VMS Software

VSI OpenVMS

VSI Reliable Transaction Router Getting Started

Document Number: DO-RTRGST-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 Getting Started

VMS Software

Copyright © 2024 VMS Software, Inc. (VSI), Boston, Massachusetts, USA

Legal Notice

Confidential computer software. Valid license from VSI required for possession, use or copying. Consistent with FAR 12.211 and 12.212, Commercial Computer Software, Computer Software Documentation, and Technical Data for Commercial Items are licensed to the U.S. Government under vendor's standard commercial license.

The information contained herein is subject to change without notice. The only warranties for VSI products and services are set forth in the express warranty statements accompanying such products and services. Nothing herein should be construed as constituting an additional warranty. VSI shall not be liable for technical or editorial errors or omissions contained herein.

HPE, HPE Integrity, HPE Alpha, and HPE Proliant are trademarks or registered trademarks of Hewlett Packard Enterprise.

Preface v

1. About VSI v
2. Intended Audience v
3. Document Structure v
4. Related Documentation v
5. Reading Path vi
6. VSI Encourages Your Comments vi
7. OpenVMS Documentation vii
8. Conventions vii

Chapter 1. Introduction 1

1.1. Reliable Transaction Router 1
1.2. RTR Continuous Computing Concepts 2
1.3. RTR Terminology 3
1.4. RTR Server Types 11
1.5. RTR Networking Capabilities 16

Chapter 2. Architectural Concepts 17

2.1. The Three-Tier Architecture 17
2.2. RTR Facilities Bridge the Gap 18
2.3. Broadcasts 18
2.4. Flexibility and Growth 18
2.5. Transaction Integrity 19
2.6. The Partitioned Data Model 19
2.7. Object-Oriented Programming 20
    2.7.1. Objects 21
    2.7.2. Messages 21
    2.7.3. Class Relationships 22
    2.7.4. Polymorphism 22
    2.7.5. Object Implementation Benefits 22
2.8. Java Support 23
2.9. XA Support 23

Chapter 3. Reliability Features 25

3.1. RTR Server Types 25
3.2. Failover and Recovery 25
    3.2.1. Router Failover 26
        3.2.1.1. Backend Restart Recovery 26
        3.2.1.2. Transaction Message Replay 26
        3.2.1.3. Link Failure Recovery 26
3.3. Recovery Scenarios 26
    3.3.1. Backend Recovery 26
    3.3.2. Router Recovery 26
    3.3.3. Frontend Recovery 26

Chapter 4. RTR Interfaces 29

4.1. Management Interfaces 29
4.2. Programming Interfaces 30
4.3. Application Development 31
4.4. RTR Management 31
    4.4.1. RTR Administrator 32
    4.4.2. RTR Manager 32
    4.4.3. RTR Explorer 33
    4.4.4. RTR Command Line Interface 34

4.4.5. Examples 35
4.5. Application Programming Interfaces 38
4.5.1. RTR Java Object-Oriented Interface 38
4.5.2. RTR C++ Object-Oriented Programming Interface 39
4.5.3. RTR C Programming Interface 42
Chapter 5. The RTR Environment 45
5.1. The RTR System Management Environment 45
5.1.1. Monitoring RTR 47
5.1.2. Transaction Management 47
5.1.3. Partition Management 48
5.2. The RTR Runtime Environment 48
5.3. What's Next? 49

Preface

This document introduces VSI Reliable Transaction Router and describes its concepts for the system manager, system administrator, and applications programmer.

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 purpose of this document is to assist an experienced system manager, system administrator, or application programmer understand the Reliable Transaction Router (RTR) product.

3. Document Structure

This document contains the following chapters:

- Chapter 1, Introduction to RTR, provides information on RTR technology, basic RTR concepts, and RTR terminology.
- Chapter 2, Architectural Concepts, introduces the RTR three-layer model and explains the use of RTR functions and programming capabilities.
- Chapter 3, Reliability Features, highlights RTR server types and failover and recovery scenarios.
- Chapter 4, RTR Interfaces, introduces the management and programming interfaces of RTR.
- Chapter 5, The RTR Environment, describes the RTR system management and runtime environments, and provides explicit pointers to further reading in the RTR documentation set.

4. Related Documentation

Additional resources in the RTR documentation kit include:

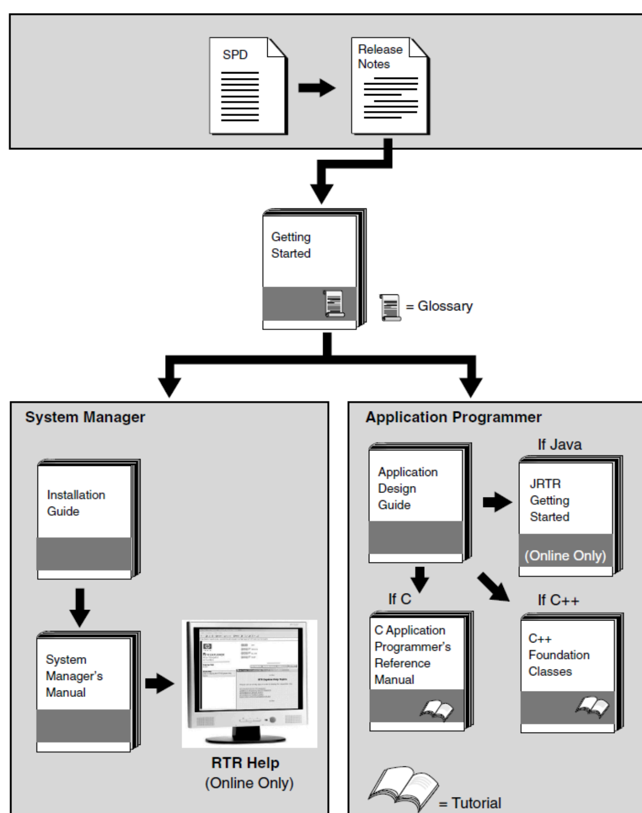
Document	Content
For all users:	
VSI Reliable Transaction Router Release Notes	Describes new features, changes, and known restrictions for RTR.
RTR *Commands*	Lists all RTR commands, their qualifiers and defaults.
For the system manager:	
VSI Reliable Transaction Router Installation Guide	Describes how to install RTR on all supported platforms.
VSI Reliable Transaction Router System Manager's Manual	Describes how to configure, manage, and monitor RTR.

Document	Content
Reliable Transaction Router Migration Guide	Explains how to migrate from RTR Version 2 to RTR Version 3 (OpenVMS only).
For the application programmer:	
VSI Reliable Transaction Router Application Design Guide	Describes how to design application programs for use with RTR, illustrated with both the C and C++ interfaces.
VSI Reliable Transaction Router C++ Foundation Classes	Describes the object-oriented C++ interface that can be used to implement RTR object-oriented applications.
VSI Reliable Transaction Router C Application Programmer's Reference Manual	Explains how to design and code RTR applications using the C programming language; contains full descriptions of the basic RTR API calls.

5. Reading Path

The reading path to follow when using the Reliable Transaction Router information set is shown in Figure 1.

Figure 1. RTR Reading Path



6. VSI Encourages Your Comments

You may send comments or suggestions regarding this manual or any VSI document by sending electronic mail to the following Internet address: `<docinfo@vmssoftware.com>`. Users who have

VSI OpenVMS support contracts through VSI can contact <support@vmssoftware.com> for help with this product.

7. OpenVMS Documentation

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

8. Conventions

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

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

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

The following conventions are also used in this manual:

Convention	Meaning
Ctrl/ *x*	A sequence such as **Ctrl/** *x* indicates that you must hold down the key labeled Ctrl while you press another key or a pointing device button.
PF1 *x*	A sequence such as PF1 *x* indicates that you must first press and release the key labeled PF1 and then press and release another key or a pointing device button.
Return	In examples, a key name enclosed in a box indicates that you press a key on the keyboard. (In text, a key name is not enclosed in a box.)
...	A horizontal ellipsis in examples indicates one of the following possibilities: • Additional optional arguments in a statement have been omitted. • The preceding item or items can be repeated one or more times. • Additional parameters, values, or other information can be entered.
. . .	A vertical ellipsis indicates the omission of items from a code example or command format; the items are omitted because they are not important to the topic being discussed.
( )	In command format descriptions, parentheses indicate that you must enclose the options in parentheses if you choose more than one.
[ ]	In command format descriptions, brackets indicate optional choices. You can choose one or more items or no items. Do not type the brackets on the command line. However, you must include the brackets in the syntax for OpenVMS directory specifications and for a substring specification in an assignment statement.

<table>
<tr><th>Convention</th><th>Meaning</th></tr>
<tr><td>[|]</td><td>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.</td></tr>
<tr><td>{ }</td><td>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.</td></tr>
<tr><td>bold text</td><td>This typeface represents the introduction of a new term. It also represents the name of an argument, an attribute, or a reason.</td></tr>
<tr><td>italic text</td><td>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).</td></tr>
<tr><td>UPPERCASE TEXT</td><td>Uppercase text indicates a command, the name of a routine, the name of a file, or the abbreviation for a system privilege.</td></tr>
<tr><td><code>Monospace type</code></td><td>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.</td></tr>
<tr><td>-</td><td>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.</td></tr>
<tr><td>numbers</td><td>All numbers in text are assumed to be decimal unless otherwise noted. Nondecimal radixes—binary, octal, or hexadecimal—are explicitly indicated.</td></tr>
</table>

Chapter 1. Introduction

This document introduces RTR and describes RTR concepts. It is intended for the system manager or administrator and for the application programmer who is developing an application that works with Reliable Transaction Router (RTR).

1.1. Reliable Transaction Router

Reliable Transaction Router (RTR) is failure-tolerant transactional messaging middleware used to implement large, distributed applications with client/server technologies. RTR helps ensure business continuity across multivendor systems and helps maximize uptime.

Failure and fault tolerance

Failure tolerance is supplied by the RTR software that enables an application to continue even when failures such as node or site outages occur. Failover is automatic. **Fault tolerance** is supplied by systems with hardware that is built with redundant components to ensure that processing survives failure of an individual component. Depending on system requirements and hardware, an RTR configuration can be both failure and fault tolerant.

Interoperability

You use the architecture of RTR to ensure high availability and transaction completion. RTR supports applications that run on different hardware and different operating systems. RTR applications can be designed to work with several database products including Oracle, Microsoft Access, Microsoft SQL Server, Sybase, and Informix. For specifics on operating systems, operating system versions, and supported hardware, refer to the *VSI Reliable Transaction Router Software Product Description* for each supported operating system.

Networking

RTR can be deployed in a local or wide area network and can use either TCP/IP or DECnet for its underlying network transport.

Transaction processing

A transaction involves the exchange of something for something else, for example, the exchange of cash for something purchased such as a book or restaurant meal. Transactions are the fodder of the commercial world in which we live. Transaction processing is the use of computers to do the bookkeeping for the physical transactions in which people engage. RTR is the premier transaction processing software available for many computer systems, offering unique benefits to ensure the integrity and correctness of transactions under its control.

Corporations and institutions benefit when the data they need is current and kept rapidly up-to-date. This ensures that they have increased control of their business or institution, and can react more effectively to changes in the business or institutional environment.

For example, an inventory management system provides current inventory for inquiry and can automatically request inventory updates when needed. Or an production manager can make production

decisions based on current status of manufacturing elements directly from the shop floor, and a credit card company can respond with accurate information when a card request for validation arrives. Or a shipping company can determine and report the location of a package in transit anywhere in the world, or a personnel system can enable an employee to update personal information online. All of these systems perform transaction processing tasks as designed by their developers.

With RTR, applications can be written to be deployed over a wide geography to take advantage of distributed resources, both computers and personnel. Implementing a transaction processing system using RTR requires analysis, planning, and considered execution.

1.2. RTR Continuous Computing Concepts

RTR provides a continuous computing environment that is particularly valuable in financial transactions, for example in banking, stock trading, or passenger reservations systems. RTR satisfies many requirements of a continuous computing environment:

- Reliability
- Failure tolerance
- Data and transaction integrity
- Scalability
- Ease of building and maintaining applications
- Interoperability with multiple operating systems

RTR additionally provides the following capabilities, essential in the demanding transaction processing environment:

- Flexibility
- Parallel execution at the transaction level
- Potential for step-by-step growth
- Comprehensive monitoring tools
- Management station for single console system management
- WAN deployability

RTR also ensures that transactions have the ACID properties that have been established as crucial in a transaction processing environment. A transaction with the ACID properties is:

- Atomic
- Consistent
- Isolated
- Durable

For more details on transactional ACID properties, see the discussion later in this document, and in the *VSI Reliable Transaction Router Application Design Guide*.

1.3. RTR Terminology

In addition to the terms previously defined, the following terms are either unique to RTR or redefined when used in the RTR context. If you have learned any of these terms in other contexts, take the time to assimilate their meaning in the RTR environment. These and other terms are also defined in the Glossary of this manual. The terms are described in the following order:

- Application
- Client, client application
- Server, server application
- Channel
- RTR configuration
- Roles
- Frontend
- Router
- Backend
- Facility
- Transaction
- Transaction controller
- Transactional messaging
- Nontransactional messaging
- Transaction ID
- Tier
- Standby server
- Transactional shadowing
- RTR journal
- Partition
- Key range
- XA

RTR Application

An RTR application is user-written software that executes within the confines of several distributed processes. The RTR application may perform user interface, business, and server logic tasks and is written in response to some business need. An RTR application can be written in one of the supported languages, C, C++, or Java and includes calls to RTR. RTR applications are composed of two kinds of actors, client applications and server applications. An application process is shown in diagrams as an oval, open for a client application (see Figure 1.1), filled for a server application (see Figure 1.2).

Client

A **client** is always a **client application**, one that initiates and demarcates a piece of work. In the context of RTR, a client must run on a node defined to have the frontend role. Clients typically deal with presentation services, handling forms input, screens, and so on. A client could connect to a browser running a browser *applet* or be a webserver acting as a gateway. In other contexts, a client can be a physical system, but in RTR and in this document, physical clients are called frontends or nodes. You can have more than one instance of a client on a node.

