VMS Software

VSI OpenVMS

VSI Reliable Transaction Router Application Design Guide

Document Number: DO-RTRADG-01A

Publication Date: April 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 Application Design Guide

Legal Notice

Preface

This guide explains how to design application programs for use with VSI Reliable Transaction Router.

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

The goal of this document is to assist the experienced application programmer in designing applications for use with Reliable Transaction Router (RTR). Here you will find conceptual information and some implementation details to assist in:

- Creating new applications
- Updating existing applications

As an application programmer, you should be familiar with the following concepts:

- Distributed systems
- Client/server environment
- C or C++ programming
- Transaction processing

If you are not familiar with these software concepts, you will need to augment your knowledge by reading, taking courses, or through discussion with colleagues. You should also become familiar with the other books in the RTR documentation kit, listed in Related Documentation. Before reading this document, become familiar with the information in *VSI Reliable Transaction Router Getting Started*.

This document is intended to be read from start to finish; later you can use it for reference.

Section and Appendix Moved

- The section on Journal Sizing from Chapter 2 has been moved to the *VSI Reliable Transaction Router System Manager's Manual*, Section 2.3, Creating a Recovery Journal.
- Appendix G, Dual-Rail Setup, has been moved to the *VSI Reliable Transaction Router System Manager's Manual*, Section 2.14, Network Transports.

---

3. Related Documents

Below is the list of books that can be helpful in developing your transaction processing application include:

- Philip A. Bernstein, Eric Newcomer, *Principles of Transaction Processing*, Morgan Kaufman, 1997

- Jim Gray, Andreas Reuter, *Transaction Processing: Concepts and Techniques*, Morgan Kaufman, 1992
- *Oracle8 Application Developer's Guide* and *Oracle8i Application Developer's Guide*

A web search for “Oracle SQL” can provide tutorial and other useful references.

4. 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.

5. OpenVMS Documentation

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

6. 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.

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

Chapter 1. Introduction

This document is for the application programmer who is developing an application for use with Reliable Transaction Router (RTR). Here you will find information on using RTR in the design and development of an application. The main emphasis is on providing design suggestions and considerations for writing the RTR application. Example designs describing existing applications that use RTR show implementations exploiting RTR features, and provide real examples where RTR is in use.

Note

Before reading this manual, read the prerequisite: *VSI Reliable Transaction Router Getting Started*, which describes basic RTR concepts and terminology.

RTR provides both a C++ object-oriented application programming interface (API) and a C API. Most of the material in this document is generic to RTR and not specific to either interface. However, some implementation specifics for each API are shown in separate chapters and appendices.

Each API also has its own reference document, the *VSI Reliable Transaction Router C++ Foundation Classes* manual for the C++ API, and the *VSI Reliable Transaction Router C Application Programmer's Reference Manual* for the C API.

In designing your application, VSI recommends that you use object-oriented analysis and design techniques, whether or not you are using the new object-oriented API. This methodology includes the following:

- Performing use-case analysis
- Tracing scenarios
- Determining actors and classes
- Establishing object interactions

Discussing this methodology is outside the scope of this document, but you can find more information in the following books, among others:

- James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, William Lorensen, *Object-Oriented Modeling and Design*, Prentice-Hall, 1991
- Martin Fowler, Kendall Scott, *UML Distilled Applying the Standard Object Modeling Language*, Addison-Wesley, 1997

When designing your application:

- Consider your application requirements fully.
- Work through the entire design and its operational results.
- Understand both the logical and physical design of your database, including any partitioning of your database.

A design flaw can be very expensive or impossible to correct in your application, so doing a thorough design, fully discussed and understood by your team, is essential to the ultimate success of your application in operation. One goal of this document is to help you understand some of the finer subtleties of the product, to help you design an optimum application.

1.1. Reliable Transaction Router

RTR is transactional messaging middleware used to implement highly scalable, distributed applications with client/server technologies. With RTR, you can design and implement transaction management applications, take advantage of high availability, and implement failure-tolerant applications to maximize uptime. RTR helps ensure business continuity across multivendor systems and helps maximize uptime.

Implementing an application to use RTR embeds high availability capabilities in the application. Furthermore, RTR greatly simplifies the design and coding of distributed applications because, with RTR, client-server interactions can be bundled together in transactions. In addition, RTR applications can easily be deployed in highly available configurations.

RTR supports applications that run on different hardware and different operating systems. RTR has not been built to work with any specific database product or resource manager (file system, communication link, or queuing product), and thus provides an application the flexibility to choose the best product or technology suited for its purpose. There are in addition some resource managers with which RTR provides tight coupling.

For more information on using this tighter coupling, see the section in this manual on Using XA. For specifics on operating systems, operating system versions, and supported hardware, see the *VSI Reliable Transaction Router Software Product Description* for each supported operating system.

RTR provides several benefits for your application:

- High availability

  An application designed to work with RTR can take advantage of RTR failover capabilities and system availability solutions such as hardware clusters. Transactional shadowing and single input no need to log on to another node after a failure) with input logging are additional features that provide RTR high availability. You can take advantage of configurations that tolerate process failure, node failure, network failure, and site failure.

- High security

  To protect your data and transactions against unauthorized tampering, you can use the authentication server (also known as the callout server) that is available with RTR, and operating system security features and tunnels.

- Data retention

  To ensure against loss of data, you can use RTR transactional shadowing. Transactional shadowing can be done at a single site or at geographically separate sites. In deploying your application, you can locate sites in different cities or on different power grids to protect against data loss.

- High transaction performance

  To design your application for scalability, you can use a partitioned model to exploit RTR data-content routing, and you should evaluate hardware constraints that may limit an application's performance in processing transactions.

RTR is a middleware product used to connect client and server applications in a distributed computing environment. At the same time, it enhances the overall solution by providing transactional semantics, XA and DECdtm integration with databases, concurrent, standby, and shadow configurations for scalability, fault tolerance and disaster tolerance.

RTR consists of three logical entities, the frontend (FE), the transaction router (TR) and the backend (BE). These entities run on computer nodes (for example OpenVMS, Windows). On any node, any combinations of these entities can be configured.

The application client programs all run on the FE nodes, which may be web servers using an access tool to connect to the RTR FE. The server programs all run on the BE nodes, for example OpenVMS applications, and connect to the RTR BE. There is no application software on the TR, but it takes care of routing messages between client and server applications, takes care of two-phase commit for distributed transactions, of failover and failback in case of errors and outages, and in general maintains overall control of the distributed system.

All RTR programs connect to RTR locally on the same node. Internode connections are established and maintained by RTR. RTR uses a mechanism of content-based routing to route client messages to server applications. By this, the application can tell RTR which field in the message is the routing key and also which range of values of that key a particular server is supposed to service.

RTR, however, is not a plug-and-play product. Client and server applications must make RTR calls (rtr_open_channel) to connect to RTR, to send a message (rtr_send_to_server, rtr_reply_to_client), and to accept or reject a transaction (rtr_accept_tx, rtr_reject_tx).

1.2. RTR Application Programming Interfaces

RTR provides programming interfaces for use in implementing applications that work with RTR, including:

- the C++ API

  The C++ API is described in the *VSI Reliable Transaction Router C++ Foundation Classes* manual and shown in this manual.

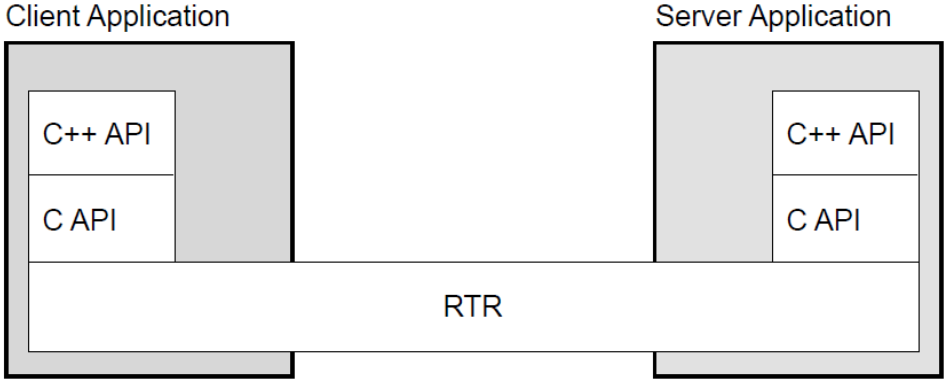
- The C API

  The C API is described in the *VSI Reliable Transaction Router C Application Programmer's Reference Manual* and shown in this manual.

1.2.1. The C++ API

The C++ API is an object-oriented application programming interface for RTR. With the C++ API, as an application developer, you can design and implement applications to take advantage of object-oriented (OO) design principles, including patterns and code reuse. As Figure 1.1 illustrates, the C++ API sits on top of the C API, which is a layer on top of RTR.

Figure 1.1. RTR Layers



Client applications and server applications can use both the C++ API and the C API or just the C++ API. The C++ API maps RTR data elements, data structures, and C API functionality into the object-oriented design world.

The C++ API:

- Is 100% compatible with existing applications.
- Provides the features of the flat C API, plus many more.
- Allows RTR applications to focus on their business logic instead of the details of RTR.
- Provides all the benefits of OO design.
- Allows existing applications to benefit from the new interface in many ways.

C++ API and RTR Technology

The C++ API provides an object-oriented framework through which you can implement fault-tolerant systems. C++ API code resides beneath application or business logic code. Thus, the C++ API interfaces directly with RTR while application code does not. This transparency simplifies the development and implementation of transaction processing solutions with RTR.

OO Benefits

The C++ API was created to assist RTR customers who:

- Need to create RTR system management routines.
- Are writing common application code.
- Can take advantage of the benefits of OO design and development.
- Write some form of OO wrapper to the existing API.

The benefits include:

- Higher quality software.
- Lower maintenance costs.
- Reduced development time.
- Ease of extensibility.

With the C++ API, applications can inherit functionality from RTR.

The C++ API Provides Ease of Use

The C++ API provides a simplified way for you to implement RTR solutions. With the C++ API:

- Each major RTR concept is represented by its own individual class.
- Class factory support is provided for data objects.
- Clients and servers connect through transaction controller objects (automates and hides C API channel use).

- You do not need to provide handling for all messages and events; default handling is provided.
- The sending and receiving of data is abstracted to a high level.
- Simple methods let you obtain RTR internal information without a need to know the internals of RTR.

C++ API Design

The C++ API upgrades RTR technology by providing a set of classes that streamlines the development and implementation of RTR transaction processing applications. The C++ API has been designed to:

- Provide 100% functional and binary compatibility (backward compatibility) with existing applications.
- Provide an object model that can be implemented in any OO language.
- Provide an object model from which new and existing applications can benefit.
- Perform "common" tasks for the application (for example, for an application to receive a message for a client/server connection).
- Provide default implementation for applications where appropriate.
- Provide an easy-to-use framework for handling messages and events that applications can extend to suit their own business logic.

Use of the C++ API

The C++ API can be used to:

- Develop entirely new applications.
- Upgrade existing applications on a single tier of their application - client or server.
- Integrate individual C++ API classes into existing applications. For example, existing applications can easily use the property classes.
- Develop system management routines for both new and existing applications. The C++ API enables you to write management routines for both new and existing applications. Additionally, C++ API management classes simplify the process of moving from application design to implementation. For example, the RTRPartitionManager class enables you to write management routines for specifying server types, while methods such as CreateBackEndPartition enable you to specify the roles of servers for tolerating process and node failure.
- Add diagnostics to the application that can be viewed in an integrated display with RTR counters.

1.2.2. The C API

The C API enables applications to be written with RTR calls using the C programming language. The C API is described in the *VSI Reliable Transaction Router C Application Programmer's Reference Manual*.

Chapter 2. Configuration and Design

To assist you in designing fault-tolerant RTR transaction processing applications, this chapter addresses configuration and design topics. Specifying how your RTR components are deployed on physical systems is called *configuration*. Developing your application to exploit the benefits of using RTR is called *design* or application design. The following topics are addressed:

- Tolerating process failure
- Tolerating storage device failure
- Tolerating node failure
- Tolerating site disaster
- Design for performance
- Configuration for operability

Short examples for both C++ and C APIs are available in appendices. The *VSI Reliable Transaction Router C++ Foundation Classes* manual and the *VSI Reliable Transaction Router C Application Programmer's Reference Manual* provide longer code examples. Code examples are also available with the RTR software kit in the examples directory.

2.1. Tolerating Process Failure

To design an application to tolerate process failure, the application designer can use concurrent servers with partitions and possibly threads with RTR.

2.1.1. Using Concurrent Servers

A concurrent server is an additional instance of a server application running on the same node as the original instance. If one concurrent server fails, the transaction in process is replayed to another server in the concurrent pool.

The main reason for using concurrent servers is to increase throughput by processing transactions in parallel, or to exploit Symmetric Multiprocessing (SMP) systems. The main constraint in using concurrent servers is the limit of available resources on the machine where the concurrent servers run. Concurrent servers deal with the same database partition. They may be implemented as multiple channels in a single process or as channels in separate processes. For an illustration of concurrent servers, refer to the *VSI Reliable Transaction Router Getting Started* manual. By default, servers are declared concurrent.

When a concurrent server fails, the server application can fail over to another running concurrent server, if one exists. Concurrent servers are useful to improve throughput, to make process failover possible, and to help minimize timeout problems in certain server applications. For more information on this topic, see the section Section 3.2.6: Server-Side Transaction Timeouts later in this document.

Concurrent servers must not access the same file or the same record in a database table. This can cause contention for a single resource, with potential for performance bottleneck. The resources that you can usefully share include the database and access to global (shared) memory. However, an application may need to take out locks on global/shared memory; this would need to be taken into account during design. With some applications, it may be possible to reduce operating system overhead by using multiple channels in a process.

Performance in a system is usually limited by the hardware in the configuration. Evaluating hardware constraints is described in the *VSI Reliable Transaction Router System Manager's Manual*. Given unlimited system resources, adding concurrency will usually improve performance. Before putting a new application or system built with RTR into production, the prudent approach is to test performance. You can then make adjustments to optimize performance. Do not design a system that immediately uses all the resources in the configuration because there will be no room for growth.

Using concurrency also improves reliability, because RTR provides server process failover (the "three strikes and you're out " rule) when you have concurrent servers.

2.1.2. Using Threads

In addition to using concurrent processes, an application can use the following methods to help improve performance:

- Multiple threads
- Multiple transaction controllers or channels
- Multiple partitions

An application designer may decide to use threads to have an application perform other tasks while waiting for RTR, for example, to process terminal input while waiting for RTR to commit a transaction or send a broadcast.

To use multiple threads, you write your application as a threaded application and use the shared thread library for the operating system on which your application runs. Use one channel per thread (with the C API), or one TransactionController per thread (with the C++ API). The application must manage the multiple processes.

To use multiple channels in a thread, use the polled receive method, polling for `rtr_receive_message` (C API) or Receive (C++ API). The application must contain code to handle the multiple channels or transaction controllers. This is by far the most complex solution and should only be used if it is not possible to use multiple threads or concurrent processes.

When using multiple threads in a process, the application must do the scheduling. One design using threads is to allocate a single channel for each thread. An alternative is to use multiple channels, each with a single thread. In this design, there are no synchronization issues, but the application must deal with different transactions on each thread.

Using multiple threads, design and processing is more streamlined. Within each thread, the application deals with only a single transaction at a time, but the application must deal with issues of access to common variables. It is often necessary to use mutexes (resource semaphores) and locks between resources.

2.1.3. Using Multiple Partitions

To use multiple partitions in your application, your database must be *designed* with multiple partitions. You may also be able to use multiple partitions when you have multiple databases. In general, using multiple partitions can reduce resource contention and improve performance. In the case where contention for database resources is causing performance degradation, partitioning or repartitioning your database can improve performance. To reduce resource contention in a database:

- Distribute the database on multiple disks.

- Partition the database to reduce or prevent row contention.
- Use partition names and key segments to route data to the appropriate database partitions in your RTR application.

When you have multiple databases to which transactions are posted, you can also design your RTR application to use partitions and thereby achieve better performance than without partitioning.

2.2. Tolerating Storage Device Failure

To configure a system that tolerates storage device failure well, consider incorporating the following in your configuration and software designs:

- Shadowed storage devices
- RAID storage devices

Further discussion of these devices is outside the scope of this document.

2.3. Tolerating Node Failure

RTR failover employs concurrent servers, standby servers, shadow servers, and journaling, or some combination of these. To survive node failure, you can use standby and shadow servers in several configurations. If the application starts up a second server for the partition, the server is a standby server by default.

Consider using a standby server to improve data availability, so that if your backend node fails or becomes unavailable, you can continue to process your transactions on the standby server. You can have multiple standby servers in your RTR configuration.

The time-to-failover on OpenVMS and Tru64 UNIX systems is virtually instantaneous; on other operating systems, this time is dictated by the cluster configuration that is in use.

The C++ API includes management classes that enable you to write management routines that can specify server type, while the C API uses flags on the open channel calls.

2.3.1. Router Failover

RTR deals with router failures automatically and transparently to the application. In the event of a router failure, neither client nor server applications need to do anything, and do not see an interruption in service. Consider router configuration when defining your RTR facility to minimize the impact of failure of the node where a router resides. If possible, place your routers on *independent* nodes, not on either the frontend or backend nodes of your configuration. If you do not have enough nodes to place routers on separate machines, configure routers with backends. This assumes a typical situation with many client applications on multiple frontends connecting to a few routers. For tradeoffs, see the sections on Section 2.5 and Section 2.6 in this chapter.

Provide multiple routers for redundancy. For configurations with a large number of frontends, the failure of a router causes many frontends to seek an alternate router. Therefore, configure sufficient routers to handle reconnection activity. When you configure multiple routers, one becomes the current router. If it fails, RTR automatically fails over to another.

For read-only applications, routers can be effective for establishing multiple sites for failover without using shadowing. To achieve this, define multiple, nonoverlapping facilities with the same facility name

in your network. Then provide client applications in the facility with the list of routers. When the router for the active facility fails, client applications are automatically connected to an alternate site. Read-only transactions can alternatively be handled by a second partition running on a standby server. This can help reduce network traffic.

When a router fails, in-progress transactions are routed to another router if one is available in that facility.

2.3.2. Server Failover

Server failover in the RTR environment can occur with failure of concurrent, standby, or transactional shadow servers. Servers in a cluster have additional failover attributes. Conceptually, server process failures can be contrasted as follows:

- A concurrent server employs a different process running on the same node.
- A standby server that becomes active employs a different process on a different node in the same cluster.
- A shadow server employs a different process on a different node in a different cluster.

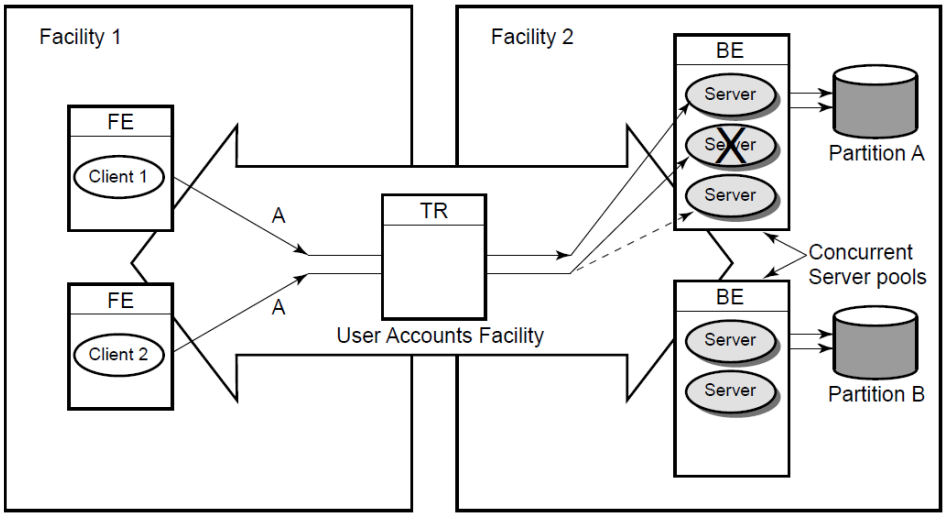
Note

A standby server can be configured over nodes that are *not* in the same cluster, but recovery of a failed node's journal is not possible until a server is restarted on the failed node.

Failover of any server is either event-driven or timer-based. For example, server loss due to process failure is event-driven and routinely handled by RTR. Server loss due to network link failure is timer-based, with timeout set by the SET LINK/INACTIVITY timer (default: 60 seconds). For more information on setting the inactivity timer, see the SET LINK command in the *VSI Reliable Transaction Router System Manager's Manual*.

For example, below illustrates the use of concurrent servers to process transactions for Partition A. When one of the concurrent servers cannot service transactions going to partition A, another concurrent server (shown by the dashed line) processes the transaction. Failover to the concurrent server is transparent to the application and the user.

Figure 2.1. Transaction Flow with Concurrent Servers and Multiple Partitions



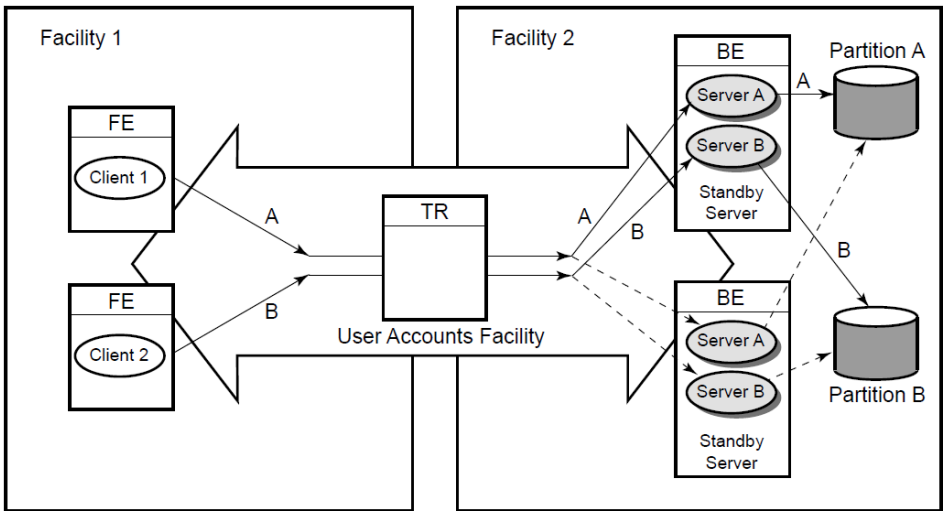
2.3.2.1. Concurrent Servers

Concurrent servers are useful in environments where more than one transaction can be performed on a database partition at one time to increase throughput.

2.3.2.2. Standby Servers

Standby servers provide additional availability and node-failure tolerance. Figure 2.2 illustrates the processing of transactions for two partitions using standby servers.

Figure 2.2. Transaction Flow on Standby Servers



When the configuration is operating normally, the primary servers send transactions to the designated partition (solid lines); transactions “A” proceed through primary server A to database partition A and transactions “B” proceed through primary server B to database partition B. However, when the primary server fails, the router reroutes transactions “A” through the standby server A ' to partition A, and transactions “B” through the standby server B ' to database partition B. Note that standby servers for different partitions can be on different nodes to improve throughput and availability. For example, the bottom node could be the primary server for partition B, with the top node the standby. The normal route is shown with a solid line, the standby route with a dashed line.

When the primary path for transactions intended for a specific partition fails, the standby server can still process the transactions. Standby servers automatically take over from the primary server if it fails, transparently to the application. Standby servers recover all in-progress transactions and replay them to complete the transactions.

As shown in Figure 2.2, there can be multiple standby servers for a partition.

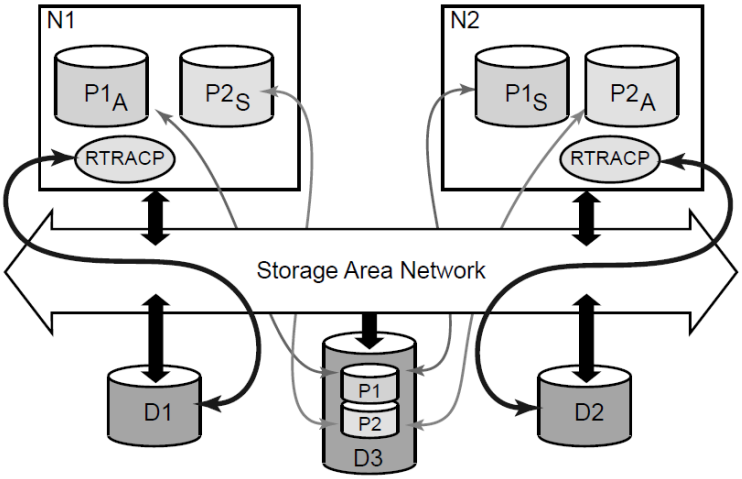
2.3.2.2.1. Standby Support in a Cluster

Failover and transaction recovery behave differently depending on whether server nodes are in a cluster configuration. Not all “cluster” systems are recognized by RTR; RTR recognizes only the more advanced or “true” cluster systems. Figure 2.3 shows the components that form an RTR standby server configuration.

Two nodes, N1 and N2, in a cluster configuration are connected to shared disks D1, D2 and D3. Disks D1 and D2 are dedicated to the RTR journals for nodes N1 and N2 respectively, and D3 is the disk that hosts the clustered database. This is a partitioned database with two partitions, P1 and P2.

Under normal operating conditions, the RTR active server processes for each partition, P1 $_A$ and P2 $_A$ run on nodes N1 and N2 respectively. The standby server processes for each partition run on the other node, that is, P1 $_S$ runs on N2 and P2 $_S$ runs on N1. In this way, both nodes in the cluster actively participate in the transactions and at the same time provide redundancy for each other. In case of failure of a node, say N1, the standby server P1 $_S$ is activated by RTR and becomes P1 $_A$ and continues processing transactions without any loss of service or loss of in-flight transactions. Both the active and standby servers have access to the same database and therefore both can process the same transactions.

Figure 2.3. RTR Standby Servers



2.3.3. The Cluster Environment

Failover between RTR standby servers behaves differently depending on the type of cluster where RTR is running. Actual configuration and behavior of each type of cluster depends on the operating system and the physical devices deployed. For RTR, configurations are either true clusters or host-based clusters.

2.3.3.1. True or Recognized Clusters

True clusters are systems that allow direct and equal access to shared disks by all the nodes in the cluster, for example OpenVMS and Tru64 UNIX (Version 5.0). Since concurrent access to files must be managed across the cluster, a distributed lock manager (DLM) is typically used as well. Since all cluster members have equal access to the shared disks, a failure of one member does not affect the accessibility of other members to the shared disks. This is the best configuration for smooth operation of RTR standby servers. In such a cluster configuration, RTR failover occurs immediately if the active node goes down.

2.3.3.2. Host-Based or Unrecognized Clusters

Host-based clusters include MSCS on Windows, Veritas for Solaris, IBM AIX and Tru64 UNIX (versions before 5.0). These clusters do not allow equal access to the disks among cluster members. There is always one host node that mounts the disk locally. This node then makes this disk available to other cluster members as a shared disk. All cluster members accessing this disk communicate through the host. In such a configuration, failure of the host node affects accessibility of the disks by the other members. They will not be able to access the disks until the host-based cluster software appoints another host node and this node has managed to mount the disks and export them. This will cause a delay in the failover, and also introduces additional network overhead for the other cluster members that need to access the shared disks.

2.3.3.3. Behavior in Recognized Clusters

The cluster systems currently recognized by RTR are: OpenVMS, TruCluster systems on Tru64 UNIX and Microsoft Cluster Server (MSCS) on Windows. Cluster behavior affects how the standby node fails over and how transactions are recovered from the RTR journal. For RTR to coordinate access to the shared file system across the clustered nodes, it uses the Distributed Lock Manager on both OpenVMS and Tru64 UNIX. On Windows and Sun, RTR uses file-based locking.

Failover in Recognized Clusters

When the active server goes down, RTR fails over to the standby server. Before that, RTR on the upcoming active node attempts to perform a scan of the failed node's journal. Since this is a clustered system, the cluster manager fails over the disks as well to the new active node. RTR will wait for this failover to happen before it starts processing new transactions.

Transaction Recovery in Recognized Clusters

In all the recognized clusters, whenever a failover occurs, RTR attempts to recover all the in-doubt transactions from the failed node's journal and replay them to the new active node. If RTR on the upcoming active server node cannot access the journal of the node that failed, it waits until the journal becomes accessible. This wait allows for any failover time in the cluster disks. This is particularly relevant in host-based clusters (for example, Windows clusters) where RTR must allow time for a new host to mount and export the disks. If after a certain time the journal is still inaccessible, the partition state goes into local recovery fail. In such a situation, the system manager must complete the failover of the cluster disks manually. If this succeeds, RTR can complete the recovery process and continue processing new transactions.

RTR does not recognize the cluster systems available for Sun Solaris.

2.3.3.4. Behavior in Unrecognized Clusters

Failover and transaction recovery in unrecognized clusters is slightly different from such operations in recognized clusters.

Failover in Unrecognized Clusters

When the active server goes down, RTR fails over to the standby server. RTR treats unrecognized cluster systems as independent non-clustered nodes. In this case, RTR scans for the failed node's journal among the valid disks accessible to it. However if it does not find it, it does not wait for it, as with recognized clusters. Instead, it changes to the active state and continues processing new transactions.

Transaction Recovery in Unrecognized Clusters

As in the case of recognized clusters, whenever a failover occurs, RTR attempts to recover all the in-doubt transactions from the failed node's journal and replay them to the new active node. If the failover of the cluster disks happens fast enough so that when the new active node does the journal scan, the journal is visible, RTR will recover any in-doubt transactions from the failed node's journal. However, if the cluster disk failover has not yet happened, RTR does not wait. RTR changes the standby server to the active state and continues processing new transactions. Note that this does not mean that the transactions in the failed node's journal have been lost, as they will remain in the journal and can be recovered. See the *VSI Reliable Transaction Router System Manager's Manual* for information on how to recover these transactions from the journal.

2.3.3.5. Clustered Resource Managers and Databases

When RTR standby servers work in conjunction with clustered resource managers such as Oracle RDB or Oracle Parallel Server, additional considerations apply. These affect mainly the performance of the application and are relevant only to unrecognized cluster systems.

Unrecognized file systems host their file systems on one node and export the file system to the remaining nodes. In such a scenario, the RTR server could be working with a resource manager on one node that has its disks hosted on another node; this is not an optimal situation. Ideally, disks should be local to the RTR server that is active. Since RTR only waits for the journals to become available, this is not synchronized with the failover of the resource manager's disks. An even worse scenario occurs if both the RTR journal and the database disks are hosted on a remote node. In this case, the use of failover scripts is recommended to assist switching over in the most optimal way. Subsequent sections discuss this in more detail.

2.3.4. Failure Scenarios with RTR Standby Servers

In this section the various failure situations are analyzed in more detail. This can help system managers to configure the system in an optimal way.

2.3.4.1. Active Server Fails

Behavior when an active server fails depends on whether concurrent or standby servers are available.

Active Server Fails: Concurrent Servers Available

When the active server fails in the midst of a transaction, if there are other RTR servers for that partition on the same node (concurrent servers), RTR simply reissues the transaction to one of the other servers. Transaction processing is continued with the remaining servers. If the server fails due to a programming error, all the servers are likely to have the same error. Thus reissuing the same transaction to the remaining servers could cause all the concurrent servers to fail, one after another.

RTR has a built-in protection against this so that if the same transaction knocks out three servers in a row, that transaction is aborted. Three servers is the default value and can be configured to suit the application requirements. This behavior is the same whether or not RTR is run in a cluster.

Active Server Fails: Standby Servers Available

After the last available active server for a specific partition has failed, RTR tries to fail over to a standby server, if any exists. If no standby servers have been configured, the transaction is aborted.

Take the case of the configuration shown in Figure 2.3. Assume that P1 $_{A}$ (active server process for partition 1) has crashed on node N1. RTR will attempt to fail over to P1 $_{S}$. Before P1 $_{S}$ can be given new transactions to process, it has to resolve any in-doubt transactions that are in N1's journal sitting on D1. Therefore RTR on N2 scans the journal of N1 looking for any in-doubt transactions. If it finds any, these are reissued to the P1 $_{S}$. Once transaction recovery is completed, P1 $_{S}$ then changes state to active and becomes the new P1 $_{A}$. In this case, since the RTR ACP is still alive, and since it is the RTR ACP on N1 that owns the journal on D1, RTR on N2 will do a journal scan using the RTR ACP on N1. This behavior is the same for both recognized and unrecognized clusters.

2.3.4.2. RTR ACP Fails

If the RTR ACP fails, behavior depends on the availability of standby servers.

