Reliable Transaction Router

Getting Started

Order Number: AA-RLE1A-TE

January 2001

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

Revision/Update Information: This is a new manual.

Software Version: Reliable Transaction Router Version 4.0

Compaq Computer Corporation
Houston, Texas

This document was prepared using VAX DOCUMENT, Version 2.1.

Contents

3 Reliability Features

4 RTR Interfaces

5 The RTR Environment

Glossary

Index

Examples

Figures

Tables

Preface

Purpose of this Document

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

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 ths 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.

Related Documentation

Additional resources in the RTR documentation kit include:

Document	Content
For all users:	
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:	
Reliable Transaction Router *Installation Guide*	Describes how to install RTR on all supported platforms.
Reliable Transaction Router *System Manager's Manual*	Describes how to configure, manage, and monitor RTR.
Reliable Transaction Router *Migration Guide*	Explains how to migrate from RTR Version 2 to RTR Version 3 (OpenVMS only).
For the application programmer:	
Reliable Transaction Router *Application Design Guide*	Describes how to design application programs for use with RTR, illustrated with both the C and C++ interfaces.
Reliable Transaction Router *C++ Foundation Classes*	Describes the object-oriented C++ interface that can be used to implement RTR object-oriented applications.
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.

You can find additional information on RTR and existing implementations on the RTR web site at http://www.compaq.com/rtr/.

Reader's Comments

Compaq welcomes your comments on this guide. Please send your comments and suggestions by email to *rtrdoc@compaq.com*. Please include the document title, date from title page, order number, section and page numbers in your message. For product information, send email to *rtr@compaq.com*.