Figure 1.1. Client Symbol

Server

A **server** is always a *server application*, one that reacts to a client's units of work and carries them through to completion. This may involve updating persistent storage such as a database file, toggling a switch on a device, or performing another predefined task. In the context of RTR, a server must run on a node defined to have the backend role. In other contexts, a server can be a physical system, but in RTR and in this document, physical servers are called backends or nodes. You can have more than one instance of a server on a node. Servers can have partition states such as primary, standby, or shadow.

Figure 1.2. Server Symbol

Channel

RTR expects client and server applications to identify themselves before they request RTR services. During the identification process, RTR provides a tag or handle that is used for subsequent interactions. This tag or handle is called an RTR **channel**. A channel is used by client and server applications to exchange units of work with the help of RTR. An application process can have one or more client or server channels. Channel management is handled transparently by the C++ and Java APIs.

RTR configuration

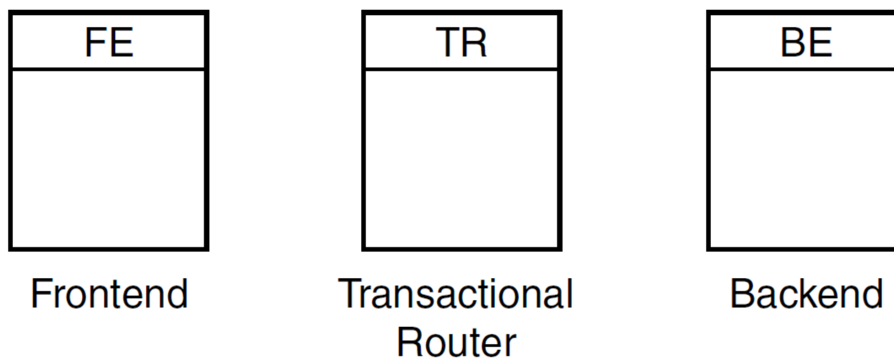
An RTR configuration consists of *nodes* that run RTR client and server applications. An RTR configuration can run on several operating systems including OpenVMS, Tru64 UNIX, and Windows NT among others (for the full set of supported operating systems, see the title page of this document, and the appropriate SPD). Nodes are connected by network *links*.

Roles

A node that runs client applications is called a **frontend** (FE), or is said to have the frontend role. A node that runs server applications is called a **backend** (BE). Additionally, the transaction **router** (TR) contains no application software but acts as a traffic cop between frontends and backends, routing transactions to the appropriate destinations. The router controls the distributed RTR nodes, and takes care of two-phase commit,failover and failback.

The router also eliminates any need for frontends and backends to know about each other in advance. This relieves the application programmer from the need to be concerned about network configuration details. The router can reside on a node running as a frontend or a backend but is often run on a node where neither backends nor frontends are running. Figure 1.3 shows the symbol for each of the RTR roles.

Figure 1.3. Roles Symbols

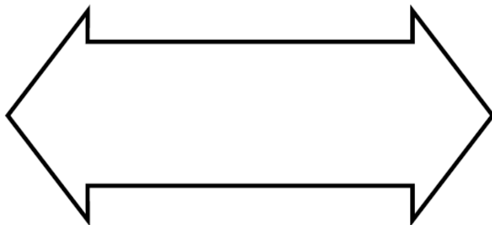


Facility

The mapping between nodes and roles is done using a facility. An RTR facility is the user-defined name for a particular configuration whose definition provides the role-to-node map for a given application. The facility symbol (see Figure 1.4) illustrates its use in the RTR environment. Nodes can share several facilities. The role of a node is defined within the scope of a particular facility. Normally a facility is defined across all roles but facility definition depends on application design

The router is the only role that knows about all three roles. A router can run on the same physical node as the frontend or backend, if that is required by configuration constraints,but such a setup would not take full advantage of failover characteristics.

Figure 1.4. Facility Symbol

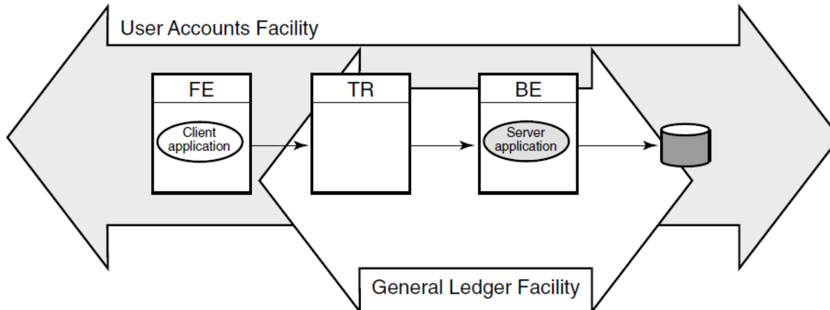


A facility name is mapped to specific physical nodes and their roles using the CREATE FACILITY command.

Figure 1.5 shows the logical relationship between client application, server application, frontends (FEs), routers (TRs), and backends (BEs) in the RTR environment. The database is represented by the cylinder.

Two facilities are shown (indicated by the large double-headed arrows), the User Accounts Facility and the General Ledger Facility. The User Accounts Facility uses three nodes, FE, TR,and BE, while the General Ledger Facility uses only two, TR and BE in the configuration shown. Its FEs are on nodes not shown in the figure, at another location

Figure 1.5. Components in the RTR Environment



Clients send messages to servers to ask that a piece of work be done. Such requests may be bundled together into transactions. An RTR transaction consists of one or more messages that have been grouped together by a client application, so that the work done as a result of each message can be undone completely, if some part of that work cannot be done. If the system fails or is disconnected before all parts of the transaction are done, then the transaction remains incomplete.

Transaction

A **transaction** is a piece of work or group of operations that must be executed together to perform a consistent transformation of data. This group of operations can be distributed across many nodes serving multiple databases. Applications use services that RTR provides.

Because typically a transaction consists of several operations, a system or network failure at any step in the process will cause the transaction to be in doubt. RTR ensures that all transactions have the ACID properties so that all transactions are in a known state. (See the description of “Transactional messaging” for further clarification of transaction integrity.)

Transaction controller

With the C++ API, the Transaction Controller manages transactions (one at a time), channels, messages, and events.

Transactional messaging

RTR provides transactional messaging in which transactions are enclosed in messages controlled by RTR.

Transactional messaging ensures that each transaction is complete, and not partially recorded. For example, a transaction or business exchange in a bank account might be to move money from a checking account to a savings account. The complete transaction is to remove the money from the checking account and add it to the savings account.

A transaction that transfers funds from one account to another consists of two individual updates: one to debit the first account, and one to credit the second account. The transaction is not complete until both actions are done. If a system performing this work goes down after the money has been debited from the checking account but before it has been credited to the savings account, the transaction is incomplete. With transactional messaging, RTR ensures that a transaction is “all or nothing” – either fully completed

or discarded; either both the checking account debit and the savings account credit are done, or the checking account debit is backed out and not recorded in the database. RTR transactions have the ACID properties.

Nontransactional messaging

An application will also contain nontransactional tasks such as writing diagnostic trace messages or sending a *broadcast* message about a change in a stock price after a transaction has been completed.

Transaction ID

Every transaction is identified on initiation with a transaction identifier or *transaction ID*, with which it can be logged and tracked. RTR guarantees that TIDs are unique.

To reinforce the use of these terms in the RTR context, this section briefly reviews other uses of configuration terminology.

Tiers

A traditional two-tier client/server environment is based on hardware that separates application presentation and business logic (the clients) from database server activities. The client hardware runs presentation and business logic software, and server hardware runs database or data manager (DM) software,also called resource managers (RM). This type of configuration is illustrated in Figure 1.6. (In all diagrams, all lines are actually bidirectional, even when represented otherwise for clarity. Fora given transaction, the initial action is typically from left to right.) In Figure 1.6, Application Presentation and Business Logic are the first tier, and the Database Server is the second tier.

Further separation into three tiers is achieved by separating presentation software from business logic on two systems,and retaining a third physical system for interaction with the database. This is illustrated in Figure 1.7, where Presentation and User Interface are the first tier, the Application Server and Business Logic are the second tier, and the Database Server is the third tier.

Figure 1.6. Two-Tier Client/Server Environment

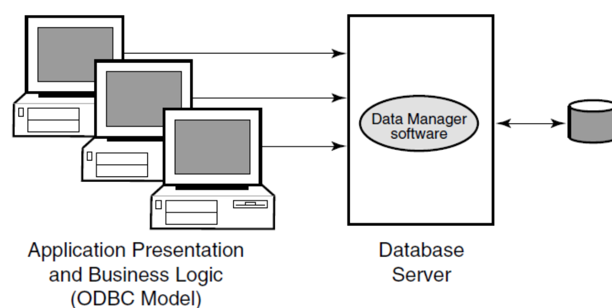
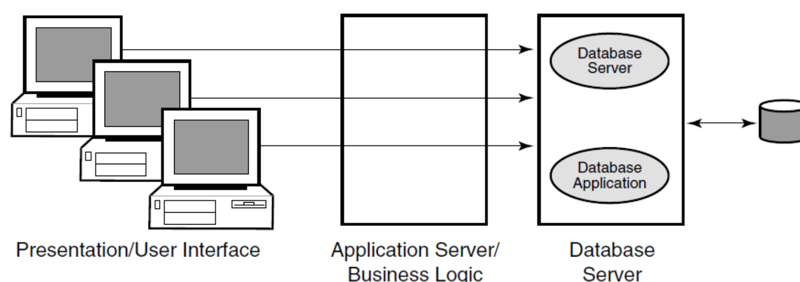


Figure 1.7. Three-Tier Client/Server Environment



RTR extends the three-tier model, which is based on hardware, to a multilayer, or multicomponent software model.

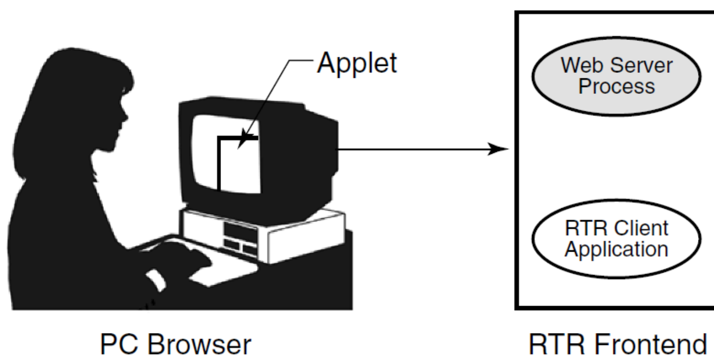
RTR SoftwareComponents

RTR provides a multicomponent software model where clients running on frontends, routers, and servers running on backends cooperate to provide reliable service and transactional integrity. Application users interact with the client (presentation layer) on the frontend node that forwards messages to the current router. The router in turn routes the messages to the current, appropriate backend, where server applications reside, for processing. The connection to the current router is maintained until the current router fails or connections to it are lost.

All RTR software components can reside on a single node but are typically deployed on different nodes to achieve modularity,scalability, and redundancy for availability. During initial application development, it can be convenient to use a single physical node for all RTR roles and application software.

With different physical systems, if one node goes down or offline, another router or backend node can take over application processing. In a slightly different configuration, you could have an application that uses an external applet component running on a browser that connects to a client running on the RTR frontend. Such a configuration is shown in Figure 1.8. In the figure, the applet is separate from but connects to the client application written to work directly with RTR.

Figure 1.8. Browser Applet Configuration



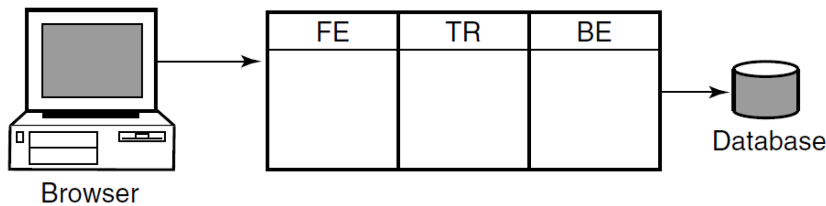
The RTR client application could be an ASP (Active Server Page) script or a process interfacing to the webserver through a standard interface such as CGI (Common Gateway Interface) script.

RTR provides automatic software failure tolerance and failure recovery in multinode environments by sustaining transaction integrity in spite of hardware, communications, application, or site failures. Automatic failover and recovery of service can exploit redundant or underutilized hardware and network links.

For example, you could use an underutilized system as a standby server in certain configurations.

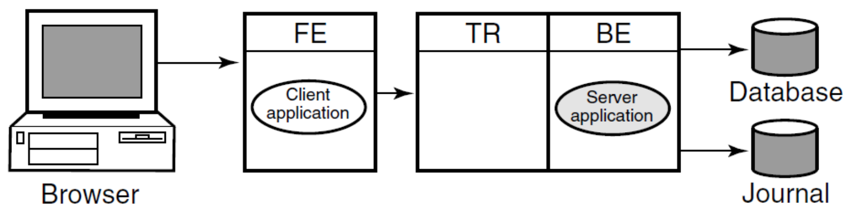
As you modularize your application and distribute its components on frontends and backends, you can add new nodes, identify usage bottlenecks, and provide redundancy to increase availability. Adding backend nodes can help divide the transactional load and distribute it more evenly. For example, you could have a single node configuration as shown in Figure 1.9: RTR with Browser, Single Node, and Database. A single node configuration can be useful during development, but would not normally be used when your application is deployed.

Figure 1.9. RTR with Browser, Single Node, and Database



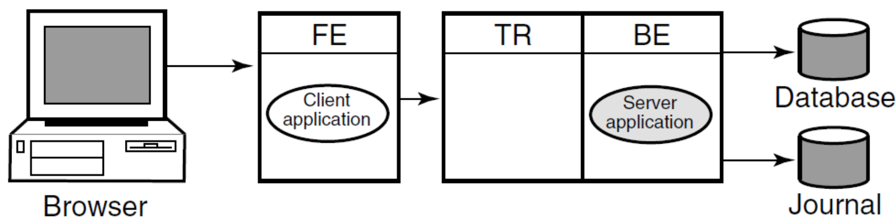
When applications are deployed, often the frontend is separated from the backend and router, as shown in Figure 1.10.