RTR ACP Fails: Standby Servers Available

If the RTR ACP fails, all the active servers on that node have their RTR channels closed and any transaction in progress is rejected. RTR tries to fail over to the standby server, if any exists. If no standby servers have been configured, the transaction is aborted. Take the case of the configuration shown in Figure 2.3. Assume that the ACP has crashed on node N1. RTR on the surviving node recognizes this and attempts to fail over to P1 $_S$. As before, a journal scan of the journal on N1 must be done before changing to active state. Since the ACP on N1 is gone, it cannot be used for the journal scan; the ACP on N2 must do the journal scan on its own. In this case, the behavior is different for recognized and unrecognized clusters.

2.3.4.3. Journal Scan

Because RTR recognizes that it is in a cluster configuration, it will wait for the cluster management to fail over the disks to N2. This failover process depends on whether it is a recognized or unrecognized cluster.

Journal Scan: Recognized Clusters

Recognized clusters allow N2 to access D.1 immediately and recover from the journal N1.J0 . This is because both N1 and N2 have equal access to the disk. Because the RTR ACP has gone down with the node, the DLM locks on N1.J0 are also released making it free for use by N2. In this cluster configuration, the RTR failover occurs immediately when the active node goes down.

Because this is a cluster configuration, both nodes N1 and N2 can access the journal N1.J0 on D1. On recognized clusters, RTR can directly access N1.J0 and on unrecognized clusters, RTR can access N1.J0 through the host node N1. Since the RTR ACP on N1 has failed, it has released locks on N1.J0 making it free for the ACP on N2 to access. There is no failover time as the failure of the ACP on N1 is detected by RTR immediately.

If a cluster transition causes D1 and D3 to be hosted on N2, this initiates the worst-case scenario, because the active server for P1 $_A$ is running on N1 but will be accessing the database partition P1 through host N2. Similarly, the RTR ACP on N1 will also access its journal N1.J0 through host N2. Note that this inefficiency is not present in recognized clusters. Thus, wherever host-based clustering is used, any re-hosting of disks should result in a matching change in the active/standby configuration of the RTR servers as well. RTR events or failover scripts can be used to achieve this.

Journal Scan: Unrecognized Clusters

Failover is more complicated in unrecognized clusters. When N1 goes down, the host for D1 also disappears. The cluster software must then select a new host, N2 in this case. It then proceeds to re-host D1 on N2. Once this has happened, D1 will become visible from N2. RTR failover time depends on cluster failover time.

Since the unrecognized clusters are all host-based, there will be a failover time required to re-host D1 on N2. RTR will not wait for this re-hosting. It performs a journal scan for N1.J0, does not find it and so does not do any transaction recovery. RTR simply moves into the active state and starts processing new transactions.

RTR treats unrecognized clusters as though they are not clusters. That is, RTR on the upcoming active server (N2) performs a journal scan. It searches among the disks accessible to it but does not specifically look for clustered disks. It also does not perform a journal scan on any NFS-mounted disks. If RTR on N2 can find the journal N1.J0, it performs a full recovery of any transactions sitting in this journal and

then continues to process transactions. If it cannot find the journal (N1.J0), it just continues to process new transactions; it does not wait for journals to become available.

2.3.4.4. Active Node Fails

Behavior when an active node fails depends on whether a standby node is available.

Active Node Fails: Standby Nodes Available

In this scenario, the node on which the active RTR servers are running fails. This causes the loss of a cluster node in addition to the RTR ACP and RTR servers. So, in addition to RTR failover, there is also a cluster failover. The RTR failover occurs as described above, with first a journal scan, transactions in the journal recovered, and then changing the standby server to active ($P1_S \rightarrow P1_A$). As this also causes a cluster failover, the effects vary according to cluster type.

2.3.5. Shadow Servers

A transactional shadow server handles the same transactions as the primary server, and maintains an identical copy of the database on the shadow. Both the primary and the shadow server receive every transaction for their key range or partition. If the primary server fails, the shadow server continues to operate and completes the transaction. This helps to protect transactions against site failure.

2.4. Tolerating Site Disaster

To prevent database loss at an entire site, you can use either transactional shadowing or standby servers. For example, for the highest level of fault tolerance, the configuration should contain two shadowed databases, each supported by a remote journal, with concurrent servers and partitions.

With such a configuration, you can use RTR shadowing to capture client transactions at two different physically separated sites. If one site becomes unavailable, the second site can then continue to record and process the transactions. This feature protects against site disaster.

2.4.1. The Role of Quorum

To understand and plan for smooth inter-node communication you must understand *quorum.*

Quorum is used by RTR to ensure facility consistency and deal with potential network partitioning. A facility achieves *quorum* if the right number of routers and backends in a facility (referred to in RTR as the *quorum threshold*), usually a majority, are active and connected.

In an OpenVMS cluster, for example, nodes communicate with each other to ensure that they have quorum, which is used to determine the state of the cluster; for cluster nodes to achieve quorum, a majority of possible voting member nodes must be available. In an OpenVMS cluster, quorum is node based. In the RTR environment, quorum is role based and facility specific. Nodes/roles in a facility that has quorum are *quorate*; a node that cannot participate in transactions becomes *inquorate.*

RTR computes a quorum threshold based on the distributed view of connected roles. The minimum value can be two. Thus a minimum of one router and one backend is required to achieve quorum. If the computed value of quorum is less than two, quorum cannot be achieved. In exceptional circumstances, the system manager can reset the quorum threshold below its computed value to continue operations, even when only a minimum number of nodes, less than a majority, is available. Note, however, that RTR uses other heuristics, not based on simple computation of available roles, to determine quorum viability.

For instance, if a missing (but configured) backend's journal is accessible, that journal is used to count for the missing backend.

A facility without quorum cannot complete transactions. Only a facility that has quorum, whose nodes/roles are quorate can complete transactions. A node/role that becomes inquorate cannot participate in transactions.

Your facility definition also has an impact on the quorum negotiation undertaken for each transaction. To ensure that your configuration can survive a variety of failure scenarios (for example, loss of one or several nodes), you may need to define a node that does not process transactions. The sole use of this node in your RTR facility is to make quorum negotiation possible, even when you are left with only two nodes in your configuration. This *quorum node* prevents a network partition from occurring, which could cause major update synchronization problems.

Note that executing the CREATE FACILITY command or its programmatic equivalent does not immediately establish all links in the facility, which can take some time and depends on your physical configuration. Therefore, do not use a design that relies on all links being established immediately.

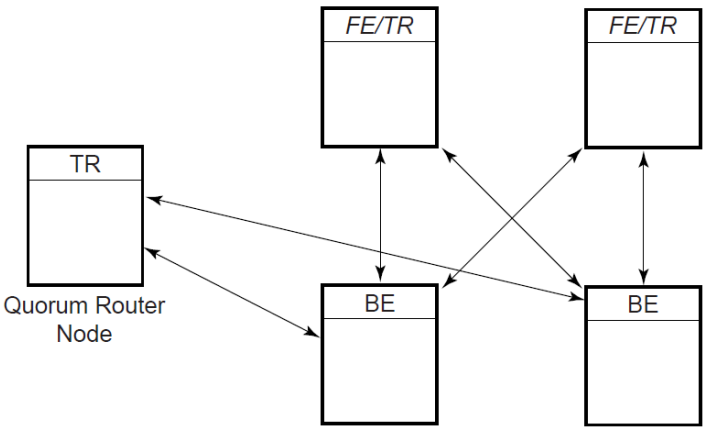
Quorum is used to:

- Detect inconsistencies in how a facility has been defined on the routers and backends of the facility. RTR checks that the facility definition on its nodes is consistent and will disallow operations if it discovers inconsistencies.
- Ensure that frontends route their transactions through a router that is properly connected to all running backends.
- Ensure that shadow servers do not operate independently if a network partition occurs. This could cause databases connected to these servers to diverge.

2.4.2. Surviving on Two Nodes

If your configuration is reduced to two server nodes out of a larger population, or if you are limited to two servers only, you may need to make some adjustments in how to manage quorum to ensure that transactions are processed. Use a quorum node as a tie breaker to ensure achieving quorum.

Figure 2.4. Configuration with Quorum Node



For example, with a five-node configuration (Figure 2.4) in which one node acts as a quorum node, processing still continues even if one entire site fails (only two nodes left). When an RTR configuration

is reduced to two nodes, the system manager can manually override the calculated quorum threshold. For details on this practice, see the *VSI Reliable Transaction Router System Manager's Manual*.

2.4.3. Partitioning

Frequently with RTR, you will partition your database.

Partitioning your database means dividing your database into smaller databases to distribute the smaller databases across several disk drives. The advantage of partitioning is improved performance because records on different disk drives can be updated independently - resource contention for the data on a single disk drive is reduced. With RTR, you can design your application to access data records based on specific key ranges. When you place the data for those key ranges on different disk drives, you have partitioned your database. How you establish the partitioning of your database depends on the database and operating systems you are using. To determine how to partition your database, see the documentation for your database system.

2.4.4. Transaction Serialization

In some applications that use RTR with shadowing, the implications of transaction serialization need to be considered.

Given a series of transactions, numbered 1 through 6, where odd-numbered transactions are processed on Frontend A (FE A) and even-numbered transactions are processed on Frontend B (FE B), RTR ensures that transactions are passed to the database engine on the shadow in the same order as presented to the primary. This is serialization. For example, the following table represents the processing order of transactions on the frontends.

Transaction Ordering on Frontends	
FE A	**FE B**
1	2
3	4
5	6

The order in which transactions are committed on the backends, however, may not be the same as their initial presentation. For example, the order in which transactions are committed on the primary server may be 2,1,4,3,5,6, as shown in the following table.

Transaction Ordering on Backend - Primary BE A
2
1
4
3
5
6

The secondary shadowed database needs to commit these transactions in the same order. RTR ensures that this happens, transparently to the application.

However, if the application cannot take advantage of partitioning, there can be situations where delays occur while the application waits, say, for transaction 2 to be committed on the secondary. The best way

to minimize this type of serialization delay is to use a partitioned database. However, because transaction serialization is not guaranteed across partitions, to achieve strict serialization where every transaction accepts in the same order on the primary and on the shadow, the application must use a single partition.

Not every application requires strict serialization, but some do. For example, if you are moving $20, say, from your savings to your checking account before withdrawing $20 from your checking account, you will want to be sure that the $20 is first moved from savings to checking before making your withdrawal. Otherwise you will not be able to complete the withdrawal, or perhaps, depending upon the policies of your bank, you may get a surcharge for withdrawing beyond your account balance. Or a banking application may require that checks drawn be debited first on a given day, before deposits. These represent dependent transactions, where you design the application to execute the transactions in a specific order.

If your application deals only with independent transactions, however, serialization will probably not be important. For example, an application that tracks payment of bills for a company would consider that the bill for each customer is independent of the bill for any other customer. The bill-tracking application could record bill payments received in any order. These would be independent transactions. An application that can ignore serialization will be simpler than one that must include logic to handle serialization and make corrections to transactions after a server failure.

In addition to dependent transactions that can make serialization more complex, if the application uses batch processing or concurrent servers, ensuring strict serialization may be difficult.

In a transactional shadow configuration using the same facility, the same partition, and the same key-range, RTR ensures that data in both databases are correctly serialized, provided that the application follows a few rules. For a description of these rules, see Chapter 3, later in this document. The shadow application runs on the backends, processes transactions based on the business and database logic required, and hands off transactions to the database engine that updates the database. The application can take advantage of multiple CPUs on the backends.

2.4.4.1. Transaction Serialization Detail

Transactions are serialized by committing them in chronological order within a partition. Do not share data records between partitions because they cannot be serialized correctly on the shadow site.

Dependent transactions operate on the same record and must be executed in the same order on the primary and the secondary servers. Independent transactions do not update the same data records and can be processed in any order.

RTR relies on database locking during its accept phase to determine if transactions executing on concurrent servers within a partition are dependent. A server that holds a lock on a data record during its vote call (AcceptTransaction for the C++ API or `rtr_accept_tx` for the C API) blocks other servers from updating the same record. Therefore only independent transactions can vote at the same time.

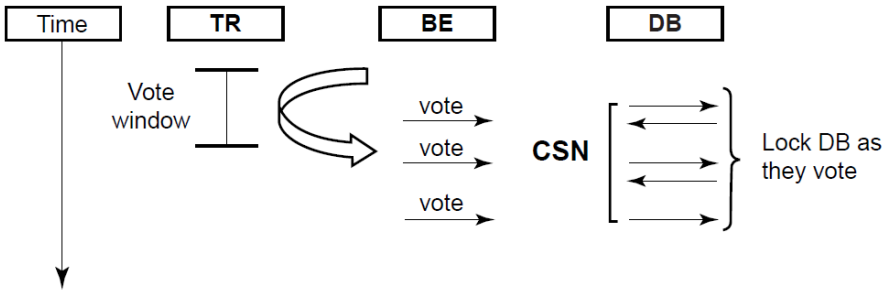
RTR tracks time in cycles using windows; a *vote window* is the time between the close of one commit cycle and the start of the next commit cycle.

2.4.4.1.1. RTR Commit Group

RTR commit grouping enables independent transactions to be scheduled together on the shadow secondary. A group of transactions that execute an AcceptTransaction (or `rtr_accept_tx` call for the C API) call within a vote window form an *RTR commit group*, identified by a unique commit sequence number (CSN). For example, given a router (TR), backend (BE), and database (DB), each transaction sent by the backend to the database server is represented by a vote. When the database

receives each vote, it locks the database and responds as *voted*. The backend responds to the router in a time interval called the vote window, during which all votes that have locked the database receive the same commit sequence number. This is illustrated in Figure 2.5.

Figure 2.5. Commit Sequence Number



To improve performance on the secondary server, RTR lets this commit group of transactions execute in any order on the secondary.

RTR reuses the current CSN if it determines that the current transaction is independent of previous transactions. This way, transactions can be sent to the shadow in a bunch.

In a little more detail, RTR assumes that transactions within the vote window are independent. For example, given a router and a backend processing transactions, as shown in Figure 2.6 for the C++ API, transactions processed between execution of AcceptTransaction and the final Receive that occurs after the SQL commit or rollback will have the same commit sequence number.

Figure 2.6. CSN Vote Window for the C++ API

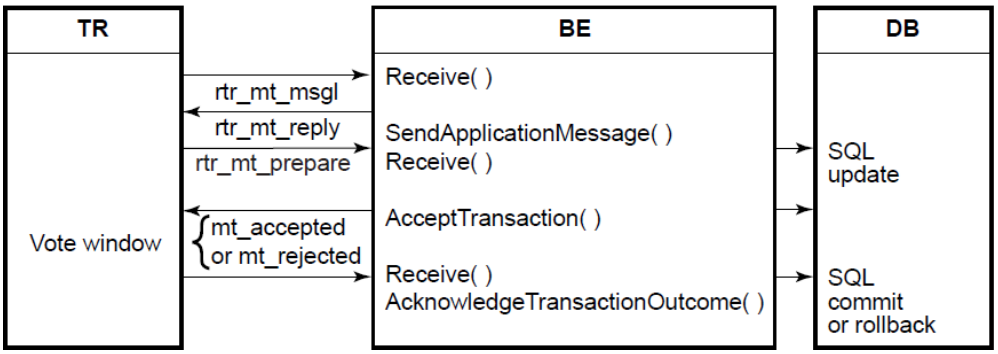
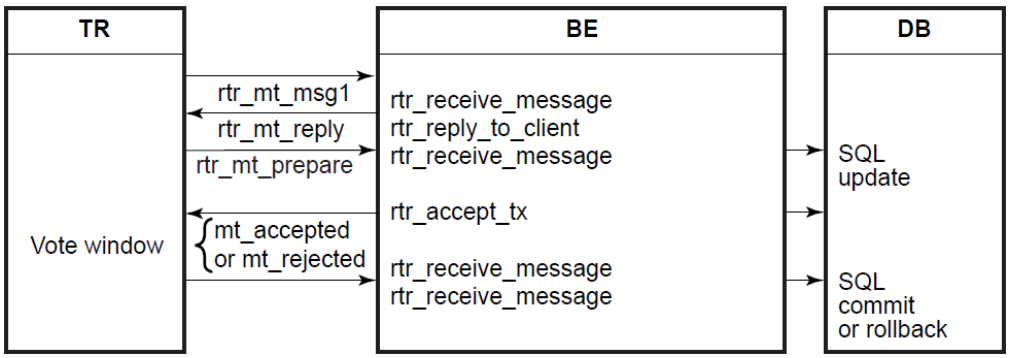


Figure 2.7 illustrates the vote window for the C API. Transactions processed between execution of the `rtr_accept_tx` call and the final `rtr_receive_message` call that occurs after the SQL commit or rollback will have the same commit sequence number.

Not all database managers require locking before the SQL commit operation. For example, some insert calls create a record only during the commit operation. For such calls, the application must ensure that the table or some token is locked so that other transactions are not incorrectly placed by RTR in the same commit group.

All database systems do locking at some level, at the database, file, page, record, field, or token level, depending on the database software. The application designer must determine the capabilities of whatever database software the application will interface with, and consider these in developing the application. For full use of RTR, the database your application works with must, at minimum, be capable of being locked at the record level.

Figure 2.7. CSN Vote Window for the C API



Not all database managers require locking before the SQL commit operation. For example, some insert calls create a record only during the commit operation. For such calls, the application must ensure that the table or some token is locked so that other transactions are not incorrectly placed by RTR in the same commit group.

All database systems do locking at some level, at the database, file, page, record, field, or token level, depending on the database software. The application designer must determine the capabilities of whatever database software the application will interface with, and consider these in developing the application. For full use of RTR, the database your application works with must, at minimum, be capable of being locked at the record level.

2.4.4.1.2. Independent Transactions

When a transaction is specified as being independent (using the SetIndependentTransaction parameter set to true in the AcceptTransaction method (C++ API) or with the INDEPENDENT flag (C API)), the current commit sequence number is assigned to the independent transaction. Thus the transaction can be scheduled simultaneously with other transactions having the same CSN, but only after all transactions with lower CSNs have been processed.

RTR tracks time in cycles using windows; a vote window is the time between the close of one commit cycle and the start of the next commit cycle. For example, independent transactions include transactions such as zero-hour ledger posting (posting of interest on all accounts at midnight), and selling bets (assuming that the order in which bets are received has no bearing on their value).

RTR examines the vote sequence of transactions executing on the primary server, and determines dependencies between these transactions. The assumption is: if two or more transactions vote within a vote window, these transactions could be processed in any order and still produce the same result in the database. Such a group of transactions is considered independent of other transaction groups. Such groups of transactions that are mutually independent may still be dependent on an earlier group of independent transactions.

2.4.4.1.3. CSN Ordering

RTR tracks these groups through CSN ordering. A transaction belonging to a group with a higher CSN is considered to be dependent on all transactions in a group with a lower CSN. Because RTR infers CSNs based on run-time behavior of servers, there is scope for improvement if the application can provide hints regarding actual dependence. If the application knows that the order in which a transaction is committed within a range of other transactions is not significant, then using independent transactions is recommended. If an application does not use independent transactions, RTR determines the CSN grouping based on its observation of the timing of the vote.

2.4.4.1.4. CSN Boundary

To force RTR to provide a *CSN boundary*, the application must:

- Use dependent transactions. This is the default behavior.
- Ensure that a transaction is voted on *after* any transactions on which it is dependent.

The CSN boundary is between the end of one CSN and the start of the next, as represented by the last transaction in one commit group and the first transaction in the subsequent commit group.

In practice, for the transaction to be voted on after its dependent transactions, it is enough for the dependent transaction to access a common database resource, so that the database manager can serialize the transaction correctly.

Dependent transactions do not automatically have a higher CSN. To ensure a higher CSN, the transaction also needs to access a record that is locked by a previous transaction. This will ensure that the dependent transaction does not vote in the same vote cycle as the transaction on which it is dependent. Similarly, transactions that are independent do not automatically all have the same CSN. In particular, for the C API, if they are separated by an independent transaction, that transaction creates a CSN boundary.

RTR commit grouping enables independent transactions to be scheduled together on the shadow secondary. Flags on `rtr_accept_tx` and `rtr_reply_to_client` enable an application to signal RTR that it is safe to schedule this transaction for execution on the secondary within the current commit sequence group. In a shadow environment, an application can obtain certain performance improvements by using independent transactions where suitable. With independent transactions, transactions in a commit group can be executed on the shadow server in a different order than on the primary. This reduces waiting times on the shadow server.

For example, transactions in a commit group can execute in the order A2, A1, A3 or the primary partition and in the order A1, A2, A3 on the shadow site. Of course independent transactions can only be used where transaction execution need not be strictly the same on both primary and shadow servers. Examples of code fragments for independent transactions are shown in the code samples appendices of this manual.

2.4.5. Batch Processing Considerations

Some of your applications may rely on batch processing for periodic activity. Application facilities can be created with batch processing. (The process for creating batch jobs is operating-system specific, and is thus outside the scope of this document.) Be careful in your design when using batch transactions. For example, accepting data in batch from legacy systems can have an impact on application results or performance. If such batch transactions update the same database as online transactions, major database inconsistencies or long transaction processing delays can occur.

2.4.6. Application Considerations with Shadowing

Although applications need not be directly concerned about shadowing matters, certain points must be considered when implementing performance boosting optimizations:

- Anything specific to the executing node should not be stored in the database, since this would lead to diverging copies.
- Any physical reference to the transaction which is unique to the executing server, e.g. Channel ID, system time, DB-key, etc., should not be passed back to the client for future references within its

subsequent messages. This could lead to inconsistent handling when a different server is involved in shadow operations.

This consideration is also valid for recovery of non-shadowed servers.

2.4.7. Journal Accessibility

The RTR journal on each node must be accessible to be used to replay transactions. When setting up your system, consider both journal sizing and how to deal with replay anomalies.

2.4.8. Journal Sizing

To size a journal, use the guidelines described in the section Creating a Recovery Journal in the *VSI Reliable Transaction Router System Manager's Manual*.

Use of large transactions generally causes poor performance, not only for initial processing and recording in the database, but also during recovery. Large transactions fill up the RTR journals more quickly than small ones.

For replay anomalies, use the RTR_STS_REPLYDIFF status message to determine if a transaction has been recorded differently during replay. For details on this and other status messages, see the *VSI Reliable Transaction Router C++ Foundation Classes* manual or the *VSI Reliable Transaction Router C Application Programmer's Reference Manual*.

You should also consider how the application is to handle secondary or shadow server errors and aborts, and write your application accordingly.

2.5. Design for Performance

In designing for performance, take the following into account:

- Consider the amount of data being transferred.
- Keep the size of transaction messages short.
- Tie up the database for as short a time as possible.
- When using transactional shadowing to two sites, have high-speed links between sites.
- Evaluate your hardware, in particular:
  - Memory (see the *VSI Reliable Transaction Router System Manager's Manual* for information on virtual memory requirements for RTR links, channels, and messages)
  - Disk striping
  - Volume shadowing
  - Disk performance/fragmentation
  - Disk controllers
- Consider tuning your operating system on nodes where RTR is running.
- With the C++ API:

  - Use the RTRServerTransactionController::SetIndependentTransaction method.
  - Use multi-transaction controller applications, which are more efficient than multiple, single transaction controller applications.
  - Use the RTRClientTransactionController::SetReadOnly method to reduce RTR journaling.
- With the C API:
  - Use the independent transaction flag.
  - Use multi-channel applications, which are more efficient than multiple, single channel applications.
  - Use the READ_ONLY flag to reduce RTR journaling.
  - Use single `accept_txn` flags for client/server calls to minimize transaction activity; for example, `send/accept` or `reply/forget`.

2.5.1. RTR Performance Guidelines

An important part of your application design will concern performance considerations: how will your application perform when it is running with RTR on your systems and network? Providing a methodology for evaluating the performance of your network and systems is beyond the scope of this document. However, to assist your understanding of the impact of running RTR on your systems and network, this section provides information on three major performance factors:

- Number of client channels for the C API (equivalent to transaction controllers (client side) and partitions (server side) for the C++ API).
- Size of messages (in bytes)
- Number of CPUs in a node

This information is roughly scalable to other CPUs and networks, and illustrates the concepts of RTR throughput on an untuned, as-shipped product and operating system. No guarantees are implied; however, these data may be used as a guide for baseline estimates.

These performance tests were run during April 2004 in the Nashua, N.H. RTR lab.

System Environment:

The tests were staged on an AlphaServer GS160 (four system building blocks (SBBs)), each with four EV6/7 730MHz CPUs. Only two of the SBBs were used in the tests and OpenVMS 7.3-1 was installed on both. Both were configured identically and were dedicated to these tests (there were no other users or processes).

The network interface card in each system was a 10/100Mbps twisted-pair Ethernet card (DEC602) enabled to run at 100Mbps. There was no other network traffic.

Test Scenario Overview:

The test goal was to provide information on RTR throughput as measured in transactions per second (TPS) and CPU load (%CPU) while varying four basic parameters: number of nodes (one and two), number of CPUs (one and quad), message size (100 to 60,000 bytes), and number of open channels (1 to 100).

This was accomplished with six test runs, varying appropriate parameters. The actual applications used to create and transfer transactions consisted of a client application, TPSREQ, and a server application, TPSSRV. These mini-applications are provided on the RTR software kit and use the RTR V4.2 C API.

RTR facilities were configured with the FE role running the client application, TPSREQ, on one node, while the router and backend roles were configured on the other. The backend was running the server application, TPSSRV.

The six test scenarios were:

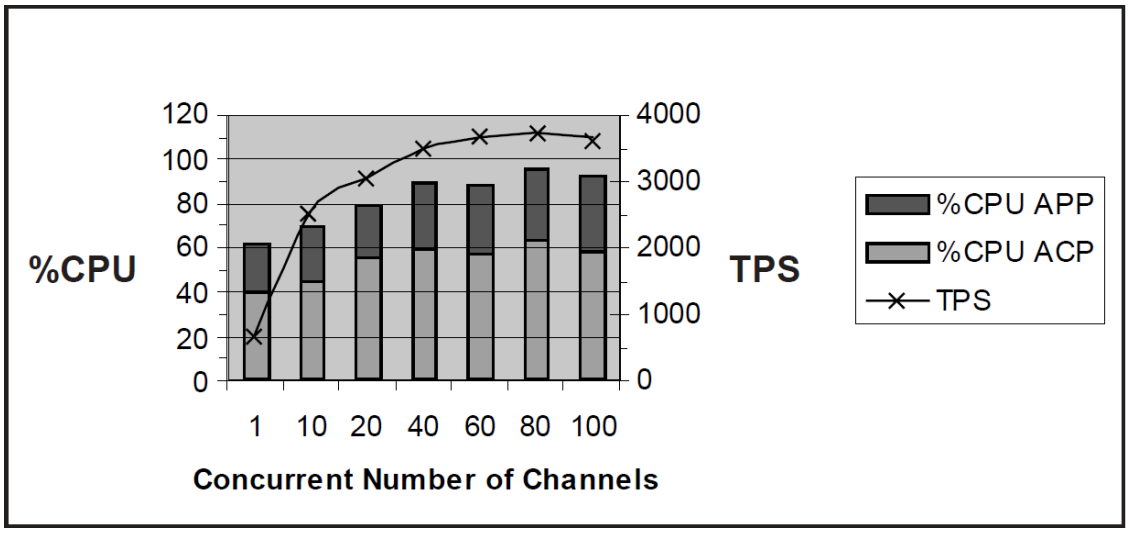
- Single node, single CPU with a fixed 100-byte message size using an open-channel count varying from 1 to 100 channels
- Single node, quad CPU with a fixed 100-byte message size using an open-channel count varying from 1 to 100 channels
- Single node, single CPU with a fixed number of open channels (80) and a message size varying from 100 to 60,000 bytes
- Single node, quad CPU with a fixed number of open channels (80) and a message size varying from 100 to 60,000 bytes
- Two node, single CPU with a fixed 100-byte message size using an open channel count varying from 1 to 100 channels
- Two node, quad CPU with a fixed 100-byte message size using an open channel count varying from 1 to 100 channels

Note that, in the following test runs, the charts indicate %CPU utilization and throughput in transactions per second (TPS). The %CPU utilization is the sum of RTRACP and application (TPSREQ, TPSSRV) CPU usage on all nodes in the facility.

To establish a baseline for network throughput, FTP (the File Transfer Protocol tool) was used to transfer a 9Mbyte file from one node to the other. This was done several times in both directions showing an average network transfer rate of about 9Mbytes/sec.

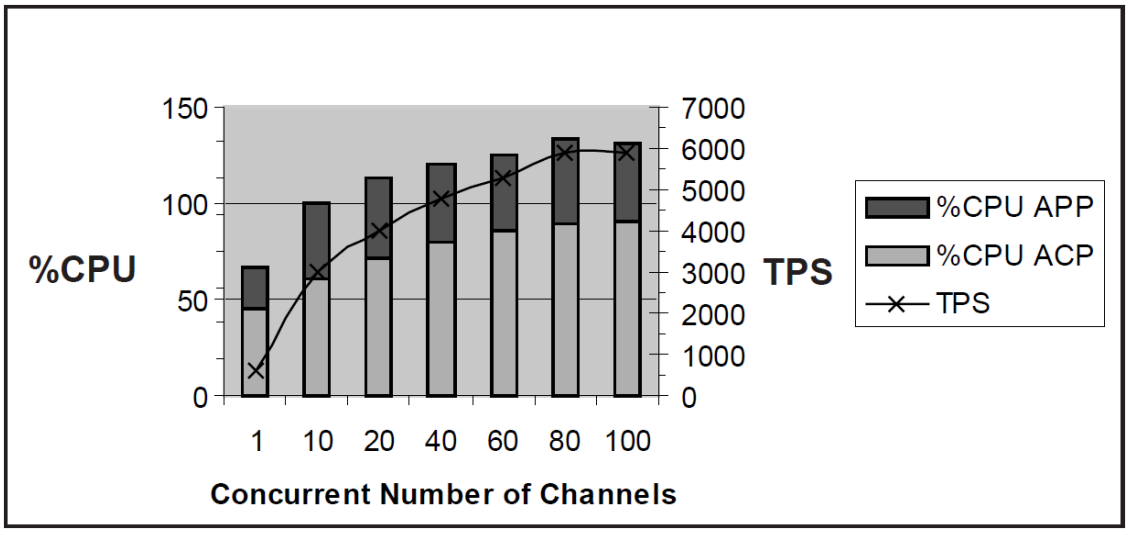
The results for a single node with an increasing number of channels are shown in Figure 2.8

Figure 2.8. Single-Node, Single CPU, TPS and CPU Load by Number of Channels



The results for a single node with four CPUs and an increasing number of channels are shown in Figure 2.9

Figure 2.9. Single-Node, Quad CPU, TPS and CPU Load by Number of Channels



The transactions used in these tests were regular read/write transactions; there was no use of optimizations such as READONLY or ACCEPT_FORGET.

The results of these tests using 100-byte messages suggest the following:

- CPU saturation limits the maximum TPS at about 3500 for the single-node, single-CPU case.
- For the single-node, quad-CPU test, the maximum rate rises, approaching 6000 TPS. The test also indicated that, for a single CPU, more than 80 channels would saturate the CPU.
- Due to more effective use of RTR optimizations to ‘batch’ I/Os for disk and interprocess communication (IPC), CPU resource cost per transaction (as more transactions are processed concurrently) goes down rapidly as offered load (number of RTR channels) increases. (As on-node RTR transfers use IPC, there is no network traffic.)
- In a multi-CPU environment , the RTRACP will likely limit the maximum TPS per system to about 6000, regardless of the number of CPUs added.

The results for a single node with a fixed number of channels (80) and increasing message size are shown in Figure 2.10 for a single CPU, and Figure 2.11 for a quad CPU.