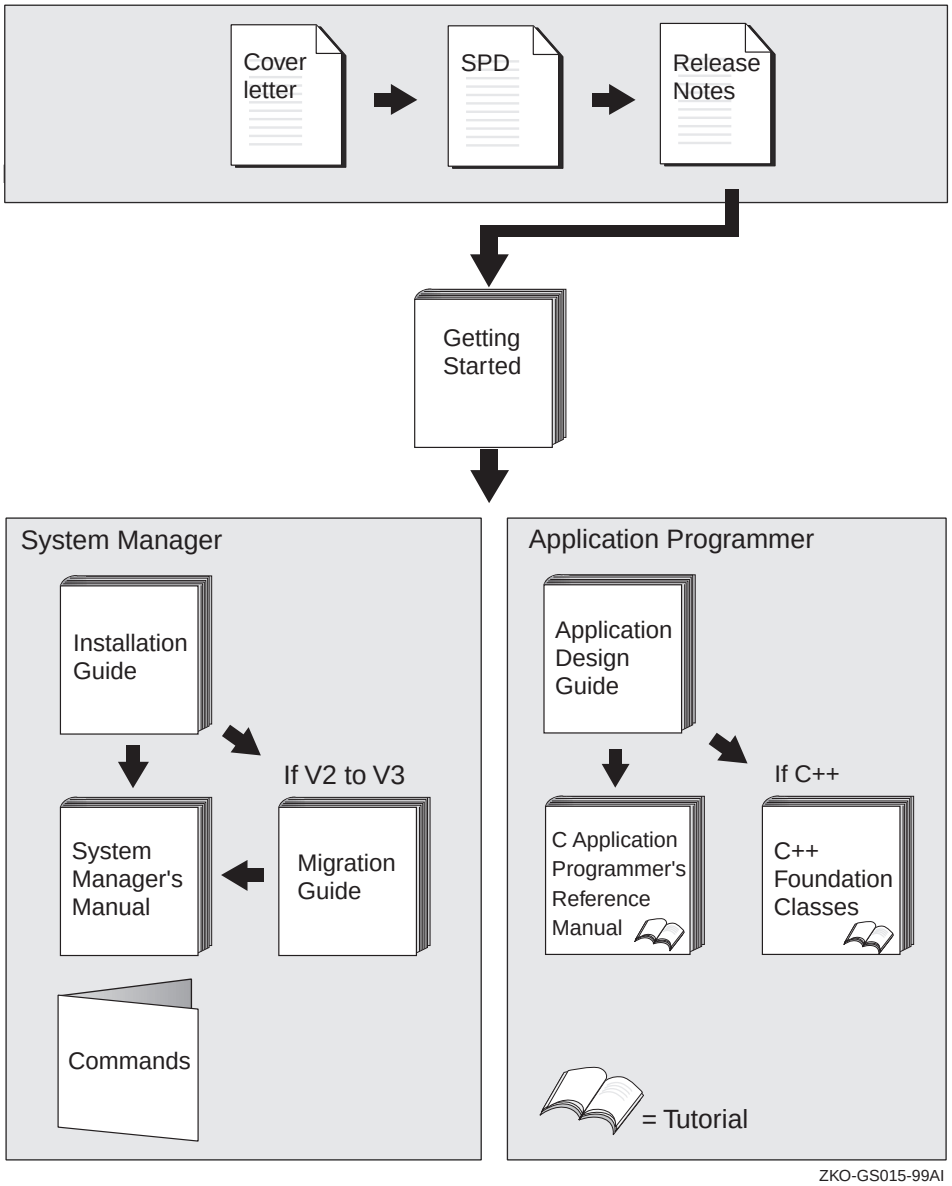
Conventions

This manual adopts the following conventions:

Convention	Description
New term	New terms are shown in bold when introduced and defined. All RTR terms are defined in the glossary at the end of this document or in the supplemental glossary in the RTR *Application Design Guide*.
`User input`	User input and programming examples are shown in a monospaced font. Boldface monospaced font indicates user input.
Terms and titles	Terms defined only in the glossary are shown in italics when presented for the first time. Italics are also used for titles of manuals and books, and for emphasis.
FE	RTR frontend
TR	RTR transaction router or router
BE	RTR backend

Reading Path

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


Cover letter
SPD
Release Notes
Getting Started
System Manager
Installation Guide
If V2 to V3
System Manager's Manual
Migration Guide
Commands
Application Programmer
Application Design Guide
If C++
C Application Programmer's Reference Manual
C++ Foundation Classes
= Tutorial
ZKO-GS015-99AI

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).

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.

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 also works 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, see the 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.

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. A transaction with the ACID properties has the following attributes:

- Atomic
- Consistent
- Isolated
- Durable

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

RTR Terminology

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. The terms are described in the following order:

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

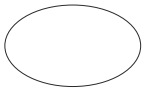
RTR Application

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 any language, commonly C or C++, 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, filled for a server application.

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.

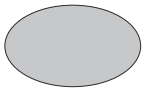
Figure 1–1 Client Symbol



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.

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

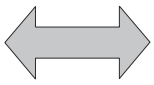
A node that runs client applications is called a **frontend** (FE), or is said to have the frontend role. A node that runs server applications is called a **backend** (BE). Additionally, the transaction **router** (TR) contains no application software but acts as a traffic cop between frontends and backends, routing transactions to the appropriate destinations. The router 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.

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. Nodes can share several facilities. The role of a node is defined within the scope of a particular facility. 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

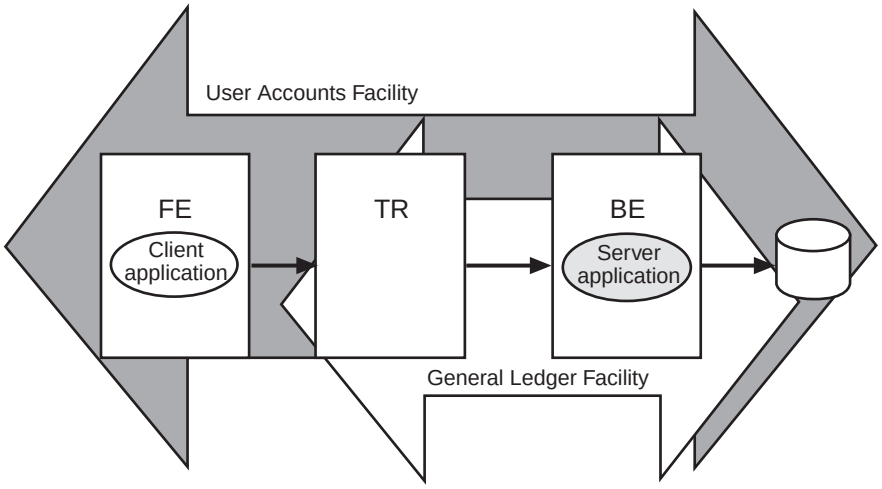


A facility name is mapped to specific physical nodes and their roles using the CREATE FACILITY command.

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.

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

Figure 1–5 Components in the RTR Environment



LKG-11203-98WI

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.

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.

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

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 bidirectional.)

Transaction Controller

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

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.