Figure 1.10. RTR Deployed on Two Nodes



In this example, the frontend with the client application resides on one node, and the router with the server application reside a node that has both the router and backend roles. This is a typical configuration where routers are placed on backends rather than on frontends. A further separation of workload onto three nodes is shown in Figure 1.11. However, in this configuration, there remain several single points of failure where one node/role or a network outage can disrupt processing of transactions.

Figure 1.11. RTR Deployed on Three Nodes



While this three-node configuration separates transaction load onto three nodes, it does not provide for continuing work if one of the nodes fails or becomes disconnected from the others. In many applications, there is a need to ensure that there is a server always available to access the database.

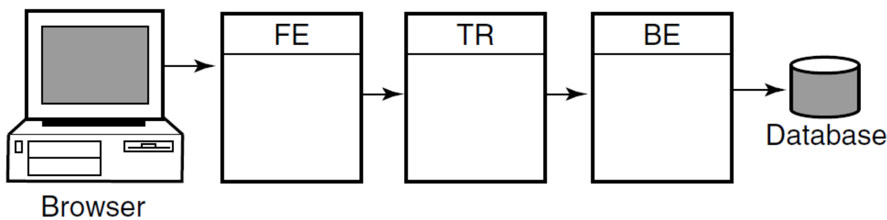
Standby server

In this case, a **standby server** will do the job. A standby server(see Figure 1.12) is a process or application that can take over when the primary server is not available, due to hardware failure, application software failure or network outage.

Both the primary and the standby server have the capability to access the same database, but the primary processes all transactions unless it is unavailable. On the other hand, the standby processes transactions only when the primary becomes unavailable. When not being used to process transactions, the standby CPU can do other work.

The standby server is usually placed on a node other than the node where the primary server runs, and should be, to avoid being a single point of failure. Network capability, clustering or disk-sharing technology, and appropriate software must be available on both primary and standby backend systems when running RTR.

Figure 1.12. Standby Server Configuration

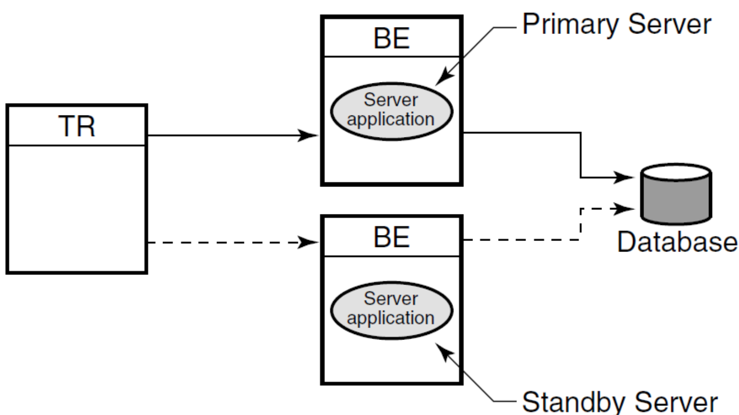


Shadow Server and Transactional Shadowing

To increase transaction processing availability, transactions can be shadowed with a shadow server, as shown in Figure 1.13.The system where the shadow server runs can be made available with clustering technology. A shadow server eliminates the single point of failure that is evident in Figure 1.12. In a shadow configuration, the second database of Figure 1.13 is available even when the first is not.

Use of a shadow server is called transactional shadowing and is accomplished by having a second location, often at a different site, where transactions are also recorded. Data are recorded in two separate data stores or databases. The router knows about both backends and sends all transactions to both backends. RTR provides the server application with the necessary information to keep the two databases synchronized.

Figure 1.13. Transactional Shadowing Configuration



RTR Journal

In the RTR environment, one data store (database or data file) is elected the primary, and a second data store is made the shadow. The shadow data store is a copy of the data store kept on the primary. If either data store becomes unavailable, all transactions continue to be processed and stored on the surviving data store. At the same time, RTR makes a record of (remembers) all transactions stored only on the shadow data store in the **RTR journal** by the shadow server.

When creating the configuration used by an application and defining the nodes where a facility has its frontends, routers,and backends, the setup must also define which nodes will have journal files. Each backend in an RTR configuration must have a journal file to capture transactions when other nodes are unavailable. When the primary server and data store become available again, RTR replays the transactions in the journal to the primary data store through the primary server. This brings the data store back into synchronization.

With transactional shadowing, there is no requirement that hardware, the data store, or the operating system at different sites be the same. You could, for example, have one site running OpenVMS and

another running Windows; the RTR transactional commit process would be the same at each site. Because the database resides at both sites, either backend can have an outage and all transactions will still be processed and recovered.

Note

Transactional shadowing shadows only transactions controlled by RTR.

For full redundancy to assure maximum availability, a configuration could employ **disk shadowing** in clusters at separate sites coupled with **transactional shadowing** across sites. Disk shadowing used in specific cluster environments copies data to another disk to ensure data availability. Transactional shadowing copies only transactional data.

Additionally, an RTR configuration typically deploys multiple frontends running client applications with connections to several routers to ensure continuing operation if a particular router fails.

1.4. RTR Server Types

In the RTR environment, in addition to the placement of frontends, routers, and servers, the application designer must determine what server capabilities to use. RTR provides four types of software servers for application use:

- Standby servers
- Transactional shadow servers
- Concurrent servers
- Callout servers

These are described in the next few paragraphs. You specify server types to your application in RTR API calls.

RTR server types help to provide continuous availability and a secure transactional environment.

Standby server

The **standby server** remains idle while the RTR primary backend server performs its work, accepting transactions and updating the database. When the primary server fails, the standby server takes over, recovers any in-progress transactions, updates the database, and communicates with clients until the primary server returns. There can be many instances of a standby server. Activation of the standby server is transparent to the user.

A typical standby configuration is shown in Figure 1.12: Standby Server Configuration. Both physical servers running the RTR backend software are assumed by RTR to connect to the same database. The primary server is typically in use, and the standby server can be either idle or used for other applications, or data partitions, or facilities. When the primary server becomes unavailable, the standby server takes over and completes transactions as shown by the dashed line. Primary server failure could be caused by server process failure or backend (node) failure.

Standby in a cluster

The intended and most common use of a standby server is in a recognized cluster environment. In a noncluster or unrecognized cluster environment, seamless failover of standbys is not guaranteed. For

RTR, clusters supported by OpenVMS and Tru64 UNIX are recognized clusters, whose processing is controlled by a lock manager. Windows and Sun clusters can use disk-sharing, unrecognized cluster technology.

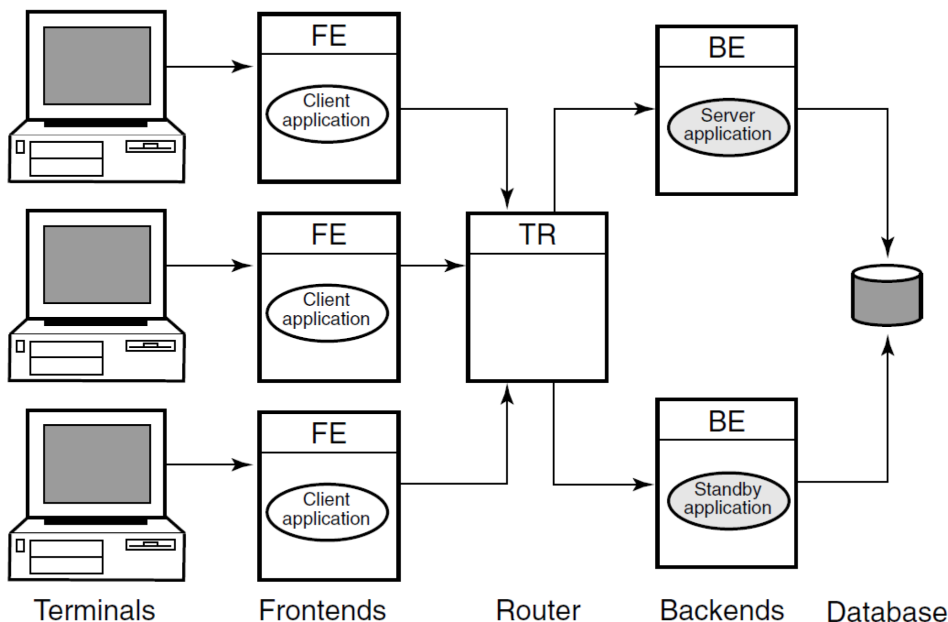
Standby servers are “spare” servers which automatically take over from the main backend if it fails. This takeover is transparent to the application.

Figure 1.14 shows a simple standby configuration. The two backend nodes are members of a cluster environment, and are both able to access the database.

For any one key range, the main or primary server (Server application) runs on one node while the standby server (Standby application) runs on the other node. The standby server process is running, but RTR does not pass any transactions to it. Should the primary node fail, RTR starts passing transactions to the Standby application.

Note that one node can contain the primary servers for one key range and standby servers for another key range to balance the load across systems. This allows the nodes in a cluster environment to act as standby for other nodes without having idle hardware. When setting up a standby server, both servers must have access to the same journal.

Figure 1.14. Standby Servers



Transactional shadow server

The **transactional shadow server** places all transactions recorded on the primary server on a second database. The transactional shadow server can be at the same site or at a different site, with networking capability available.

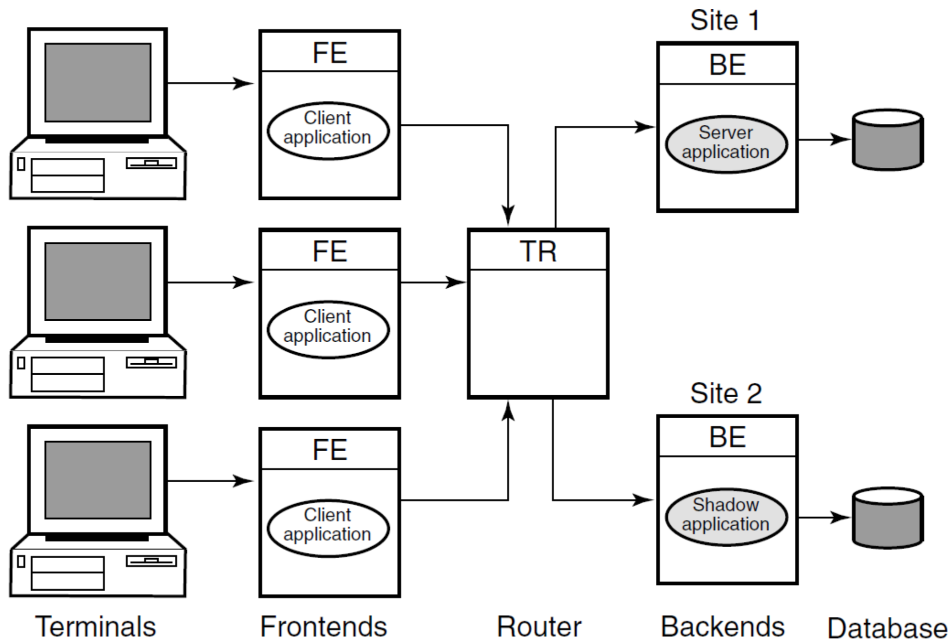
When one member of a shadow set fails, RTR remembers the transactions executed at the surviving site in a journal,and replays them when the failed site returns. Only after all journaled transactions are recovered does the recovering site fully process new online transactions. During recovery, new transactions are processed at the surviving site and added to the journal for the recovering site.

Transactional shadowing is done by partition. A transactional shadow configuration can have only two members of the shadowset.

Shadow servers are servers on separate backends that handle the same transactions in parallel on identical copies of the database.

Figure 1.15 shows a simple shadow configuration. The main backend server application at Site 1 and the shadow server(Shadow application) at Site 2 both receive every transaction for the data partition they are servicing. Should Site 1 fail,Site 2 continues to operate without interruption. Sites can be geographically remote, for example, available at separate locations in a wide area network (WAN).

Figure 1.15. Shadow Servers

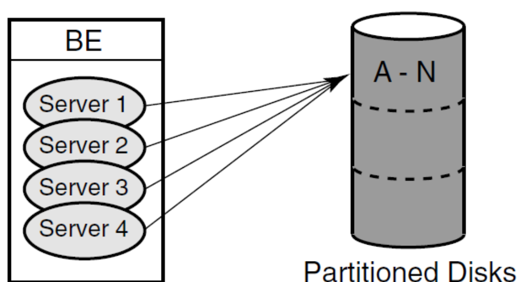


Concurrent server

The **concurrent server** is an additional instance of a server application running on the same node. RTR delivers transactions to a free server from the pool of concurrent servers. If one server fails, the transaction in process is replayed to another server in the concurrent pool. Concurrent servers are designed primarily to increase throughput and can exploit Symmetric Multiprocessing (SMP) systems. Figure 1.16: Concurrent Servers, illustrates the use of concurrent servers sending transactions to the same partition on a backend, the partition A-N.

Concurrent servers allow transactions to be processed in parallel to increase throughput. Concurrent servers deal with the same database partition, and may be implemented as multiple channels within a single process or as one channel in separate processes. The application designer must determine if transactions can be processed concurrently by the database or server application. Deadlocks can occur if every transaction competes for the same database lock outside the RTR server application.

Figure 1.16. Concurrent Servers

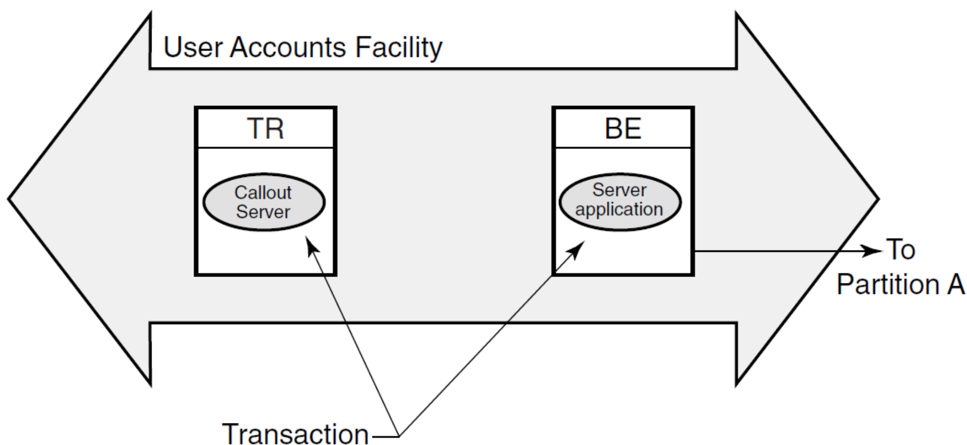


Callout server

The **callout server** enables message authentication on transaction requests made in a given facility, and could be used, for example, to provide audit trail logging. A callout server can run on either backend or router nodes. A callout server receives a copy of all messages in a facility. Because the callout server votes on the outcome of each transaction it receives, it can veto any transaction that does not pass its security checks.