Figure 2.10. Single-Node, Single CPU, TPS and CPU Load by Message Size

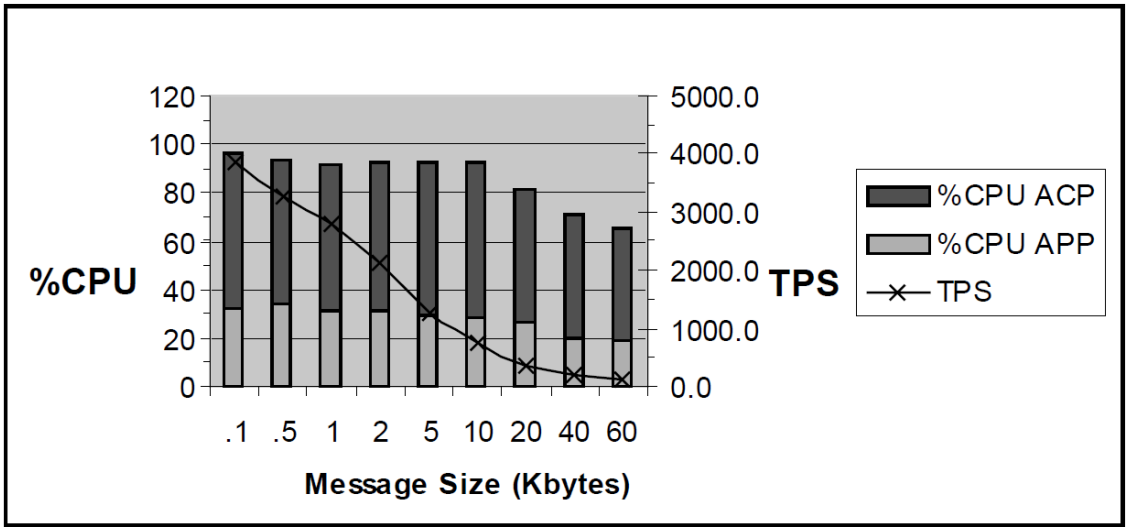
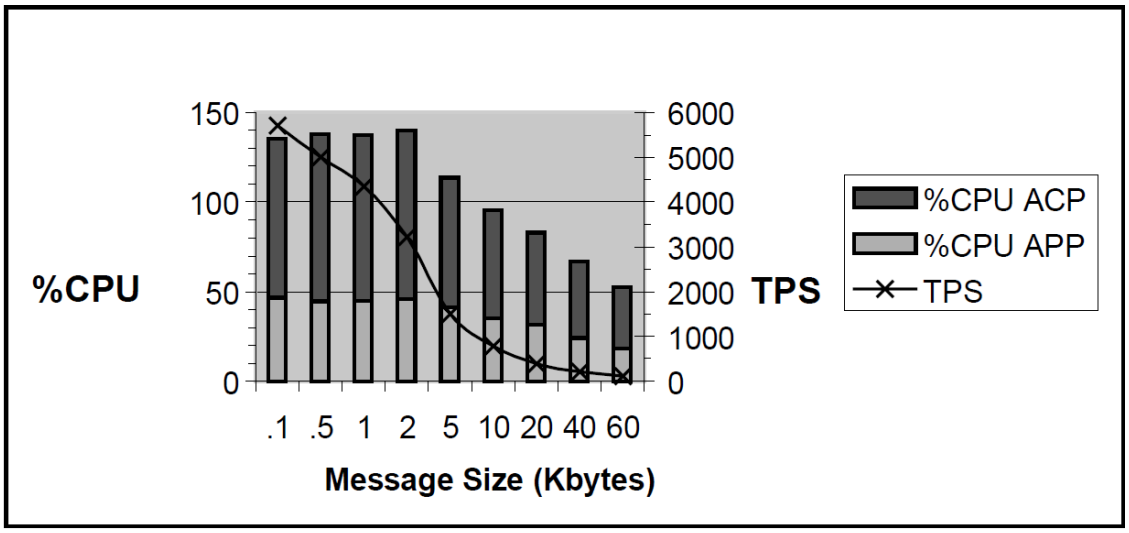


Figure 2.11. Single-Node, Quad CPU, TPS and CPU Load by Message Size



These tests using 80 client and server channels suggest that:

- CPU saturation appears to limit TPS for small message sizes.
- Disk I/O rates appear to limit TPS for large messages.

The results for the two-node single and quad CPU configurations are shown in Figure 2.12 and Figure 2.13.

Figure 2.12. Two-Node, Single CPU TPS and CPU Load by Number of Channels

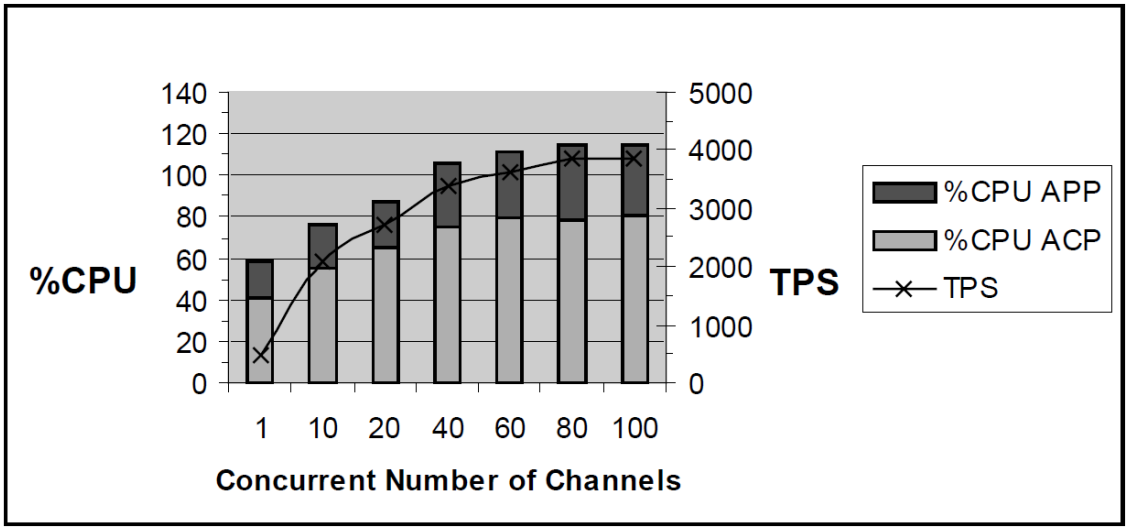
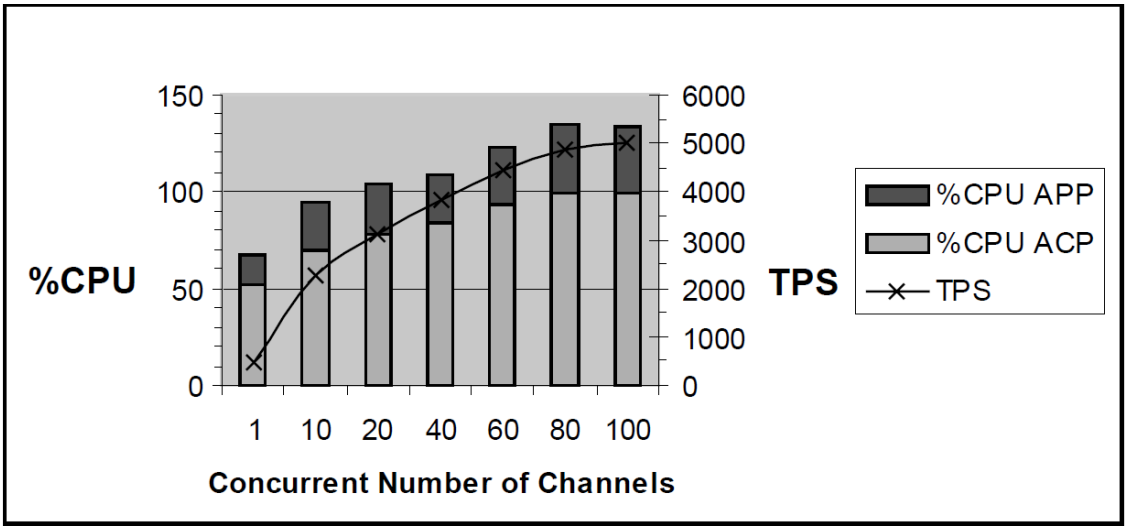


Figure 2.13. Two-Node, Quad CPU TPS and CPU Load by Number of Channels



This two-node test using 100-byte messages shows CPU usage with totals for frontend and backend combined (thus CPU utilization can be more than 100 percent). This test shows TPS increasing as the number of channels increases.

The two-node/quad-CPU test indicates that the constraint appears to be network bandwidth (values not shown in the figures) because the TPS rate flattens out at a network traffic level consistent with that measured on the same LAN by another independent test (using FTP to transfer data across the same network links). For example, using FTP on this Ethernet, multi-CPU configuration, shows transfer rates up to 8.9Mbytes per second, while using the RTR mini-applications shows transfer rates of about 6.4Mbytes per second. Thus improving network bandwidth could likely improve TPS rates when using RTR.

2.5.1.1. Summary

Determining the factors that limit performance in a particular configuration can be complex. While the previous performance data can be used as a rough guide to what can be achieved in particular configurations, they should be applied with caution. Performance will certainly vary depending on the

capabilities of the hardware, operating system, and RTR version in use, as well as the work performed by the user application (the above tests employ a dummy application which does no real end-user work.)

In general, performance in a particular case is constrained by contention for a required resource. Typical resource constraints are:

- CPU saturation
- Disk storage I/O bandwidth and latency
- Network bandwidth and delays
- Server application I/O delays
- Database tuning
- Optimum database connection bandwidth
- Size of messages
- Number of transaction controllers or channels

Additionally, achieving a high TPS rate can be limited by:

- Lack of applied client load

For suggestions on examining your RTR environment for performance, see Appendix F in this document.

2.5.2. Concurrent Servers

Use concurrent servers in database applications to optimize performance and continue processing when a concurrent server fails.

When programming for concurrency, you must ensure that the multiple threads are properly synchronized so that the program is thread-safe and provides a useful degree of concurrency without ever deadlocking. Always check to ensure that interfaces are thread-safe. If it is not explicitly stated that a method is thread-safe, you should assume that the routine or method is not thread-safe. For example, to send RTR messages in a different thread, make sure that the methods for sending to server, replying to client and broadcasting events are safe. You can use these methods provided that the:

- Sending thread owns the object being sent.
- Transaction controller has been completely constructed before any other threads use it.
- Transaction controller is not destructed before other threads have stopped using it.

2.5.3. Partitions and Performance

Partitioning data enables the application to balance traffic to different parts of the database on different disk drives. This achieves parallelism and provides better throughput than using a single partition. Using partitions may also enable your application to survive single-drive failure in a multi-drive environment more gracefully. Transactions for the failed drive are logged by RTR while other drives continue to record data.

2.5.4. Facilities and Performance

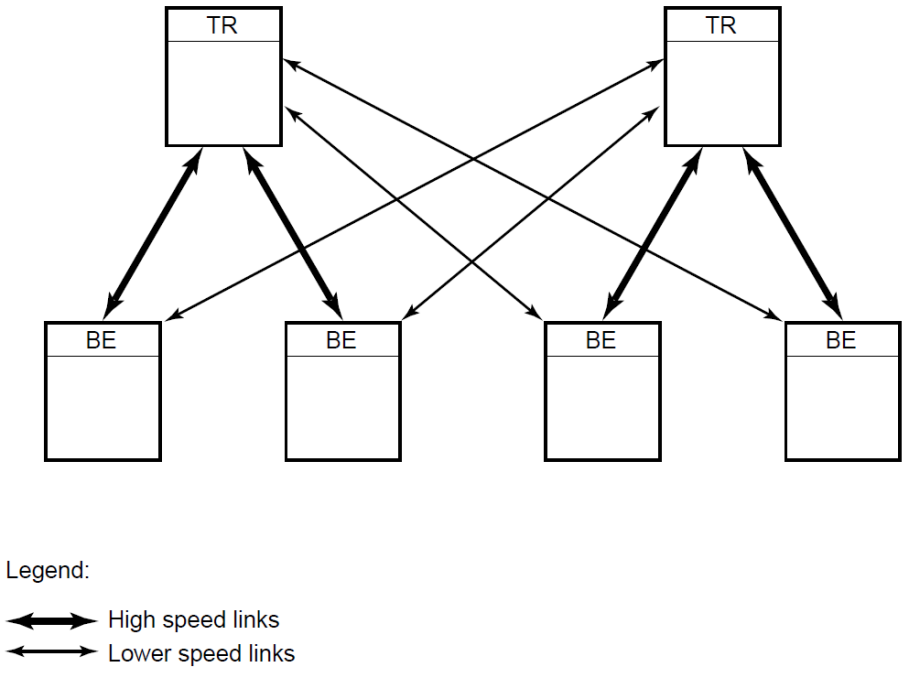
To achieve performance goals, you should establish facilities spread across the nodes in your physical configuration using the most powerful nodes for your backends that will have the most traffic.

In some applications with several different types of transactions, you may need to ensure that certain transactions go only to certain nodes. For example, a common type of transaction is for a client application to receive a stock sale transaction, which then proceeds through the router to the current server application. The server may then respond with a broadcast transaction to only certain client applications. This exchange of messages between frontends and backends and back again can be dictated by your facility definition of frontends, routers, and backends.

2.5.5. Router Placement

Placement of routers can have a significant effect on your system performance. With connectivity over a wide-area network possible, do not place your routers far from your backends, if possible, and make the links between your routers and backends as high speed as possible. However, recognize that site failover may send transactions across slower-speed links. For example, Figure 2.14 shows high-speed links to local backends, but lower-speed links that will come into use for failover.

Figure 2.14. Two-Site Configuration



Additionally, placing routers on separate nodes from backends provides better failover capabilities than placing them on the same node as the backend.

In some configurations, you may decide to use a dual-rail or multihome setup for a firewall or to improve network-related performance. (See the *VSI Reliable Transaction Router System Manager's Manual* section on Network Transports for information on this setup.)

2.5.6. Broadcast Messaging

When a server or client application sends out a broadcast message, the message passes through the router and is sent to the client or server application as appropriate. A client application sending a broadcast

message to a small number of server applications will probably have little impact on performance, but a server application sending a broadcast message to many, potentially hundreds of clients, can have a significant impact. Therefore, consider the impact of frequent use of large messages broadcast to many destinations. If your application requires use of frequent broadcasts, place them in messages as small as possible. Broadcasts could be used, for example, to inform all clients of a change in the database that affects all clients.

Figure 2.15 illustrates message fan-out from client to server, and from server to client.

You can also improve performance by creating separate facilities for sending broadcasts.

2.5.6.1. Making Broadcasts Reliable

To help ensure that broadcasts are received at every intended destination, the application might number them with an incrementing sequence number and have the receiving application check that all numbers are received. When a message is missing, have a retransmit server re-send the message.

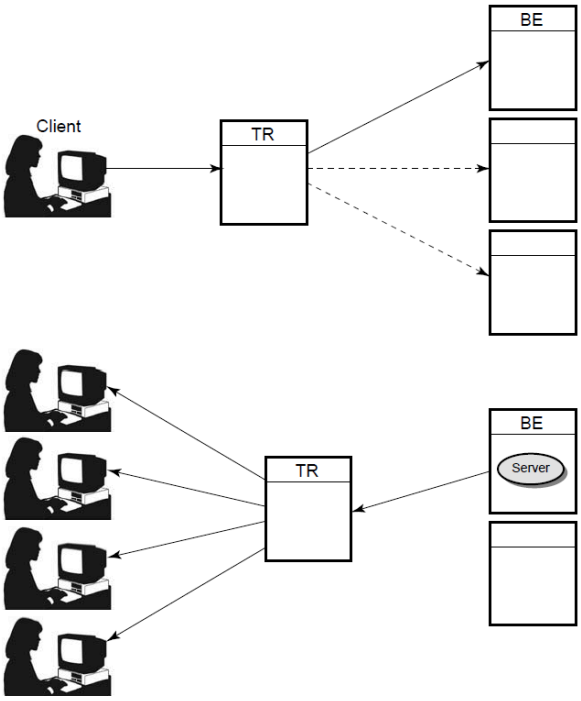
2.5.7. Large Configurations

Very large configurations with unstable or slow network links can reduce performance significantly. In addition to ensuring that your network links are the fastest you can afford and put in place, examine the volume of inter-node traffic created by other uses and applications. RTR need not be isolated from other network and application traffic, but can be slowed down by them.

2.5.8. Using Read-Only Transactions

Read-only transactions can significantly improve throughput because they do not need to be journaled. A read-only database can sometimes be updated only periodically, for example, once a week rather than continuously, which again can reduce application and network traffic.

Figure 2.15. Message Fan-Out



2.5.9. Making Transactions Independent

When using transactional shadowing, it can enhance performance to process certain transactions as independent. When transactions are declared as independent, processing on the shadow server proceeds without enforced serialization. Your application analysis must establish what transactions can be considered independent, and you must then write your application accordingly. For example, bets placed at a racetrack for a specific race are typically independent of each other. In another example, transactions within one customer's bank account are typically independent of transactions within another customer's account. For examples of code snippets for each RTR API, see the appendices of samples in this manual.

2.6. Configuration for Operability

To help make your RTR system as manageable and operable as possible, consider several tradeoffs in establishing your RTR configuration. Review these tradeoffs before creating your RTR facilities and deploying an application. Make these considerations part of your design and validation process.

- Define your facilities with an eye to the number and placement of frontends, routers, and backends.
- To avoid problems with quorum resolution, design your configuration with an odd number of routers to ensure that quorum can be achieved.
- Separate your routers from your backends to improve failover, so that failure of one node does not take out both the router and the backend.
- If your application requires frontend failover when a router fails, frontends must be located on separate nodes from the routers, but frontends and routers must of course be in the same facility. For frontend failover, a frontend must be in a facility with multiple routers. You use frontend failover with nested transactions.
- To identify a node used only for quorum resolution, define the node as a router or as a router and frontend. Define all backends in the facility, but no other frontends.
- With a widely dispersed set of nodes, for example, nodes distributed across an entire country, use local routers to deal with local front ends. This can be more efficient than having many dispersed frontends connecting to a small number of distant routers.
- In many configurations, it may be more effective to place routers near backends.

2.6.1. Firewalls and RTR

For security purposes, your application transactions may need to pass through firewalls in the path from the client to the server application. RTR provides this capability within the CREATE FACILITY syntax. See the *VSI Reliable Transaction Router System Manager's Manual*, Network Transports, for specifics on how to specify a node to be used as a firewall, and how to set up your application tunnel through the firewall.

2.6.2. Avoiding DNS Server Failures

Nodes in your configuration are often specified with names and IP or DECnet addresses fielded by a name server. When the name server goes down or becomes unavailable, the name service is not available and certain requests may fail. To minimize such outages, declare the referenced node name entries in a local host names file that is available even when the name server is not. Using a host names file can

also improve performance for name lookups. For details on this, see the *VSI Reliable Transaction Router System Manager's Manual* section on Network Transports.

2.6.3. Batch Procedures

Operations staff often create batch or command procedures to take snapshots of system status to assist in monitoring applications. The character cell displays (ASCII output) of RTR can provide input to such procedures. Be aware that system responses from RTR can change with each release, which can cause such command procedures to fail. If possible, plan for such changes when bringing up new versions of the product.

Chapter 3. Implementing an Application

In addition to understanding the RTR run-time and system management environments, you must also understand the RTR applications environment and the implications of that environment on your implementation. This section provides information on requirements that transaction processing applications must take into account and deal with effectively. It also cites rules to follow that can help prevent your application from violating the rules for ensuring that your transactions are ACID compliant. The requirements and rules complement each other and sometimes repeat a similar concept. Your application must take both into account.

3.1. RTR Requirements on Applications

Applications written to operate in the RTR environment should adhere to the following rules:

- Be transaction aware
- Avoid server-specific data
- Optionally, have independent transactions
- Optionally, use two identical databases for transactional shadow servers
- Make transactions self-contained
- Lock shared resources

3.1.1. Be Transaction Aware

RTR expects server applications to be transaction aware; an application must be able to roll back an appropriate amount of work when asked. Furthermore, to preserve transaction integrity, rollback must be all or nothing. Each transaction incurs some overhead, and the application must be prepared to deal with failures and concomitant rollback gracefully. When designing your client and server applications, note the outcome of transactions. Transactional applications often store data in variables that pertain to the operation taking place outside the control of RTR. Depending on the outcome of the RTR transaction, the values of these variables may need to be adjusted. RTR guarantees delivery of messages (usually to a database), but RTR does not know about any data not passed through RTR.

The rule is:

Code your application to preserve transaction integrity through failures.

3.1.2. Avoid Server-Specific Data

The client and server applications must not exchange any data that makes sense on only one node in the configuration. Such data can include, for example, a memory reference pointer, whose purpose is to allow the client to reference this context in a later transaction, indexes into files, node names, or database record numbers. These values only make sense on the machine on which they were generated. If your application sends data to another machine, that machine will not be able to interpret the data correctly. Furthermore, data cannot be shared across servers, transaction controllers, or channels.

The rule is: How you track state must be meaningful on all nodes where your application runs.

3.1.3. Be Independent of Time of Processing

Transactions are assumed to contain all the context information required to be successfully executed. An RTR transaction is assumed to be independent of time of processing. For example, in a shadow environment, if the secondary server cannot credit an account because it is past midnight, but the transaction has already been successfully committed on the primary server, this would cause an inconsistency between the primary and secondary databases. Or, in another example, Transaction B cannot rely on the fact that Transaction A performed some operation before it.

Make no assumptions about the amount of time that will occur between transactions, and avoid using a transaction to establish a session with a server application that can time out. Such a timeout might occur in a client application that logs onto a server application that sets a timer to determine when to log the client off. If a crash occurs after a successful logon, subsequent transactions may fail because the logon session is no longer valid.

The rule is:

If you have operations that must not be shadowed, identify them and exclude them from your application. Furthermore, do not keep a state that can become stale over time.

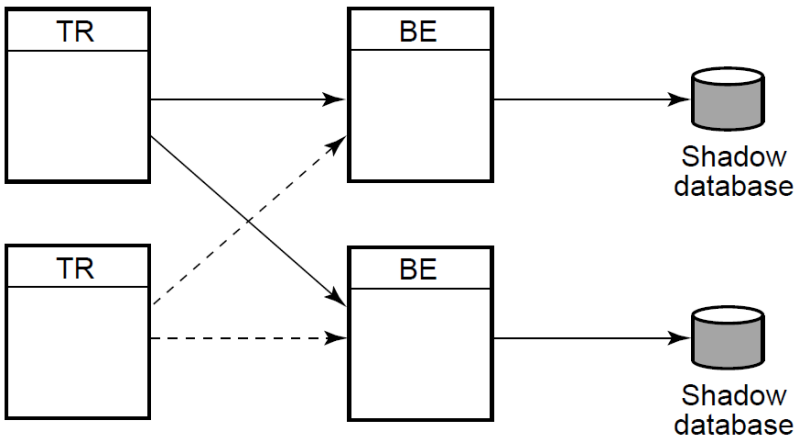
In your application, you can define transactions as independent with the C++ API, using the SetIndependentTransaction method in your transaction controller AcceptTransaction or SendApplicationMessage calls. Using the C API, you use the independent transaction flag in your `rtr_accept_tx` or `rtr_reply_to_client` calls.

For more information on the independent transaction methods in the RTRServerTransactionController class, refer to the *VSI Reliable Transaction Router C++ Foundation Classes* manual. For more information on the independent transaction flag and the different uses of these calls, refer to the *VSI Reliable Transaction Router C Application Programmer's Reference Manual*.

3.1.4. Use Two Identical Databases for Shadow Servers

Shadow server use is aimed at keeping two identical copies of the database synchronized. For example, Figure 3.1 illustrates a configuration with a router serving two backends to two shadow databases. The second router is for router failover.

Figure 3.1. Transactional Shadow Servers



If an update of a copy triggers the update of a third common database, the application must determine whether it is running as a primary or a secondary, and only perform an update if it is the primary. Otherwise, there can be complex failure scenarios where duplication can occur.

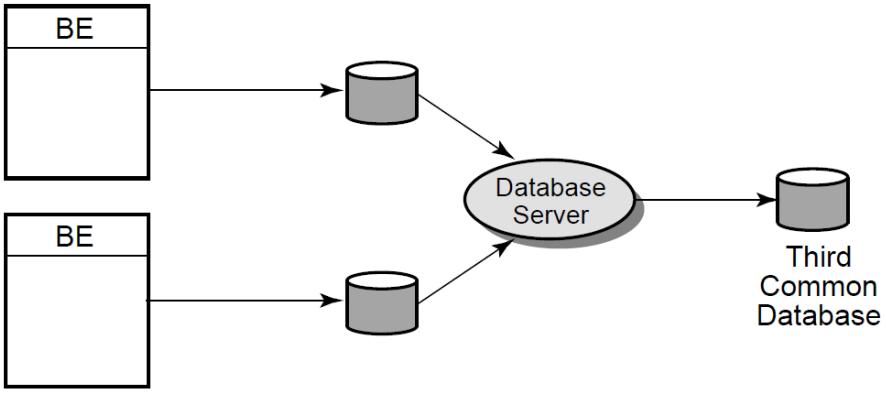
For example, RTR has no way to determine if a transaction being shadowed is a one-time-only transaction, such as a bookstore debiting your credit card for the purchase of a book. If this transaction is processed on the primary node and the processed data fed to a third common database, and the transaction is later processed on the secondary node, your account would incorrectly be double charged. The application must handle this situation correctly.

The rule is:

Design your application to deal correctly with transactions, such as debiting a credit card or bank account, that must never be performed more than once.

Figure 3.2 shows a configuration with two shadow servers and a third, independent server for a third, common database. This is not a configuration recommended for use with RTR without application software that deals with the kind of failure situation described above. Another method is to decouple the shadow message from the other branch.

Figure 3.2. Shadow Servers and Third Common Database (not recommended)



When updating a single resource through multiple paths, the recommended method is to use the RTR standby functionality.

3.1.5. Make Transactions Self-Contained

All information required to process a transaction from the perspective of the server application should be contained within the transaction message. For example, if the application required a user-ID established earlier to successfully execute the transaction, the user-ID should be included in the transaction message.

The rule is:

Construct complete transaction messages within your application.

3.1.6. Lock Shared Resources

While a server application is processing a transaction, and particularly before it "accepts" the transaction, it must ensure that all shared resources accessed by that transaction are locked. Failure to do so can cause unpredictable results in shadowing or recovery.

The rule is:

Lock shared resources while processing each transaction.

3.2. Ensuring ACID Compliance

To ensure that your application deals with transactions correctly, its transactions must be:

- Atomic
- Consistent
- Isolated
- Durable

3.2.1. Ensuring Atomicity

For the atomic attribute, the result of a transaction is all or nothing, that is, either totally committed or totally rolled back. To ensure atomicity, do not use a data manager that cannot roll back its updates on request. All standard data managers or database management systems have the atomicity attribute. However, in some cases, when interfacing to an external legacy system, a flat-file system, or an in-memory database, a transaction may not be atomic.

For example, a client application may believe that a transaction has been rejected, but the database server does not. With a database manager that can make this mistake, the application itself must be able to generate a compensating transaction to roll back the update.

Data managers that do not use XA/DTC, DECdtm or Microsoft DTC to integrate with RTR using XA or DECdtm must be programmed to handle rtr_mt_msg1_uncertain messages.

For example, to illustrate the atomicity rules, Figure 3.3 shows the uncertain interval in a transaction sequence that the application program must be aware of and take into account, by performing appropriate rollback.

Figure 3.3. Uncertain Interval for Transactions

C++ API	C API
. . .	. . .
Receive (rtr_mt_msg1)	rtr_receive_message (rtr_mt_msg1)
SQL update . . .	**SQL update . . .**
rtr_mt_msg1	rtr_mt_msg1
AcceptTransaction( )	rtr_accept
rtr_mt_msg1_uncertain	rtr_mt_msg1_uncertain
Receive (rtr_mt_accepted)	rtr_receive_message (rtr_mt_accepted)
SQL commit	**SQL commit**
. . .	. . .

If there is a crash *before* the AcceptTransaction method (`rtr_accept_tx` statement for the C API) is executed, on recovery, the transaction is replayed as `rtr_mt_msg1` because the database will have rolled back the prior transaction instance. However, if there is a crash *after* the AcceptTransaction method or `rtr_accept_tx` statement is executed, on recovery, the transaction is replayed as

`rtr_mt_msg1_uncertain` because RTR does not know the status of the prior transaction instance. Your application must understand the implications of such failures and deal with them appropriately.

3.2.2. Ensuring Consistency

A transaction either creates a new and valid state of data, or, if any failure occurs, returns all data to its state as it was before the start of the transaction. This is called *consistency*.

Several rules must be considered to ensure consistency:

- Shadowed applications must lock records being updated and hold these locks while RTR performs commit processing. Note that shadowed transactions can be executed at different times and in a different order between two sites because RTR assumes that transactions are independent.
- Client applications that rely on system-dependent information sent from the server application should use the REPLYDIFF option in RTR to ensure that a single transaction does not span multiple nodes. For example, if data are returned to the client application before the server crashes, the recovered transaction may produce results that conflict with the message previously returned to the client.
- Applications should never access shadowed data sets outside the RTR environment. This could, for example, be the case if your application relies on batch processing that does not use RTR or database maintenance. Updating between the two sites will not be serialized if all updates do not pass through RTR. Updating two sites only after quiescing shadow systems is impractical in most mission-critical applications.
- Partition your data so that all access is done through a single RTR partition (key range). It is unusual to partition by function or use shared tables. Although partitioning is recommended for improving concurrency, RTR serializes transactions only on a partition basis. If two partitions have overlapping data records, these records might be updated in differing order across the two sites, unless the application deals with them correctly. Do not overlap partitions; make them discrete.
- Perform all data-constraint checking before committing the database. This can be an issue with certain database optimizations.

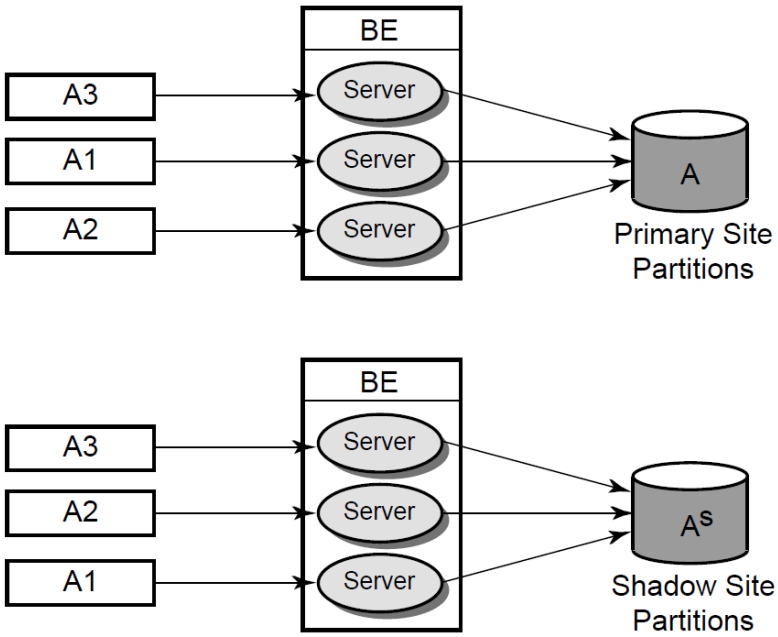
3.2.3. Ensuring Isolation

The changes to shared resources that a transaction causes do not become visible outside the transaction until the transaction commits. This makes transactions serializable. To ensure isolation:

- Hold record locks throughout the RTR commit cycle. If the server crashes, the transaction could be recovered after changes to the data by a dependent transaction, generating results that are different from those already sent to the client. Also, new transactions can overtake the completion of a previous transaction from the same client if shared records are not locked.
- Do not use RTR concurrent servers if your data manager does not support record locking. This can be important, for example, with in-memory databases. Concurrency relies on the independence of two operations that may affect common data records. Record locking ensures that a concurrent transaction cannot affect the consistency of data being operated on by another transaction.
- Avoid certain site-dependent actions when running RTR shadow servers. For example, using transaction sequence numbers or time and date comparisons, can introduce problems. Shadowed transactions are serialized based on commit groups. If your application requires absolute transaction serialization, you cannot run concurrent servers. For example, Figure 3.4 illustrates the serialization of commit groups. The first commit group is A1, processed on the primary backend; it is followed by commit group A2, followed by A3. Commit to the database, however, is in the order A3, A1,

A2, as shown in the diagram. On the shadow site, commit to the database will be in the order A3, A2, A1, due to use of concurrent servers. If absolute serialization of CSNs is important in your application, you cannot use concurrent servers.

Figure 3.4. Concurrent Server Commit Grouping



RTR commit grouping allows independent transactions to be scheduled together on the shadow secondary.

3.2.4. Ensuring Durability

For a transaction to be durable, the changes caused by transaction commitment must survive subsequent system and media failure. Thus transactions are both persistent and stable.

For example, your bank deposit is durable if, once the transaction is complete, your account balance reflects what you have deposited.

The durability rule is:

- Standby servers that update the database must have access to each other's RTR journal, and use cluster-aware data managers such as Oracle Parallel Servers. If a node running as a standby server crashes, in-progress transactions will be recovered from the failed node's journal files.

3.2.5. Transaction Dependencies with Concurrent Servers

If there are dependencies between separate RTR transactions, these should be considered carefully because the locking mechanisms of resource managers can cause unexpected behavior. These issues around locking mechanisms occur only if there is more than one server for the same partition.

For example, consider the case where there is a transaction T1 which inserts a record in the database and a subsequent transaction T2 which uses that record to make another update. If the partition has been configured with concurrent servers, it can happen that the update transaction T2 which has been given to a free server will begin executing and reach the database before the insert operation issued by transaction

T1 has completed the commit process. In this scenario, the inserted record is not yet visible to the update transaction T2 because the commit is not yet complete. This will cause transaction T2 to fail. However, if the database table being updated is locked for the duration of the insert, transaction T2 will block (wait) until the insert has committed and there will be no possibility of transaction T2 overtaking transaction T1.

In another example, the first transaction T1 makes an update to the table and a second transaction T2 uses the updated value in its transaction. If the resource manager does not lock the row being accessed by transaction T1 right at the start of the update, that row can be queried by the second transaction T2 which has started on a concurrent server. However, transaction T2 will in this case be working with the old and not the updated value that was the result of T1. To prevent such unexpected and potentially undesirable behavior, check the locking mechanisms of the resource managers being used before using concurrent servers.