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

Figure 1–6 Two-Tier Client/Server Environment

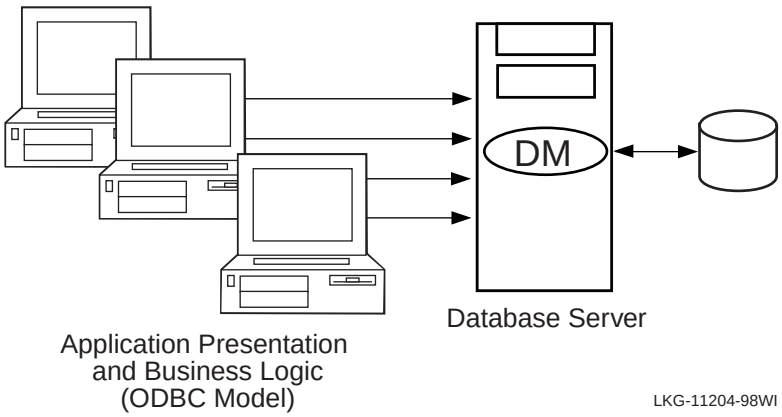
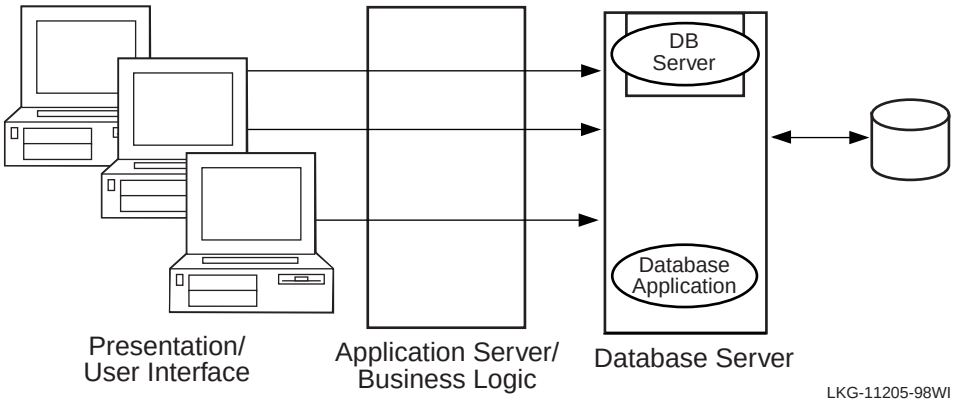


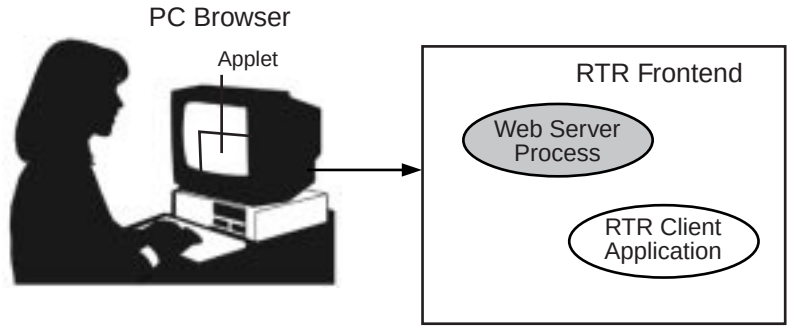
Figure 1–7 Three-Tier Client/Server Environment



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 components can reside on a single node but are typically deployed on different nodes to achieve modularity, scalability, and redundancy for availability. With different systems, if one physical node goes down or off line, another router and backend node takes over. In a slightly different configuration, you could have an application that uses an external applet running on a browser that connects to a client running on the RTR frontend. Such a configuration is shown in Figure 1–8.

Figure 1–8 Browser Applet Configuration



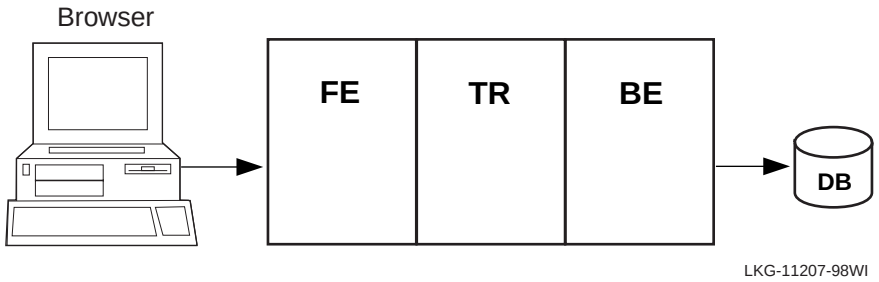
The RTR client application could be an ASP (Active Server Page) script or a process interfacing to the webserver through a standard interface such as CGI (Common Gateway Interface).