A callout server is facility based, not partition based; any message arriving at the facility is routed to both the server and the callout. A callout server is enabled when the facility is defined. Figure 1.17 illustrates the use of a callout server that authenticates every transaction in a facility.

Figure 1.17. A Callout Server



To authenticate any part of a transaction, the callout server must vote on the transaction, but does not write to the database. RTR does not replay a transaction that is only authenticated.

Authentication

RTR callout servers provide partition-independent processing for authentication. For example, a callout server can enable checks to be carried out on all requests in a given facility.

Callout servers run on backend or router nodes. They receive a copy of every transaction either delivered to or passing through the node.

Callout servers offer the following advantages:

- The security check can run in parallel with the database updates thus improving response times.
- The security check can be run on the router hardware.
- The security checking code is completely separated from other application code.

Since this technique relies on backing out unauthorized transactions, it is most suitable when only a small proportion of transactions are expected to fail the security check, so as not to have a performance impact.

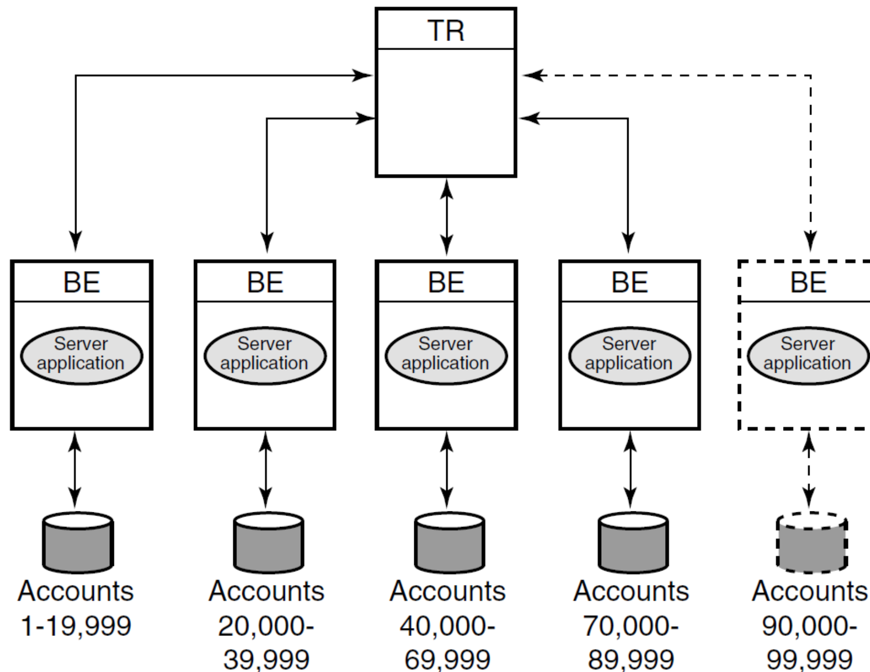
Partition

When working with database systems, partitioning the database can be essential to ensuring smooth and untrammeled performance with a minimum of bottlenecks. When you **partition** your database, you

locate different parts of your database on different disk drives to spread both the physical storage of your database onto different physical media and to balance access traffic across different disk controllers and drives.

For example, in a banking environment, you could partition your database by account number, as shown in Figure 1.18. A partition is a segment of your database.

Figure 1.18. Bank Partitioning Example



Key range

Once you have decided to partition your database, you use key ranges in your application to specify how to route transactions to the appropriate database partition. A **key range** is the range of data held in each partition. For example, the key range for the first partition in the bank partitioning example goes from 00001 to 19999.

You can assign a partition name in your application program or have it set by the system manager. Note that sometimes the terms key range and partition are used as synonyms in code examples and samples with RTR, but strictly speaking, the key range defines the partition. A partition has both a name, its partition name, and an identifier generated by RTR — the partition ID. The properties of a partition (callout, standby, shadow, concurrent, key segment range) can be defined by the system manager with a CREATE PARTITION command. For details of the command syntax, see the *VSI Reliable Transaction Router System Manager's Manual*.

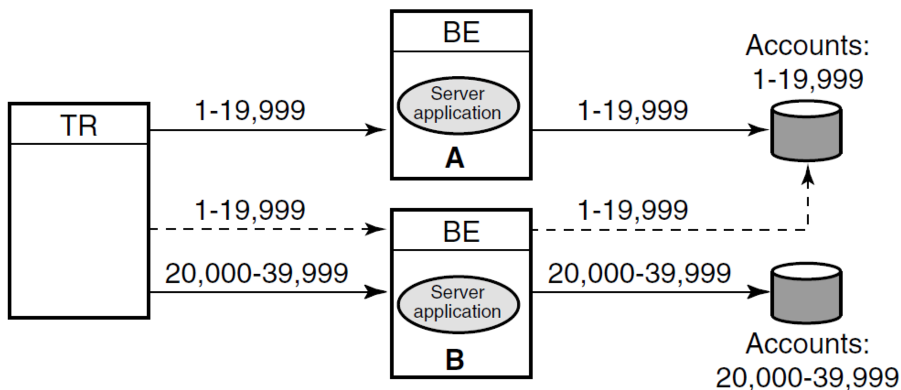
A significant advantage of the partitioning shown in the bank example is that you can add more account numbers without making changes to your application; you need only add another server and disk drive for the new account numbers. For example, say you need to add account numbers from 90,000 to 99,999 to the basic configuration of Figure 1.18: Bank Partitioning Example. You can add these accounts and bring them on line easily. The system manager can change the key range with a command, for example, in an overnight operation, or you can plan to do this during scheduled maintenance.

A partition can also have multiple standby servers.

Standby Server Configurations

A node can be configured as a primary server for one key range and as a standby server for another key range. This helps to distribute the work of the standby servers. Figure 1.19 illustrates this use of standbys with distributed partitioning. As shown in Figure 1.19, Application Server A is the primary server for accounts 1 to 19,999 and Application Server B is the standby for these same accounts. Application Server B is the primary for accounts 20,000 to 39,999 and Application Server A can be the standby for these same accounts (not shown in the figure). For clarity, account numbers are shown only for primary servers and one standby server.

Figure 1.19. Standby with Partitioning



Anonymous clients

RTR supports anonymous clients, that is, clients can be set up in a configuration using wildcarded node names.

Tunnel

RTR can also be used with firewall tunneling software, which supports secure internet communication for an RTR connection, either client-to-router, or router-to-backend.

1.5. RTR Networking Capabilities

Depending on operating system, RTR uses TCP/IP or DECnet as underlying transports for the virtual network (RTR facilities) and can be deployed in both local area and wide area networks. PATHWORKS 32 is required for DECnet configurations on Windows NT.

Chapter 2. Architectural Concepts

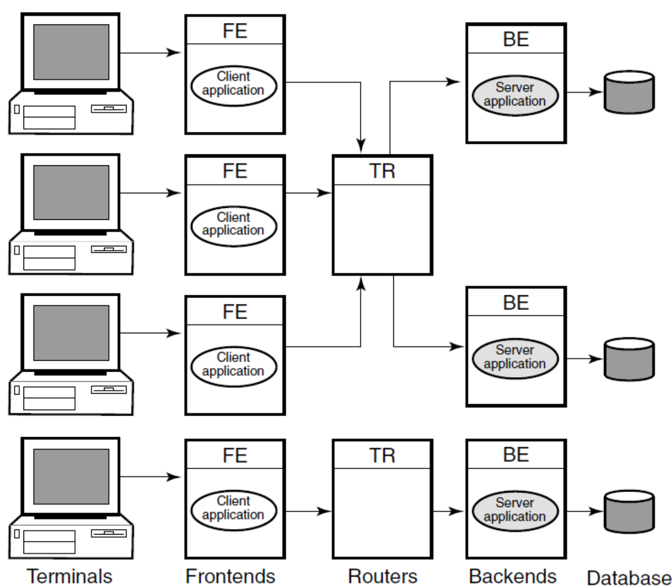
This chapter introduces concepts on basic transaction processing and RTR architecture.

2.1. The Three-Tier Architecture

RTR is based on a three-tier physical architecture consisting of frontend (FE) roles, backend (BE) roles and router (TR) roles. The roles are shown in Figure 2.1. (In this and subsequent diagrams, rectangles represent physical nodes, ovals represent application software, and cylinders represent the disks storing the database. The nodes connected to the actual database usually run the database software that controls the database.)

In addition to the physical configuration where RTR is deployed, software plays a critical part, extending the tier concept to more than three tiers. On certain pieces of hardware, client application software runs, and on others, server application software runs. Users can connect to nodes that are running the frontend role with appropriate non-RTR software. For example, a user can have a PC where RTR runs; in this case, the PC has the frontend role. Or a user could use a PC running, say Pathworks, to connect to another node that has the frontend role and run the RTR client application from there. This would be a multitier configuration.

Figure 2.1. The Multitier Model



Client application processes run on nodes defined to have the frontend role. This tier allows computing power to be provided locally at the end-user site for transaction acquisition and presentation.

Server processes (represented by “Server application” in Figure 2.1) run on nodes defined to have the backend role. This tier:

- Allows the database to be distributed geographically
- Permits replication of servers to cope with network, node or site failures
- Allows computer resources to be added to meet performance requirements
- Allows performance or geographic expansion while protecting the investments made in existing hardware and application software

The router tier contains no application software unless running callout servers. This tier reduces the number of logical network links required on frontend and backend nodes and helps ensure good performance even in an unstable network. It also decouples the backend tier from the frontend tier so that configuration changes in the (frequently changing) user environment have little influence on the transaction processing and database (backend) environment.

The three-tier model can be mapped to any system topology. More than one role may be assigned to any particular node. For example, on a system with few frontends, the router and backend tiers can be combined in the same nodes. During application development and test, all three roles can be combined in one node.

The nodes used by an application and their configuration roles are specified using RTR configuration commands. RTR lets application code be completely location and configuration independent.

2.2. RTR Facilities Bridge the Gap

Many applications can use RTR at the same time without interfering with one another. This is achieved by defining a separate facility for each application. A facility can be thought of as an application network.

When an application calls the `rtr_open_channel()` routine to declare a channel as a client or server, it specifies the name of the facility it will use.

Refer to the *VSI Reliable Transaction Router System Manager's Manual* for information on how to define facilities.

2.3. Broadcasts

Sometimes an application has a requirement to send unsolicited messages to multiple recipients.

An example of such an application is a commodity trading system, where the clients submit orders and also need to be informed of the latest price changes.

The RTR broadcast capability meets this requirement.

Recipients subscribe to a class of broadcasts; a sender broadcasts a message in this class, all interested recipients receive the message. However, broadcast reception is not guaranteed; network or node outages can cause a particular client to fail to receive a broadcast message.

RTR permits clients to broadcast messages to one or more servers, or servers to broadcast to one or more clients. If a server needs to broadcast a message to another server, it must open a second channel as a client.

2.4. Flexibility and Growth

RTR allows you to cope easily with changes in:

- Network demand
- User access patterns
- The volume of data

Since an RTR-based system can be built using multiple systems at each functional layer, it easily lends itself to step-by-step growth, avoiding unused capacity at each stage. With your system still up and running, it is possible to:

- Create and delete concurrent server processes.
- Add or remove nodes (frontend, router or backend).

This means you do not need to provide spare capacity to allow for growth.

RTR also allows parallel execution. This means that different parts of a single transaction can be processed in parallel by multiple servers.

RTR provides a comprehensive set of monitoring tools to help you evaluate the volume of traffic passing through the system. This can help you respond to unexpected load changes by altering the system configuration dynamically.

2.5. Transaction Integrity

RTR greatly simplifies the design and coding of distributed applications, because, with RTR, database actions can be bundled together into transactions.

To ensure that your application deals with transactions correctly,its transactions must have the ACID properties, fundamental properties of transaction processing systems. A transaction that has the ACID properties is:

- Atomic
- Consistent
- Isolated
- Durable

An atomic transaction is all or nothing; that is, either the entire transaction is totally committed or totally rolled back. A consistent transaction either creates a new, valid state of data,or, from any failure, returns all data to its state as it was before the start of the transaction. An isolated transaction does not cause changes to shared resources until commitment of the transaction. A durable transaction survives system and media failures after transaction commitment. A durable transaction is thus both persistent and stable.

For more detail on these properties and their use in transaction processing, refer to the *VSI Reliable Transaction Router Application Design Guide*.

2.6. The Partitioned Data Model

One goal in designing for high transaction throughput is reducing the time that users must wait for shared resources.

While many elements of a transaction processing system can be duplicated, one resource that must be shared is the database. Users compete for a shared database in three ways:

- For use of the disk
- For locks on database records
- For the CPU resources needed to access the database

This competition can be alleviated by spreading the database across several backend nodes, each node being responsible for a subset of the data, or partition. RTR enables you to implement this partitioned data model, shown roughly in Figure 2.2 where the database has three partitions. RTR routes messages to the correct partition on the basis of an application-defined key. For a more complete description of partitioning as provided with RTR, refer to the *VSI Reliable Transaction Router Application Design Guide*.

Each RTR API provides the capability to use partitions. For specific information on declaring and using partitions, refer to the RTR documentation for the system manager and the applications programmer.

2.7. Object-Oriented Programming

Java objects and the RTR the C++ foundation classes map traditional RTR functional programming concepts into an object-oriented programming model. Using the power and features of these foundation classes requires a basic understanding of the differences between functional and object-oriented programming concepts. Table 2.1 compares the worlds of functional programming and object-oriented programming.