3.2.6. Server-Side Transaction Timeouts

RTR provides client applications the option to specify a transaction timeout, but has no provision for server applications to specify a timeout on transaction duration. If there is a scarcity of server application processes, all other client transactions remain queued. If these transactions have also specified timeouts, they are aborted by RTR (assuming that the timeout value is less than 2 minutes).

To avoid this problem, the application designer has two choices:

- Use concurrent server processes
- Have the application administer its own timeout

The first (and easier) option is to use concurrent server processes. This allows transaction requests to be serviced by other free servers, even if one server is occupied by such a transaction that is taking a long time to disappear. The second option is to design the server application so that it can abort the transaction independently.

There are three cases where this use of concurrent servers is not ideal. First, there is an implicit assumption about how many such lingering transactions might remain on the system. In the worst case, this could exceed or equal the number of client processes. But having so many concurrent server processes to cater to this contingency is wasteful of system resources. Second, use of concurrent servers is beneficial when the servers do not need to access a common resource. For instance, if all these servers needed to update the same record in the database, they would simply be waiting on a lock taken by the first server. Additional servers do not resolve this issue. Third, it must make business sense to have additional servers. For example, if transactions must be executed in the exact order in which they entered the system, concurrent servers may introduce sequencing problems.

Take the example of the order matcher in a stock trading application. Business rules may dictate that orders be matched on a first-come, first- matched basis; using concurrent servers would negate this rule.

The second option is to let the server application process administer its own timeout and abort the transaction when it sees no activity on its input stream.

3.2.7. Two-Phase Commit Process

To ensure that transactions are fully executed and that the database is consistent, RTR uses the *two-phase commit* process for committing a transaction. The two-phase commit process has both a prepare phase and a commit phase. Transactions must reach the commit phase before they are hardened in the database.

The two-phase commit mechanism is initiated by the client when it executes a call to RTR that declares that the client has accepted the transaction. The servers participating in the transaction are then asked to be prepared to accept or roll back the transaction, based on a subsequent request.

3.2.7.1. Prepare Phase

Transactions are prepared before being committed by accept processing. Table 3.1 lists backend transaction states that represent the steps in the prepare phase.

Table 3.1. Backend Transaction States

Phase	State	Meaning
Phase 0	WAITING	Waiting for a server to become free.
	RECEIVING	Processing client messages.
Phase 1	VREQ	Vote of server requested.
	VOTED	Server has voted and awaits final transaction status.
Phase 2	COMMIT	Final status of a committed transaction delivered to server.
	ABORT	Final status of an aborted transaction delivered to server.

The RTR frontend sees several transaction states during accept processing. Table 3.2 lists frontend transaction states that represent the steps in the prepare phase.

Table 3.2. Frontend Transaction States

State	Meaning
SENDING	Processing, not ready to accept.
VOTING	Accept processing in process; frontend has issued an `rtr_accept_tx` call, but the transaction has not been acknowledged.
DONE	Transaction is complete, either accepted or rejected.

Implementation details are shown in the separate chapters for the RTR APIs.

3.3. RTR Messaging

With RTR, client/server messaging enables the application to send:

- Transactional messages
- Broadcast messages

3.3.1. Transactional Messages

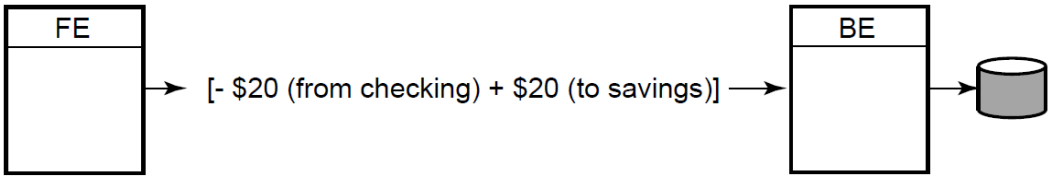
With RTR, client and server applications communicate by exchanging messages in a transaction dialog. Transactional messages are grouped in a unit of work called a transaction. RTR takes ownership of a message when called by the application.

A transaction is a group of logically connected messages exchanged in a transaction dialog. Each dialog forms a transaction in which all participants have the opportunity to accept or reject the transaction. A transaction either commits or aborts. When the transaction is complete, all participants are informed of the transaction's completion status. The transaction succeeds if all participants accept it, but fails if even one participant rejects it.

In the context of a transaction, an RTR client application sends one or more messages to the server application, which responds with zero or more replies to the client application. Client messages can be grouped to form a transaction. All work within a transaction is either fully completed or all work is undone. This ensures transaction integrity from client initiation to database commit with the cooperation of the server application.

For example, say you want to take $20 from your checking account and add it to your savings account. With an application using RTR you are assured that this entire transaction is completed; you will not be left at the point where you have taken $20 from your checking account but it has not yet been deposited in your savings account. This feature of RTR is transactional integrity, illustrated in Figure 3.5.

Figure 3.5. Transactional Messaging



The transactional message is either all or nothing for everything enclosed in brackets [ ] in Figure 3.5.

An RTR client application sends one or more messages to one or more server applications and receives zero or more responses from one or more server applications. For example:

The client application:

- Initiates the transaction.
- Sends the messages that contain routing information used by RTR to select the appropriate server.
- Receives the replies.
- Commits the transaction. The client application must be on the RTR frontend.

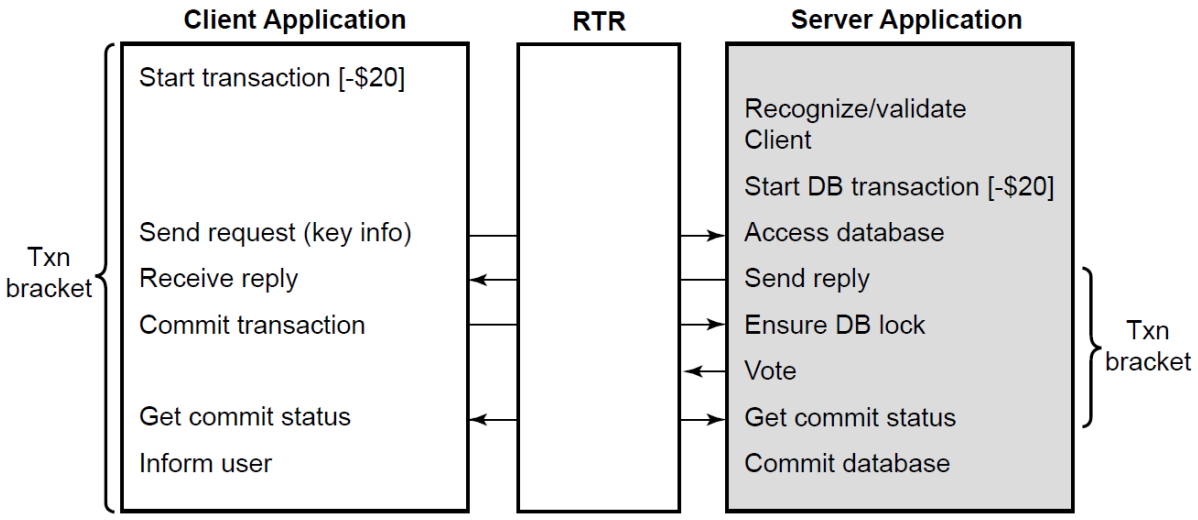
The server application:

- Waits to receive the transaction.
- Receives messages from the client.
- Processes requests from the client.
- Sends responses.
- Votes its acceptance of the transaction, either committing or rolling back the transaction to the database, depending on the outcome. The server application is deployed on the RTR backends.

RTR generates a unique identifier, the transaction ID or TID, for each transaction. The client can inject also its own TID into RTR. Doing so will make RTR treat the transaction as a nested transaction.

Figure 3.6 illustrates frontend/backend interaction with pseudo-code for transactions and shows transaction brackets. The transaction brackets show the steps in completing all parts of a transaction, working from left to right and top to bottom. The transaction (txn) is initiated at "Start txn" at the frontend, and completed after the "Commit txn" step on the backend. The transaction ID is encoded to ensure its uniqueness throughout the entire distributed system. In the prepare phase on the server, the application should lock the relevant database (db) records. The commit of a transaction hardens the commit to the database. The `rtr_start_tx` message specifies the characteristics of the transaction. RTR identifies the server based on key information in the transaction.

Figure 3.6. Transactional Messaging Interaction



3.3.2. Broadcast Messages

Broadcast messaging lets client and server applications send non- transactional information to recipients. Recipients must be declared with the opposite role; that is, a client can send a broadcast message to servers, and a server can send a broadcast message to clients. Broadcasts are delivered on a "best try" basis; they are not guaranteed to reach every potential recipient. A broadcast message can be directed to a specific client or server, or be sent to all reachable recipients.

This point-to-point messaging using broadcast semantics is a feature to use instead of transactions when the information being sent is not recorded as a transaction in the database, and when you need to send information to several clients (or servers) simultaneously. For example, in a stock trading environment, when a trade has been completed and the stock price has changed, the application can use a broadcast message to send the new stock price to all trading stations. Another use for such messages is to inform the applications about a state change in the environment (for example, the fact that the exchange is now closed for business).

Other considerations when using broadcast messages include:

- Flow control issues
- Sequencing of broadcasts
- Sequencing relative to transaction delivery
- Recovery/replay (or lack thereof)
- Under what conditions broadcasts can be lost

- Design issues on how to cope with loss of broadcast

3.3.2.1. Flow Control

Broadcast messages are subject to flow control. A broadcaster may be blocked and unable to send messages when traffic is high and recipients are unable to process the broadcasts. The broadcaster sends at the minimum rate (MINIMUM_BROADCAST_RATE) which can be set to send "no matter what" for a given node. However, if an application does this, the application may in practice hold up broadcasts for others, and application design must take this into account. For example, no client application should be able to issue a Control S (^ S) to hold up all broadcasts. If an application doing broadcasts works with transactions that might get held up, it may be time to consider using multiple channels on multiple threads.

3.3.2.2. Sequencing of Broadcasts

RTR guarantees that broadcasts are received in the same order as sent by a specific sender. However, if there is more than one sender in an application, different recipients can receive broadcasts in different orders. For example, Sender A could send broadcasts ABC and Sender B, broadcasts XYZ. These could be received by two different recipients as ABCXYZ or XYZABC. If this is important in your application, correct application design is to use one sender that collects and distributes all input needed for such broadcasts.

3.3.2.3. Sequencing Relative to Transaction Delivery

Consider a shadowed trading environment that initiates 5PM processing with a broadcast for closing of the exchange. Application design should send broadcasts and transactions through different pipes. Because RTR does not guarantee receipt of a broadcast at all servers, but does guarantee receipt of transactions, this critical "broadcast" could be most effectively handled by being sent in a transaction as an event through the transaction pipe.

3.3.2.4. Recovery of Broadcasts

There is no replay or recovery for broadcasts.

3.3.2.5. Lost Broadcasts

A broadcast can sometimes be lost. This can be caused by link loss, low memory in the RTRACP, or excessively high volume.

3.3.2.6. Coping with Broadcast Loss

There is overhead associated with managing and correcting for loss of broadcasts. Thus VSI recommends that applications do not use broadcasts for critical information. If, however, an application decides to use broadcasts and wants to ensure that all broadcasts are accounted for, one approach is to add a tracking sequence number to each broadcast that is sent out. All recipients can then check for missing sequence numbers and request a resend of any missing broadcasts.

3.4. Broadcast Messaging Processes

A client or server application may need to send unsolicited messages to one or more participants. Applications tell RTR which broadcast classes they want to receive.

The sender sends one message received by several recipients. Recipients subscribe to a specific type of message. Delivery is not guaranteed. Broadcast messages can be:

- Application messages from client to server applications
- Application messages from server to client applications
- One-to-many or one-to-one

Clients cannot broadcast to other clients, and servers cannot broadcast to other servers. To enable communication between two applications of the same type, open a second instance of the application of the other type. Messaging destination names can include wildcards, enabling flexible definition of the subset of recipients for a particular broadcast.

Broadcast types include **user events** and **RTR events**; both are numbered.

3.4.1. User Events

Event numbers are provided as a list beginning with `RTR_EVTNUM_USERDEF` and ending with `RTR_EVTNUM_ENDLIST`. To subscribe to all user events, an application can use the range indicators `RTR_EVTNUM_USERBASE` and `RTR_EVTNUM_USERMAX`, separated by `RTR_EVTNUM_UP_TO`, to specify all possible user event numbers.

A user broadcast is named or unnamed. An unnamed broadcast does a match on user event number; the event number completely describes the event. A named broadcast does a match on both user event number and recipient name. The recipient name is a user-defined string. Named broadcasts provide greater control over who receives a particular broadcast.

Named events specify an event number and a textual recipient name. The name can include wildcards (% and *).

For all unnamed events specify the *evtnum* field and `RTR_NO_RCPSPC` as the recipient name.

3.4.2. RTR Events

RTR delivers status information to which client and server applications can subscribe. Status information is delivered as messages, where the type of each message is an RTR event.

RTR events are numbered. The base value for RTR events is defined by the symbol `RTR_EVTNUM_RTRBASE`; its maximum value is defined by the symbol `RTR_EVTNUM_RTRMAX`. RTR events and event numbers are listed in the Reliable Transaction Router API manuals and in the RTR header files rtr.h and rtrapi.h.

An application can subscribe to RTR events to receive notification of external events that are of interest to the application. For example, a shadow server may need to know if it is a primary or a secondary server to perform certain work, such as uploading information to a central database, that is done at only one site.

To subscribe to all RTR events, use the range indicators `RTR_EVTNUM_RTRBASE` and `RTR_EVTNUM_RTRMAX`. RTR events are delivered as messages of type `rtr_mt_rtr_event`.

In application design, consider creating separate facilities for sending broadcasts. By separating broadcast notification from transactional traffic, performance improvements can be substantial. Facilities can further be reconfigured to place the RTR routers strategically to minimize wide-area traffic.

A server application can expect to see a primary or secondary event delivered only in certain transaction states. For more detail, see the state diagrams in Appendix C, Server States.

3.5. Location Transparency

With location transparency, applications do not need to be modified when the hardware configuration is altered, whether changes are made to systems running RTR services or to the network topology. Client and server applications do not know the location of one another so services can be started anywhere in the network. Actual configuration binding is a system management operation at run time, through the assignment of roles (frontend/backend/router) within a given facility to the participant nodes.

For RTR to automatically take care of failover, server applications need to specify certain availability attributes for the partition.

Because RTR automatically takes care of failover, applications need not be concerned with specifying the location of server resources.

3.6. Handling Error Conditions

RTR can provide information to an application with the RTRMessage and RTREvent classes (for the C ++ API). Certain inherited methods within these classes translate RTR internal error message values to informational text meaningful to the reader. For the C API, this is done with the `rtr_error_text` call.

If an application encounters an error, it should log the error message received. Error messages are more fully described in rtrapi.h for the C++ API and in rtr.h for the C API, where each error code is explained.

For example, the following short program uses the standard C library output function to display the text of an error status code.

```
Program "prog":
#include "rtr.h"
```

or

```
#include <rtr.h>
main() {
 printf("%s",
  rtr_error_text(RTR_STS_NOLICENSE));
 }
```

When this program is run, it produces the following output:

```
$run prog
 No license installed
```

The several hundred error or status codes reside in the rtr.h header file; status codes can come from any RTR subsystem. A few codes that an application is likely to encounter are described in Table 3.3.

Table 3.3. RTR Error Codes

Status Code	Meaning
`RTR_STS_COMSTAUNO`	Commitment status unobtainable. The fate of the transaction currently being committed is unobtainable; this may be due to a hardware failure.
`RTR_STS_DLKTXRES`	The transaction being processed was aborted due to deadlock with other transactions using the same

Status Code	Meaning
	servers. RTR will replay the transaction after the deadlock has been resolved and cleared.
RTR_STS_EARVINC	Early Vote option, which by default is enabled when NORECOVERY partition option is selected, causes secondary ACP process to vote on a transaction even before primary does the same. Loss of primary site before voting on the active transactions results in an inconsistent transaction states at surviving shadow site. RTR aborts such transactions and passes status information to the applications.
RTR_STS_FELINLOS	Frontend link lost; probably due to a network failure.
RTR_STS_INVFLAGS	Invalid flags.
RTR_STS_NODSTFND	No destination found; no server had declared itself to handle the key value specified in the sent message. Probably a server down or disconnected. If a RTR router does not know of a backend that has a server running for the specified key range, then the router may abort the transaction with this status. NOTE: If a client application receives a NODSTFND error, then the client can try to resend the transaction, since the cause may have been only temporary.
RTR_STS_REPLYDIFF	Two servers respond with different information during a replay; transaction aborted.
RTR_STS_TIMOUT	Timeout expired; transaction aborted.
RTR_STS_SRVDIED	Probably a server image exited, for example because a node is down.
RTR_STS_SRVDIEDVOT	A server exited before committing a transaction.
RTR_STS_SRVDIEDCOM	A server exited after being told to commit a transaction.

RTR can abort a transaction at any time, so the application must be prepared to deal with such aborted transactions. Server applications are expected to roll back transactions as the need arises, and must be built to take the correct action, and subsequently carry on to deal with new transactions that are received.

A client application can also get a reject and must also be built to deal with the likely cases it will encounter. The application must be built to decide on the correct course of action in the event of a transaction abort.

3.7. Using Locks

When using a database system with RTR, an application designer must be aware of how the database system works and how it handles database locks. Because Oracle is a frequently used database system, this section provides a short summary of Oracle locking methods. The application designer must use

Oracle documentation to supplement this brief description. This material is fully discussed in the *Oracle8 Application Developer's Guide* and *Oracle8i Application Developer's Guide*, specifically in the chapters on Processing SQL Statements, Explicit Data Locking, Explicitly Acquiring Row Locks, Serializable and Row Locking Parameters, User Locks, Non-Default Locking, and Concurrency Control Using Serializable Transactions. Oracle database operations are performed using Structured Query Language (SQL).

3.7.1. Oracle Locking

3.7.1.1. Privileges Required

In its own schema, an application can automatically acquire any type of table locks. However, to acquire a table lock on a table in another schema, the application must have the LOCK ANY TABLE system privilege or an object privilege such as SELECT or UPDATE for the table.

3.7.1.2. Overriding Default Locking

By default, Oracle locks data structures automatically. However, an application can request specific data locks on rows or tables when it needs to override default locking. Explicit locking lets an application share or deny access to a table for the duration of a transaction.

An application can explicitly lock entire tables using the LOCK TABLE statement, but locking a table means that no other transaction, user, or application can access it. This can cause performance problems.

With the SELECT FOR UPDATE statement, an application explicitly locks specific rows of a table to ensure the rows do not change *before* an update or a delete. Oracle automatically obtains row-level locks at update or delete time, so use the FOR UPDATE clause only to lock the rows before the update or delete.

A SELECT statement with Oracle does not acquire any locks, but a SELECT … FOR UPDATE does. For example, the following is a typical SELECT … FOR UPDATE statement:

```
SELECT partno FROM parts FOR UPDATE OF price
```

This statement starts a transaction to update the parts table with a price change for a specific part.

3.7.1.3. Oracle Explicit Data Locking

To ensure data concurrency, integrity, and statement-level read consistency, Oracle always performs necessary locks. However, an application can override default locks. This can be useful when:

- *An application needs transaction-level read consistency or repeatable reads* . For example, when transactions must query a consistent set of data for the duration of the transaction, and the application must be sure that the data have not been changed by any other transactions. To achieve transaction-level read consistency, an application can use:
  - Explicit locks
  - Read-only transactions
  - Serializable transactions
  - Default-lock overrides
- An application requires a transaction to have exclusive access to a resource. A transaction with exclusive access need not wait for other transactions to complete.

Overrides to Oracle locks can be done at two levels:

- Transaction level
- System level

At *transaction level*: Transactions override Oracle default locks with the following SQL statements:

- LOCK TABLE
- SELECT … FOR UPDATE
- SET TRANSACTION … READ ONLY
- SET TRANSACTION … ISOLATION LEVEL SERIALIZABLE

At *system level*: Oracle can start an instance with non-default locking by adjusting the following initialization parameters:

- SERIALIZABLE
- ROW_LOCKING

If an application overrides any Oracle default locks, the application itself must:

- Ensure that the overriding locking procedures work correctly.
- Guarantee data integrity.
- Ensure acceptable data concurrency.
- Ensure that deadlocks cannot occur or are handled appropriately.

3.7.1.4. Table Locks

When a LOCK TABLE statement executes, it overrides default locking, and a transaction explicitly acquires the specified table locks. A LOCK TABLE statement on a view locks the underlying base tables (see Table 3.4).

Table 3.4. LOCK TABLE Statements

Statement	Meaning
LOCK TABLE tablename IN EXCLUSIVE MODE [NOWAIT];	Acquires exclusive table locks. Locks all rows of the table. No other user can modify the table. With NOWAIT, the application acquires the table lock only if the lock is immediately available, and Oracle issues an error if not. Without NOWAIT, the transaction does not proceed until the requested table lock is acquired. If the wait for a table lock reaches the limit set by the initialization parameter DISTRIBUTED_LOCK_TIMEOUT, a distributed transaction can time out. As no data will have been modified due to the timeout, the application can proceed as if it has encountered a deadlock.

Statement	Meaning
LOCK TABLE tablename IN ROW SHARE MODE; LOCK TABLE tablename IN ROW EXCLUSIVE MODE;	These offer the highest degree of concurrency. Consider if the transaction must prevent another transaction from acquiring an intervening share, share row, or exclusive table lock for a table before the table can be updated in the transaction. If another transaction acquires an intervening share, share row, or exclusive table lock, no other transactions can update the table until the locking transaction commits or rolls back.
LOCK TABLE tablename IN SHARE MODE;	Consider this share table lock if: • Your transaction only queries the table and requires a consistent set of table data for the duration of the transaction (requires transaction-level read consistency for the locked table). • Other transactions to update the locked table concurrently can be made to wait until all transactions with the share table locks commit or roll back. • Other transactions can acquire concurrent share table locks on the same table, providing them transaction-level read consistency. **Note:** If multiple transactions concurrently hold share table locks for the same table, NO transaction can update the table. Thus if share table locks on the same table are common, deadlocks will be frequent and updates will not proceed. For such a case, use share row exclusive or exclusive table locks.
LOCK TABLE tablename IN SHARE ROW EXCLUSIVE MODE;	Acquire a share row exclusive table lock when: • Your transaction requires both transaction-level read consistency for the specified table, and the ability to update the locked table. • Other transactions can obtain explicit row locks. • The application needs only a single transaction to have this behavior.

3.7.1.5. Acquiring Row Locks

The SELECT … FOR UPDATE statement acquires exclusive row locks of selected rows. The statement can be used to lock a row without changing the row. Acquiring row locks can also be used to ensure that only a single interactive application user updates rows at a given time. For information on using this statement with cursors or triggers, see the Oracle8 or Oracle8i documentation. To acquire a row lock only when it is immediately available, include NOWAIT in the statement.

Each row in the return set of a SELECT … FOR UPDATE statement is individually locked. The statement waits until a previous transaction releases the lock. If a SELECT … FOR UPDATE statement locks many rows in a table, and the table is subject to moderately frequent updates, it may improve performance to acquire an exclusive table lock rather than using row locks.

If the wait for a row lock reaches the limit set by the initialization parameter DISTRIBUTED_LOCK_TIMEOUT, a distributed transaction can time out. As no data will have been modified, the application can proceed as if it has encountered a deadlock.

3.7.1.6. Setting SERIALIZABLE and ROW_LOCKING Parameters

How an instance handles locking is determined by the SERIALIZABLE option on the SET TRANSACTION or ALTER SESSION command, and the initialization parameter ROW_LOCKING. By default, SERIALIZABLE is set to *false* and ROW_LOCKING is set to *always*.

Normally these parameters should never be changed. However they may be used for compatibility with applications that run with earlier versions of Oracle, or for sites that must run in ANSI/ISO-compatible mode. Performance will usually suffer with non-default locking.

3.7.1.7. Using the LOCK TABLE Statement

The application uses the LOCK TABLE statement to lock entire database tables in a specified lock mode to share or deny access to them. For example, the statement below locks the parts table in row-share mode. Row-share locks allow concurrent access to a table; they prevent other users from locking the entire table for exclusive use. Table locks are released when your transaction issues a commit or rollback.

```
LOCK TABLE parts IN ROW SHARE MODE NOWAIT;
```

The lock mode determines which other locks can be placed on the table. For example, many users can acquire row-share locks on a table at the same time, but only one user at a time can acquire an exclusive lock. While one user has an exclusive lock on a table, no other users can insert, delete, or update rows in that table. For more information about lock modes, see the *Oracle8 Server Application Developer's Guide* or *Oracle8i Server Application Developer's Guide*.

A table lock never keeps other users from querying a table, and a query never acquires a table lock. Only if two different transactions try to modify the same row will one transaction wait for the other to complete.

If your program includes SQL locking statements, make sure the Oracle users requesting the locks have the privileges needed to obtain the locks.

Table 3.5. Summary of Locking Options

Case	Description	SERIALIZABLE	ROW_LOCKING
0	Default settings	FALSE	ALWAYS
1	As Oracle Version 5 and earlier (no concurrent inserts, updates or deletes in a table)	FALSE (disabled)	INTENT
2	ANSI compatible	Enabled	ALWAYS
3	ASNI compatible with table-level locking (no concurrent inserts,	Enabled	INTENT

Case	Description	SERIALIZABLE	ROW_LOCKING
	updates or deletes in a table)		

Table 3.6. Non-Default Locking Behavior

Statement	Case 0	Case 1		Case 2		Case 3	
SERIALIZABLE	FALSE (disabled)	Disabled		Enabled		Enabled	
ROW_LOCKING	ALWAYS	INTENT		ALWAYS		INTENT	
		Row	Table	Row	Table	Row	Table
SELECT		-	-	-	S	-	S
INSERT		X	SRX	X	RX	X	SRX
UPDATE		X	SRX	X	SRX	X	SRX
DELETE		X	SRX	X	SRX	X	SRX
SELECT … FOR UPDATE		X	RS	X	S	X	S
LOCK TABLE … IN..							
ROW SHARE MODE		RS	RS	RS	RS	RS	RS
ROW EXCLUSIVE MODE		RX	RX	RX	RX	RX	RX
SHARE MODE		S	S	S	S	S	S
SHARE ROW EXCLUSIVE MODE		SRX	SRX	SRX	SRX	SRX	SRX
EXCLUSIVE MODE		X	X	X	X	X	X
DDL Statements		-	X	-	X	-	X

Modes:	X = EXLUSIVE
	RS = ROW SHARE
	RX = ROW EXCLUSIVE
	S = SHARE
	SRX = SHARE ROW EXCLUSIVE

The information in this table comes from the *Oracle8i Application Developer's Guide*.

3.7.2. Distributed Deadlocks

A deadlock or deadly embrace can occur when transactions lock data items in a database. The typical scenario is with two transactions txn1 and txn2 executing concurrently with the following sequence of events:

1. txn1 write-locks data item A.

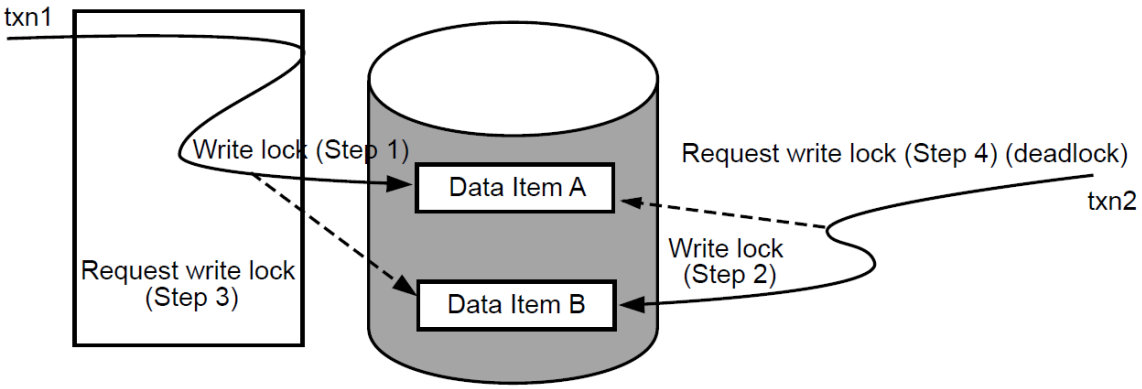
2. txn2 write-locks data item B.

3. txn1 requests a lock on data item B but must wait because txn2 still has a lock on data item B.

4. txn2 requests a lock on data item A but must wait because txn1 still has a lock on data item A.

Neither txn1 nor txn2 can proceed; they are in a deadly embrace. Figure 3.7 illustrates a deadly embrace.

Figure 3.7. Deadly Embrace



With RTR, to avoid such deadlocks, follow these guidelines:

1. Always engage servers in the same order, and wait for the reply before each send.

2. Provide several concurrent servers to minimize contention. Estimate the number of concurrent servers needed by determining the volume of transactions the servers must support, considering periods of maximum activity, and allowing for growth. The larger the volume on your servers, the more likely it is that your application will benefit from using concurrent servers.

RTR attempts to resolve deadlocks by aborting one deadlocked transaction path with error code `RTR_STS_DLKTXRES` and replaying the transaction. Other paths are not affected. Server applications should be written to handle this status appropriately.

The RTR status code `RTR_STS_DLKTXRES` can occur under several environmental conditions that RTR detects and works around. The application need not take any explicit action other than releasing the resources connected with the active transaction such as doing a rollback on the database transaction.

For example, RTR may issue an `RTR_STS_DLKTXRES` status when:

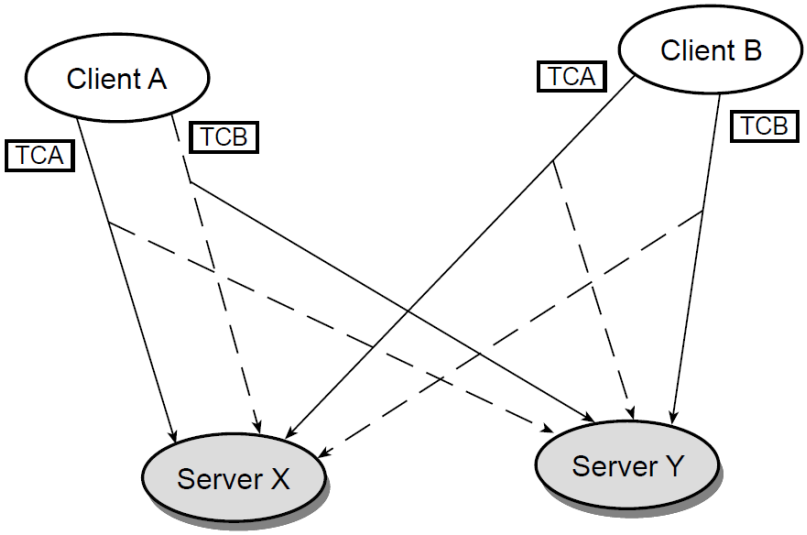
- There are at least two client applications, or one client with two or more channels.

- There are multi-participant transactions, where each transaction contains two or more messages involving more than one server application.

- The number of servers is less than or equal to the number of transaction branches that can be simultaneously active.

As an example of the first case, consider clients A and B both performing transactions TCA and TCB, where both TCA and TCB include a message to both server X and server Y followed by an ACCEPT. There is only one instance of Server X and Server Y available, and due to the quirks of distributed processing, only Server X receives the message belonging to TCA and only Server Y receives the message belonging to TCB. Figure 3.8 reflects this scenario. Because Server Y has no chance of accepting TCA until TCB is processed to completion and Server X has no chance of accepting TCB until TCA is processed to completion, Server X and Y are in a distributed deadlock. In such a case, RTR selects TCA or TCB to abort with `DLKTXRES` and replays it in a different order.

Figure 3.8. Scenario for Distributed Deadlock



Sometimes RTR needs to abort a transaction and reschedule it. For example, it can happen that a state change is needed after the primary server started to process a transaction but RTR had to change its role to secondary before the transaction was completed. Thus the transaction would be executed on the other node as primary and later played to this server as secondary. RTR uses the same status code `RTR_STS_DLKTXRES` when aborting the transaction.

3.7.3. Providing Parallel Processing

One method for improving response time is to send multiple messages from clients without waiting for a reply. The messages can be sent to different partitions to provide parallel processing of transactions.

3.7.4. Establishing Read-Only Sites

For certain read-only applications, RTR can be used without shadowing to establish sites to protect against site failure. The method is to define multiple non-overlapping facilities with the same facility name across a network that is geographically dispersed. In the facility, define a failover list of routers, for example, some in one city, some in another. Then when the local router fails, a client is automatically reconnected to another node. If all local nodes in the facility are unavailable, the client is automatically connected to a node at the alternate site.