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.

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

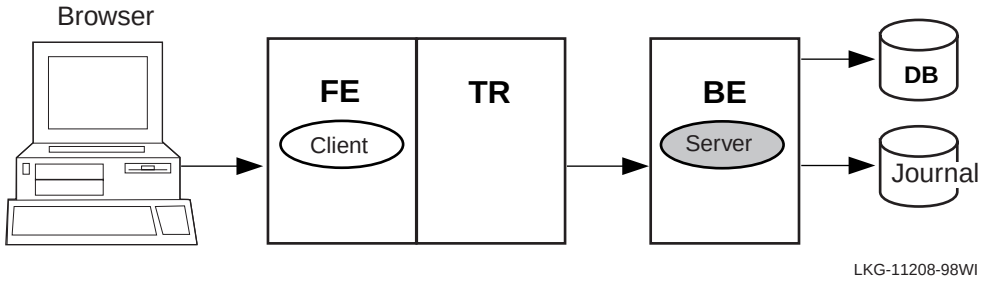
single node configuration can be useful during development, but would not normally be used when your application is deployed.

Figure 1–9 RTR with Browser, Single Node, and Database



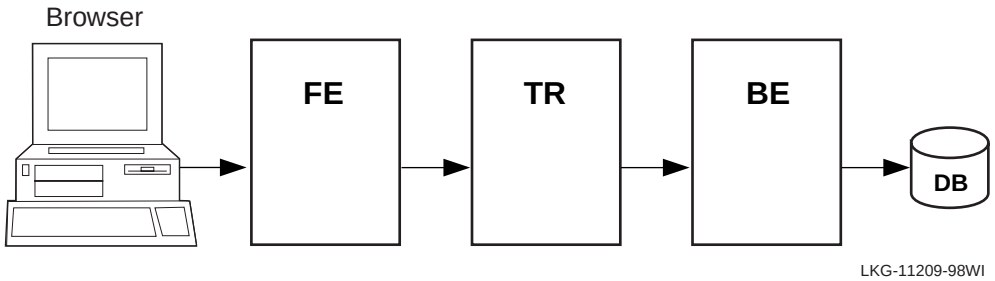
When creating the configuration used by an application and defining the nodes where a facility has its frontends, routers, and backends, the setup must also define which nodes will have journal files. Each backend in an RTR configuration must have a journal file to capture transactions when other nodes are unavailable. When applications are deployed, often the backend is separated from the frontend and router, as shown in Figure 1–10.

Figure 1–10 RTR Deployed on Two Nodes



In this example, the frontend with the client and the router reside on one node, and the server resides on the backend. Frequently, routers are placed on backends rather than on frontends. A further separation of workload onto three nodes is shown in Figure 1–11.

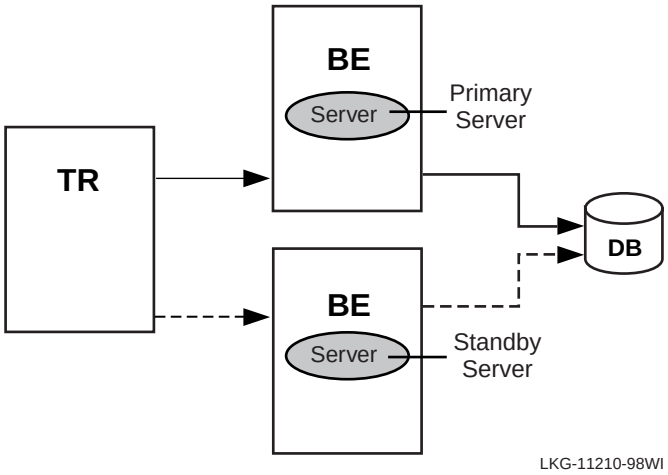
Figure 1–11 RTR Deployed on Three Nodes



This three-node configuration separates transaction load onto three nodes, but 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.

In this case, a **standby server** will do the job. A standby server (see Figure 1–12 is a process that can take over when the primary server is not available. Both the primary and the standby server access the same database, but the primary processes all transactions unless it is unavailable. The standby processes transactions only when the primary is unavailable. At other times, the standby can do other work. The standby server is often placed on a node other than the node where the primary server runs.