Figure 2.2. Partitioned Data Model

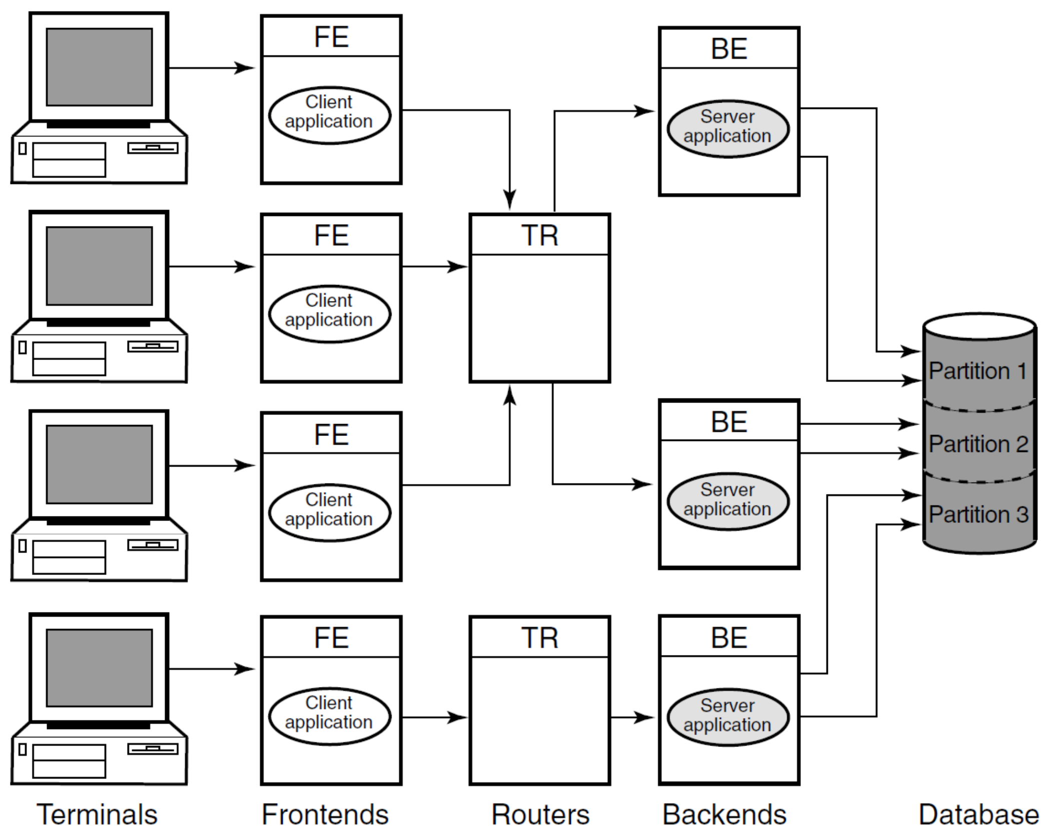


Table 2.1. Functional and Object-Oriented Programming Compared

Functional Programming	Object-Oriented Programming
A program consists of data structures and algorithms.	A program consists of a team of cooperating objects.
The basic programming unit is the function, that when run, implements an algorithm.	The basic programming unit is the class, that when instantiated, implements an object.

Functional Programming	Object-Oriented Programming
Functions operate on elemental data types or data structures.	Objects communicate by sending messages.
An application's architecture consists of a hierarchy of functions and sub-functions.	An applications architecture consists of objects that model entities of the problem domain. Objects' relationships can vary.

2.7.1. Objects

In the object-oriented environment, a program or application is a grouping of cooperating objects. The basic programming unit is the class. Instantiating, or declaring an instance of, a class implements an object. RTR provides object-oriented programming capabilities with the C++ API, described in the *VSI Reliable Transaction Router C++ Foundation Classes*. Objects are instances of a class. In a transaction class, each transaction is an object. An object is an instantiated (declared) class. Its state and behavior are determined by the attributes and methods defined in the class. An object or class is defined by its:

- State (attributes)
- Behavior (methods)
- Identity (name at instantiation)

The name given at object declaration is its identity. In Example 2.1, the two dog objects King and Fifi are instances of Dog. The Dog class is declared in a header (Dog.h) file and implemented in a .cpp file.

Example 2.1. Objects-Defined Sample

```
Dog.h:
class Dog
{ ...
};
main.cpp:
#include "Dog.h"
main()
{
   Dog King;
   Dog Fifi;
}
```

2.7.2. Messages

Objects communicate by sending messages. This is done by calling an object's methods.

Some principal categories of messages are:

- Constructors: Create objects
- Destructors: Delete objects
- Selectors: Return part or all of an object's state. For example, a Get method
- Modifiers: Change part or all of an object's state. For example, a Set method
- Iterators: Access multiple element objects within a container object. For example, an array.

2.7.3. Class Relationships

Classes can be related in the following ways:

- Simple association: One class is aware of another class. For example, "Dog object is associated with a Master object." This is a "Knows a" relationship.
- Composition: One class contains another class as part of its attributes. For example, "Dog objects contains Leg objects." This is a "Has a" relationship.
- Inheritance A child class is derived from one or more parent, or base, classes. For example, "Mutt object derives from Collie object and Boxer object which both derive from Dog object." This is an "Is a" relationship. Inheritance enables the use of polymorphism.

2.7.4. Polymorphism

Polymorphism is the ability of objects, inherited from a common base or parent class, to respond differently to the same message. This is done by defining different implementations of the same method name within the individual child class definitions. For example: A DogArray object, "DogArray OurDogs[2];" refers to two element objects of class Dog, the base class:

- King, of class Doberman, is a derived or child class of Dog.
- Fifi, of class Minipoodle, is a derived or child class of Dog.

If, in a program, OurDogs[n]->Bark() is called in a loop, then:

- In iteration one ([1]), method King::Bark() is called.
- In iteration two ([2]), method Fifi::Bark() is called.

King's bark does not sound like Fifi's bark because each Bark() call is a separately defined method within its child object definition. The virtual parent class (Dog) method Bark() is defined in the class definition of Dog.

2.7.5. Object Implementation Benefits

The benefits of creating RTR solutions with C++ foundation classes include the following:

- Each major RTR concept is represented by its own individual foundation class.
- Simple methods within RTR classes transform features of RTR for streamlined solutions.
- Major classes include Get and Set methods for changing transaction states.
- Default handling code is provided for all Messages and Events, where appropriate.
- You do not need to provide handling code for all messages and events.
- The sending and receiving of data is abstracted to a higher level with transaction controller and data classes.
- No buffers and links coding is needed.
- Internal RTR information is accessible without a need to know RTR internals.

2.8. Java Support

RTR clients and servers can be Java applications that obtain the benefits of high availability, fault tolerance and scalability provided by RTR. RTR clients and servers employing Java technology use standard Java and J2EE interfaces for transaction management, data input/output, and database access.

For additional information, see the *VSI Reliable Transaction Router Getting Started* manual and associated online documentation.

2.9. XA Support

Within its C API, RTR provides the capability of using the XA interface to work with XA-compliant database systems. The XA interface is part of the X/Open DTP (Distributed Transaction Processing) standard. It defines the interface that transaction managers (TM) and resource managers (RM) use to perform the two-phase commit protocol. (Resource managers are underlying database systems such as ORACLE RDBMS, Microsoft SQL Server, and others.) This interface is used by TM-to-RM exchanges to coordinate a transaction from within an application program.

If your database application supports XA, you have less to implement in your application environment; use of XA can also increase the portability of your application.

For details on using XA with RTR, refer to the *VSI Reliable Transaction Router C Application Programmer's Reference Manual* and the *VSI Reliable Transaction Router Application Design Guide*.

Chapter 3. Reliability Features

This chapter addresses:

- RTR server types
- Failover and recovery
- Recovery scenarios

3.1. RTR Server Types

Reliability in RTR is enhanced by the use of:

- Concurrent servers
- Standby servers
- Shadow servers
- Callout servers
- Router failover

Note that, conceptually, servers can be contrasted as follows:

- Concurrent servers handle *similar* transactions which access the same data partition and run on the same node.

  When transaction throughput is constrained by your server application, consider adding a second instance of your server application with a concurrent server.

- Shadow servers handle the *same* transactions and run on different nodes.

  When there is concern that your database is a single point of failure, add a shadow server, if possible at a different physical location.

- Standby servers provide a node that can take over processing on a data partition when the primary server or node fails.

  When there is concern that your server application or the node where it is running is a single point of failure in your configuration, configure a standby server to be ready to take over.

- Callout servers run on backends or routers and receive all messages within a facility so that authentication and logging operations can be performed in parallel.

  Use a callout server to add processing logic (authentication or logging) to your transactions without modifying your server application.

All servers are further described in the earlier section on RTR Terminology.

3.2. Failover and Recovery

RTR provides several capabilities to ensure failover and recovery under several circumstances.

3.2.1. Router Failover

Frontend nodes automatically connect to another router if the one being used fails. This reconnection is transparent to the application.

Routers are responsible for coordinating the two-phase commit for transactions. If the original router coordinating a transaction fails, backend nodes select another router that can ensure correct transaction completion.

3.2.1.1. Backend Restart Recovery

Transactions in the process of being committed at the time of a failure are recovered from RTR's disk journal. Recovery could be with a concurrent server, a standby server, or a restarted server created when the failed backend restarts.

Correct ordering of the execution of transactions against the database is maintained.

3.2.1.2. Transaction Message Replay

Transaction messages which are lost in transit are re-sent when possible. The frontend and backend nodes keep an in-memory copy of all active messages for this purpose.

3.2.1.3. Link Failure Recovery

In the event of a communications failure, RTR tries to reconnect the link or links until it succeeds.

3.3. Recovery Scenarios

This section describes how RTR recovers from different hardware and software failure. For additional information on failure and recovery scenarios, refer to the *VSI Reliable Transaction Router Application Design Guide*.

3.3.1. Backend Recovery

If standby or shadow servers are available on another backend node, operation of the rest of the system will continue without interruption, using the standby or shadow server.

If a backend processor is lost, any transactions in progress are remembered by RTR and later recovered, either when the backend restarts, or by a standby if one is present. Thus, the distributed database is brought back to a transaction-consistent state.

3.3.2. Router Recovery

If a router fails and another router node is available, all in-progress transactions are transparently re-routed by the other router. System operation will continue without interruption.

3.3.3. Frontend Recovery

If a frontend is lost:

- All transactions committed but not completed on the frontend node at the time of failure will be completed.

- All transactions started but not committed on the frontend node at the time of failure will be aborted.

Chapter 4. RTR Interfaces

RTR provides interfaces for system management (the management interfaces) and for development of transaction processing and management applications (the programming or application development interfaces).

4.1. Management Interfaces

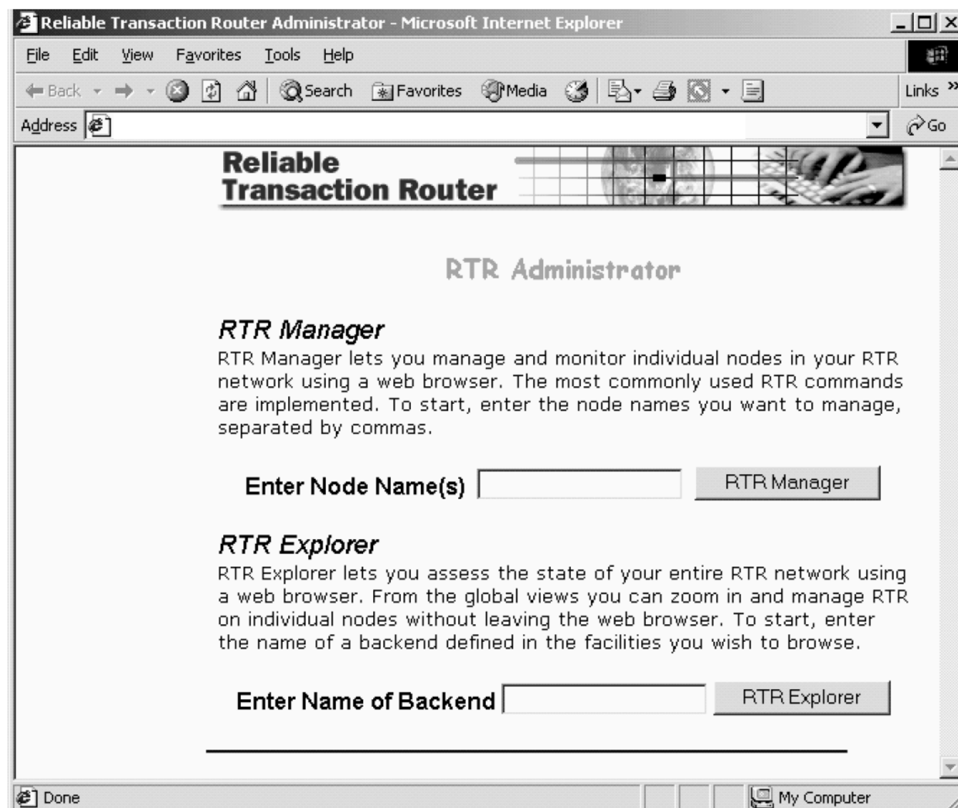
The management interfaces are:

- The RTR Administrator, a browser interface. This interface includes:
  - RTR Manager
  - RTR Explorer
- The command line interface or CLI

The RTR Administrator lets you manage RTR, its facilities, nodes, and network links, with a point-and-click interface. It contains extensive help, both as inline popups, as linked help, and as links to current information. For example, inline popups describe short headings more fully, and the system manager can view many types of status as RTR and the applications under its control run.

Figure 4.1 shows the RTR Adminstrator screen where you select whether to use the RTR Manager or the RTR Explorer.