Another method is to define a partition on a standby server for read-only transactions. This minimizes network traffic to the standby. A read-only partition on a standby server can reduce node-to-node transaction locking.

3.8. Resolving Idempotency Issues

Generally, databases (and applications built to work with them) are required to be idempotent. That is, given a specific state of the database, the same transaction applied many times would always produce the same result. Because RTR relies on replays and rollbacks, if there is a server failure before a transaction is committed, RTR assumes the database will automatically roll back to the previous state, and the replayed transaction will produce results identical to the previous presentation of the transaction. RTR assumes that the database manager and server application provide idempotency.

For example, consider an internet transaction where you log into your bank account and transfer money from one account to another, perhaps from savings to checking. If you interrupt the transfer, and replay

it two hours later, the transfer may not succeed because it would be required to have been done within a certain time interval after the login. Such a transaction is not idempotent.

3.9. Designing for a Heterogenous Environment

In a heterogeneous environment, you can use RTR with several hardware architectures, both little endian and big endian. RTR does *data marshalling* in your application so that you can take advantage of such a mixed environment.

If you are constructing an application for a heterogeneous environment:

- Use RTR data marshalling for smooth transfer from one architecture to another.
- Do not use binary data, if at all possible (although you could adopt a convention for passing binary data between your machines).
- Make your applications as portable as possible, for example, adopting ANSI C as your programming language.
- Use C-style arguments.
- Check the network byte-order of your systems, and prepare your application accordingly.
- Check compiler settings and switches to ensure they produce consistent results (compilers may change spacing of messages based on how their switches are set).

3.10. Using the Multivendor Environment

With RTR, applications can run on systems from more than one vendor. You can mix operating systems with RTR, and all supported operating systems and hardware architectures can interoperate in the RTR environment. For example, you can have some nodes in your RTR configuration running OpenVMS and others running Windows.

To develop your applications in a multivendor environment:

- Develop your applications on one system, for example, on Windows using Microsoft Visual C++ following strict ANSI C implementation.
- When both the server and client code are debugged, move them to the non-NT system.
- Build and debug them on the non-NT system.

3.11. Upgrading from RTR Version 2 to RTR Versions 3 and 4

An existing application written using RTR Version 2 with OpenVMS will still operate with RTR Versions 3 and 4. Refer to the *Reliable Transaction Router Migration Guide* for pointers on using RTR Version 2 applications with RTR Version 3, and moving RTR Version 2 applications to RTR Version 3 or 4.

Chapter 4. Design with the C++ API

This chapter provides information on RTR transaction model and recovery concepts for client and server applications implemented with the C++ API. Topics include :

- Transactional messaging with the C++ API
- Transaction message processing
- Transaction recovery

Additional information on RTR transactions and recovery can be found in the Application Implementation chapter of this guide and in *VSI Reliable Transaction Router Getting Started*.

4.1. Transactional Messaging with the C++ API

Figure 4.1 illustrates frontend/backend interaction with pseudo-code for transactions and shows transaction brackets. The transaction brackets show the steps in completing all parts of a transaction, working from left to right and top to bottom. In the figure, TC stands for transaction controller.

The transaction is initiated at "Start transaction" on the frontend, and completed after the "Commit transaction" step on the backend. The transaction ID is encoded to ensure its uniqueness throughout the entire distributed system. In the prepare phase on the server, the application should lock the relevant database (DB) records. The commit of a transaction hardens the commit to the database. Figure 4.2 illustrates a typical call sequence between a client and server application. These calls are RTRClientTransactionController and RTRServerTransactionController class methods. The first call in both the client and server transaction controllers is to create a new transaction controller, for example, in the server, use RTRServerTransactionController::RTRServerTransactionController.

Figure 4.1. Transactional Messaging with the C++ API

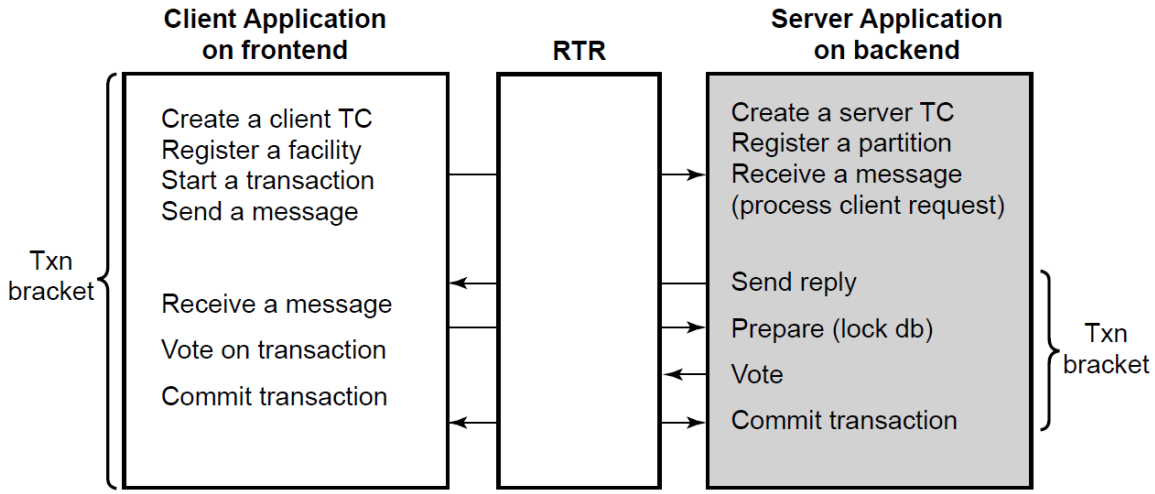
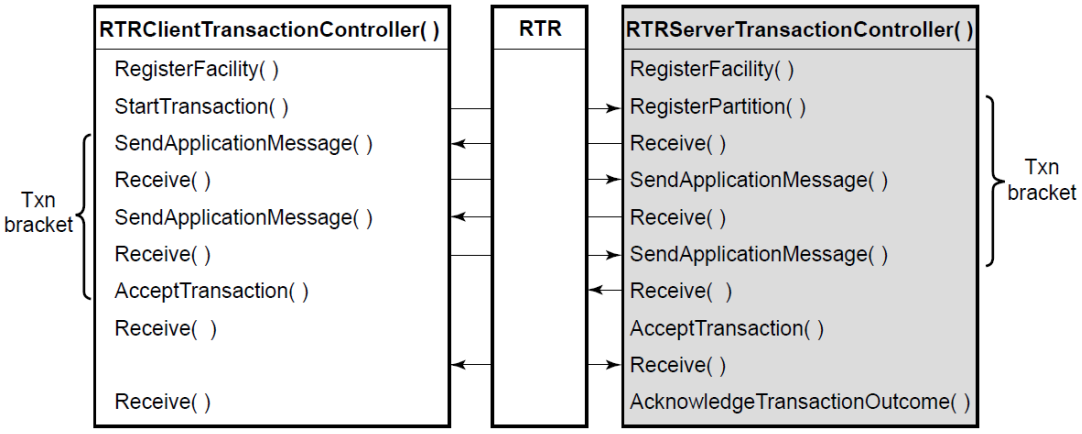


Figure 4.2. C++ API Calls for Transactional Messaging



For a client, an application typically uses the following methods in this order:

- RTRClientTransactionController constructor

Client first creates a transaction controller and a facility.

- RegisterFacility
- StartTransaction
- SendApplicationMessage

The client sends a request to the server.

- Receive

After the server has processed the request, the client calls Receive and the data object contains the RTR message `rtr_mt_reply`, causing the client message handler OnInitialize and OnApplicationMessage methods to be called.

- AcceptTransaction

The client calls AcceptTransaction, if all went well.

- Receive

For a server, an application typically uses the following methods in this order:

- RTRServerTransactionController constructor
- RegisterFacility
- RegisterPartition
- Initialize
- Receive

On the first server transaction controller Receive, RTRData contains `rtr_mt_msg1`. With event-driven processing (the default behavior) the server message handler calls OnInitialize and then calls OnApplicationMessage.

On a second Receive from a client SendApplicationMessage, the RTR message received in the data object contains `rtr_mt_msgn`, causing OnApplicationMessage to be called by the server message handler.

- SendApplicationMessage

After processing the client's request, the server calls SendApplicationMessage.

- Receive

When the client accepts the transaction, the server Receive call includes `rtr_mt_prepare` from RTR.

- AcceptTransaction

The server accepts the transaction.

- Receive

RTR sends `rtr_mt_accepted`

- AcknowledgeTransactionOutcome

With event-driven processing, the appropriate methods in the RTRClientMessageHandler and RTRServerMessageHandler classes are called by RTRData::Dispatch.
You must use the RTRClientTransactionController::RegisterHandlers and RTRServerTransactionController::RegisterHandlers methods to enable this default handling.

4.1.1. Data-Content Routing with the C++ API

Data-content routing is the capability provided by RTR to support the partitioned data model. With the C++ API, you define partitions with the RTRPartitionManager class. Partitions are defined with partition name, facility name and KeySegment attributes. Using RTR data-content routing, message content determines message destination. The routing key, defined in the C++ API RTRKeySegment class is embedded within the application message. When a server is started, the server informs RTR that it serves a particular partition.

When a client sends a message that references that particular partition, RTR knows where to find a server for that partition and routes the message accordingly. Even if the server is moved to another location, RTR can still find it. The client and the application do not need to worry about locating the server. This is location independence.

The benefits of data-content routing are simpler application development and flexible, scalable growth without application changes. RTR hides the underlying network complexity from the application programmer, thus simplifying development and operation.

4.1.2. Changing Transaction States

With Set State methods within the RTRServerTransactionProperties class, users can change a transaction state in their application servers. With the event-driven processing model, your states are shown through the message handlers. In the polling model, the states are accessed with an RTRData::GetMessageType() call.

Consider the following scenario: upon receiving an `rtr_mt_accepted` message indicating that a transaction branch is committed, the server normally performs an SQL commit to write all changes to

the underlying database. However, the server that is to detect the SQL commit or the underlying database may be temporarily unavailable.

With the C++ API, you can change the state of a transaction to EXCEPTION using the SetStateToException method, to temporarily put this transaction in the exception queue. When things get better, users can call SetStateToCommit or SetStateToDone to change the transaction state back to COMMIT or DONE, respectively.

The following example shows how a transaction state can be changed using the set state methods. See the RTRServerTransactionProperties class documentation in the *VSI Reliable Transaction Router C++ Foundation Classes* manual for more details.

```
(RTRServerTransactionProperties::SetStateToException();)
RTRServerTransactionProperties_object_name.SetStateToException(stCurrentTxnState);)
```

The parameter `stCurrentTxnState` is a transaction state of type `rtr_tx_jnl_state_t`.

Normally at the RTR command level, users need to provide at least facility name and partition name qualifiers to help RTR select a desired set of transactions to be processed. Because a transaction's TID `(rtr_tid_t)` is unambiguously unique, a user needs only to specify the transaction's current state and its TID.

Note that if a transaction has multiple branches running on different partitions simultaneously on this node, RTR will reject this set transaction request with an error. RTR can only change state for one branch of a multiple partition transaction at a time.

4.1.3. RTR Message Types

RTR calls (and responses to them) contain RTR message types (mt) such as `rtr_mt_reply` or `rtr_mt_rejected`. There are four groups of message types:

- Transactional
- Status
- Event-related
- Informational

Table 4.1 lists all RTR message types.

Table 4.1. RTR Message Types

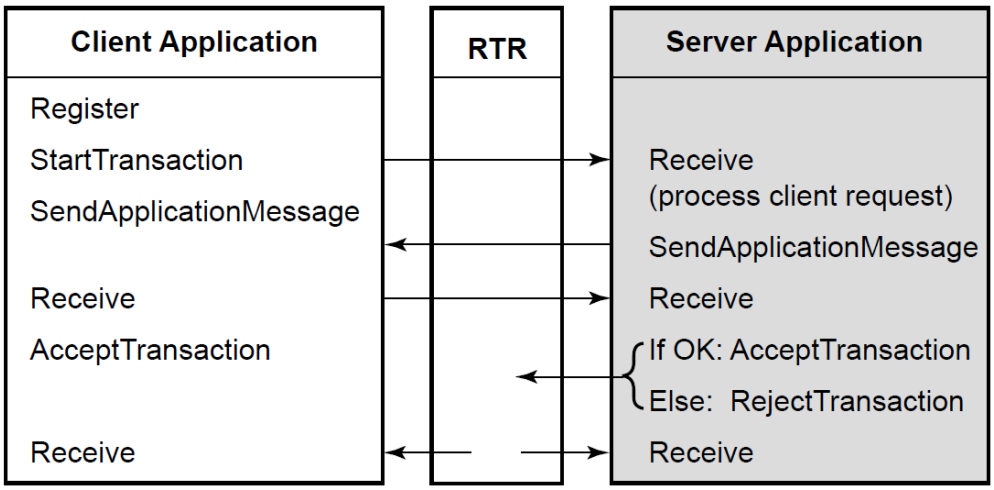
Transactional	Status	Event-related	Informational
rtr_mt_msg1	rtr_mt_accepted	rtr_mt_user_event	rtr_mt_opened
rtr_mt_msg1_uncertain	rtr_mt_rejected	rtr_mt_rtr_event	rtr_mt_closed
rtr_mt_msgn	rtr_mt_prepare		rtr_mt_request_info
rtr_mt_reply	rtr_mt_prepared		rtr_mt_rettosend

Applications should include code for all RTR expected return message types. Message types are returned to the application in the message status block. For more detail on message types, see the *VSI Reliable Transaction Router C Application Programmer's Reference Manual*.

4.2. Transactional Message Processing

Figure 4.3 illustrates transactional messaging interaction between a client application and a server application using C++ API calls.

Figure 4.3. Flow of a Transaction



In Figure 4.3:

1. The first send call in the client application starts a transaction and sends a message.
2. The server processes the transaction then sends a message back to the client.
3. The client receives the message and either accepts or rejects the transaction. The client's vote goes to RTR, not the server application.
4. The server then votes after receiving a message or event from RTR.

RTR sends both the server and the client a message that either declares that the transaction was accepted or rejected.

All the above steps comprise two parts of a transaction:

- Message processing
- Accept processing

4.2.1. Message Processing Sequence

Message processing involves a message being sent by a client and a reply coming back from one or more servers. This is where the business application processing takes place.

The typical steps during message processing are:

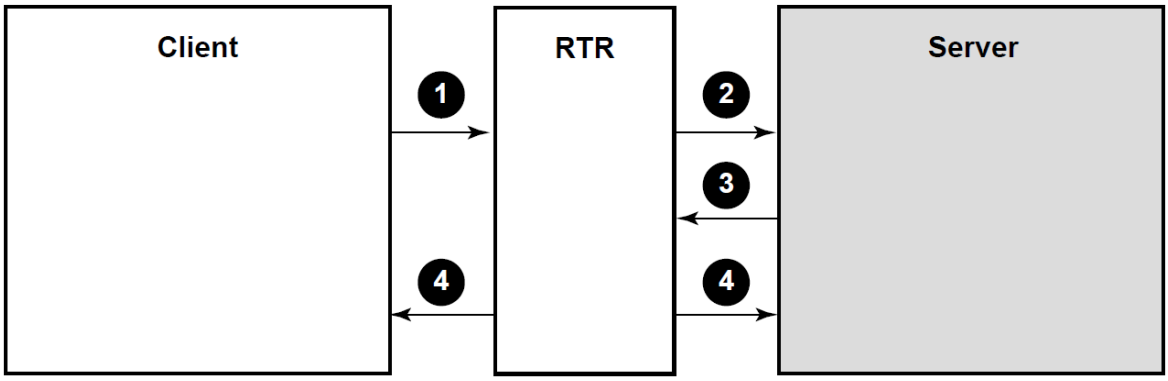
- Client starts a transaction
- Client sends a message to the server

- Server receives the message
- Server processes the request from the client
- Server replies to client
- Client receives message from server

4.2.2. Accept Processing

Figure 4.4 illustrates the accept-processing portion of a completed RTR transaction.

Figure 4.4. Accept Processing



The steps in accept processing are:

- ❶

  Client sends accept message to RTR.

- ❷

  Server receives prepare message.

- ❸

  Server accepts or rejects transaction.

- ❹

  Client and server receive final transaction state from RTR.

4.2.3. Starting a Transaction

There is one way to start a transaction - explicitly by creating a server transaction controller object, registering a facility, and using StartTransaction. The SendMessage call sends a message as part of a transaction from a client. If there is no transaction currently active on the transaction controller, a new one is started. The AcceptTransaction can be bundled with the last message. The SendMessage call also sends a reply message from a server to the client. The reply message is part of the transaction initiated by

the client. A transaction is defined as a group of messages initiated by the client. The server knows that a transaction has begun when it receives a message of one of the following types:

- OnInitialize and then OnApplicationMessage
- OnUncertainTransaction

If transaction timeouts are not required, the transaction starts on the next SendMessage call.

The Register call enables the transaction to join another transaction or set a transaction timeout. When a transaction is started implicitly, the timeout feature is disabled. A client has two options for message delivery after a failure:

- Return to sender
- Accept transaction

4.2.4. Identifying the Transaction

When a message is received, the message status block contains the transaction identifier (TID).

You can use the GetTID call to obtain the RTR transaction identifier for the current transaction. This identifier is a unique number generated by RTR for each transaction. The application can use the TID if the client needs to know the TID to take some action before receiving a response.

4.2.5. Accepting a Transaction

The AcceptTransaction call by the client begins the prepare phase of the two- phase commit protocol. An accepted transaction is not complete until a message of type `rtr_mt_accepted` or `rtr_mt_rejected` is received.

The application can specify a reason on the AcceptTransaction method so that the caller can specify an accept reason that is passed on to all participants in the transaction. If more than one transaction participant specifies a reason, the reason values are ORed together by RTR. The accept is final: the caller cannot reject the transaction later. The caller cannot send any more messages for this transaction.

A client can accept a transaction in one of two ways: with the AcceptTransaction call or by using the SetAcceptTransaction method. Using the SetAcceptTransaction method removes the need to issue an AcceptTransaction method and can help optimization of client traffic. Merging the data and accept messages in one call puts them in a single network packet. This can make better use of network resources and improve throughput.

4.2.6. Rejecting a Transaction

Any participant in the transaction can call RejectTransaction. The reject is final and it is impossible for the caller to accept the transaction later. The RejectTransaction method rejects a transaction. Once the transaction has been rejected, the caller receives no more messages for this transaction.

The server can call the ForceRetry method to have RTR redeliver the transaction beginning with `rtr_mt_msg1` without aborting the transaction for other participants. Using the RejectTransaction method, the application can specify a reason that the caller can pass on to all participants in the transaction. If more than one transaction participant specifies a reason, the reason values are ORed together by RTR.

4.2.7. Ending a Transaction

A server application can end a transaction by either accepting or rejecting the transaction. RTR can reject a transaction at any time after the transaction is started but before it is committed. For example, if RTR cannot deliver a transaction to its destination, it rejects the transaction and delivers the reject completion status to all participants that know about the transaction.

A transaction is accepted explicitly with the AcceptTransaction method, and rejected explicitly with the RejectTransaction method. RTR can reject a transaction at any time once the transaction is started but before it is committed. If RTR cannot deliver a transaction to its destination, it rejects the transaction explicitly and delivers the reject completion status to all participants.

A transaction participant can specify a reason for an accept or reject on the AcceptTransaction and RejectTransaction methods. If more than one transaction participant specifies a reason, RTR uses the OR operator to combine the reason values together. For example, with two servers, A and B, each providing a reason code of 1 or 2, respectively, the client receives the result of the OR operation, reason code 3, in its message buffer in RTRData.

A transaction is done once a client or server application receives a completion message, either an `rtr_mt_accepted` or `rtr_mt_rejected` message from RTR. An application no longer receives messages related to a transaction after receiving a completion message or if the application uses RejectTransaction A client or server can also specify SetForgetTransaction to signal its acceptance and end its involvement in a transaction early. RTR returns no more messages (including completion messages) associated with the transaction; any such messages received will be returned to the caller.

A client or server application no longer receives messages related to a transaction after it receives an OnAccepted or OnRejected message from RTR, or if the application called RejectTransaction.

4.2.8. Processing Summary

This section summarizes the default behavior of the client and server transaction controller objects when they interact in processing a transaction. When a transaction controller receives an RTRData object, it receives an RTR message. With the event-driven model of processing, Dispatch is automatically called and the appropriate methods, based on the RTR message, execute by default. For tables of the RTR to C++ API mapping of messages and events, see the *VSI Reliable Transaction Router C++ Foundation Classes* manual. Table 4.2 lists commonly used server transaction controller methods.

Table 4.2. RTRServerTransactionController Methods

Methods	RTR Message within RTRData	Event-Driven Default Calls	Default Behavior
RTRServerTransactionController::RegisterPartition()			
	NA (not applicable)	NA	Creates a partition.
RTRServerTransactionController::RegisterHandlers()			
	NA	NA	Registers server message and event handlers with the server transaction controller.
RTRServerTransactionController::Receive()			
	rtr_mt_msg1	OnInitialize	Receives application message.

Methods	RTR Message within RTRData	Event-Driven Default Calls	Default Behavior
		OnApplicationMessage	
RTRServerTransactionController::Receive()			
	rtr_mt_msgn	OnApplicationMessage	Receives application message.
RTRServerTransactionController::SendApplicationMessage()			
	NA	NA	Sends application message.
RTRServerTransactionController::Receive()			
	rtr_mt_prepare	OnPrepare	Receives application message.
RTRServerTransactionController::AcceptTransaction			
	NA	NA	Accepts transaction.
RTRServerTransactionController::Receive()			
	rtr_mt_accepted	OnAccepted	Receives application message.
RTRServerTransactionController::Receive()			
	rtr_mt_rejected	OnRejected	Receives application message.

To register two partitions, create two partitions and call Register once for each. Table 4.3 lists basic client transaction controller methods.

Table 4.3. RTRClientTransactionController Methods

ClientTransactionController	RTR Message within RTRData	Event- Driven Default Calls
RTRClientTransactionController:: Receive()	rtr_mt_reply	RTRServerMessageHandler:: (OnInitialize) RTRServerMessageHandler:: (OnApplicationMessage)
RTRClientTransactionController:: AcceptTransaction()	rtr_mt_accept	
RTRClientTransactionController:: Receive()	rtr_mt_accepted	OnAccepted

For more information on RTRTransactionController methods, refer to the *VSI Reliable Transaction Router C++ Foundation Classes* manual.

4.2.9. Administering Transaction Timeouts

RTR provides a relatively simple way to administer a transaction timeout in the server. Use of timeout values on the `Receive()` method lets a server application specify how long it is prepared to wait for the next message. (Of course, the server should be prepared to wait forever to get a new transaction or for the result of an already-voted transaction.)

One way to achieve this would be to have a transaction controller-specific global variable, say, called `SERVER_INACTIVITY_TIMEOUT`, which is set to the desired value (in milliseconds-that is, use a value of 5000 to set a 5-second timeout). Note that this timeout value should be used *after* receiving the first message of the transaction. The value should be reset to `RTR_NO_TIMOUTMS` after receiving the `rtr_mt_prepare` message. Whenever the Receive method completes with an `RTR_STS_TIMOUT`, the server transaction controller calls RejectTransaction to abort the partially processed transaction. This would prevent transactions from occupying the server process beyond a reasonable time.

4.2.10. Two-Phase Commit

A prepared application votes its intention using the AcceptTransaction method. An application that does not agree to commit to this transaction votes with the RejectTransaction method. This is the first (or prepare) phase of the two-phase commit process.

4.2.10.1. Initiating the Prepare Phase

The two-phase commit process with the C++ API is initiated by the client application when it issues a call to RTR indicating that the client "accepts" the transaction. This does not mean that the transaction is fully accepted, only that the client is *prepared* to accept it. RTR then asks the server applications participating in the transaction if they are prepared to accept the transaction. A server application that is prepared to accept the transaction votes its intention by issuing the AcceptTransaction method, an "accept" vote. A server application that is not prepared to accept the transaction issues the RejectTransaction method, a "not accept" vote. Issuing all votes concludes the prepare phase.

RTR provides an optional message to the server, OnPrepareTransaction, to expose the prepare phase. This message indicates to the server that it is time to prepare any updates for a later commit or rollback operation.

4.2.10.2. Commit Phase

In the second phase of the commit, RTR collects votes from all the servers. If they are all votes to accept, then RTR tells all participant servers that they can now commit the transaction to the database. RTR also informs the client that the transaction has completed successfully. If any server rejects the transaction, all participants are informed of this and the database is left unchanged. Your application can use the level of participation that makes the most sense for your business and operations needs.

4.2.10.3. Explicit Accept, Explicit Prepare

To request an explicit accept and explicit prepare of transactions, the server receives both prepare and accept messages. The server then explicitly accepts or rejects a transaction when it receives the prepare message. The transaction sequence for an explicit prepare and explicit accept is as follows:

Client	**RTR**	**Server**
SendApplicationMessage	→ rtr_mt_msg1	→ Receive
AcceptTransaction	→ rtr_mt_prepare	→ Receive
		← AcceptTransaction
Receive	← rtr_mt_accepted	→ Receive

With explicit transaction handling, the following steps occur:

- The server application waits for a message from the client application.

- The server application receives the `rtr_mt_prepare` request message from RTR.
- The server application issues the accept or reject.

A participant can reject the transaction up to the time RTR has sent the `rtr_mt_prepare` message to the server using the AcceptTransaction method and the AcceptTransaction method is executed. Once the client application has used the AcceptTransaction method, the result cannot be changed.

4.2.10.4. Implicit Prepare, Explicit Accept

The sequence for an implicit prepare and explicit accept is as follows:

Client	RTR	Server
SendApplicationMessage	→ rtr_mt_msg1	→ Receive
AcceptTransaction	→	← AcceptTransaction
Receive	← rtr_mt_accepted	→ Receive

In certain database applications, where the database manager does not let an application explicitly prepare the database, transactions can simply be accepted or rejected. For server optimization, the server can signal its acceptance of a transaction with either the SetAcceptTransaction method, or with the client calling the SetAcceptTransaction method. This helps minimize network traffic for transactions by increasing the likelihood that the data message and the RTR accept message will be sent in the same network packet.

4.2.10.5. Server Transaction States

The server transaction states depend on whether the transaction is in prepare or commit phase. Table 4.4 lists server transaction states in the prepare phase.

Table 4.4. Prepare Phase Server States

State	Meaning
RECEIVING	This state represents the server not yet voting on the transaction.
VOTING	The transaction state changes to VOTING when the client has accepted the transaction and RTR has asked the server to vote but the server has not yet answered.

Table 4.5 lists server transaction states in the commit phase.

Table 4.5. Commit Phase Server States

State	Meaning
COMMIT	The server commits the transaction after receiving a message from RTR. If all servers vote to accept (AcceptTransaction), all participants receive a commit message.
ABORT	The server aborts the transaction after receiving a message from RTR. If any server votes to abort

State	Meaning
	(RejectTransaction), all participants receive an abort message.

4.2.10.6. Router Transaction States

The transaction states for the router depend on whether the transaction is in prepare or commit phase. Table 4.6 lists states in the prepare phase.

Table 4.6. Prepare Phase Router States

State	Meaning
SENDING	The router state changes to VREQ except on a failed transaction, in which case it changes to ABORTING.
VREQ	This state represents RTR waiting for the server to vote by issuing AcceptTransaction or RejectTransaction. Once a vote is received, the state changes to either COMMITTING or ABORTING.

4.3. Transaction Recovery

When a transaction fails in progress, RTR provides recovery support using RTR replay technology. RTR, as a distributed transaction manager, communicates with a database resource manager in directing the two-phase commit process. Table 4.7 lists the typical server application transaction sequences for committing a transaction to the database. The sequence depends on which processing model you implement, polling or event driven.

Table 4.7. Typical Server Application Transaction Sequences

Polling Model	Event-Driven Model
1. RTRServerTransactionController:: Receive(rtr_mt_msg1)	1. RTRServerTransactionController:: Receive(rtr_mt_msg1)
2. SQL update	2. RTRServerMessageHandler:: OnInitialize()
3. RTRServerTransactionController:: AcceptTransaction()	3. RTRServerMessageHandler:: OnApplicationMessage()
4. RTRServerTransactionController:: Receive(rtr_mt_accepted)	4. SQL update
5. SQL commit	5. RTRServerTransactionController:: AcceptTransaction()
6. RTRServerTransactionController:: AcknowledgeTransactionOutcome()	6. RTRServerTransactionController:: Receive(rtr_mt_accepted)

Polling Model	Event-Driven Model
	7. RTRServerMessageHandler:: OnAccepted()
	8. SQL commit
	9. RTRServerTransactionController:: AcknowledgeTransactionOutcome()

4.3.1. Recovery Examples

The impact of a crash on application recovery depends on where in the process the crash occurs. RTR handles the recovery, with the assistance of the application.

The typical server application transaction sequence using the event-driven processing model is as follows:

1. Receive(OnInitialize, OnApplicationMessage)
2. SQL update
3. AcceptTransaction
4. Receive(OnAccepted)
5. SQL commit
6. AcknowledgeTransactionOutcome

RTR transaction recovery summarized:

- RTR always replays the transaction with `rtr_mt_msg1` if the crash occurs between steps 1 and 3.
- With transaction manager coordination, RTR does not replay the transaction after step 3. The transaction is completed transparently to the application.
- Without transaction manager coordination, RTR replays the transaction with OnUncertainTransaction if the crash occurs between steps 3 and 6.

4.3.1.1. Recovery: Before Server Accepts

If the failure occurs before the server accepts a transaction, the sequence is as follows:

1. Receive(OnInitialize, OnApplicationMessage) **failure**
2. SQL update
3. AcceptTransaction
4. Receive(OnAccepted)
5. SQL commit
6. Receive

If a crash occurs before the server accepts a transaction (between steps 1 and 3):

- With no database replication: the transaction is aborted and the database is rolled back.
- With a concurrent server: the database is rolled back and the transaction is replayed to another instance of the server.
- With a standby server: the database is rolled back and the transaction is replayed to the standby.
- With a shadow server: the shadow server completes the transaction as a lone member. On recovery, the database is rolled back and the transaction is replayed.

If another server (concurrent or standby) is available, RTR replays the transaction to that other server.

4.3.1.2. Recovery: After Server Accepts

If the failure occurs after a server accepts a transaction, the sequence is as follows:

1. Receive(OnInitialize, OnApplicationMessage)
2. SQL update
3. AcceptTransaction **failure**
4. Receive(OnAccepted) **failure**
5. SQL commit
6. Receive

If a failure occurs after the AcceptTransaction method is executed, but before the SQL Commit, the transaction is replayed. The type of the first message is then `rtr_mt_uncertain` when the server is restarted. Servers should check to see if the transaction has already been executed in a previous presentation. If not, it is safe to re-execute the transaction because the database operation never occurred.

After the failure where the server accepts a transaction, but before the database is committed (between steps 3 and 5) the following occurs:

- With no database replication: the database is rolled back and the transaction is replayed as uncertain. The server should re-execute the transaction.
- With a concurrent server: the database is rolled back and the transaction is replayed as uncertain. The server should re-execute the transaction.
- With a standby server: the database is rolled back and the transaction is replayed as uncertain. The server should re-execute the transaction.
- With a shadow server: the shadow server completes the transaction as a lone member. When the failed server returns, the database is rolled back and transaction is replayed as uncertain. The server should re-execute the transaction.

If a failure occurs after the transaction has been accepted, but before it has been committed, the message is `rtr_mt_uncertain` when the server is restarted. It is safe to re-execute the transaction since the database commit operation never occurred.

4.3.1.3. Recovery: After Database is Committed

If the failure occurs after the database is committed (for example, after the SQL commit but before receipt of a message starting the next transaction), RTR does not know the difference. The sequence is as follows:

- Receive(OnInitialize, OnApplicationMessage)
- SQL update
- AcceptTransaction
- Receive(OnAccepted)
- SQL commit
- Receive

If failure occurs after an SQL commit is made, but before the receipt of a message starting the next transaction (between steps 5 and 6), the sequence is as follows:

- With no database replication: the transaction is replayed as uncertain. The server should ignore the transaction.
- With a concurrent server: the transaction is replayed as uncertain. The server should ignore the transaction.
- With a standby server: the transaction is replayed as uncertain. The server should ignore the transaction.
- With a shadow server: the shadow server completes the transaction as a lone member. On recovery, the transaction is replayed as uncertain. The server should ignore the transaction.

In this case, the transaction should not be re-executed because the database commit operation has already occurred.

4.3.1.4. Recovery: After Beginning a New Transaction

If the failure occurs after executing a Receive method to begin a new transaction, RTR assumes a successful commit (if the Receive occurs after receiving the `rtr_mt_accepted` message) and will forget the transaction. There is no replay following these events. The sequence is as follows:

1. Receive(OnInitialize, OnApplicationMessage)
2. SQL update
3. AcceptTransaction
4. Receive(OnAccepted)
5. SQL commit
6. Receive **failure**

If a crash occurs after a Receive call is made to begin a new transaction (after step 6), the sequence is as follows:

- With no database replication: No replay. Prior transaction was forgotten.