Figure 1–12 Standby Server Configuration



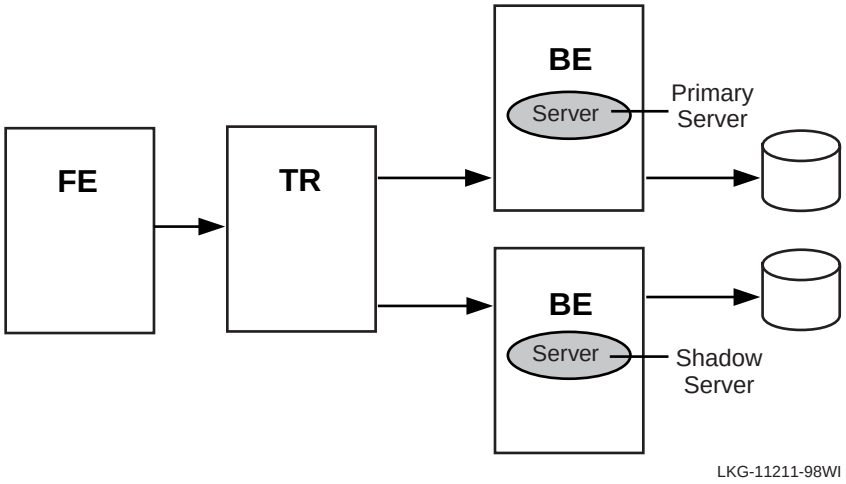
Transactional shadowing

To increase transaction availability, transactions can be shadowed with a **shadow server**. This is called **transactional shadowing** and is accomplished by having a second location, often at a different site, where transactions are also recorded. This is illustrated in Figure 1–13. 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.

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 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.

Figure 1–13 Transactional Shadowing Configuration



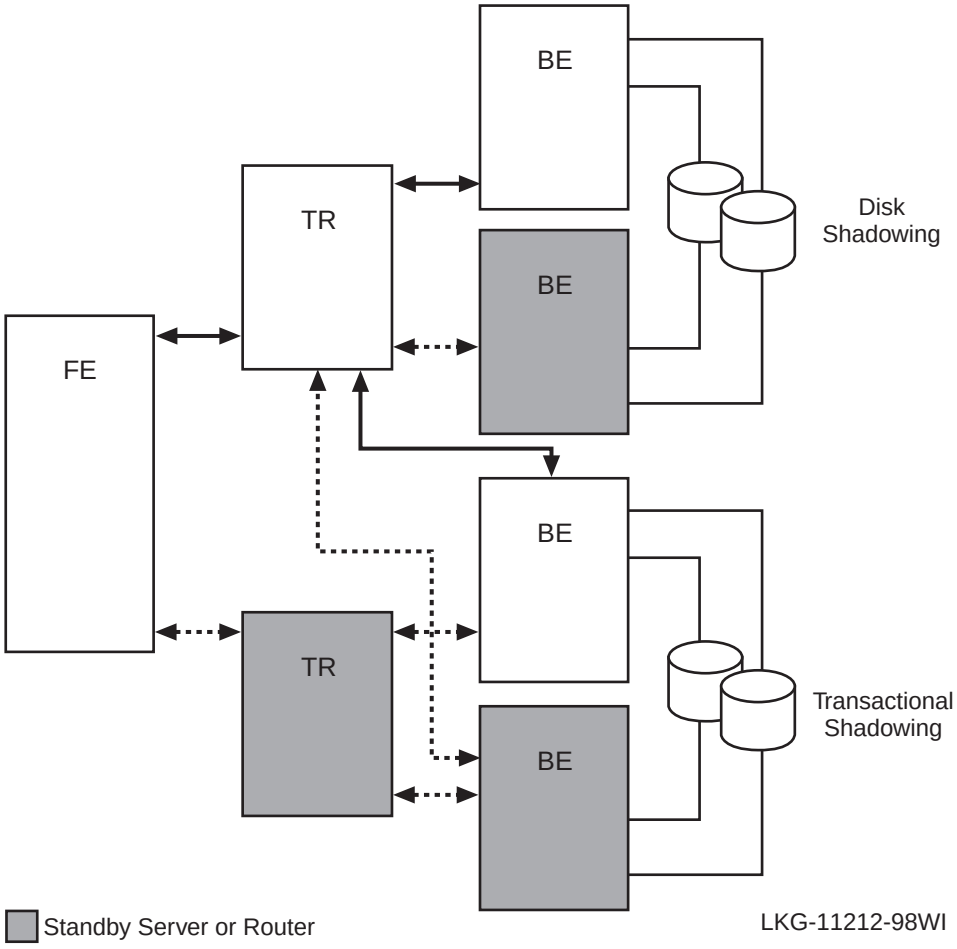
With transactional shadowing, there is no requirement that hardware, the data store, or the operating system at different sites be the same. You could, for example, have one site running OpenVMS and another running Windows NT; the RTR transactional commit process would be the same at each site.