Figure 4.1. RTR Administrator

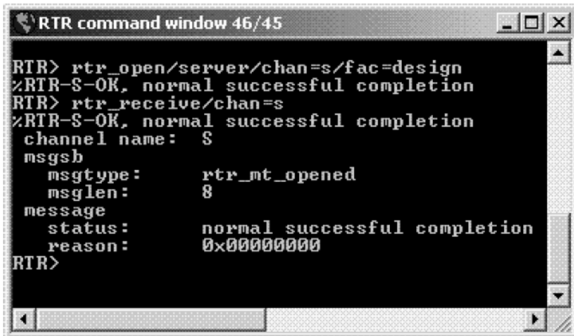


The RTR CLI contains all RTR system manager commands and calls to all RTR C API routines such as `rtr_open_channel` or `rtr_create_facility`. You can use either the RTR Manager or

the RTR CLI to manage your RTR configuration. You can also use the command line interface to write short RTR C applications for testing and experimentation. The CLI is described in the *VSI Reliable Transaction Router System Manager's Manual*. Its use is illustrated in this chapter.

Figure 4.2 shows the RTR command line interface.

Figure 4.2. RTR Command Line Interface



4.2. Programming Interfaces

RTR provides several programming or application development interfaces for design and implementation of transaction processing programs. They include the following:

- The object-oriented RTR Java interface

  You can use this API for new development, and, where appropriate, for new development with existing applications. This API can be used to implement applications with RTR using Java and J2EE technologies.

- The object-oriented API for C++ programming

  You can use this API for new development and, where appropriate, for new work with existing applications. An application can contain both object-oriented classes and Portable API calls. The C ++ API can be used to implement both management and transaction processing applications on all platforms supported by RTR.

- The RTR API for C programming

  This interface was the first multiplatform API available with RTR.

- An interface that enables use of an X/Open Distributed Transaction Processing-conformant resource manager

  This interface, invoked through the RTR management interfaces, enables RTR applications to be used with X/Open-compliant resource managers such as Oracle8.

- The OpenVMS API containing OpenVMS calls

  This API, supported on OpenVMS only, is obsolete for new development. To take advantage of new RTR features and capabilities, such applications can be rewritten with one of the newer APIs. Older applications will continue to run with later versions of RTR.

The RTR application programming interfaces, where available, are identical on all hardware and operating system platforms that support RTR. The object-oriented C++ API is fully described in the *VSI Reliable Transaction Router C++ Foundation Classes* manual. The C-programming API is fully

described in the *VSI Reliable Transaction Router C Application Programmer's Reference Manual*. Both APIs are used in examples in the *VSI Reliable Transaction Router Application Design Guide*. The Java J2EE interface is described in the JTA material in the RTR JRTR kit. The XA interface is described in materials from X/Open.

4.3. Application Development

The transaction processing environment poses special challenges for the development of applications, challenges best addressed by following a defined methodology such as the software development life cycle or object-oriented design fleshed out with use cases.

The software development life cycle consists of the following phases that are to be viewed as iterative:

- Gathering requirements (what is needed?)
- Developing a high-level design (what will the transaction processing application do?)
- Constructing a detailed design (explicitly, what will each part of the application do and what are its intended results?)
- Coding and unit testing
- Integration and system testing/deployment
- Maintenance

Many books are available to assist the developer with both design and development, including:

- J. Gray, A. Reuter, *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann, San Mateo, CA, 1992
- Philip A. Bernstein, Eric Newcomer, *Principles of Transaction Processing*, Morgan Kaufmann, San Francisco, CA, 1997

Object-oriented methods and practice are described and elaborated on in many books, including:

- James Rumbaugh, Michael Blaha, William Lorenson, *Object-Oriented Modeling and Design*, Prentice Hall, Englewood CLiffs, NJ, 1991
- Martin Fowler with Kendall Scott, *UML Distilled*, Addison-Wesley, Reading, MA 1997

Table 4.1 summarizes the RTR interfaces and their typical use.

Table 4.1. RTR Interfaces and their Use

With this interface:	You can write:
RTR Java interface	application programs
RTR C++ Foundation Classes	application programs system management programs
RTR C programming interface	application programs

4.4. RTR Management

You can manage RTR from several locations:

- from a node on which RTR is running

- from a remote node from which you send RTR commands to a node running RTR
- from a web browser that can be on or access a node running RTR

4.4.1. RTR Administrator

With the RTR Administrator, you have a network-browserlike display from which you can view RTR status and issue certain RTR commands with a point-and-click operation. The RTR Administrator contains both the RTR Manager, with which you set up and manage your RTR configuration, and the RTR Explorer, with which you observe and monitor your RTR configuration. Online help provided with the RTR Administrator includes popups for screen headings, popups for some RTR Explorer information, and extensive help for RTR commands.

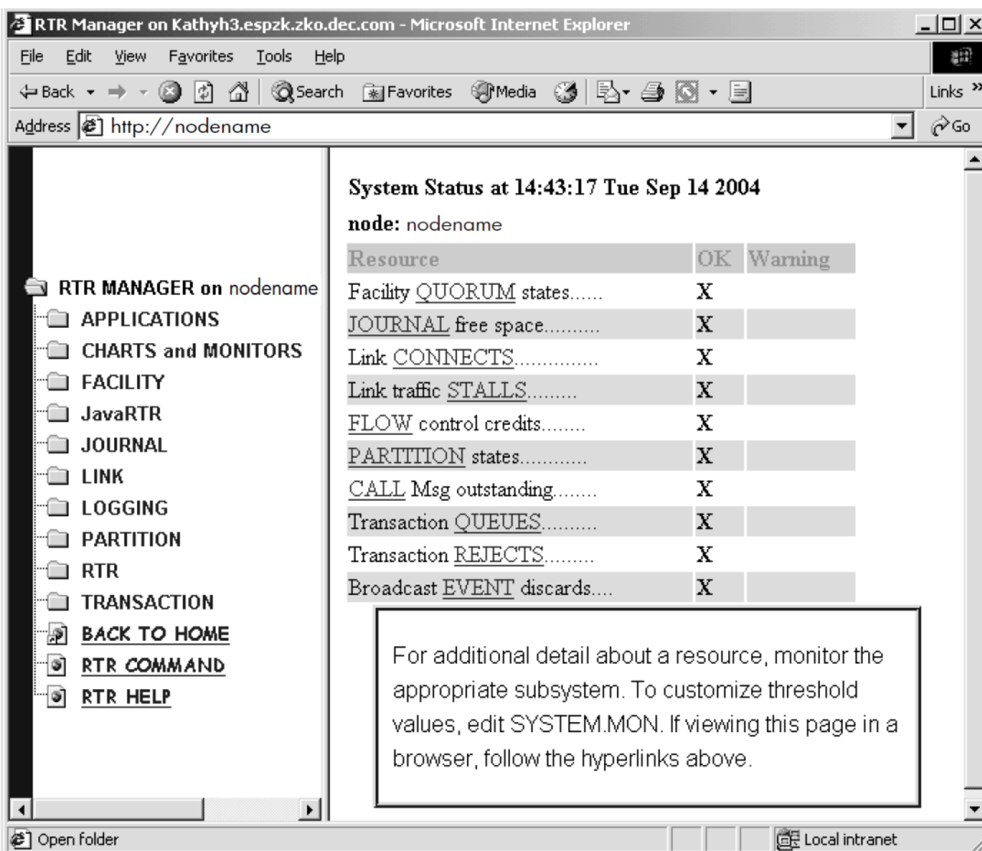
In the RTR Manager or RTR Explorer, buttons have the following effect:

This button:	Takes you to:
Back	The previous level
Back to Home	To the RTR Administrator

4.4.2. RTR Manager

Figure 4.3 shows the first RTR Manager screen through which you manage RTR and RTR applications. Not all RTR CLI commands are accessible from the RTR Manager RTR Command link; those rarely used are available only through the RTR CLI command window. The RTR Manager provides help for input screens, logging windows, and links between displays.

Figure 4.3. RTR Manager

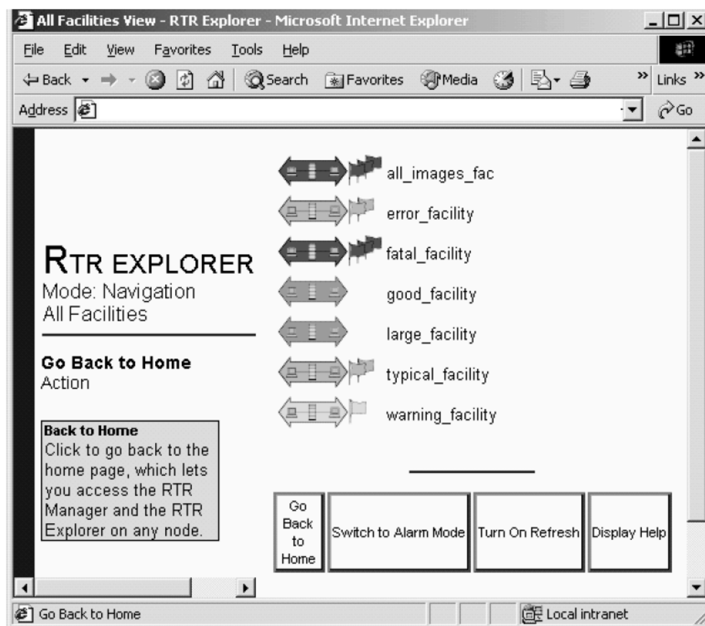


4.4.3. RTR Explorer

Figure 4.4 shows a sample screen of the RTR Explorer with several defined facilities. The RTR Explorer lets the system manager or administrator view the entire RTR configuration by facility and by node, and assess the state of the RTR network and the state of any individual node or facility in the network.

The All Facilities View (Figure 4.4) shows all facilities by name, with icons showing status of normal or one of three levels of alert (warning, error, or fatal). Additional views either by facility or node enable the manager to drill down, with a simple point-and-click, to the facility or node of interest.

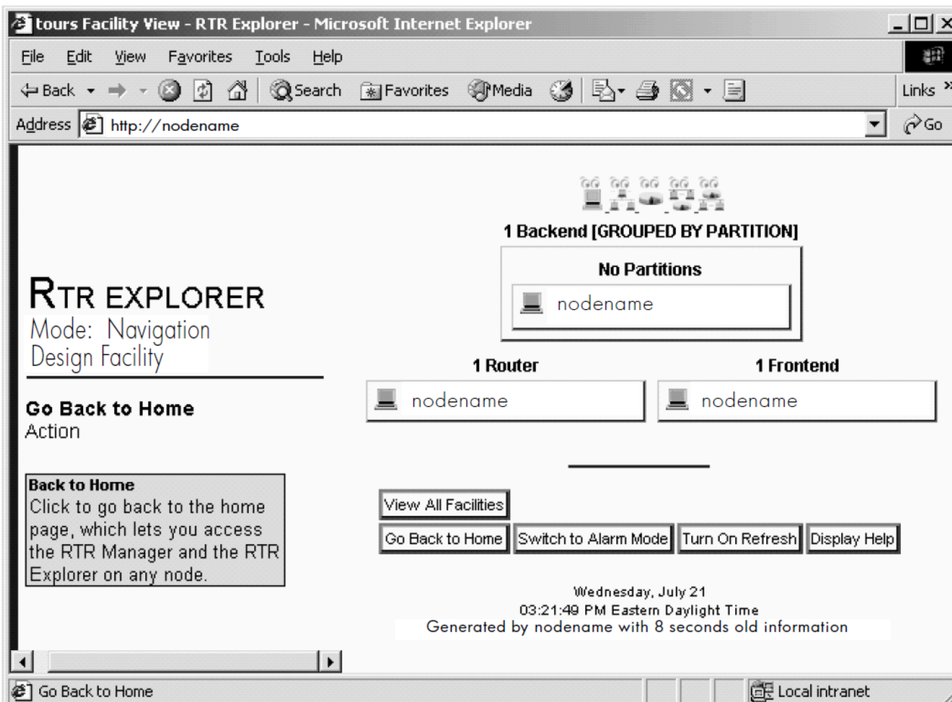
Figure 4.4. RTR Explorer: View of All Facilities



Information available for each facility includes the facility name, the alerts associated with nodes participating in the facility, and the state of the facility. Information for each node includes its name, role, cluster name, partition names, partition states, node state, and alerts.

Additional views either by facility or by node enable the administrator to zoom, with a simple point-and-click, to the facility or node of interest. From the node view, the administrator can open the RTR Manager for the node. Hence, RTR Explorer enables the administrator to both view and manage the state of every facility and node.

Figure 4.5. RTR Explorer: View of One Facility



Nodes can monitor themselves for alerts. Each alert can be set at progressive levels of severity – first *Warning*, second *Error*, and third *Fatal*. The severity of an alert indicates the urgency of the alert. *Warning* means RTR may or may not be operating normally, but something needs to be looked at. *Error* means that RTR is likely not operating normally, but may be able to continue operating. *Fatal* means that RTR cannot continue to operate unless the alert is resolved. See the REMEMBER EXPRESSION command in the *VSI Reliable Transaction Router System Manager's Manual* for how to define alerts.

The state of a facility shown by its icon in a view of all facilities (see Figure 4.4) indicates the worst severity for all defined alerts for all nodes in that facility. Similarly, the state of a node or group of nodes shown by its icon in a view of a single facility (see Figure 4.5) is the worst severity of all alerts for that node or group of nodes. If there are no flags, the node or group is operating normally.

4.4.4. 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. Figure 4.2 shows the RTR CLI interface. For example, the commands shown in the examples below start RTR and exchange a message between a client and a server.

Note

The channel identifier identifies the application process to the ACP. The client and server process must each have a *unique* channel identifier. In this example, the channel identifier for the client is C and for the server is S. Both use the facility called DESIGN.

Sample TP

The examples that follow show transaction processing (TP) communication between a client and a server created by entering commands at a terminal keyboard. The client application is executing on the frontend and the server on the backend.

The operational process is:

- Create the journal
- Create a facility
- Create a partition
- Open channels
- Start a transaction
- Accept the transaction

All applications will use some variants of these steps in the same order.

The user is called user, the facility being defined is called *DESIGN*, a client and a server are established, and a test message containing the words "Kathy's text today" is sent from the client to the server. After the server receives this text, the user on the server enters the words "And this is my response."

System responses begin with the characters %RTR-. Notes on the procedure are enclosed in square brackets [ ]. For clarity, commands you enter are shown in bold. You can view the status of a transaction with the SHOW TRANSACTION command.