- With a concurrent server: No replay. Prior transaction was forgotten.
- With a standby server: No replay. Prior transaction was forgotten.
- With a shadow server: No replay. Prior transaction was forgotten.

As an application design suggestion, the application can maintain the list of transaction identifiers (TID) on a per-process, per-transaction controller basis to keep the list from growing infinitely.

4.3.2. Exception Transaction Handling

RTR keeps track of how many times a transaction is presented to a server application before it is VOTED. The rule is: three strikes and you're out! After the third strike, RTR rejects the transaction with the reason `RTR_STS_SRVDIED`. The server application has committed the transaction and the client believes that the transaction is committed. The transaction is flagged as an exception and the database is not committed. Such an exception transaction can be manually committed if necessary. This process eliminates the possibility that a single rogue transaction can crash all available copies of a server application at both primary and secondary sites.

Application design can change this behavior. The application can specify the retry count to use when in recovery using the SetRecoveryRetryCount method in the RTRBackEndPartitionProperties class, or the system administrator can set the retry count from the RTR CLI with the SET PARTITION command. If no recovery retry count is specified, RTR retries replay three times. For recovery, retries are infinite. For more information on the SET PARTITION command, refer to the *VSI Reliable Transaction Router System Manager's Manual*; for more information on the SetRecoveryRetryCount method, refer to the *VSI Reliable Transaction Router C++ Foundation Classes* manual.

When a node is down, the operator can select a different backend to use for the server restart. To complete any incomplete transactions, RTR searches the journals of all backend nodes of a facility for any transactions for the key range specified in the server's RegisterPartition call.

4.3.3. Coordinating Transactions

RTR provides two mechanisms for coordinating RTR transactions with database transactions (or database managers): transaction management coordination (XA, DECdtm) and RTR replay technology.

4.3.3.1. Integration of RTR with Resource Managers

When RTR is used with a resource manager (database or DB manager), two transactions are defined: an RTR transaction and that of the resource manager. These must be coordinated to ensure that a transaction committed by the resource manager is also committed by RTR. The same applies to rejected transactions.

4.3.3.2. Distributed Transaction Model

A distributed transaction manager provides a link for RTR to communicate with the resource manager in directing the two-phase commit process. Without the transaction manager, there is no path for RTR to communicate with a resource manager, so RTR must rely on the application for this communication. In general, two-phase commit coordination with the resource manager is not available to application programs.

Without a transaction manager, RTR uses replay technology to handle coordination with a resource manager. With a transaction manager, the transaction manager is the common agent to link the two transactions.

4.3.4. Broadcast Messaging Processes

To use broadcast messaging with the C++ API, client and server applications can overload their event handlers.

To enable communication between two applications of the same type, create a second transaction controller of the other type. Messaging destination names can include wildcards, enabling flexible definition of the subset of recipients for a particular broadcast.

Use the SendApplicationEvent method to broadcast a user-event message. Broadcast types include *user events* and *RTR events*; both are numbered and can be named.

4.3.4.1. User Events

A user broadcast is named or unnamed. An unnamed broadcast performs a match on the user event number. A named broadcast performs a match on both user event number and recipient name, a user-defined string. Named broadcasts provide greater control over who receives a particular broadcast.

Named events specify an event number and a textual recipient name. The name can include wildcards (% and *).

For all unnamed events, specify the *evtnum* field and `RTR_NO_RCPSPC` as the recipient name.

For example, the following code specifies named events for all subscribers:

```
SendApplicationEvent( evUserEventNumber, "*", RTR_NO_MSGFMT)
```

For a broadcast to be received by an application, the recipient name specified by the subscribing application on its RegisterFacility method for clients and RegisterPartition method for servers must match the recipient specifier used by the broadcast sender on the SendApplicationEvent call.

Note

`RTR_NO_RCPSPC` is not the same as "*".

An application receives broadcasts with the Receive method. A message type returned in the RTRData buffers informs the application of the type of broadcast received. For example,

```
Receive(
...pmsg,maxlen,
   pmsgsb);
```

The user event would be in `msgsb.msgtype == rtr_mt_user_event`. User broadcasts can also contain a broadcast message. This message is returned in the message buffer provided by the application. The size of the user's buffer is determined by the *maxlen* field. The number of bytes actually received is returned by RTR in the *msglen* field of the message status block. Table 4.8 summarizes these rules.

Table 4.8. C++ API Named and Unnamed Broadcast Events

Broadcast Type	Match On	Specify
Named	user event number	*evtnum* field
	recipient name	`rcpspc=` *subscriber* (for all subscribers, use `="*"`)

Broadcast Type	Match On	Specify
Unnamed	user event number	*evtnum* field
		RTR_NO_RCPSPC

4.3.4.2. RTR Events

RTR delivers status information to which client and server applications can subscribe. Status information is delivered as messages, where the type of each message is an RTR event.

RTR events are numbered. The base value for RTR events is defined by the symbol RTR_EVTNUM_RTRBASE; its maximum value is defined by the symbol RTR_EVTNUM_RTRMAX. RTR events and event numbers are listed in the Programming with the C++ API chapter of this guide and in the RTR header file `rtrapi.h`.

An application can subscribe to RTR events to receive notification of external events that are of interest to the application. For example, a shadow server may need to know if it is a primary or a secondary server to perform certain work, such as uploading information to a central database, that is done at only one site.

To subscribe to all RTR events, use the range indicators RTR_EVTNUM_RTRBASE and RTR_EVTNUM_RTRMAX.

RTR events are delivered as messages of type `rtr_mt.rtr.event`. Read the message type to determine what RTR has delivered. For example,

```
rtr_status_t Receive( *pRTRData )
```

The event number, *evtnum,* is returned in the RTRData. The following RTR events return key range data back to the client application:

- RTR_EVTNUM_KEYRANGEGAIN
- RTR_EVTNUM_KEYRANGELOSS

In application design, you may wish to consider creating separate facilities for sending broadcasts. By separating broadcast notification from transactional traffic, performance improvements can be substantial. Facilities can further be reconfigured to place the RTR routers strategically to minimize wide-area traffic.

A server application can expect to see a primary or secondary event delivered only in certain transaction states. For more detail, see the state diagrams in Appendix C.

4.3.5. Authentication Using Callout Servers

RTR callout servers enable security checks to be carried out on all requests using a facility. Callout servers can run on backend or router nodes. They receive a copy of every transaction delivered to or passing through the node, and they vote on every transaction. To enable a callout server, use the /CALLOUT qualifier when issuing the *RTR CREATE FACILITY* command. Callout servers are facility based, not key-range or partition based.

An application enables a callout server with the CreateFacility method in the RTRFacilityManager class. For a router callout, the application sets the EnableRouterCallout boolean to true:

```
RTRFacilityManager.CreateFacility(...
```

```
                                    bEnableRouterCallout = true
                                      ...);
```

For a backend callout server, the application sets the EnableBackendCallout boolean to true:

```
RTRFacilityManager.CreateFacility(...
                                    bEnableBackendCallout = true);
```

Chapter 5. Design with the C API

5.1. RTR C Application Programming Interface

RTR provides a C application programming interface (API) that features transaction semantics and intelligent routing in the client/server environment. It provides software fault tolerance using shadow servers, standby servers, and concurrent server processing. It can provide authentication with callout servers. RTR makes single-point failures transparent to the application, guaranteeing that, within the limits of reliability of the basic infrastructure and the physical hardware used, a transaction will arrive at its destination.

The RTR C API contains 16 calls that address four groups of activity:

- Initialization/termination calls
- Messaging calls
- Transactional calls
- Informational calls

The initialization call signals RTR that a client or server application wants to use RTR services and the termination call releases the connection once the requested work is done.

Messaging calls enable client and server applications to send and receive messages and broadcasts.

Transactional calls collect groups of messages as transactions (txn).

Informational calls enable an application to set RTR options or interrogate RTR data structures.

Table 5.1. RTR C API Calls by Category

Initiation/ termination Calls	Messaging Calls	Transactional Calls	Informational Calls	Manipulation
rtr_open_channel	rtr_broadcast_event	rtr_start_tx	rtr_request_info	rtr_set_info
rtr_close_channel	rtr_reply_to_client	rtr_accept_tx	rtr_get_tid (tid is the transaction identifier)	rtr_set_user_ handle
	rtr_send_to_server	rtr_reject_tx	rtr_error_text	rtr_set_wakeup
	rtr_receive_message			
	rtr_ext_broadcast_event			

To execute these calls using the RTR CLI, precede each with the keyword CALL. For example,

```
RTR> CALL RTR_OPEN_CHANNEL
```

Table 5.2 provides additional information on RTR API calls, which are listed in alphabetical order.

Table 5.2. RTR C API Calls

Routine Name	Description	Client and Server	Client Only	Server Only	Returns completion status as a message
rtr_accept_tx()	Votes on a transaction (server).			Yes	Yes
	Commits a transaction (client).		Yes		
rtr_broadcast_event()	Broadcasts an event message.	Yes			
rtr_close_channel()	Closes a previously opened channel.	Yes			
rtr_error_text()	Converts RTR message numbers to message text	Yes			
rtr_ext_broadcast_event()	Broadcasts an event message.	Yes			
rtr_get_tid()	Gets the current transaction ID.	Yes			
rtr_open_channel()	Opens a channel for sending and receiving messages.	Yes			Yes
rtr_receive_message()	Receives the next message (transaction message, event or completion status); returns a message and a message status block.	Yes			
rtr_reject_tx()	Rejects a transaction.	Yes			
rtr_reply_to_client()	Sends a response from a server to a client.			Yes	
rtr_request_info()	Requests information from RTR.	Yes			Yes
rtr_send_to_server()	Sends a message from a client to the server(s).		Yes		
rtr_set_info()	Sets an RTR parameter.	Yes			Yes
rtr_set_user_handle()	Associates a user value with a transaction.	Yes			
rtr_set_wakeup()	Sets a function to be called on message arrival.	Yes			
rtr_start_tx()	Explicitly starts a transaction and specifies its characteristics.		Yes		

5.1.1. Using the rtr.h Header File

The `rtr.h` file included with the product defines all RTR data, status, and message types, including text that can be returned in error messages from an application. You must use it in application compilation.

To support the multi-operating system environment, error codes are processed by RTR using data values in `rtr.h`, and translated into text messages. Status codes are described in the *VSI Reliable Transaction Router C Application Programmer's Reference Manual*.

5.2. RTR Command Line Interface

The command line interface (CLI) to the RTR API enables the programmer to write short RTR applications from the RTR command line. This can be useful for testing short program segments and exploring how RTR works. For an example of a sequence of commands that starts RTR and exchanges messages between a client and a server, see *VSI Reliable Transaction Router Getting Started*.

5.3. Designing an RTR Client/Server Application

The design of an RTR client/server application should capitalize on the RTR multi-layer model, separating client activity in the application from server and database activity. The RTR client software layer passes messages transparently to the router layer, and the router layer sends messages and transactions reliably and transparently, based on message content, to appropriate server processes. Server application software typically interacts with database software to update and query the database, and respond back to the client.

All RTR calls complete synchronously. Subsequent completion events are returned as messages to the application (see Table 5.2).

The client/server environment has both pluses and minuses. Performing processing on the client that does not need to be handled by the server is a plus, as it enables the client to perform tasks that the server need have no knowledge of, and need expend no resources to support. With RTR as the medium for moving transactions from client to server, the application need not be concerned in detail about how the transactions are sent, or what to do in the event of node or site failures. RTR handles all required journaling and recovery without direct intervention by the application. The application needs no code to deal with server and link failures. However, the application must deal with transactions that get aborted, such as business-logic cases (for example, insufficient funds in a bank account) where rules dictate the abort. Table 5.3 lists the types of transaction aborts.

Table 5.3. How RTR Reports Aborted Transactions

Transaction Aborted	Action
Business logic cases. For example, insufficient funds in a bank account.	Application reports error to user. Server or client aborts transactions.
RTR generated abort, `RTR_STS_NODSTFND`. RTR has exhausted all redundancy paths.	Application reports system unavailable, perhaps temporarily.
Timeout abort.	Task dependent. Try again or inform user.
Deadlock aborts.	None. RTR reschedules.

In a client/server environment, application design becomes more complex because the designer must consider what tasks the clients are to perform, and what the servers must do. Typically the client application will capture information entered by the user, while the server interacts with the database to update the database with transactions passed to it by the router.

5.3.1. The RTR Journal

The RTR journal is always in use, recording transaction activity to persistent storage. The journal thus provides the capability of recovery from any single hardware or process failure. When a server application no longer provides service, for example, when it goes off line, goes down, or is taken out of service temporarily, RTR aborts all transactions for that service with `RTR_STS_NODSTFND`. When doing transactional shadowing, the journal at the active site is used to record transactions, for the out-of-service site. Journaling on frontends is required to support nested transactions to record the final state of a transaction.

If transactions do not update the database, specify them as read-only by using the `RTR_F_SEN_READONLY` flag on the `rtr_send_to_server` call. Read-only transactions are not recovered as uncertain transactions. Also, in a shadowed environment, these transactions would not be remembered and would therefore save on journal space.

5.4. Data Content Routing with Partitions or Key Ranges

Client applications use data content routing to route each transactional request to the appropriate server servicing the correct database segment. The key value supplied by the client application in the key fields (as defined in the definition of the partition) is used by RTR to achieve data content or data partition routing.

5.4.1. Partitions or Key Ranges

RTR enables an application to route transactions by partition or key range, rather than sending all transaction requests to a single server application.

When RTR has chosen a specific server for a request within a given transaction, RTR ensures that all requests within that transaction are routed to the same server application channel.

Key ranges, or data partitions, are a major concept in RTR. An application is said to service a partition. Failovers and other availability attributes are applied to all applications, which service a given partition. All server applications able to service a specific data partition on a given node are called concurrents of one another. Concurrent servers may be either multiple channels within a given process, or separate processes.

Partitions can be given names. This lets the system manager manipulate the attributes for a partition at runtime. For example, the operator can temporarily suspend the presentation of online transactions to the partition. This provides time to initiate a database backup operation.

It is possible to give a different name to the same partition on each backend. However, this is not recommended, because RTR Explorer displays the partition name when grouping backends by partition. If a partition is not given the same name on all backends, RTR Explorer will display a string representation of the key range instead of one of the names.

Partitions, in RTR, are the essence of routing. The server application declares its intention to service a certain partition or key range by associating itself with a name, or explicitly defining the key format and range of values.

A partition has many attributes, some of which are specified by the programmer or operator. Some key attributes include:

- Name
- Facility (domain or name space)
- Key format (location in the message where the key value can be found, size, and type)
- Range of values
- Redundancy attributes (concurrency, either/or, parallel processing)
- Resource manager bindings (XA, DTC, DDTM)
- Failover preferences
- Node priority

To plan for future expansion, consider using compound keys rather than single field keys. For example, using a bank example, with a bank that has multiple branches, an application that routes data to each bank can use a BankID key field or partition. Over time, the application may also need to further subdivide transactions not only by bank but also by customer ID. If the application is initially written with a compound key providing both a BankID and a CustomerID key, it can be simpler to make such a change in future.

5.4.1.1. Multithreading

An application can be multithreaded. There are several ways to use multithreading in the application architecture. Check the *VSI Reliable Transaction Router Release Notes* and SPD for the current extent of support for multithreaded programming for a given platform.

The two common ways of using multithreading are:

- Dedicating a thread for every RTR channel. This way, any channel can decide to wait to receive from RTR without affecting the ability of the other channels to continue processing.
- Having a dedicated channel for receiving messages from RTR. Other channels send messages to RTR, or perform other activity such as reading from a different device.

One word of caution for application developers who intend to deploy on OpenVMS: AST programming and threading do not mix well, and may cause intermittent deadlocks. It is therefore prudent not to use ASTs when using threads on OpenVMS with RTR.

5.4.1.2. RTR Call Sequence

For a client, an application typically uses the following calls in this order:

C API Client Call	**Instantiation**
rtr_open_channel() rtr_receive_message()	one per channel
rtr_start_tx() rtr_send_to_server() [one or more] rtr_accept_tx() rtr_receive_message()[zero or more receive messages for the expected number of replies]	each per transaction

For a server, an application typically uses the following calls in this order:

C API Server Call	Instantiation
rtr_open_channel() rtr_receive_message()	one per channel
rtr_receive_message() [one or more] rtr_reply_to_client() [zero or more] rtr_accept_tx() rtr_receive_message()	each per transaction

The `rtr_receive_message` call returns a message and a message status block (`MsgStatusBlock`). For example,

```
... status = rtr_receive_message(&Channel,
                                                RTR_NO_FLAGS,
                                                RTR_ANYCHAN,
                                                MsgBuffer,
                                                DataLen,
                                                RTR_NO_TIMOUTMS,
                                                &MsgStatusBlock);
```

The message status block contains the message type, the user handle, if any, the message length, the TID, and the event number, if the message type is `rtr_mt_rtr_event` or `rtr_mt_user_event`. For more information on the message status block, refer to the descriptions of `rtr_receive_message` and `rtr_set_user_handle` in the *VSI Reliable Transaction Router C Application Programmer's Reference Manual*.

5.4.1.3. RTR Message Types

RTR calls and responses to them contain RTR message types (mt) such as `rtr_mt_reply` or `rtr_mt_rejected`. The four groups of message types, listed in Table 5.4, are:

- Transactional
- Transactional status
- Event-related
- Informational

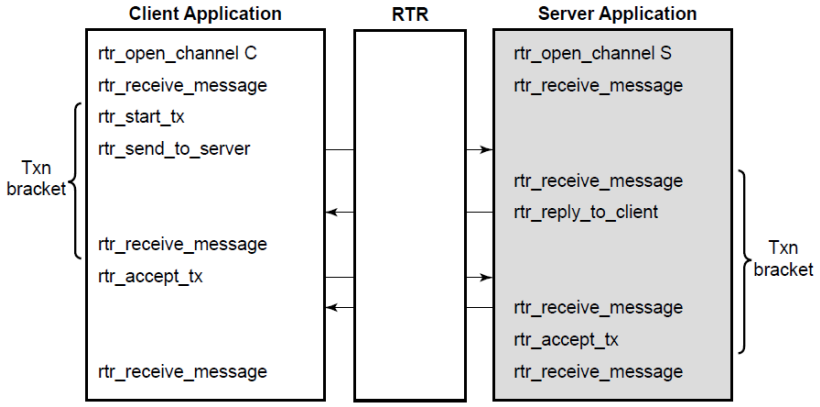
Table 5.4. RTR Message Types

Transactional	Status	Event-related	Informational
rtr_mt_msg1	rtr_mt_accepted	rtr_mt_user_event	rtr_mt_opened
rtr_mt_msg1_uncertain	rtr_mt_rejected	rtr_mt_rtr_event	rtr_mt_closed
rtr_mt_msgn	rtr_mt_prepare		rtr_mt_request_info
rtr_mt_reply	rtr_mt_prepared		rtr_mt_rettosend

Applications should include code for all RTR expected return message types. Message types are returned to the application in the message status block. For more detail on message types see the *VSI Reliable*

Transaction Router C Application Programmer's Reference Manual. Figure 5.1 shows the RTR C API calls you use to achieve transactional messaging in your application.

Figure 5.1. RTR C API Calls for Transactional Messaging



5.4.1.4. Message Format Definitions

To work in a mixed-operating system environment, an application can specify a message format definition on the following calls:

- `rtr_send_to_server`
- `rtr_reply_to_client`
- `rtr_broadcast_event`
- `rtr_ext_broadcast_event`

RTR performs data marshaling, evaluating and converting data appropriately as directed by the message format descriptions provided in the application.

The following example shows an RTR application using a message format declaration; first the RTR call specifying that `TXN_MSG_FMT` contains the actual format declaration, then the definition used by the call.

RTR application call:

```
status = rtr_send_to_server(p_channel,
                            RTR_NOFLAGS,
                            &txn_msg,
                            msg_size,
                            TXN_MSG_FMT );
```

Data definition: `#define TXN_MSG_FMT "%1C%UL%7UL%31C"`

This data structure accommodates an 8-bit signed character field (C) for the key, a 32-bit unsigned field (UL) for the server number, a 224-bit (7x32) field (7UL) for the transaction ID, and a 31-bit byte (248-bit) field (31C) for character text. For details of defining message format for a mixed-endian environment, refer to the *VSI Reliable Transaction Router C Application Programmer's Reference Manual*.

5.5. Using the XA Protocol

You use the XA protocol supported by RTR to communicate with an X/Open Distributed Transaction Processing (DTP) conformant resource manager. This eliminates the need for an application program

to process `rtr_mt_uncertain` messages. For examples of system setup, refer to the *VSI Reliable Transaction Router System Manager's Manual*, Appendix C.

5.5.1. XA Oracle Example

Note

In this example, error checking has been omitted for clarity.

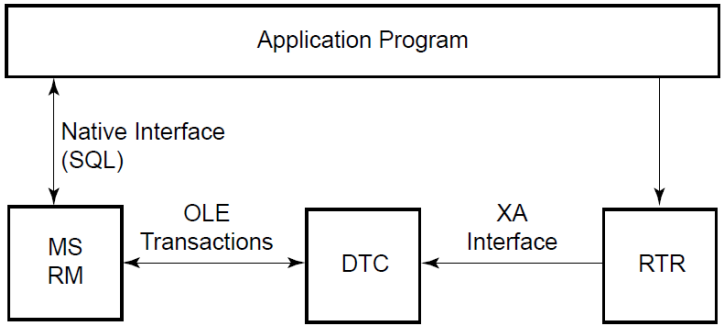
```
int main( int argc, char argv[] )
 {
      server_key[0].ks_type = rtr_keyseg_unsigned;
      server_key[0].ks_length = sizeof(rtr_uns_8_t);
      server_key[0].ks_offset = 0;
      server_key[0].ks_lo_bound = &low;
      server_key[0].ks_hi_bound = &high;
      server_key[1].ks_type = rtr_keyseg_rmname;          /* RM name */
      server_key[1].ks_length = 0;                        /* not applicable */
      server_key[1].ks_offset = 0;
      server_key[1].ks_lo_bound = rm_name;
      server_key[1].ks_hi_bound = xa_open_string;

      (flag = RTR_F_OPE_SERVER |
       RTR_F_OPE_NOSTANDBY |
       RTR_F_OPE_XA_MANAGED |                       /* XA flag */
       RTR_F_OPE_EXPLICIT_PREPARE |
       RTR_F_OPE_EXPLICIT_ACCEPT;
rtr_open_channel(&server_channel, flag, fac_name, NULL, RTR_NO_PEVTNUM,
                 NULL, 2, server_key);
while (rtr_receive_message(&server_channel, RTR_NO_FLAGS, RTR_ANYCHAN,
    &receive_msg,sizeof(receive_msg), RTR_NO_TIMOUTMS, &msgsb)
           == RTR_STS_OK)
    { ...
     msg = receive_msg.receive_data_msg;
           (switch(msgsb.msgtype)
         {
      case rtr_mt_msg1:
      case rtr_mt_msgn:
        switch(msg.txn_type)
          {
      case ...
         EXEC SQL ...
          }
              ... rtr_reply_to_client(server_channel, RTR_NO_FLAGS, &reply_msg,
  sizeof(reply_msg), RTR_NO_MSGFMT);
              ...
      case rtr_mt_prepare:
              ...
 rtr_accept_tx(s_chan,RTR_NO_FLAGS,RTR_NO_REASON);
              ...
      case rtr_mt_accepted:
       /* EXEC SQL COMMIT;   Comment out SQL Commits */
       break;
      case rtr_mt_rejected:
       /* EXEC SQL ROLLBACK;  Comment out SQL rollbacks */
       break;
       /*       case rtr_mt_msg1_uncertain:
              ...
       */
        }
      ...
     }
         EXEC SQL COMMIT WORK RELEASE;
```

```
... }
```

5.5.2. Using XA with MS DTC

The XA software architecture of RTR provides interoperability with the Distributed Transaction Controller of Microsoft, MS DTC. Thus RTR users can develop application programs that update MS SQL databases, MS MQ, or other Microsoft resource managers under the control of a true distributed transaction. RTR as a distributed transaction manager communicates directly with MS DTC to manage a transaction or perform recovery using the XA protocol. For each standard XA call received from RTR, MS DTC translates it into a corresponding OLE transaction call that SQL Server or MS MQ expects to perform database updates. This is shown in Figure 5.2.

Figure 5.2. MS DTC and RTR



For example, using XA and DTC (VSI Tru64 UNIX and Microsoft Windows only) eliminates the need to process uncertain messages `rtr_mt_msg1_uncertain`). To use the XA protocol with RTR, you must:

- Use the RTR REGISTER RM (register resource manager) command to register the resource manager, specifying the appropriate libraries. For Oracle 7RDBMS, the V$XATRANS$ view must exist on the database.
- Create a facility with the referenced resource manager by executing the CREATE FACILITY/RM= *xxx* command, where *xxx* is the same as that used in the previous REGISTER RM command. For more details on the REGISTER RM command and other commands for working with other database managers, refer to the *VSI Reliable Transaction Router System Manager's Manual*.
- In the `rtr_open_channel` call in the server application, specify which resource manager the application is using. The application must submit a resource manager instance name with [OPEN_STRING] to the `rtr_open_channel` call before accessing the desired database.
- Remove unnecessary SQL calls from existing server code such as commit or rollback, to avoid vendor-specific warnings.

Both the resource manager instance name and the database (RM) name in [OPEN- STRING] must be identical to that in the previously executed REGISTER RM command. The information is stored in the RTR key segment structure, and the `RTR_F_OPE_XA_MANAGED` flag associates the channel with the XA interface.

Only one transaction at a time is processed on an RTR channel; thus a server process or thread of control can only open one channel to handle a single XA request. Better throughput can be achieved by using a multithreaded application.

For example, the following code from a sample server application shows use of the RM key, the XA flag, and commenting out commits and rollbacks for the Oracle and DTC environments.

5.5.3. XA DTC Example

The following XA/DTC server application example is for a Windows environment only.

Note

In this example, error checking has been omitted for clarity.

```
int main( int argc, char argv[] )
{
     server_key[0].ks_type = rtr_keyseg_unsigned;
     server_key[0].ks_length = sizeof(rtr_uns_8_t);
     server_key[0].ks_offset = 0;
     server_key[0].ks_lo_bound = &low;
     server_key[0].ks_hi_bound = &high;
     server_key[1].ks_type = rtr_keyseg_rmname;          /* RM name */
     server_key[1].ks_length = 0;                        /* not applicable */
     server_key[1].ks_offset = 0;
     server_key[1].ks_lo_bound = rm_name;
     server_key[1].ks_hi_bound = xa_open_string;

     (flag = RTR_F_OPE_SERVER |
      RTR_F_OPE_NOSTANDBY |
      RTR_F_OPE_XA_MANAGED |                        /* XA flag */
      RTR_F_OPE_EXPLICIT_PREPARE |
      RTR_F_OPE_EXPLICIT_ACCEPT;
rtr_open_channel(&server_channel, flag, fac_name, NULL, RTR_NO_PEVTNUM,
                 NULL, 2, server_key);
while (rtr_receive_message(&server_channel, RTR_NO_FLAGS, RTR_ANYCHAN,
    &receive_msg,sizeof(receive_msg), RTR_NO_TIMOUTMS, &msgsb)
         == RTR_STS_OK)
   { ...
    msg = receive_msg.receive_data_msg;

    (switch(msgsb.msgtype)
       {
     case rtr_mt_msg1:
     case rtr_mt_msgn:
       switch(msg.txn_type)
         {
     case ...
        EXEC SQL ...
         }
             ...
rtr_reply_to_client(server_channel, RTR_NO_FLAGS, &reply_msg,
sizeof(reply_msg), RTR_NO_MSGFMT);
             ...
     case rtr_mt_prepare:
             ...
rtr_accept_tx(s_chan,RTR_NO_FLAGS,RTR_NO_REASON);
             ...
     case rtr_mt_accepted:
      /* EXEC SQL COMMIT;   Comment out SQL Commits */
      break;
     case rtr_mt_rejected:
      /* EXEC SQL ROLLBACK;  Comment out SQL rollbacks */
      break;
      /*
     case rtr_mt_msg1_uncertain:
             ...
      */
       }      ...
    }
```

```
        EXEC SQL COMMIT WORK RELEASE;
                ...
}
```

5.6. Using DECdtm

You can use the DECdtm protocol to communicate with OpenVMS Rdb. This provides a two-phase commit capability. For additional information on using this protocol, refer to the OpenVMS documentation, for example, Managing DECdtm Services in the *VSI OpenVMS System Manager's Manual*, the *VSI OpenVMS System Services Reference Manual*, the *VSI OpenVMS Programming Concepts Manual* and the *Oracle Rdb Guide to Distributed Transactions* available from Oracle.

5.7. RTR Transaction Processing

To pass transactions from client to server, RTR (with the C API) uses channels as identifiers. Each application communicates with RTR on a particular channel. In a multithreaded application, when multiple transactions are outstanding, the application uses the channel to inform RTR which transaction a command is for.

With RTR, the client or server application can:

- Open a channel
- Send a transaction to a channel
- Bind a transaction to a channel
- Receive a transaction on a channel
- Specify a receive programming style (blocked, polled, or signaled)
- Close a channel

To open a channel, the application uses the `rtr_open_channel` call. This opens a channel for communication with a client or server application on a specific facility. Each application process can open up to 255 channels.

For example, the `rtr_open_channel` call in this client application opens a single channel for the facility called DESIGN:

```
status = rtr_open_channel(&Channel,
         RTR_F_OPE_CLIENT, [1]
         DESIGN,            /* Facility name */ [2]
         client_name,
         rtrEvents,
         NULL,              /* Access key / [3]
         RTR_NO_NUMSEG,
         RTR_NO_PKEYSEG     /* Key range */ [4]
         );
```

The application uses parameters on the `rtr_open_channel` call to define the application environment. Typically, the application defines the:

- **[1]** Role of the application (client or server)

- **[2]** Name of the application facility
- **[3]** Facility access key, a password; in this case no password is used
- **[4]** Key range or key segment for data content routing

For a server application, the `rtr_open_channel` call additionally supplies the number of key segments, `numseg`, and the partition name, in `pkeyseg`.

The syntax of the `rtr_open_channel` call is as follows:

```
    status = rtr_open_channel (pchannel,flags,facnam,rcpnam,
     pevtnum,access,numseg,pkeyseg)
```

You can set up a variable section in your client program to define the required parameters and then set up your `rtr_open_channel` call to pass those parameters. For example, the variables definition would contain code similar to the following:

```
/*
** ---------------- Variables -------------------
*/
rtr_status_t Status;
rtr_channel_t Channel;
rtr_ope_flag_t Flags  = RTR_F_OPE_CLIENT;
rtr_facnam_t        Facility  = "DESIGN";
rtr_rcpnam_t        Recipient  = RTR_NO_RCPNAM;
rtr_access_t        Access  = RTR_NO_ACCESS;
```

The `rtr_open_channel` call would contain:

```
status = rtr_open_channel(&Channel,
                          Flags,
                          Facility,
                          Recipient,
                          Evtnum,
                          Access,
                          RTR_NO_NUMSEG,
                          RTR_NO_PKEYSEG);
if (Status != RTR_STS_OK)
/*
{ Provide for error return */}
```

You will find more complete samples of client and server code in the appendix of this document and on the RTR software kit in the Examples directory.

5.7.1. Channel Identifier

To specify the location to return the channel identifier, use the channel argument in the `rtr_open_channel` call. For example,

```
rtr_channel_t channel;
```
or
```
rtr_channel_t *p_channel = &channel;
```

This parameter points to a valid channel identifier when the application receives an `rtr_mt_opened` message.

5.7.2. Flags Parameter

To define the application role type (client or server), use the *flags* parameter. For example,

```
rtr_ope_flag_t
flags = RTR_F_OPE_CLIENT;
```

or

```
 flags = RTR_F_OPE_SERVER;
```

5.7.3. Facility Name

The facility name is a required string supplied by the application. It identifies the RTR facility used by the application. The default facility name for the RTR CLI only is `RTR$DEFAULT_FACILITY`; there is no default facility name for an RTR application. You must supply one.

To define the facility name, use the *facnam* parameter. For example,

```
rtr_facnam_t
facnam = "DESIGN";
```

5.7.4. Recipient Name

To specify a recipient name, use the *rcpnam* parameter, which is case sensitive. For example,

```
  rtr_rcpnam_t rcpnam  = "* Rogers";
```

5.7.5. Event Number

To specify user event numbers, use the *evtnum* parameter. For example,

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

There are both RTR events and user events. For additional information on employing events, see the Broadcast Messaging Processes section of this chapter, and the section on RTR Events in the *VSI Reliable Transaction Router C Application Programmer's Reference Manual*.

5.7.6. Access Key

You can use the facility access key to restrict client or server access to a facility. The key acts as a password to restrict access to the specific facility for which it is declared.