Note

Transactional shadowing shadows only transactions controlled by RTR.

For full redundancy to assure maximum availability, a configuration could employ both *disk shadowing* in clusters at separate sites coupled with transactional shadowing across sites with standby servers at each site. This configuration is shown in Figure 1–14. For clarity, not all possible connections are shown. In the figure, backends running standby servers are shaded, connected to routers by dashed lines. Only one site (the upper site) does full disk shadowing; the lower site is the shadow for transactions, shadowing all transactions being done at the upper site.

Figure 1–14 Two Sites: Transactional and Disk Shadowing with Standby Servers



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 cluster environment. In a non- cluster environment, seamless failover of standbys is not guaranteed.

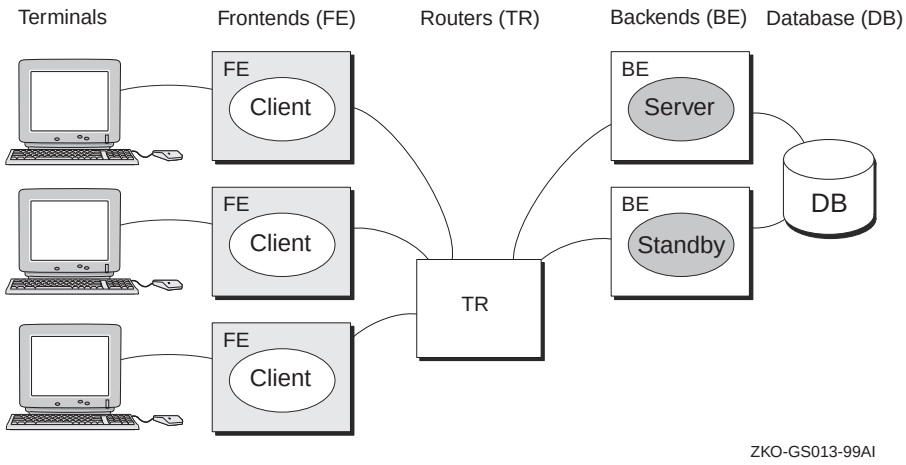
Standby servers are "spare" servers which automatically take over from the main backend if it fails. This takeover is transparent to the application.

Figure 1–15 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) runs on one node while the standby server (Standby) 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 (Standby). 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 1–15 Standby Servers



Transactional shadow server

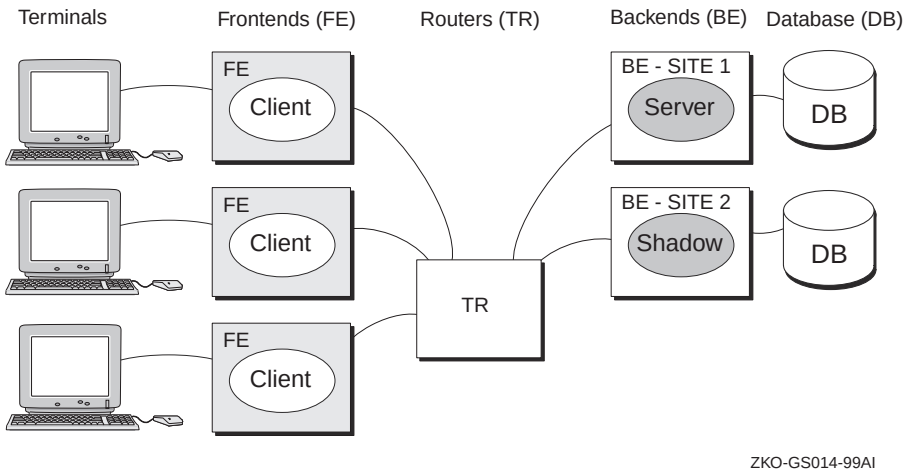
The **transactional shadow server** places all transactions recorded on the primary server on a second database. The transactional shadow server can be at the same site or at a different site, and must exist in a networked environment.

A transactional shadow server can also have standby servers for greater reliability. 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 receive new online transactions. Transactional shadowing is done by partition. A transactional shadow configuration can have only two members of the shadow set.