The exchange of messages you observe in executing these commands illustrates RTR activity. You need to retain a similar sequence in your own designs for starting up RTR and initiating your own application.

You can use RTR SHOW and MONITOR commands to display status and examine system state at any time from the CLI. For more information on RTR commands, refer to the *VSI Reliable Transaction Router System Manager's Manual*.

Note

The `rtr_receive_message` command waits or blocks if no message is currently available. When using the `rtr_receive_message` command in the RTR CLI, use the /TIME=0 qualifier or TIMEOUT to poll for a message, if you do not want your `rtr_receive_message` command to block.

4.4.5. Examples

Example 4.1. The user issues the following commands on the server application where RTR is running on the backend.

```
$ RTR
Copyright 1994, 2003 Hewlett-Packard Development Company, L.P.
RTR> set mode/group
%RTR-I-STACOMSRV, starting command server on node NODEA
%RTR-I-GRPMODCHG, group changed from " " to "username"
%RTR-I-SRVDISCON, server disconnected on node NODEA
RTR> CREATE JOURNAL
%RTR-I-STACOMSRV, starting command server on node NODEA in group "username"
%RTR-S-JOURNALINI, journal  has been created on device D:
RTR> SHOW JOURNAL
Journal configuration on NODEA in group "username" at Mon Aug 28 14:54:11
 2000:-

Disk:   D:\  Blocks:    1000
```

```
RTR> start rtr
%RTR-I-NOLOGSET, logging not set
%RTR-S-RTRSTART, RTR started on node NODEA in group "username"
RTR> CREATE FACILITY DESIGN/ALL_ROLES=(NODEA)
     [- or /all=NODEA,NODEB]
%RTR-S-FACCREATED, facility DESIGN created
RTR> SHOW FACILITY
Facilities on node NODEA in group "username" at Mon Aug 28 15:00:28 2000:
Facility                                Frontend        Router     Backend
DESIGN                                  yes             yes        yes
RTR> rtr_open/server/accept_explicit/prepare_explicit/chan=s/fac=DESIGN
%RTR-S-OK, normal successful completion
RTR> RTR_RECEIVE_MESSAGE/CHAN=S
%RTR-S-OK, normal successful completion
 channel name: S
   .
   .
   .
   msgtype:    rtr_mt_opened
   .
   .
   .
   status:     normal successful completion
```

Example 4.2. When the next command is issued, RTR waits for the message from the client, which does not appear until after the client sends it

```
RTR> RTR_RECEIVE_MESSAGE/CHAN=S
%RTR-S-OK, normal successful completion
 channel name: S
  msgsb
   msgtype:     rtr_mt_msg1
   msglen:      19
   usrhdl:     0
      Tid:     63b01d10,0,0,0,0,2e59,43ea2002
  message
   offset bytes                                    text
   000000 4B 61 74 68 79 27 73 20 74 65 78 74 20 74 6F 64 Kathy's text tod
   000010 61 79 00                                      ay.
   reason:     Ox00000000

RTR> RTR_REPLY_TO_CLIENT/CHAN=S "And this is my response."
%RTR-S-OK, normal successful completion
RTR> show transaction
Frontend transactions on node NodeA in group "username" at Mon Aug 28
 15:12:10 2000
Tid                               Facility       FE-User    State
63b01d10,0,0,0,0,2e59,43ea2002   DESIGN         username.      SENDING
Router transactions on node NodeA in group "username" at Mon Aug 28
 15:12:10 2000:
63b01d10,0,0,0,0,2e59,43ea2002   DESIGN         username.      SENDING
Backend transactions on node NodeA in group "username" at Mon Aug 28
 15:12:10 2000:
63b01d10,0,0,0,0,2e59,43ea2002   DESIGN         username.      RECEIVING
RTR> RTR_RECEIVE_MESSAGE/CHAN=S
%RTR-S-OK, normal successful completion
 channel name: S
```

```
   msgsb
     msgtype:    rtr_mt_prepare
        [if OK, use: RTR_ACCEPT_TX
        else, use:  RTR_REJECT_TX]
RTR> RTR_RECEIVE_MESSAGE/TIME=0
RTR> STOP RTR  [Ends example test.]
```

Example 4.3. Commands and system response at client.

```
$ RTR
RTR> START RTR
%RTR-S-RTRSTART, RTR started on node NODEA in group "username"

RTR> RTR_OPEN_CHANNEL/CHANNEL=C/CLIENT/fac=DESIGN
%RTR-S-OK, normal successful completion

RTR> RTR_RECEIVE_MESSAGE/CHANNEL=C/tim
     [to get mt_opened or mt_closed]
%RTR-S-OK, normal successful completion
 channel name: C
   msgsb
     msgtype:     rtr_mt_opened
     msglen:      8
   message
     status:      normal successful completion
     reason:     Ox00000000
RTR> RTR_START_TX/CHAN=C
%RTR-S-OK, normal successful completion
RTR> RTR_SEND_TO_SERVER/CHAN=C "Kathy's text today." [text sent to the
 server]
%RTR-S-OK, normal successful completion
RTR> show transaction
Frontend transactions on node NodeA in group "username" at Mon Aug 28
 15:05:43 2000
Tid                               Facility       FE-User    State
63b01d10,0,0,0,0,2e59,43ea2002   DESIGN         username.      SENDING
Router transactions on node NodeA in group "username" at Mon Aug 28
 15:06:43 2000:
63b01d10,0,0,0,0,2e59,43ea2002   DESIGN         username.      SENDING
Backend transactions on node NodeA in group "username" at Mon Aug 28
 15:06:43 2000:
63b01d10,0,0,0,0,2e59,43ea2002   DESIGN         username.      SENDING

RTR> RTR_RECEIVE_MESSAGE/TIME=0/CHAN=C
```

Example 4.4. The following lines arrive at the client from RTR after the user enters commands at the server.

```
%RTR-S-OK, normal successful completion
 channel name: C
  msgsb
   msgtype:    rtr_mt_reply
   msglen:     25
   usrhdl:     0
   tid:        63b01d10,0,0,0,0,2e59,43ea2002
  message
   offset bytes                                           text
   000000 41 6E 64 20 74 68 69 73 20 69 73 20 6D 79 20 72 And this is my r
```

```
   000010 65 73 70 6F 6E 73 65 2E 00                      esponse..


RTR> RTR_ACCEPT_TX/CHANNEL=C
%RTR-S-OK, normal successful completion

RTR> show transaction
Frontend transactions on node NodeA in group "username" at Mon Aug 28
 15:17:45 2000
Tid                            Facility       FE-User    State
63b01d10,0,0,0,0,2e59,43ea2002   DESIGN         username.      VOTING
Router transactions on node NodeA in group "username" at Mon Aug 28
 15:17:45 2000:
63b01d10,0,0,0,0,2e59,43ea2002   DESIGN         username.      VOTING
Backend transactions on node NodeA in group "username" at Mon Aug 28
 15:17:45 2000:
63b01d10,0,0,0,0,2e59,43ea2002   DESIGN         username.      COMMIT

RTR> RTR_RECEIVE_MESSAGE
%RTR-S-OK, normal successful completion
 channel name: S
   .
   .
   .
   msgtype:  rtr_mt_accepted
   .
   .
   .
RTR> STOP RTR
```

4.5. Application Programming Interfaces

You write application programs and management applications with the RTR application programming interfaces.

4.5.1. RTR Java Object-Oriented Interface

You can use Java and J2EE technology to write applications that use RTR. For additional information on these technologies, see the documentation that is part of the JRTR downloadable kit. This documentation includes the *JRTR Getting Started* manual and other supplementary materials, including a sample application.

Java Technology

The following Java technology is used by:

RTR Clients	RTR Servers
UserTransaction Interface	InputStreams
	OutputStreams

J2EE Technology

The following J2EE technology is used by RTR servers:

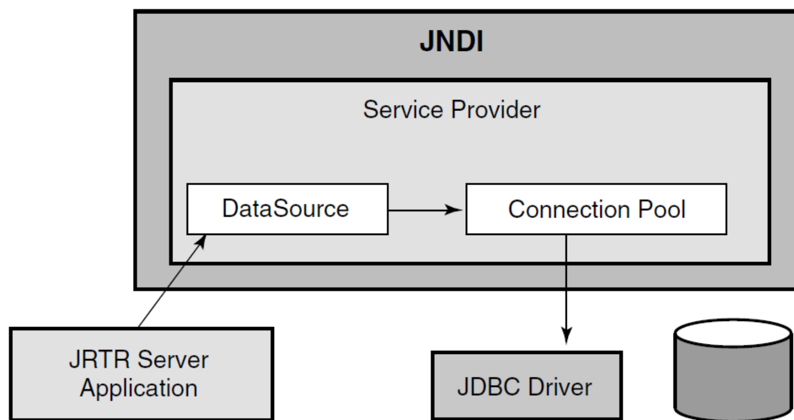
- a database and a JDBC driver that supports the JDBC 2.0 Optional Package javax.sql (required)

- a JNDI service provider (optional)

Figure 4–6 illustrates the required connection that must be defined for a service provider, with the links to the connection pool and the JDBC driver that are set up by the system administrator. The application program needs only to know about the service provider to use the connection.

RTR Java J2EE-based applications need to be able to locate and access database resources external to the application. The J2EE JDBC 2.0 javax.sql package addresses these needs through datasources, connections and connection pooling. For any particular database, the database vendor must provide a JDBC driver which supports the JDBC 2.0 Optional Package (formerly known as the JDBC 2.0 Standard Extension). This package defines datasources and connection pools.

Figure 4.6. RTR Service Provider



A database resource is represented by a datasource object. For the application to locate the datasource representing the database resource, a naming service that implements the Java Naming and Directory Interface (JNDI) must be present. Registering a datasource with the JNDI service enables the RTR Java J2EE-based application to locate the datasource and connect to its corresponding database. Once the datasource is located and the datasource object is instantiated, the datasource method getConnection( ) is called to obtain a connection object.

Sample Java server code

The sample Java code from a server Java application illustrates Java use of a datasource and a connection pool.

```
// Get a datasource that has been configured by the administrator
   DataSource ds = (DataSource)LookupFromJNDI("myDataSource");

// Get a connection to the database
   Connection con = ds.getConnection();
```

Some complex applications require multiple connections to one or more databases. Connection pooling allows applications to offload the high overhead of time and computing resources involved in creating and maintaining multiple connections to one or more databases. This is accomplished by using connectionpool objects. Connection pools (like datasources) are registered with a JNDI service. For more information on JDBC, refer to the Sun Java web site link for the JDBC Standard Extension API.

4.5.2. RTR C++ Object-Oriented Programming Interface

You can use the object-oriented programming interface to write C++ applications that use RTR. For more information on the C++ object-oriented programming interface, refer to the *VSI Reliable*

Transaction Router C++ Foundation Classes manual and the *VSI Reliable Transaction Router Application Design Guide*.

Sample C++ client code

The following example illustrates object creation in a program that is to act as an RTR client application. The first step is to create a Transaction Controller. This is followed by creating an RTR Data Object that will hold the ASCII message for the server, sending the message to the server application, and finally accepting the transaction.

```
//
// Create a Transaction Controller to receive incoming messages
// and events from a client.
//
RTRClientTransactionController *pTransaction = new
 RTRClientTransactionController();
//
// Create an RTRData object to hold an ASCII message for the server.
//
RTRData *pMessage1 = new RTRData("You are pretty easy to use!!!");
//
// Send the Server a message
//
sStatus = pTransaction->SendApplicationMessage(pMessage1);
ASSERT(RTR_STS_OK == sStatus);
//
// Since we have successfully finished our work, tell RTR we accept the
// transaction.
//
pTransaction->AcceptTransaction();
```

Sample C++ server code

The following example illustrates creation of an object in a server application that is to act as an RTR server. First it creates a key segment for a specific range of ASCII values (A to L) and creates a data object to hold each incoming message or event. Then it loops continuously, receiving messages and dispatching them to the handlers.

```
void CombinationOrderProcessor::StartProcessingOrdersAtoL()
{
//
// Create an RTRKeySegment for all ASCII values between "A" and "L."
//
m_pkeyRange = new RTRKeySegment (rtr_keyseg_string, //To process strings.
                                 1,                 //Length of the key.
                           OffsetIntoApplicationProtocol, //Offset value.
                           "A",       //Lowest ASCII value for partition.
                           "L");      //Highest ASCII value for partition.
StartProcessingOrders(PARTITION_NAMEAToL,m_pKeyRange);
}

//
// Create an RTRData Oobject to hold each incoming message or event. This
// object will be reused.
//
RTRData *pDataReceived= new RTRData();
//
```

```
// Continually loop, receiving messages and dispatching them to the
 handlers.
//
while(true)
{
  sStatus = pTransaction->Receive(&pDataReceived);
  ASSERT(RTR_STS_OK == sStatus);

  sStatus = pDataReceived->Dispatch();
  ASSERT(RTR_STS_OK == sStatus);
}
```

Sample system management code

The following examples illustrates creation of objects in an application to perform system management tasks for RTR.

Example 4.5. Creating a Facility with the C++ API

```
// Use the C++ interface to create an RTR facility
RTRFacilityManager::CreateFacilityWithAllRoles_3()
{
bool bOverallResult = true;
//Create facility manager, abort if creation fails
    RTRFacilityManager * pFacilityManager;
    pFacilityManager = new RTRFacilityManager;
if ( IsFailure(pFacilityManager != NULL) )
    {
return false;
    }
// Create the facility
    rtr_status_t stsCreateFacility;
stsCreateFacility =
    pFacilityManager->CreateFaclity("FaclityWithAllRoles_3",
                                     GetDefaultRouterName(),
                                     GetDefaultFrontendName(),
                                     GetDefaultBackendName(),
                                     true,
                                     false);
    //If facility creation is not successful, report it
    if ( IsFailure( stsCreateFaciltiy == RTR_STS_OK ) )
{
bOverallResult = false;
OutputStatus( stsCreateFacility);
}
    else // Delete a successfully created facility
    {
    rtr_status_t stsDeleteFacility;
stsDeleteFacility =
        pFacilityManager->DeleteFacility("FacilityWithAllRoles_3");
    if ( IsFailure( stsDeleteFacility == RTR_STS_OK ) )
    {
    bOverallResult = false;
    OutputStatus) stsDeleteFacility);
    }
    }
// Clean up and return
    delete pFacilityManager;
```

```
return bOverallResult;
}
```

Sample C++ system management code

The following examples perform specific RTR system management tasks. They can be used individually or together.