To define the facility access key, use the *access* parameter. For example,

```
  rtr_access_t
   access = "amalasuntha";
```

The facility access key is a string supplied by the application. The first server channel in an RTR facility defines the access key; all subsequent server and client open channel requests must specify the same access value. To use no access key, use `RTR_NO_ACCESS` or `NULL` for the *access* argument.

You can also use this feature for version control. By changing the access code whenever an incompatible protocol change is made in the application message format, client applications are prevented from processing transactions with the server applications.

5.7.7. Key Segments

To specify the number of key segments defined for a server application, use the *numseg* parameter. For example,

```
rtr_numseg_t numseg = 2;
```

5.7.8. Partition or Key Range

To specify the key range for a partition to do data-content routing, the server application defines the routing key when it opens a channel on a facility with the `rtr_open_channel` call. All servers in a facility must specify the same offset, length, and data type for the key segments in the `rtr_open_channel` call; only high and low bounds (`*ks_lo_bound, *ks_hi_bound`) can be unique to each server key segment. By application convention, the client places key data in the message at the offset, length, and data type defined by the server.

5.7.9. Channel-Open Operation

The channel-open operation completes asynchronously. Call completion status is returned in a subsequent message. RTR sends a message to the application indicating successful or unsuccessful completion; the application receives the status message using an `rtr_receive_message` call. If status is `rtr_mt_opened`, the open operation is successful. If status is `rtr_mt_closed`, the open operation is unsuccessful, and the application must examine the failure and respond accordingly. The channel is closed.

5.7.10. Determining Message Status

Data returned in the user buffer with `rtr_mt_opened` and `rtr_mt_closed` include both the status and a reason. For example,

```
case rtr_mt_opened:
  printf(" Channel %d opened\n", channel);
  status = RTR_STS_OK;
  break;
case rtr_mt_closed:
  p_status_data = (rtr_status_data_t *)txn_msg;
  printf(" cannot open channel because %s\n",
   rtr_error_text(p_status_data->status));
  exit(-1);
```

Use the call `rtr_error_text` to find the meaning of returned status. A client channel will receive no message at all if a facility is configured but no server is available. Once a server becomes available, RTR sends the `rtr_mt_opened` message.

5.7.11. Closing a Channel

To close a channel, the application uses the `rtr_close_channel` call. A channel can be closed at any time after it has been opened. Once closed, no further operations can be performed on a channel, and no further messages for the channel are received.

5.7.12. Receiving on a Channel

To receive on a channel and obtain status information from RTR, use the `rtr_receive_message` call. To receive on any open channel, use the `RTR_ANYCHAN` value for the *p_rcvchan* parameter in the `rtr_receive_message` call. To receive from a list of channels, use the *p_rcvchan* parameter as a pointer to a list of channels, ending the list with `RTR_CHAN_ENDLIST`. An application can receive on one or more opened channels. RTR returns the channel identifier. A pointer to a channel is supplied on the `rtr_open_channel` call, and RTR returns the channel identification (ID) by filling in the contents of that pointer.

5.7.13. User Handles

To simplify matching an RTR channel ID with an application thread, an application can associate a user handle with a channel. The handle is returned in the message status block with the `rtr_receive_message` call. The application can use the message status block (`MsgStatusBlock`) to identify the message type of a received message. For example,

```
{
rtr_receive_message (&channel, RTR_NO_FLAGS, RTR_ANYCHAN, txn_msg, maxlen,
RTR_NO_TIMOUTMS,
MsgStatusBlock);
} . . .
   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;
```

RTR delivers both RTR and application messages when the `rtr_receive_message` call completes. The application can use the message type indicator in the message status block to determine relevant buffer format. For further details on using message types and interpreting the contents of the user buffer, refer to the *VSI Reliable Transaction Router C Application Programmer's Reference Manual*.

5.8. Message Reception Styles

An application can specify one of three reception styles for the `rtr_receive_message` call. These are:

- Blocked
- Polled
- Signaled

5.8.1. Blocking Receive

An application can use a blocking receive to wait until a message arrives. To use a blocking receive, include `RTR_NO_TIMOUTMS` in the `rtr_receive_message` call. The call completes only when a message is available on one of the specified channels. For example,

```
rtr_receive_message (&channel, RTR_NO_FLAGS, RTR_ANYCHAN,
```

```
  MsgBuffer, DataLen, RTR_NO_TIMOUTMS, &MsgStatusBlock);
```

5.8.2. Polled Receive

An application can use a polled receive to poll RTR with a specified timeout. To use a polled receive, the application can set a value in milliseconds on the *timoutms* parameter. The timeout can be:

- Immediate: *timoutms* = 0 (Do not wait for messages; RTR notifies the application if there are any.)
- Infinite: `RTR_NO_TIMOUTMS`
- A user-specified value. Current clock granularity is 1 second (1000 milliseconds).

The call completes after the specified timeout or when a message is available on one of the specified channels.

For example, the following declaration sets polling at 1 second (1000 milliseconds).

```
rtr_receive_message(&channel, RTR_NO_FLAGS, RTR_ANYCHAN, MsgBuffer,
 DataLen,
1000, &MsgStatusBlock);
```

Note

The `rtr_receive_message` timeout is not the same as a transaction timeout.

5.8.3. Signaled Receive

An application can use a signaled receive to be alerted by RTR when a message is received. The application establishes a signal handler using the `rtr_set_wakeup` call, informing RTR where to call it back when the message is ready.

To use a signaled receive, the application uses the `rtr_set_wakeup` call and provides the address of a routine to be called by RTR when a message is available. When the wakeup routine is called, the application can use the `rtr_receive_message` call to get the message. For example,

```
rtr_status_t
rtr_set_wakeup(
  procedure  )
void
wakeup_handler(void){
 rtr_receive_message();
}

main(){
 rtr_set_wakeup(wakeup_handler);
 sleep();
}
```

Note

To disable wakeup, call `rtr_set_wakeup` with a null routine address.

When using `rtr_set_wakeup` in a multithreaded application, be careful not to call any non-reentrant functions or tie up system resources unnecessarily inside the callback routine.

The rtr_open_channel parameters are further described in the *VSI Reliable Transaction Router C Application Programmer's Reference Manual* .

5.9. Starting a Transaction

There are two ways to start a transaction:

- Explicitly, using rtr_start_tx
- Implicitly, using rtr_send_to_server

5.9.1. Using the rtr_start_tx Call

Use the rtr_start_tx call when the application must set a client-side transaction timeout to ensure that both client and server do not wait too long for a message. When a transaction is started with rtr_send_to_server, no timeout is specified.

For example,

```
rtr_start_tx(&Channel,
        RTR_NO_FLAGS,
        RTR_NO_TIMOUTMS,
        RTR_NO_JOINCHAN); //or NULL
```

5.9.2. Using the rtr_send_to_server Call

The rtr_send_to_server call sends a message as part of a transaction from a client. If there is no transaction currently active on the channel, a new one is started. The transaction accept can be bundled with the last message. A client has two options for message delivery after a failure:

- Return to sender

  Use the RTR_F_SEN_RETURN_TO_SENDER flag to tell RTR to return the message with a message type of rtr_mt_rettosend if delivery fails. This lets a client determine which message failed in a multiple message stream.

- Accept transaction

  Use the RTR_F_SEN_EXPENDABLE flag to tell RTR not to reject the transaction associated with the message if the message cannot be delivered. This lets other non-expendable messages be delivered without creating a dependency on the flagged message. RTR does not abort the transaction if delivery fails. To specify a read-only server operation for which neither shadowing nor journaling are used, use the RTR_F_SEN_READONLY flag.

5.9.3. Using the rtr_reply_to_client Call

The rtr_reply_to_client call sends a reply message from a server to the client. The reply message is part of the transaction initiated by the client. For example,

```
status = rtr_reply_to_client (&Channel,
                              RTR_NO_FLAGS,
                              MsgBuffer,
                              DataLen,
```

```
                                RTR_NO_MSGFMT);
```

The reply message format can be of any form as designed by the application. For example,

```
struct acct_inq_msg_t {
                             char reply_text[80];
                            } acct_reply_msg;
```

5.10. Identifying a Transaction

When an application receives a message with the `rtr_receive_message` call, the message status block (`MsgStatusBlock`) contains the transaction identifier. For example,

```
status = rtr_receive_message (&Channel,
                              RTR_NO_FLAGS,
                              RTR_ANYCHAN,
                              MsgBuffer,
                              DataLen,
                              RTR_NO_TIMOUTMS,
                              &MsgStatusBlock);
```

The pointer `&MsgStatusBlock` points to the message status block that describes the received message. For example,

```
 typedef struct {rtr_msg_type_t         msgtype;
                 rtr_usrhdl_t           usrhdl;
                 rtr_msglen_t           msglen;
                 rtr_tid_t              tid;
/*If a transactional message, the transaction ID or tid, msgsb.tid */
                 rtr_evtnum_t           evtnum;
                } rtr_msgsb_t;
```

Use the `rtr_get_tid` call to obtain the RTR transaction identifier for the current transaction. The TID (transaction identifier) is a unique number generated by RTR for each transaction. The application can use the TID if the client needs to know the TID to take some action before receiving a response.

Use the `rtr_set_user_handle` call to set a user handle on a transaction instead of using a channel. A client application with multiple transactions outstanding can match a reply or completion status with the appropriate transaction by establishing a new user handle each time a transaction is started.

5.11. Committing a Transaction

A server application ends a transaction by accepting or rejecting it. A transaction is accepted explicitly with the `rtr_accept_tx` call, and rejected explicitly with the `rtr_reject_tx` call. RTR can reject a transaction at any time once the transaction is started, but before it is committed. If RTR cannot deliver a transaction to its destination, it rejects the transaction explicitly and delivers the reject completion status to all participants.

A transaction participant can specify a reason for an accept or reject on the `rtr_accept_tx` and `rtr_reject_tx` call. If more than one transaction participant specifies a reason, RTR uses the OR operator to combine the reason values together. For example, with two servers A and B, each providing a reason code of 1 and 2, respectively, the client receives the result of the OR operation, reason code 3, in its message buffer.

```
Server A                        Server B
 rtr_reason_t                   rtr_reason_t
```

```
  reason = 1 ;                reason=2 ;
  rtr_reject_tx (             rtr_reject_tx (
  channel,                    channel,
  flags,                      flags,
  reason );                   reason );
typedef struct {
  rtr_status_t status;
  rtr_reason_t reason;
} rtr_status_data_t;
```

The client receives the results of the OR operation in its message buffer:

```
rtr_status_data_t
    msgbuf;
    msgbuf.reason = 3;
```

A transaction is done once a client or server application receives a completion message, either an `rtr_mt_closed, rtr_mt_accepted`, or `rtr_mt_rejected` message from RTR. An application no longer receives messages related to a transaction after receiving a completion message or if the application calls `rtr_reject_tx`. A client or server can also specify `RTR_F_ACC_FORGET` on the `rtr_accept_tx` call to signal its acceptance and end its involvement in a transaction early. RTR returns no more messages (including completion messages) associated with the transaction; any such messages received will be returned to the caller.

When issuing the `rtr_accept_tx` call with `RTR_NO_FLAGS` on the call, the caller expresses its request for successful completion of the transaction, and may give an accept reason that is passed on to all participants in the transaction. The accept is final; the caller cannot reject the transaction later. The caller cannot send any more messages for this transaction.

A client can accept a transaction in one of two ways: with the `rtr_accept_tx` call or by using the `RTR_F_SEN_ACCEPT` flag on the `rtr_send_to_server` call.

When the client sets `RTR_F_SEN_ACCEPT` on the `rtr_send_to_server` call, this removes the need to issue an `rtr_accept_tx` call and can help optimization of client traffic. Merging the data and accept messages in one call puts them in a single network packet. This can make better use of network resources and improve throughput.

The `rtr_reject_tx` call rejects a transaction. Any participant in a transaction can call `rtr_reject_tx`. The reject is final; the caller cannot accept the transaction later. The caller can specify a reject reason that is passed to all accepting participants of the transaction. Once the transaction has been rejected, the caller receives no more messages for this transaction.

The server can set the retry flag `RTR_F_REJ_RETRY` to have RTR redeliver the transaction beginning with msg1 without aborting the transaction for other participants. Issuing an `rtr_reject_tx` call with this flag can let another transaction proceed if locks held by this transaction cause a database deadlock.

5.12. Uncertain Transactions

If there is a crash *before* the `rtr_accept_tx` statement is executed, on recovery, the transaction is replayed as `rtr_mt_msg1` because the database will have rolled back the prior transaction instance. However, if there is a crash *after* the `rtr_accept_tx` statement is executed, on recovery, the transaction is replayed as `rtr_mt_msg1_uncertain` because RTR does not know the status of the prior transaction instance. Your application must understand the implications of such failures and deal with them appropriately.

5.13. Administering Transaction Timeouts

RTR provides a relatively simple way to administer a transaction timeout in the server. Use of timeout values on the `rtr_receive_message` function lets a server application specify how long it is prepared to wait for the next message. (Of course, the server should be prepared to wait forever to get a new transaction or for the result of an already-voted transaction.)

One way to achieve this would be to have a channel-specific global variable, say, called `SERVER_INACTIVITY_TIMEOUT`, which is set to the desired value (in milliseconds-that is, use a value of 5000 to set a 5 second timeout). Note that this timeout value should be used *after* receiving the first message of the transaction. The value should be reset to `RTR_NO_TIMOUTMS` after receiving the `rtr_mt_prepare` message. Whenever the `rtr_receive_message` completes with a `RTR_STS_TIMOUT`, the server calls the `rtr_reject_tx` function on that channel to abort the partially- processed transaction. This would prevent transactions from occupying the server process beyond a reasonable time.

5.13.1. Two-Phase Commit

The two-phase commit process includes prepare and commit phases. A transaction is tentatively accepted or rejected during the prepare phase.

5.13.2. Prepare Phase

To initiate the prepare phase, the server application specifies the `RTR_F_OPE_EXPLICIT_PREPARE` flag when opening the channel, and can use the message `rtr_mt_prepare` to check commit status. The message indicates to the server application that it is time to prepare any updates for a later commit or rollback operation. RTR lets the server application explicitly accept a transaction using the `RTR_F_OPE_EXPLICIT_ACCEPT` flag on the `rtr_open_channel` call. Alternatively, RTR implicitly accepts the transaction after receiving the `rtr_mt_accepted` message when the server issues its next `rtr_receive_message` call.

The commit process is initiated by the client application when it issues a call to RTR indicating that the client "accepts" the transaction. This does not mean that the transaction is fully accepted, only that the client is prepared to accept it. RTR then asks the server applications participating in the transaction if they are prepared to accept the transaction. A server application that is prepared to accept the transaction votes its intention by issuing the `rtr_accept_tx` call, an "accept" vote. A server application that is not prepared to accept the transaction issues the `rtr_reject_tx` call, a "not accept" vote. Issuing all votes concludes the prepare phase.

5.13.3. Commit Phase

When RTR has collected all votes from all participating server applications, it determines if the transaction is to be committed. If all collected votes are "accept," the transaction is committed; RTR informs all participating channels. If any vote is "not accept," the transaction is not committed. A server application can expose the prepare phase of two-phase commit by using the `rtr_mt_prepare` message type with the `RTR_F_OPE_EXPLICIT_PREPARE` flag. If the application's `rtr_open_channel` call sets neither the `RTR_F_OPE_EXPLICIT_ACCEPT` nor `RTR_F_OPE_EXPLICIT_PREPARE` flag, both prepare and accept processing are implicit.

The server application can participate in the two-phase commit process fully, somewhat, a little, or not at all. To participate fully, the server does an explicit prepare and an explicit accept of the transaction. To participate somewhat, the server does an explicit prepare and an implicit accept of the transaction.

To participate a little, the server does an explicit accept of the transaction. To participate not at all, the server does an implicit accept of the transaction. Table 5.5 summarizes the level of server participation:

Table 5.5. Server Participation

Commit Phase	Full	Somewhat	Little	Not at all
Explicit prepare	yes	yes		
Explicit accept	yes		yes	
Implicit accept		yes		yes

Your application can use the level of participation that makes the most sense for your business and operations needs.

5.13.4. Explicit Accept, Explicit Prepare

To request an explicit accept and explicit prepare of transactions, the server channel is opened with the `RTR_F_OPE_EXPLICIT_PREPARE` and `RTR_F_OPE_EXPLICIT_ACCEPT` flags. These specify that the channel will receive both prepare and accept messages. The server then explicitly accepts or rejects a transaction when it receives the prepare message. The transaction sequence for an explicit prepare and explicit accept is as follows:

Client	RTR	Server
`rtr_start_tx`		
`rtr_send_to_server`	→ `rtr_mt_msg1`	→ `rtr_receive_message`
`rtr_accept_tx`	→ `rtr_mt_prepare`	→ `rtr_receive_message`
		← `rtr_accept_tx`
`rtr_receive_message`	← `rtr_mt_accepted`	← `rtr_receive_message`

With explicit transaction handling, the following steps occur:

1. The server application waits for a message from the client application.
2. The server application receives the `rtr_mt_prepare` request message from RTR.
3. The server application issues the accept or reject.

A participant can reject the transaction up to the time RTR has sent the `rtr_mt_prepare` message type to the server in the `rtr_accept_tx` call. A participant can reject the transaction up to the time the `rtr_accept_tx` call is executed. Once the client application has called `rtr_accept_tx`, the result cannot be changed.

5.13.5. Implicit Prepare, Explicit Accept

The sequence for an implicit prepare and explicit accept is as follows:

Client	RTR	Server
`rtr_start_tx`		
`rtr_send_to_server`	→ `rtr_mt_msg1`	→ `rtr_receive_message`
`rtr_accept_tx`	→	← `rtr_accept_tx`
`rtr_receive_message`	← `rtr_mt_accepted`	← `rtr_receive_message`

In certain database applications, where the database manager does not let an application explicitly prepare the database, transactions can simply be accepted or rejected. Server applications that do not specify the RTR_F_EXPLICIT_ACCEPT flag in their rtr_open_channel call implicitly accept the in-progress transaction when an rtr_receive_message call is issued after the last message has been received for the transaction. This call returns the final status for the transaction, rtr_mt_accepted or rtr_mt_rejected. If neither the RTR_F_OPE_EXPLICIT_ACCEPT nor the RTR_F_OPE_EXPLICT_PREPARE flags are set in the rtr_open_channel call, then both prepare and accept processing will be implicit.

For server optimization, the server can signal its acceptance of a transaction with either rtr_reply_to_client, using the RTR_F_REP_ACCEPT flag, or with the client issuing the rtr_send_to_server call, using the RTR_F_SEN_ACCEPT flag. This helps to minimize network traffic for transactions by increasing the likelihood that the data message and the RTR accept message will be sent in the same network packet.

5.14. Transaction Recovery

When a transaction fails in progress, RTR provides recovery support using RTR replay technology. RTR, as a distributed transaction manager, communicates with a database resource manager in directing the two-phase commit process. When using the XA protocol, the application does not need to process rtr_mt_uncertain messages (see the section Using XA, for more details on using XA).

The typical server application transaction sequence for committing a transaction to the database is as follows:

```
rtr_receive_message (rtr_mt_msg1)
SQL update
rtr_accept_tx
rtr_receive_message (rtr_mt_accepted)
SQL commit
rtr_receive_message [wait for next transaction]
```

This sequence is also illustrated in Figure 2.7, CSN Vote Window for the C API.

A failure can occur at any step in this sequence; the impact of a failure depends on when (at which step) it occurs, and on the server configuration.

5.14.1. Failure before rtr_accept_tx

If a failure occurs before the rtr_accept_tx call is issued, RTR causes the following to occur:

- With no database replication: the transaction is aborted and the database rolled back.
- With a concurrent server: the database is rolled back and the transaction replayed to another server instance.
- With a standby server: the database is rolled back and the transaction replayed to the standby.
- With a shadow server: the shadow server completes the transaction as a lone member. When the failed server returns, the database is rolled back and the transaction is replayed.

5.14.2. Failure after rtr_accept_tx

If a failure occurs after the rtr_accept_tx call is issued but before the rtr_receive_message, the transaction is replayed. The type of the first message is then

`rtr_mt_uncertain` when the server is restarted. In this case, servers should check to see if the transaction has already been executed in a previous presentation. If not, it is safe to re-execute the transaction because the database operation never occurred. After the failure, the following occurs:

- With no database replication: the database is rolled back and the transaction is replayed as uncertain. The server should re-execute the transaction.
- With a concurrent server: the database is rolled back and the transaction is replayed as uncertain to another concurrent server. The server should re-execute the transaction.
- With a standby server: the database is rolled back and the transaction is replayed as uncertain to the standby server. The server should re-execute the transaction.
- With a shadow server: the shadow server completes the transaction as a lone member. When the failed server returns, the database is rolled back and the transaction is replayed as uncertain. The primary server should re-execute the transaction.

If a failure occurs after the SQL commit but before receipt of a message starting the next transaction, RTR does not know the difference.

If a failure occurs after an `rtr_receive_message` call is made to begin a new transaction, RTR assumes a successful commit if a server calls `rtr_receive_message` after receiving the `rtr_mt_accepted` message and will forget the transaction. There is no replay following these events.

5.14.3. Changing Transaction State

Under certain conditions, transactions may become blocked or hung, and applications can use certain RTR features to clear such roadblocks. Often, a blocked state can be cleared by the system manager using the SET TRANSACTION CLI command to change the state of a transaction, for example, to DONE. Only certain states, however, can be changed, and changing the state of a transaction manually can endanger the integrity of the application database. For information on using the SET TRANSACTION command from the CLI, see the *VSI Reliable Transaction Router System Manager's Manual*.

To achieve a change of state within an application, the designer can use the `rtr_set_info` C API call. For example, the following code fragments illustrate how such application code could be written. Such code must be used in conjunction with a design that uses a change of state only when transactions are blocked.

```
rtr_tid_t tid;
rtr_channel_t pchannel;
rtr_qualifier_value_t select qualifiers[4];
rtr_qualifier_value_t set_qualifiers[3];
int select_idx = 0;
int set_idx = 0;

rtr_get_tid(pchannel, RTR_F_TID_RTR, &tid);

/* Transaction branch to get current state */
 select_qualifiers[select_idx].qv_qualifier = rtr_txn_state;
 select_qualifiers[select_idx].qv_value = &rtr_txn_state_commit;
 select _ idx++;
/* Transaction branch to set new state */
 set_qualifiers[set_idx].qv_qualifier = rtr_txn_state;
 set_qualifiers[set_idx].qv_value = &rtr_txn_state_exception;
```

```
 set_idx++;

sts = rtr_set_info(pchannel,
     (rtr_flags_t) 0,
     (rtr_verb_t)verb_set,
     rtr_transaction_object,
     select_qualifiers,
     set_qualifiers);
if(sts != RTR_STS_OK)
       write an error;

sts = rtr_receive_message(
  /* channel  */ &channel_id,
  /* flags    */ RTR_NO_FLAGS,
  /* prcvchan */ pchannel,
  /* pmsg     */ msg,
  /* maxlen   */ RTR_MAX_MSGLEN,
  /* timoutms */ 0,
  /* pmsgsb   */ &msgsb);
if (sts == RTR_STS_OK){
 const rtr_status_data_t *pstatus = (rtr_status_data_t *)msg;
 rtr_uns_32_t num;

 switch(pstatus -> status)
 {
 case RTR_SETTRANDONE: /*Set Tran done successfully */
 memcpy(&num,(char)msg+sizeof(rtr_status_data_t),
  sizeof(rtr_uns_32_t));

 printf(" %d transaction(s) have been processed\n");
 break;
default;
}
}
```

5.14.4. Exception on Transaction Handling

RTR keeps track of how many times a transaction is presented to a server application before it is VOTED. The rule is: three strikes and you're out! After the third strike, RTR rejects the transaction with the reason `RTR_STS_SRVDIED`. The server application has committed the transaction and the client believes that the transaction is committed. The transaction is flagged as an exception and the database is not committed. Such an exception transaction can be manually committed if necessary. This process eliminates the possibility that a single rogue transaction can crash all available copies of a server application at both primary and secondary sites.

Application design can change this behavior. The application can specify the retry count to use when in recovery using the `/recovery_retry_count` qualifier in the `rtr_set_info` call, or the system administrator can set the retry count from the RTR CLI with the SET PARTITION command. If no recovery retry count is specified, RTR retries replay three times. For recovery, retries are infinite. For more information on the SET PARTITION command, refer to the *VSI Reliable Transaction Router System Manager's Manual*; for more information on the `rtr_set_info` call, refer to the *VSI Reliable Transaction Router C Application Programmer's Reference Manual*.

When a node is down, the operator can select a different backend to use for the server restart. To complete any incomplete transactions, RTR searches the journals of all backend nodes of a facility for any transactions for the key range specified in the server's `rtr_open_channel` call.

5.15. Broadcast Messaging

Use the `rtr_broadcast_event` call to broadcast a user event message.

A user broadcast is named or unnamed. An unnamed broadcast does a match on user event number. A named broadcast does a match on user event number and recipient name, a user-defined string. Named broadcasts provide greater control over who receives a particular broadcast. Named events specify an event number and a textual recipient name. The name can include wildcards (% and *).

For all unnamed events, specify the *evtnum* field and `RTR_NO_RCPSPC` (no recipient specification) for the recipient name.

For example, the following code specifies named events for all subscribers (`rcpnam="*"`):

```
rtr_status_t
rtr_open_channel {
 ...
 rtr_rcpnam_t rcpnam = "*";
 rtr_evtnum_t evtnum = {
  RTR_EVTNUM_USERDEF,
   RTR_EVTNUM_USERBASE,
   RTR_EVTNUM_UP_TO,
   RTR_EVTNUM_USERMAX,
  RTR_EVTNUM_ENDLIST
 };
rtr_evtnum_t *p_evtnum = &evtnum;
```

For a broadcast to be received by an application, the recipient name specified by the subscribing application on its `rtr_open_channel` call must match the recipient specifier used by the broadcast sender on the `rtr_broadcast_event` call.

Note

`RTR_NO_RCPSPC` is not the same as "*".

An application receives broadcasts with the `rtr_receive_message` call. A message type returned in the message status block informs the application of the type of broadcast received. Table 5.6 summarizes the rules.

Table 5.6. C API Named and Unnamed Broadcast Events

Broadcast Type	Match On	Specify
Named	user event number	*evtnum* field
	recipient name	`rcpspc=` *subscriber* (for all subscribers use ="*")
Unnamed	user event number	*evtnum* field
		`RTR_NO_RCPSPC`

5.16. Authentication Using Callout Servers

RTR callout servers enable security checks to be carried out on all requests using a facility. Callout servers can run on either backend or router nodes. They receive a copy of every transaction delivered

to, or passing through, the node, and they vote on every transaction. To enable a callout server, use the */CALLOUT* qualifier when issuing the *RTR CREATE FACILITY* command. Callout servers are facility based, not key-range or partition based.

5.16.1. Router Callout Server

An application enables a callout server by setting a flag on the rtr_open_channel call.

For a router callout server, the application sets the following flag on the `rtr_open_channel` call:

```
rtr_ope_flag_t
        flags=RTR_F_OPE_SERVER |RTR_F_OPE_TR_CALL_OUT
```

5.16.2. Backend Callout Server

For a backend callout server, the application sets the following flag on the rtr_open_channel call:

```
rtr_ope_flag_t
        flags=RTR_F_OPE_SERVER|RTR_F_OPE_BE_CALL_OUT
```

Appendix A. RTR Design Examples

To provide information for the design of new applications, this section contains scenarios or descriptions of existing applications that use RTR for a variety of reasons. They include:

- A transportation example, which shows a nationwide use of partitioned, distributed databases and surrogate clients.
- A stock exchange example, which shows use of reliable broadcasts, database partitioning, standby and concurrent servers.
- A banking example, which shows use of application multithreading and an FDDI cluster.
- Customer names are not used, but these designs reflect successfully implemented, working applications.

A.1. Transportation Example

A.1.1. Brief History

In the 1980s, a large railway system implemented a monolithic application in FORTRAN for local reservations with local databases separated into five administrative domains or regions: Site A, Site B, Site C, Site D, and Site E. By policy, rail travel for each region was controlled at the central site for each region, and each central site owned all trains leaving from that site. For example, all trains leaving from Site B were owned by Site B. The railway system supported reservations for about 1000 trains.

One result of this architecture was that for a passenger to book a round-trip journey, from, say, Site B to Site A and return, the passenger had to stand in two lines, one to book the journey from Site B to Site A, and the second to book the journey from Site A to Site B.

The implementation was on an VSI OpenVMS cluster at each site, with a database engine built on RMS, using flat files. The application displayed a form for filling out the relevant journey and passenger information: (train, date, route, class, and passenger name, age, sex, concessions). The structure of the database was the same for each site, though the content was different. RTR was not used. Additionally, the architecture was not scalable; it was not possible to add more terminals for client access or add more nodes to the existing clusters without suffering performance degradation.

A.1.2. New Implementation

This example implements partitioned, distributed databases and surrogate clients.

New requirements from the railway system for a national passenger reservations system included the goal that a journey could be booked for any train from anywhere to anywhere within the system. Meeting this goal would also enable distributed processing and networking among all five independent sites. In addition to this new functionality, the new architecture had to be more scalable and adaptable for PCs to replace the current terminals, part of the long-term plan.

With these objectives, the development team rewrote their application in C, revamped their database structure, adopted RTR as their underlying middleware, and significantly improved their overall application. The application became scalable, and additional features could be introduced. Key objectives

of the new design were improved performance, high reliability in a moderately unstable network, and high availability, even during network link loss.

The structure of the database at all sites was the same, but the data were for each local region only. The database was partitioned by train ID (which included departure time), date, and class of service, and RTR data content routing was used to route a reservation to the correct domain, and bind reservation transactions as complete transactions across the distributed sites to ensure booking without conflicts. This neatly avoided booking two passengers in the same seat, for example. Performance was not compromised, and data partitioning provided efficiency in database access, enabling the application to scale horizontally as load increased. This system currently deals with approximately three million transactions per day. One passenger reservation represents a single business transaction, but may be multiple RTR transactions. An inquiry is a single transaction.

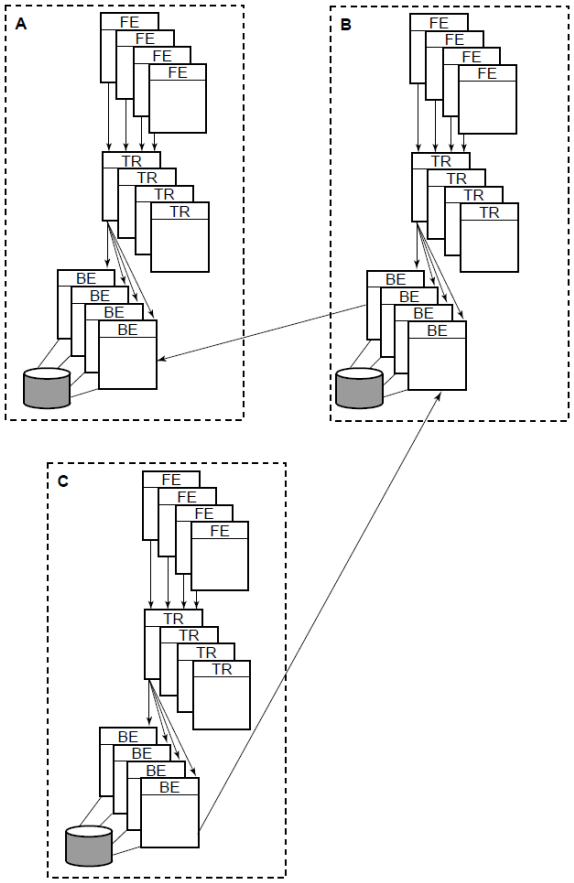
An important new feature was the use of surrogate clients at each local site that act as clients of the remote sites using a store and forward mechanism. The implementation of these surrogate clients made it possible to book round-trip tickets to any of the regions from a single terminal. This design addressed the problem of frequent RTR quorum negotiations caused by network link drops and ensured that these would not affect local transactions.

The local facility defined in one location (say, Site B) includes a gateway server acting as a surrogate client that communicates with the reservation server at the remote site (say, Site C). For example, to make a round-trip reservation in one client request from Site B to Site C and return, the reservation agent enters the request with passenger ID, destination, and date. For the Site B to Site C trip, the destination is Site C, and for the Site C to Site B trip, the destination is Site B. This information is entered only at Site B. The reservation transaction is made for the Site-B-to-Site-C trip locally, and the transaction for the return trip goes first to the surrogate client for Site C.

The surrogate forwards the transaction to the real Site C server that makes the reservation in the local Site C database. The response for the successful transaction is then sent back to the surrogate client at Site B, which passes the confirmation back to the real client, completing the reservation. There are extensive recovery mechanisms at the surrogate client for transaction binding and transaction integrity. When transaction recovery fails, a locally developed store- and-forward mechanism ensures smooth functioning at each site.