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

Figure 1–16 shows a simple shadow configuration. The main (BE) Server at Site 1 and the shadow server (Shadow) 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 1–16 Shadow Servers



Note that each shadow server can also have standby servers.

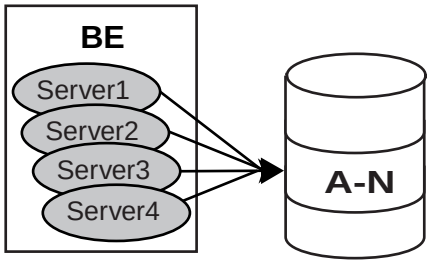
Concurrent server

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 1–17, 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.

Figure 1–17 Concurrent Servers



LKG-11275-98WI

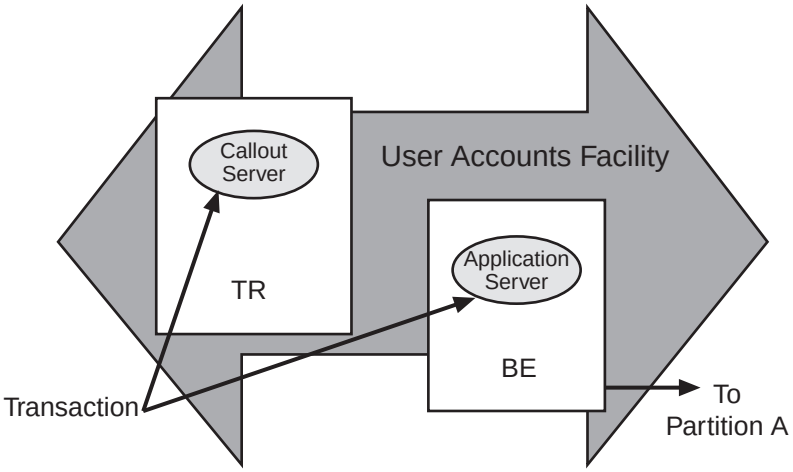
Callout server

The **callout server** provides 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 1–18 illustrates the use of a callout server that authenticates every transaction (txn) in a facility.

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.

Figure 1–18 A Callout Server



LKG-11276-98WI

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.

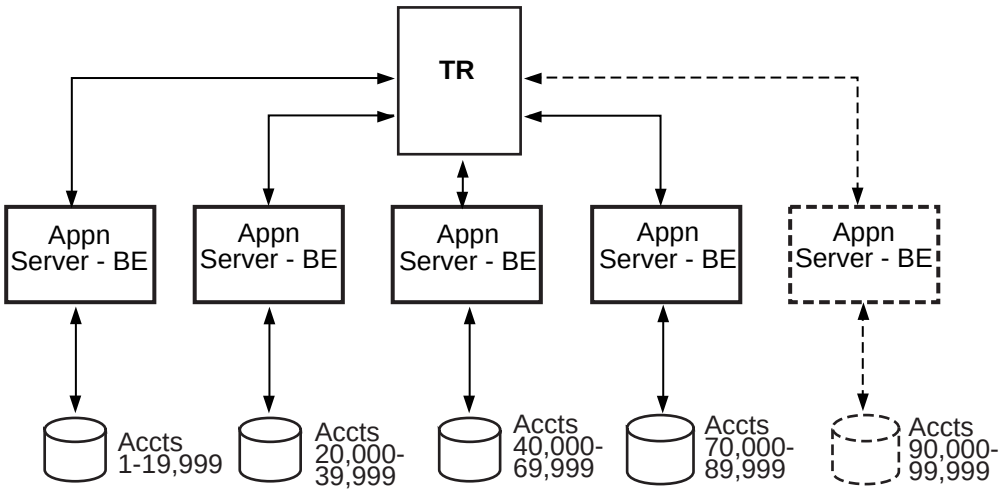
Since this technique relies on backing out unauthorized transactions, it is most suitable when only a small proportion of transactions are expected to fail the security check, so as not to have a performance impact.

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 1–19. A partition is a segment of your database.

Figure 1–19 Bank Partitioning Example



LKG-11213-98WI

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 RTR *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 1–19, 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 1–20 illustrates this use of standbys with distributed partitioning. As shown in Figure 1–20, 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.

