &channel,
                    RTR_NO_FLAGS,
                    RTR_ANYCHAN,
                    &receive_msg,
                    sizeof(receive_msg),
                    receive_time_out,
                    &msgsb);

     check_status( "rtr_receive_message", status);
```

Again, we use the RTR rtr_msgsb_t structure that RTR will place information in, and the user-defined receive_msg_t data structure (see adg_header.h) 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 in msgsb; all the server really wants to know is that the channel has been opened successfully. If it hasn't, the server application will write out an error message and exit with a failure status.

```
if ( msgsb.msgtype != rtr_mt_opened )
{
       fprintf(fpLog, " Error opening rtr channel : \n");
       fprintf(fpLog,
             "%s",
       rtr_error_text(receive_msg.receive_status_msg.status);
fclose (fpLog);
       exit(-1)
}

fprintf(fpLog, "  Server channel successfully opened \n");
return;
```

And now that the channel 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.

```
while ( RTR_TRUE ) /* always, or until we exit */
{
        status = rtr_receive_message(
                    &channel,
                    RTR_NO_FLAGS,
                    RTR_ANYCHAN,
                    &receive_msg,
                    sizeof(receive_msg),
                    receive_time_out,
                    &msgsb);

         check_status( "rtr_receive_message", status);
```

Like the client, upon receiving the message the server checks the rtr_msgsb_t structure's msgtype field to see what kind of message it is. Some are messages directly from RTR and others are from the client. When the message is from the client, your application will read the data structure 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 favorite 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.

```
switch (msgsb.msgtype)
{
        case rtr_mt_msg1_uncertain:
         case rtr_mt_msg1:
```

The first message this server application prepares to handle is the *rtr_mt_msg1_uncertain* message. This is combined with the handler for the *rtr_mt_msg1* message.

The msg1 messages identify the beginning of a new transaction. Rtr_mt_msg1 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 structure pointed to by the fourth parameter of this call. The client and server have agreed on a common data structure that the client will send to the server whenever it makes a request: this is the message_data_t we looked at in the client section of this document. RTR has copied the data from the client's data structure into the one whose memory has been supplied by the server. The server's responsibility when receiving this message is to process it.

Recovered Transactions:

The *rtr_mt_msg1_uncertain* message type tells the server that this is the first message in a *recovered* transaction. 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. In the examples, *rtr_msgsb_t* must be declared as a variable, as *rtr_msgsb_t msgsb;*.

```
if (msgsb.msgtype ==  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.

```
reply_msg.sequence_number =
         receive_msg.receive_data_msg.sequence_number;

status = rtr_reply_to_client (
               channel,
               RTR_NO_FLAGS,
               &reply_msg,
               sizeof(reply_msg),
               RTR_NO_MSGFMT);
```

The rtr_reply_to_client call is one you haven't seen before. Obviously, it is responding to a client's request. This call may not be used on a channel in an application that has declared itself a client.

The server is using the rtr_reply_to_client 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 reply_msg structure which, like the send_msg structure, has been agreed upon by both the client and the server. RTR will copy sizeof bytes from the server's copy of the reply_msg into the client's copy.

```
 check_status( "rtr_reply_to_client", status);
           break;

case rtr_mt_prepare:
```

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. To make this request, the server must use the RTR_F_OPE_EXPLICIT_PREPARE flag in the flags parameter when opening the channel.

After determining whether it is possible to complete the transaction based on what has occurred to this point, the server can either call rtr_reject_tx to reject the transaction, or set all of the required locks on the database before calling rtr_accept_tx to accept the transaction.

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.

```
                status = rtr_accept_tx (
                      channel,
                      RTR_NO_FLAGS,
                      RTR_NO_REASON);

                check_status( "rtr_accept_tx", status);

                break;

        case rtr_mt_rejected:
```

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.

```
          break;

        case rtr_mt_accepted:
```

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.

```
  break;

        } /* end of switch */
    } /* end of while loop */
```

Note that there is no close_channel call in the server. This is because the RTR router closes the channel and stops the server when it sees fit. RTR makes this decision.

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 adg_server.c and adg_shared.c module 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:

```
        cc -o server adg_server.c adg_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 adg_client.c and adg_shared.c module on the operating system that will run your client application.

To build on UNIX:

```
% cc -o client adg_client.c adg_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
```

But Wait! There's More!

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 manual will tell you in detail how to implement each capability.

Training Services offers training classes for RTR. If you'd like to attend any of them, contact your local 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 when it opens its channel to the router. Callout servers are asked to check all requests in a facility, and are asked to vote on every transaction.

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 for RTR to 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. So as you can see, if your database and transactions are important enough to you, you have the opportunity to double protect them with an RTR configuration that includes some of the following:

- A standby software server on a primary hardware backend system
- A shadow backend system replicating all transactions on a duplicate database
- Failover backend systems for each of your primary backends
- Failover routers
- Concurrent servers

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 CLI) 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.