The system configuration is illustrated in Figure A-1. For clarity, only three sites are shown, with a single set of connections. All other connections are in use, but not shown in the figure. Local connections are shown with single-headed arrows, though all are duplex; connections to other sites by network links are shown with double-headed arrows. Connections to the local databases are shown with solid lines. Reservations agents connect to frontends.

Figure A.1. Transportation Example Configuration



Currently the two transactions (the local and the remote) are not related to each other. The application has to make compensations in case of failure because RTR does not know that the second transaction is a child of the first. This ensures that reservations are booked without conflicts.

The emphasis in configurations is on availability; local sites keep running even when network links to other sites are not up. The disaster-tolerant capabilities of RTR and the system architecture made it easy to introduce site-disaster tolerance, when needed, virtually without redesign.

A.2. Stock Exchange Example

A.2.1. Brief History

For a number of years, a group of banks relied on traditional open-outcry stock exchanges in several cities for their trades in stocks and other financial scrip (paper). These were three separate markets, with three floor-trading operations and three order books. In the country, financial institutions manage a major portion of international assets, and this traditional form of stock trading inhibited growth. When the unified stock exchange opened, they planned to integrate these diverse market operations into a robust and standards-compliant system, and to make possible electronic trading between financial institutions throughout the country.

The stock exchange already had an implementation based on OpenVMS, but this system could not easily be extended to deal with other trading environments and different financial certificates.

A.2.2. New Implementation

This example implements reliable broadcasts, database partitioning, and uses both standby and concurrent servers.

For their implementation using RTR, the stock exchange introduced a wholly electronic exchange that is a single market for all securities listed in the country, including equities, options, bonds, and futures. The hardware superstructure is a cluster of 64-bit VSI AlphaServer systems with a network containing high-speed links with up to 120 gateway nodes connecting to over 1000 nodes at financial institutions throughout the country.

The stock exchange platform is based on the VSI OpenVMS cluster technology, which achieves high performance and extraordinary availability by combining multiple systems into a fault-tolerant configuration with redundancy to avoid any single point of failure. The standard trading configuration is either high-performance AlphaStations or Sun workstations, and members with multi-seat operations such as banks use AlphaServer 4100 systems as local servers. Due to trading requirements that have strict time dependency, shadowing is not used. For example, it would not be acceptable for a trade to be recorded on the primary server at exactly 5:00:00 PM and at 5:00:01 PM on the shadow.

From their desks, traders enter orders with a few keystrokes on customized trading workstation software running UNIX that displays a graphical user interface. The stock exchange processes trades in order of entry, and within seconds:

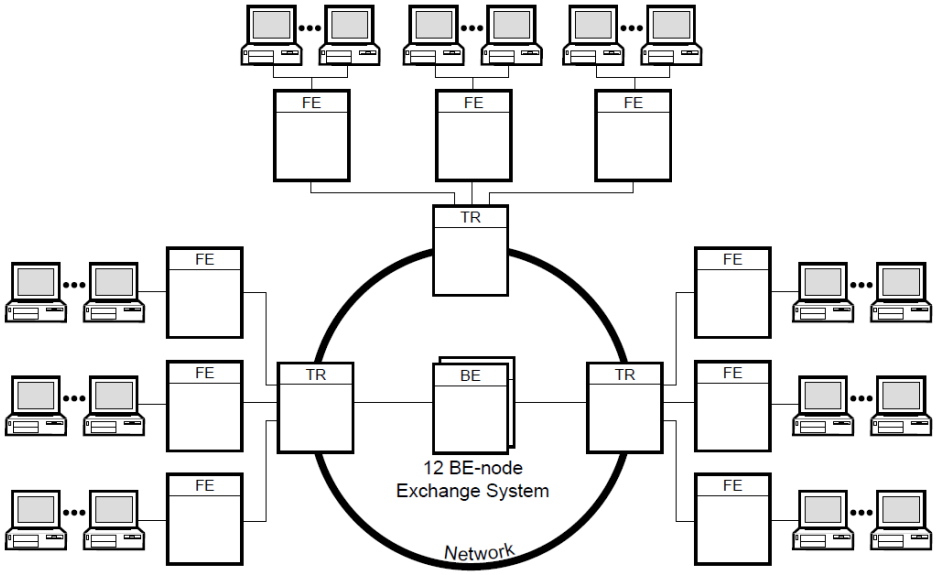
- Sends the trader a detailed confirmation statement
- Dispatches the trade to clearing and settlement systems

Traders further have access to current and complete market data and can therefore more effectively monitor and manage risks. The implementation ensures that all members receive the same information at the same time, regardless of location, making fairness a major benefit of this electronic exchange. (In RTR itself, fairness is achieved using randomization, so that no trader would receive information first, all the time. Using RTR alone, no trader would be favored.)

The stock exchange applications work with RTR to match, execute, and confirm buy/sell orders, and dispatch confirmed trades to the portfolio management system of the securities clearing organization, and to the international settlement system run by participating banks.

The stock exchange designed their client/server frontend to interface with the administrative systems of most banks; one result of this is that members can automate back-room processing of trades and greatly reduce per-order handling expenses. VSI server reliability, VSI clustering capability, and cross- platform connectivity are critical to the success of this implementation. RTR client application software resides on frontends on the gateways that connect to routers on access nodes. The access nodes connect to a 12-node VSI OpenVMS cluster where the RTR server application resides. The configuration is illustrated in Figure A-2. Only nine trader workstations are shown at each site, but many more are in the actual configuration. The frontends are gateways, and the routers are access points to the main system.

Figure A.2. Stock Exchange Example



A further advantage of the RTR implementation is that the multivendor, multiprotocol 1000-node environment can be managed with only five staff people. This core staff can manage the network, the operating systems, and the applications with their own software that detects anomalies and alerts staff members by pagers and mobile computers. Using RTR also employs standard two-phase-commit processing, providing complete transaction integrity across the distributed systems. With this unique implementation, RTR swiftly became the underpinning of nationwide stock exchanges. RTR also provides ease of management, and with two-phase commit, makes it easier than previously to manage and control the databases.

The implementation using RTR also enables the stock exchange to provide innovative services and tools based on industry and technology standards, cooperate with other exchanges, and introduce new services without reengineering existing systems. For example, with RTR as the foundation of their systems, they plan an Oracle 7 data warehouse of statistical data off a central Oracle Rdb database, with VSI Object Broker tools to offer users rapid and rich ad-hoc query capabilities. Part of a new implementation includes the disaster-tolerant VSI Business Recovery Server solution and replication of its OpenVMS cluster configuration across two data centers, connected with the VSI DEChub 900 GIGAswitch/ATM networking technology.

The unique cross-platform scalability of these systems further enables the stock exchange to select the right operating system for each purpose. Systems range from the central OpenVMS cluster, to frontends based on UNIX or Microsoft Windows. To support trader desktops with spreadsheets, an in-process implementation uses Windows with Microsoft Office to report trading results to the trader workstation.

A.3. Banking Example

A.3.1. Brief History

Several years ago a large bank recognized the need to devise and deliver more convenient and efficient banking services to their customers. They understood both the expense of face-to-face transactions at a bank office and wanted to explore new ways to reduce these expenses and to improve customer convenience with 24-hour service, a level of service not available at a bank office or teller window.

A.3.2. New Implementation

This example shows use of application multithreading in an FDDI cluster.

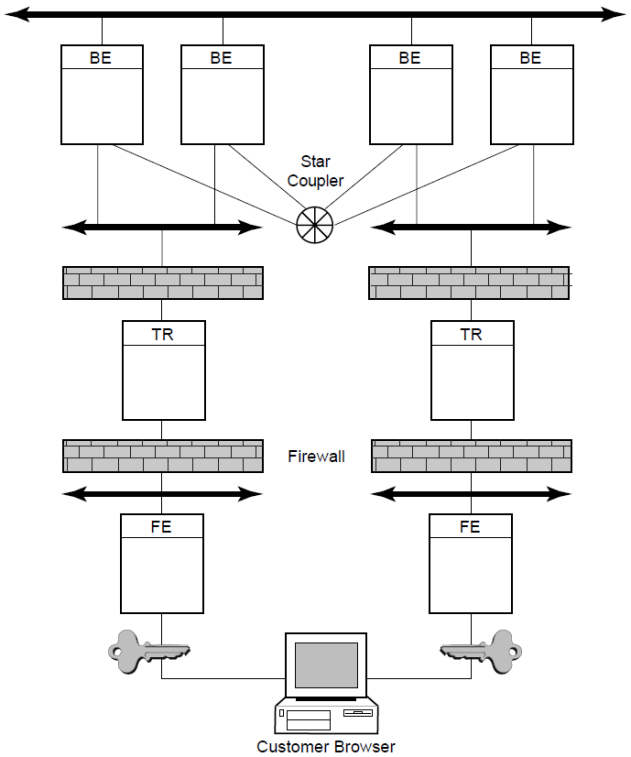
The bank had confidence in the technology, and with RTR, was able to implement the world's first secure internet banking service. This enabled them to lower their costs as much as 80% and provide 24 x 365 convenience to their customers. They were additionally able to implement a global messaging backbone that links 20,000 users on a broad range of popular mail systems to a common directory service.

With the bank's electronic banking service, treasurers and CEOs manage corporate finances, and individuals manage their own personal finances, from the convenience of their office or home. Private customers use a PC- based software package to access their account information, pay bills, download or print statements, and initiate transactions to any bank in the country, and to some foreign banks.

For commercial customers, the bank developed software interfaces that provide import and export links between popular business financial packages and the electronic banking system. Customers can use their own accounting system software and establish a seamless flow of data from their bank account to their company's financial system and back again.

The bank developed its customized internet applications based on Microsoft Internet Information Server (IIS) and RTR, using VSI Prioris servers running Windows as frontend web servers. The client application runs on a secure HTTP system using 128-bit encryption and employs CGI scripts in conjunction with RTR client code. All web transactions are routed by RTR through firewalls to the electronic banking cluster running OpenVMS. The IIS environment enabled rapid initial deployment and contains a full set of management tools that help ensure simple, low-cost operation. The service handles 8,000 to 12,000 users per day and is growing rapidly. Figure A-3 illustrates the deployment of this banking system.

Figure A.3. Banking Example Configuration



The RTR failure-tolerant, transaction-messaging middleware is the heart of the internet banking service. Data is shadowed at the transactional level, not at the disk level, so that even with a network failure, in-progress transactions are completed with integrity in the transactional shadow environment.

The banking application takes full advantage of the multiplatform support provided by RTR; it achieves seamless transaction-processing flow across the backend OpenVMS clusters and secure web servers based on Windows frontends. With RTR scalability, backends can be added as volume increases, load can be balanced across systems, and maintenance can be performed during full operation.

For the electronic banking application, the bank used RTR in conjunction with an Oracle Rdb database. The security and high availability of RTR and OpenVMS clusters provided what was needed for this sensitive financial application, which supports more than a quarter million customer accounts, and up to 38 million transactions a month with a total value of U.S. $300 to $400 million.

The bank's electronic banking cluster is distributed across two data centers located five miles apart and uses VSI GIGAswitch/FDDI systems for ultra-fast throughput and instant failover across sites without data loss. The application also provides redundancy into many elements of the cluster. For example, each data center has two or more computer systems linked by dual GIGAswitch systems to multiple FDDI rings, and the cluster is also connected by an Ethernet link to the LAN at bank headquarters.

The cluster additionally contains 64-bit Very Large Memory (VLM) capabilities for its Oracle database; this has increased database performance by storing frequently used files and data in system memory rather than on disk. All systems in the electronic banking cluster share access to 350 gigabytes of SCSI-based disks. Storage is not directly connected to the cluster CPUs, but connected to the network through the FDDI backbone. Thus, if a CPU goes down, storage survives, and is accessible to other systems in the cluster.

The multi-operating system cluster is very economical to run, supported by a small staff of four system managers who handle all the electronic banking systems. Using clusters and RTR enables the bank to provide very high levels of service with a very lean staff.

Appendix B. RTR Cluster Configurations

The *cluster environment* can be important to the smooth failover characteristics of RTR. This environment is slightly different on each operating system. The essential features of clusters are availability and the ability to access a common disk or disks. Basic cluster configurations are illustrated below for the different operating systems where RTR can run.

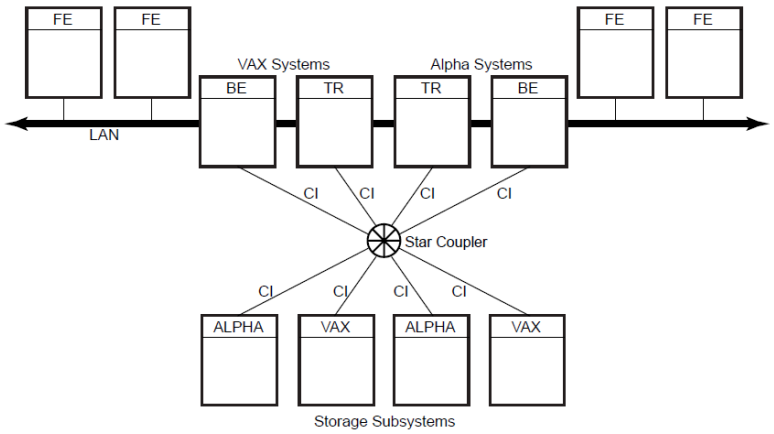
B.1. OpenVMS Cluster

An OpenVMS cluster provides disk shadowing capabilities, and can be based on several interconnects including:

- CI
- FDDI

Figure B-1 shows a CI-based OpenVMS cluster configuration. Client applications run on the frontends; routers and backends are established on cluster nodes, with backend nodes having access to the storage subsystems. The LAN is the Local Area Network, and the CI is the Computer Interconnect joined by a Star Coupler to the nodes and storage subsystems. Network connections can include GIGAswitch subsystems.

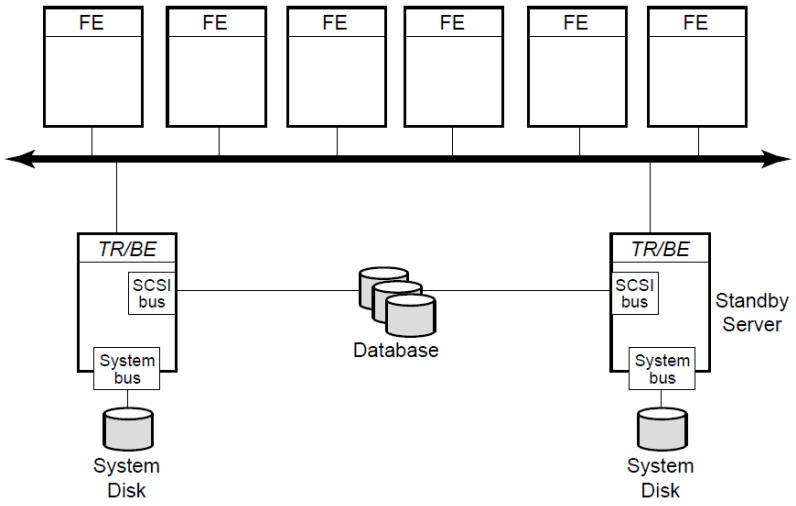
Figure B.1. OpenVMS CI-based Cluster



B.2. Windows Cluster

In the Windows environment, two Intel servers managed and accessed as a single node comprise an NT cluster. You can use RAID storage for cluster disks with dual redundant controllers. A typical configuration would place the RTR frontend, router, and backend on the cluster nodes, as shown in Figure B-3 and would include an additional tie-breaker node on the network to ensure that quorum can be achieved.

Figure B.2. Windows Cluster

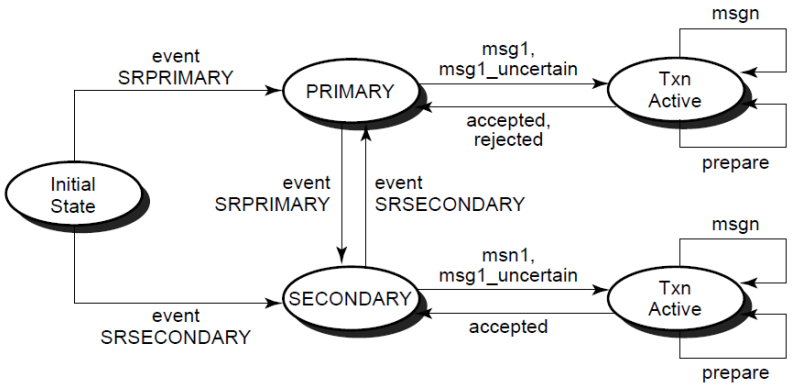


Appendix C. Server States

C.1. Server and Active Transaction States in a Shadow Server

Figure C.1 shows server states after delivery of a primary or secondary event, and message types used with primary and secondary servers.

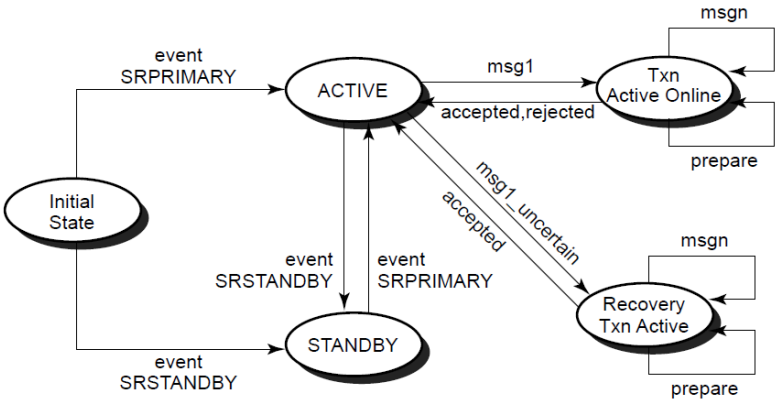
Figure C.1. Server Events and States with Active Transaction Message Types



C.2. Server and Transaction States in a Standby Server

Figure C.2 shows server states after delivery of a standby event, and message types used with transactions that are active or in recovery.

Figure C.2. Server States after Standby Events



Appendix D. RTR C++ API Samples

D.1. Specifying Server Type

The application must specify server type with boolean attributes using the CreateBackEndPartition method in the RTRManager class. For example, the following declaration establishes a standby server with concurrency:

```
CreateBackEndPartition( *pszPartitionName,
                        pszFacility,
                        pKeySegment
                        bShadow=false
                        bConcurrent=true
                        bStandby=true);
```

To add a transactional shadow server, use: `bShadow = true`

To disallow a standby server, use: `bStandby = false`

D.1.1. Server Failover

With the C++ API, you enable RTR failover behavior with the CreateBackEndPartition method in the RTRPartitionManager management class.

D.2. Concurrent Servers

For the C++ API, concurrent servers can be implemented as many server transaction controllers in one process or as one or many server transaction controllers in many processes.

RTR delivers transactions to any open transaction controllers, so each application thread must be ready to receive and process transactions.

An application creates a transaction controller and registers a partition with the RegisterPartition method. To specify whether or not a server is concurrent, the application uses the CreateBackendPartition method in the RTRPartitionManager class. The rules are as follows:

- Set the bConcurrent parameter to true for the server to have other concurrent servers.
- Set the bConcurrent parameter to false for the server not to be concurrent.

For example, the following declaration establishes a concurrent server that is also a standby:

```
CreateBackEndPartition( *pszPartitionName,
                        pszFacility,
                        pKeySegment
                        bShadow=false
                        bConcurrent=true
                        bStandby=true);
```

The following declaration establishes a server with no concurrency:

```
CreateBackEndPartition( *pszPartitionName,
                        pszFacility,
                        pKeySegment
```

```
                        bShadow=false
                        bConcurrent=false
                        bStandby=true);
```

For more information on the CreateBackEndPartition method, see the *VSI Reliable Transaction Router C ++ Foundation Classes* manual.

D.2.1. Standby Servers

RTR manages the activation of standby servers at run time.

When an application creates a server partition with the CreateBackEndPartition method in the RTRPartitionManager class, it specifies whether a server is to be standby or not as follows:

- Sets the bStandby parameter true so the server can have standby servers.
- Sets the bStandby parameter to false to specify that the server is not to be a standby nor to have standbys. For example, the following declaration establishes a concurrent server that is not a standby.

```
CreateBackEndPartition ( *pszPartitionName,
           pszFacilityName,
         *pKeySegment,
          bShadow = false,
          bConcurrent = true,
          bStandby = false);
```

D.2.2. Shadow Servers

When an application creates a server partition with the CreateBackEndPartition method in the RTRPartitionManager class, it specifies whether a server is to be a shadow or not as follows:

- Sets the bShadow parameter to false so the server is not a shadow server.
- Specifies that the server is to be a shadow by setting the bShadow parameter to true. For example:

```
CreateBackEndPartition ( *pszPartitionName,
          pszFacilityName,
         *pKeySegment,
          bShadow = true,
          bConcurrent = true,
          bStandby = false);
```

Only one primary and one secondary shadow server can be established. Shadow servers can have concurrent servers.

When partition state is important to an application, the application can determine if a shadow server is in the primary or secondary partition state after server restart and recovery following a server failure. The application does this using methods in the RTRServerEventHandler class such as OnServerIsPrimary, OnServerIsStandby, and OnServerIsSecondary. For example:

```
OnServerIsPrimary(*pRTRData, *pController);
```

D.2.3. Making Transactions Independent

Within your application server code, you identify those transactions that can be considered independent, and set the state of the transaction controller object with the bIndependent attribute

in the AcceptTransaction method, as appropriate. The following example illustrates how to set the bIndependent parameter to true with the AcceptTransaction method to make a transaction independent.

```
RTRServerTransactionController *pController= new
                                 RTRServerTransactionController();
pController->AcceptTransaction(RTR_NO_REASON, true);
```

Another example:

```
RTRServerTransactionController stc;
/* Determine from our business logic if this transaction is independent of
 our
other transactions. */
If (true == Independent())
{
  stc.AcceptTransaction(RTR_NO_REASON,true)
}
else
{
  stc.AcceptTransaction()
}
```

Appendix E. RTR C API Samples

E.1. Specifying Server Type

The application specifies the server type in the `rtr_open_channel` call as follows:

```
rtr_status_t
rtr_open_channel (
 .
 rtr_ope_flag_t
```

To add a transactional shadow server, include the following flags: `flags = RTR_F_OPE_SERVER RTR_F_OPE_SHADOW;`

To disallow concurrent and standby servers, use the following flags:

```
flags = RTR_F_OPE_SERVER | RTR_F_OPE_NOCONCURRENT |
RTR_F_OPE_NOSTANDBY;
```

E.1.1. Server Failover

With the C API, you enable RTR failover behavior with flags set when your application executes the `rtr_open_channel` statement or command.

E.2. Concurrent Servers

For the C API, concurrent servers can be implemented as many channels in one process or as one or many channels in many processes. By default, a server channel is declared as concurrent.

RTR delivers transactions to any open channels, so each application thread must be ready to receive and process transactions. The main constraint in using concurrent servers is the limit of available resources on the machine where the concurrent servers run.

When an application opens a channel with the `rtr_open_channel` call, it specifies whether the server is to be concurrent or not, as follows:

- Does nothing (omits the flag) so the server can have other concurrent servers. This is the default.
- Uses the `RTR_F_OPE_NOCONCURRENT` flag to indicate that the server is not to be concurrent.

For example, the following code fragment establishes a server with concurrency:

```
rtr_open_channel(&Channel,
  RTR_F_OPE_SERVER,
  FACILITY_NAME,
  NULL,
  RTR_NO_PEVTNUM,
  NULL,
  Key.GetKeySize(),
  Key.GetKey() !=  RTR_STS_OK);
```

If an application starts up a second server for a partition on the same node, the second server is a concurrent server by default.

The following example establishes a server with no concurrency:

```
rtr_open_channel(&Channel,
  RTR_F_OPE_SERVER|RTR_F_OPE_NOCONCURRENT,
  FACILITY_NAME,
  NULL,
  RTR_NO_PEVTNUM,
  NULL,
  Key.GetKeySize(),
  Key.GetKey() != RTR_STS_OK);
```

When a concurrent server fails, the server application can fail over to another running concurrent server, if one exists.

Concurrent servers are useful both to improve throughput using multiple channels on a single node, and to make process failover possible. Concurrent servers can also help to minimize timeout problems in certain server applications. For more information on this topic, see the section later in this manual on Server-Side Transaction Timeouts.

For more information on the `rtr_open_channel` call, see the *VSI Reliable Transaction Router C Application Programmer's Reference Manual* and the discussion later in this document.

E.2.1. Standby Servers

RTR manages the activation of standby servers at run time.

When an application opens a channel, it specifies whether or not the server is to be standby, as follows:

- Does nothing (omits the flag) so the server can have standby servers. This is the default.
- Includes the `RTR_F_OPE_NOSTANDBY` flag so the server is not to be a standby nor to have standbys.

E.2.2. Shadow Servers

When an application opens a channel, it specifies whether the server is to have the capability to be a transactional shadow server or not, as follows:

- Does nothing (omits the flag) so the server is not a shadow server. This is the default.
- Includes the RTR_F_OPE_SHADOW flag so the server is to be a shadow server.

Only one primary and one secondary shadow server can be established. Shadow servers can also have concurrent servers.

When partition state is important to an application, the application can determine if a shadow server is in the primary or secondary partition state after server restart and recovery following a server failure. The application does this using RTR events in the `rtr_open_channel` call, specifying the events `RTR_EVTNUM_SRPRIMARY` and `RTR_EVTNUM_SRSECONDARY`. For example, the following is the usual `rtr_open_channel` declaration:

```
rtr_status_t
rtr_open_channel (
 rtr_channel_t *p_channel,  //Channel
 rtr_ope_flag_t flags, //Flags
 rtr_facnam_t facnam, //Facility
```

```
 rtr_rcpnam_t rcpnam,     //Name of the channel
 rtr_evtnum_t *p_evtnum,  //Event number list
                              //(for partition states)
 rtr_access_t access,  //Access password
 rtr_numseg_t numseg,    //Number of key segments
 rtr_keyseg_t *p_keyseg  //Pointer to key-segment data
)
```

To enable receipt of RTR events that show shadow state, used if an application needs to include logic depending on partition state, the application enables receipt of RTR events that show shadow state.

The declaration includes the events as follows:

```
rtr_evtnum_t evtnum = {
  RTR_EVTNUM_RTRDEF,
  RTR_EVTNUM_SRPRIMARY,
  RTR_EVTNUM_SRSECONDARY,
  RTR_EVTNUM_ENDLIST
};
rtr_evtnum_t *p_evtnum = &evtnum;
```

Broadcasts deliver using name and subscription name. For details, see the descriptions of `rtr_open_channel` and `rtr_broadcast_event` in the *VSI Reliable Transaction Router C Application Programmer's Reference Manual*.

E.2.3. Making Transactions Independent

Within your application server code, you identify those transactions that can be considered independent, and process them with the independent transaction flags on `rtr_accept_tx` or `rtr_reply_to_client` calls, as appropriate. For example, the following code fragment illustrates use of the independent transaction flag on the `rtr_accept_tx` call:

```
  case rtr_mt_prepare:
  /* if (txn is independent).*/
 status  = rtr_accept_tx (channel,
                                  RTR_F_ACC_INDEPENDENT,
                                  RTR_NO_REASON\\);
 if (status != RTR_STS_OK)
```

You can also use the independent flag on the `rtr_reply_to_client` call. For example,

```
rtr_reply_to_client(channel,
   RTR_F_REP_INDEPENDENT,
   pmsg, msglen, msgfmt);
```

E.2.4. RTR Events

An application subscribes to an RTR event with the `rtr_open_channel` call. For example,

```
rtr_status_t
rtr_open_channel(
 .
 rtr_rcpnam_t rcpnam = RTR_NO_RCPNAM;
  rtr_evtnum_t evtnum = {
   RTR_EVTNUM_RTRDEF,
   RTR_EVTNUM_SRPRIMARY,
   RTR_EVTNUM_ENDLIST
```

```
                };
      rtr_evtnum_t *p_evtnum = &evtnum; )
```

You read the message type to determine what RTR has delivered. For example,

```
rtr_status_t
rtr_receive_message (
 .
 rtr_msgsb_t *p msgsb
 )
```

Use a data structure of the following form to receive the message:

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

The event number is returned in the message status block in the *evtnum* field. The following RTR events return key range data back to the client application:

```
     RTR_EVTNUM_KEYRANGEGAIN
     RTR_EVTNUM_KEYRANGELOSS
```

These data are included in the message (*pmsg*); size is `msglen_sizeof(rtr_msgsb_t)`. Other events do not have additional data.

Appendix F. Evaluating Application Resource Requirements

F.1. Diagnosing Performance Problems

Use the following brief checklist to help diagnose a particular performance problem:

1. Check the CPU load on the machines involved. A machine loaded over 60% is generally suspect, if reasonable response times are desired.

   Possible fixes:

   - Buy a more powerful CPU.
   - Add more SMP processors.
   - Partition the application workload over multiple machines.
   - Profile application CPU usage and optimize code hotspots.

2. Measure:

   - Disk I/O rate
   - Data rate on the disks used for the RTR journal
   - Data rate of the application

   These rates should be comfortably below the rated capacity of the controller and drives. If not, you may be on the trail of a performance constraint. Try:

   - Using faster disks or perhaps using disk caching. Note that disk caching can be vulnerable to data loss unless backed up by adequate auxiliary power supply.
   - Spreading the load over more disks, for example, using separate disks for RTR journal I/O and for application I/O.
   - Getting more concurrency in RTR journal I/O by increasing the number of RTR server processes or threads. RTR can combine the data transfers for multiple transactions into a single disk I/O, if these are being processed concurrently.
   - Reducing the size of messages sent from RTR client to server (if the problem is data-rate rather than I/O rate). Client-to-server user data are written to the RTR journal file, so removing redundancy from messages may help lower the data rate on the journal disk.

3. Measure RTR network traffic generated by the application. Use RTR MONITOR TRAFFIC for this while the application is running under load. Add the total bytes/second sent and received, and subtract the bytes/second sent and received from the local node to itself (intra-node data does not use the network). This total should be substantially lower than the measured capacity of the network.

   A rough-and-ready way to measure available network capacity is to do a file- transfer of a large file using FTP or some other program between the nodes, and divide the file size by the time taken. Note

that multiple network connections may share the same hardware infrastructure, so you may need to try multiple simultaneous measurements between different node-pairs.

If the RTR network traffic measured is not substantially less than the measured capacity of the network, then this may be the cause of the performance constraint for which you are looking. Try:

- Using faster network hardware
- Reducing the size of the application messages sent with RTR
- Often, networks are tuned for high performance when transferring large files, but perform badly for bursty traffic. Buffering of either side of the transfer and of intermediate hops ensures smooth data flow. Check each hop to see if packets are being retransmitted due to excessive loss, and tune your network accordingly.

4. Measure delays in transmission through the network. Use "ping" to measure delay times between nodes whilst the system is under load. If reported round-trip delays are not in the low-millisecond range, you may be on to something. Additionally, use RTR MONITOR STALLS to measure whether delays are taking place in the acceptance of outgoing data by the network.

   If MONITOR STALLS shows a large number of stalls, especially in the columns for stalls longer than three seconds, then you very likely have a packet-loss problem in the network. Try:

   - Getting rid of that satellite link
   - Increasing the capacity of the network hardware
   - Checking that sufficient buffers are available for the network drivers on your machines
   - Upgrading or tuning network routers, bridges, and so on, if they are reporting packet losses.

     (If MONITOR STALLS reports lots of long stalls, but standard network analysis indicates that the network is operating as expected, check network utilization more closely. Packet losses which cause these glitches are usually caused by overload *peaks* in network traffic. You may still see disturbing long delays or link-losses when the system gets busy, even if *average* traffic is well below the capacity of the hardware.)

   Network monitors generally look at overall performance, measured over a period of time. It is often possible to show a 20 percent utilization of network bandwidth over time plotted at 5 minute intervals, but miss the peaks that last for 5 seconds and lose 50 packets. It is those 50 packets that account for the odd transaction getting a response time of 45 seconds instead of the usual 200 msec.

5. Check whether the throughput on your backend machines is being limited because all the servers are busy. Measure this by issuing the command RTR SHOW PARTITION/BACKEND /FULL on the backend machines. To observe this information with automatic updating of the display, use the MONITOR QUEUE or MONITOR GROUP command.

Note

Excessive use of a MONITOR command can be disruptive to the system. For example, running several MONITOR commands simultaneously steals cycles away from RTR to do real work. To minimize the impact of using MONITOR commands, increase the sample size interval using /INTERVAL= *no-of-seconds*.

If the SHOW PARTITION command consistently shows the number of "Free Servers" as zero and the number of "Txns Active" larger than the number of servers, then a performance problem may be caused by queues building up because an inadequate number of server applications are ready to process incoming transactions. Try the following:

- Fix any resource constraints limiting the application server's ability to do its work (CPU-load, disk-saturation, DB lock contention).
- Increase the number of server processes or threads, so more work can be done concurrently.

6. If none of the above results in the TPS-rate you would like to see, are you sure that you are generating enough work for the servers to do? To check this, try increasing the number of clients accessing the system.