- The first starts RTR.
- The second creates a facility.
- The third creates a partition in the previously created facility.

Example 4.6. Starting RTR with the C++ API

```
//Start RTR.

RTR rtr;
rtr.Start();
```

Example 4.7. Creating a Facility with the C++ API

```
//Create facility named "myFacility".

RTRFacilityManager FacMgr;
rtr_status_t sts = FacMgr.CreateFacility("myFacility",
                                          "router",
                                          "frontend",
                                          "backend",
                                          false,
                                          false) ;
```

Example 4.8. Creating a Partition with the C++ API

```
//Create a partition named "myPartition" in facility "myFacility."

int low = 100;
int max = 199;
RTRKeySegment Key100To199( rtr_keyseg_unsigned,
                           sizeof(int),
                           0,
                           &low,
                           &max );

RTRPartitionManager PartitionMgr;
sts = PartitionMgr.CreateBackendPartition( "myPartition",
                                            "myFacility",
                                            Key100To199,
                                            false,
                                            false) ;
```

4.5.3. RTR C Programming Interface

You can use the C programming interface to write C applications that use RTR. For more information on the C programming interface, refer to the *VSI Reliable Transaction Router C Application Programmer's Reference Manual* and the *VSI Reliable Transaction Router Application Design Guide*.

Snippets from client and server programs using the RTR C-programing API follow and are more fully shown in the *VSI Reliable Transaction Router Application Design Guide*.

Sample C client code

Example of an open channel call in an RTR client program:

```
status = rtr_open_channel(&Channel,
                              Flags,
                              Facility,
                              Recipient,
                              RTR_NO_PEVTNUM,
                              Access,
                              RTR_NO_NUMSEG,
                              RTR_NO_PKEYSEG);
if (Status != RTR_STS_OK)
```

Sample C server code

Example of a receive message call in an RTR server program:

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

A client can have one or multiple channels, and a server can have one or multiple channels. A server can use concurrent servers, each with one channel. How you create your design depends on whether you have a single CPU or a multiple CPU machine, and on your overall design goals and implementation requirements. For a more complete discussion of application designs, refer to the *VSI Reliable Transaction Router Application Design Guide*.

Chapter 5. The RTR Environment

The RTR environment has two parts:

- The system management environment
- The runtime environment

5.1. The RTR System Management Environment

You manage your RTR environment from a management station, which can be on a node running RTR or on some other node. You can manage your RTR environment either from your management station running a network browser, or from the command line using the RTR CLI. From a management station using a network browser, processes use the http protocol for communication.

The RTR system management environment contains four processes:

- The RTR Control Process, RTRACP
- The RTR Command Line Interface, RTR CLI
- The RTR Command Server Process, RTRCOMSERV
- The RTR daemon, RTRD
- The RTR Problem Detection Process, RTRDETECT

The RTR Control Process, RTRACP, is the master program. It resides on every node where RTR has been installed and is running. RTRACP performs the following functions:

- Manages network links
- Sends messages between nodes
- Handles all transactions and recovery

RTRACP handles interprocess communication traffic, network traffic, and is the main repository of runtime information. ACP processes operate across all RTR roles and execute certain commands both locally and at remote nodes. These commands include:

- FACILITY
- SET LINK/NODE
- SET/CREATE PARTITION
- SHOW NODE
- STOP RTR

RTR CLI is the Command Line Interface that:

- Accepts commands entered locally by the system manager

- Sends commands to the Command Server Process RTRCOMSERV
- Can initiate commands on one node and execute them on another in most cases

Commands executed directly by the CLI include:

- DISPLAY
- DO (to the local operating system)
- MONITOR commands
- RECALL
- SET ENVIRONMENT
- SPAWN
- HELP

RTR COMSERV is the Command Server Process that:

- Receives commands from RTR
- Remains temporarily waiting for another command
- Exits automatically when idle for some time

The Command Server Process executes commands both locally and across nodes. Commands that can be executed at the RTR COMSERV include:

- START RTR
- CREATE/MODIFY JOURNAL
- SHOW LINK/FACILITY/SERVER/CLIENT (ACP must be running)
- Application programmer commands (for testing and demonstration)

RTRDETECT is the problem detection process that:

- Detects problems associated with RTR on each node participating in an RTR network.
- Is an intermittent process created on a set interval and dies after it completes the problem detection.
- Enables the web-based RTR Explorer to report problems on every node in the RTR network.

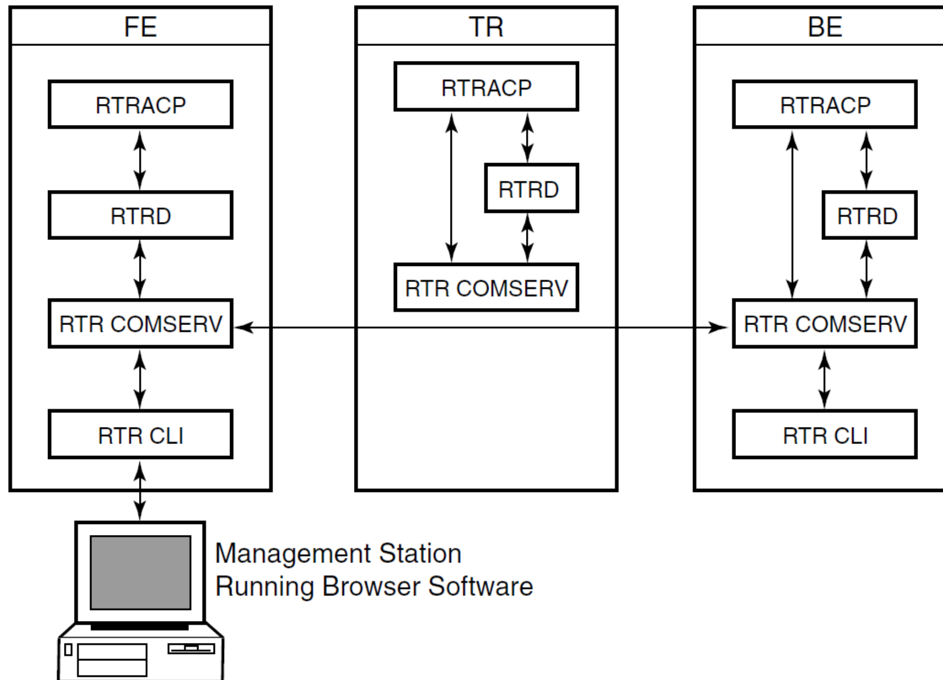
See the commands REMEMBER EXPRESSION and SET NODE /DETECTION_INTERVAL in the *VSI Reliable Transaction Router System Manager's Manual* for more details on controlling the behavior of RTR's problem detection.

Figure 5.1illustrates the RTR system management environment.

While the figure shows the RTR Management Station on a node declared as a frontend (FE), you can manage RTR from any node where RTR is running, whether the node is declared as a frontend, router, or backend. The RTR COMSERV must be running to manage RTR. The RTR Management Station runs web browser software with which you manage RTR. It could alternatively be running the RTR CLI.

For further details on the RTR entities such as RTRACP and the RTR COMSERV, see RTR Runtime Environment later in this manual.

Figure 5.1. RTR System Management Environment



5.1.1. Monitoring RTR

RTR Monitor pictures or the RTR Monitor let you view the status and activities of RTR and your applications. A monitor picture is dynamic, its data periodically updated. RTR SHOW commands that also let you view status are snapshots, giving you a view at one moment in time. A full list of RTR Monitor pictures is available in the *VSI Reliable Transaction Router System Manager's Manual* "RTR Monitoring" chapter and in the help file under RTR_Monitoring. Many RTR Monitor pictures are available using the RTR browser interface.

5.1.2. Transaction Management

The RTR transaction is the heart of an RTR application, and *transaction state* characterizes the current condition of a transaction. As a transaction passes from one state to another, it undergoes a state transition. Transaction states are maintained in memory, and some are stored in the RTR journal for use in recovery.

RTR uses three transaction states to track transaction status:

- transaction runtime state
- transaction journal state
- transaction server state

Transaction runtime state describes how a transaction progresses from the point of view of RTR roles (FE, TR, BE). A transaction, for example, can be in one state as seen from the frontend, and in another as seen from the router.

Transaction journal state describes how a transaction progresses from the point of view of the RTR journal. The transaction journal state, not seen by frontends and routers, managed by the backend, is used by RTR for recovery replay of a transaction after a failure.

Transaction server state, also managed by the backend, describes how a transaction progresses from the point of view of the server. RTR uses this state to determine if a server is available to process a new transaction, or if a server has voted on a particular transaction.

The RTR SHOW TRANSACTION command shows transaction status, and the RTR SET TRANSACTION command can be used, under certain well-constrained circumstances, to change the state of a live transaction. For more details on use of SHOW and SET commands, see the *VSI Reliable Transaction Router System Manager's Manual*.

5.1.3. Partition Management

Partitions are subdivisions of a routing key range of values used with a partitioned data model and RTR data-content routing. Partitions exist for each range of values in the routing key for which a server is available to process transactions. Redundant instances of partitions can be started in a distributed network, to which RTR automatically manages the state and flow of transactions. Partitions and their characterisitcs can be defined by the system manager or operator, as well as within application programs.

RTR management functions enable the operation to manage many partition-based attributes and functions including:

- Creation/deletion of a partition with a user-specified name
- Defining/changing a key-range definition
- Selecting a preferred primary node
- Selecting failover precedence between local and cross-site shadows
- Suspending/resuming operations to synchronize database backup with transaction flow
- Overriding the automatic recovery procedures of RTR with manual recovery procedures, for added flexibility
- Specifying retry limits for problem transactions

The operator can selectively inspect transactions, modify states, or remove transactions from the journal or the running RTR system. This allows for greater operational control and enhanced management of a system where RTR is running.

For more details on managing partitions and their use in applications, see the *VSI Reliable Transaction Router System Manager's Manual* chapter “Partition Management.”

5.2. The RTR Runtime Environment

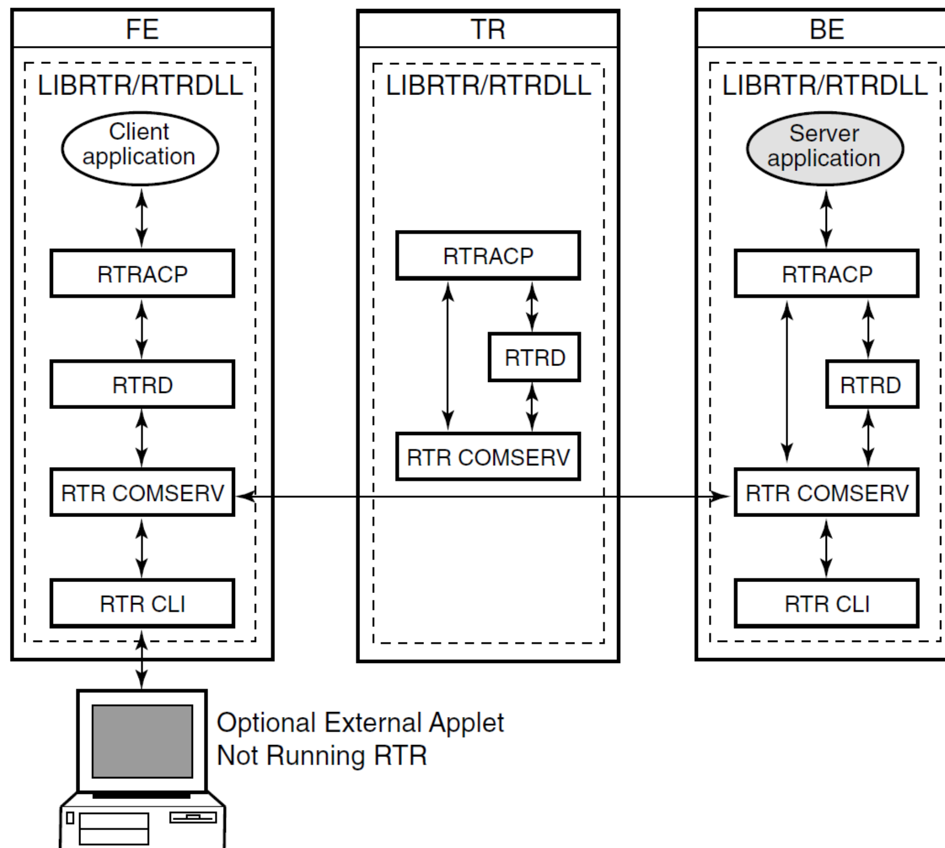
When all RTR and application components are running, the RTR runtime environment contains:

- Client application
- Server application
- RTR shareable image, LIBRTR

- RTR control process, RTRACP
- RTR daemon, RTRD
- RTR command line interface, RTR CLI
- RTR command server, RTR COMSERV

Figure 5.2 shows these components and their placement on frontend, router, and backend nodes. The frontend, router, and backend can be on the same or different nodes. If these are all on the same node, there is only one RTRACP process.

Figure 5.2. RTR Runtime Environment



5.3. What's Next?

This concludes the material on RTR concepts and capabilities that all users and implementors should know. For more information, proceed as follows:

If you are:	Read these documents:
a system manager, system administrator, or software installer	1. *VSI Reliable Transaction Router Release Notes* 2. *VSI Reliable Transaction Router Installation Guide* 3. *VSI Reliable Transaction Router System Manager's Manual*

If you are:	**Read these documents:**
an applications or system management developer, programmer, or software engineer	1. *VSI Reliable Transaction Router Application Design Guide* 2. *JRTR Getting Started* 3. *VSI Reliable Transaction Router C++ Foundation Classes* 4. *VSI Reliable Transaction Router C Application Programmer's Reference Manual*