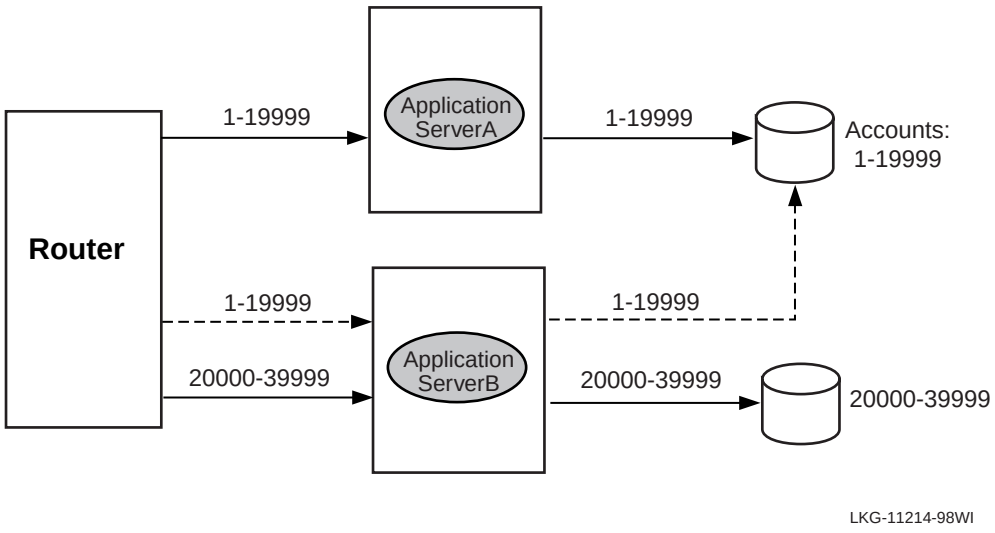
Anonymous clients

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.

Figure 1–20 Standby with Partitioning



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.

2
Architectural Concepts

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

The Three-Layer Model

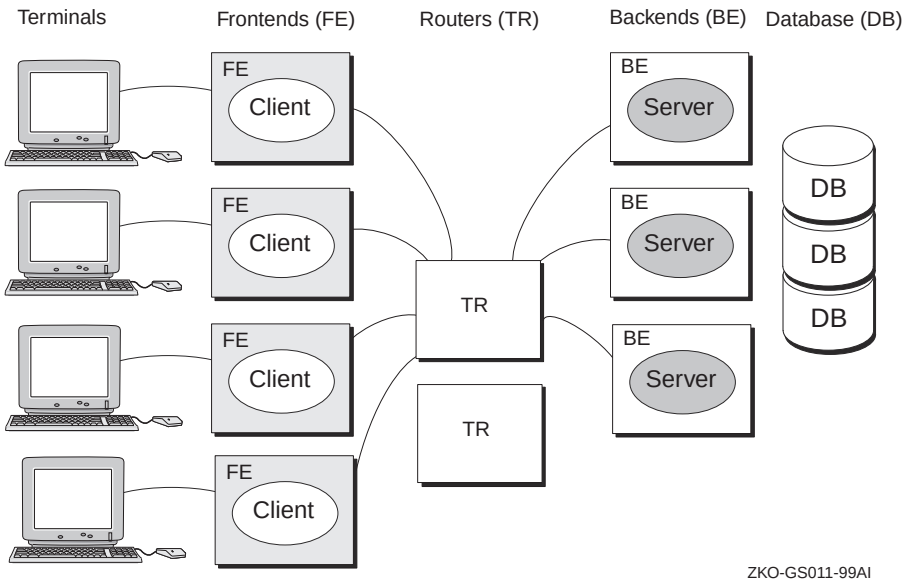
RTR is based on a three-layer architecture consisting of frontend (FE) roles, backend (BE) roles and router (TR) roles. The roles are shown in Figure 2–1. In this and subsequent diagrams, rectangles represent physical nodes, ovals represent application software, and DB represents the disks storing the database (and usually the database software that runs on the server).

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

Server processes (represented by "Server" in Figure 2–1) run on nodes defined to have the backend role. This layer:

- Allows the database to be distributed geographically.
- Permits replication of servers to cope with either network, node or site failures.
- Allows computer resources to be added to meet performance requirements.

Figure 2–1 The Three Layer Model



- Allows performance or geographic expansion while protecting the investments made in existing hardware and application software.

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

The three layer 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 frontend layers 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.

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.

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.

See the RTR *System Manager's Manual* for information on how to define facilities.

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.

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.

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.

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 be:

- Atomic
- Consistent
- Isolated
- Durable

These are the ACID properties of transactions. For more detail on these properties, see the Reliable Transaction Router *Application Design Guide*.

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, see the Reliable Transaction Router *Application Design Guide*.

Object-Oriented Programming

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

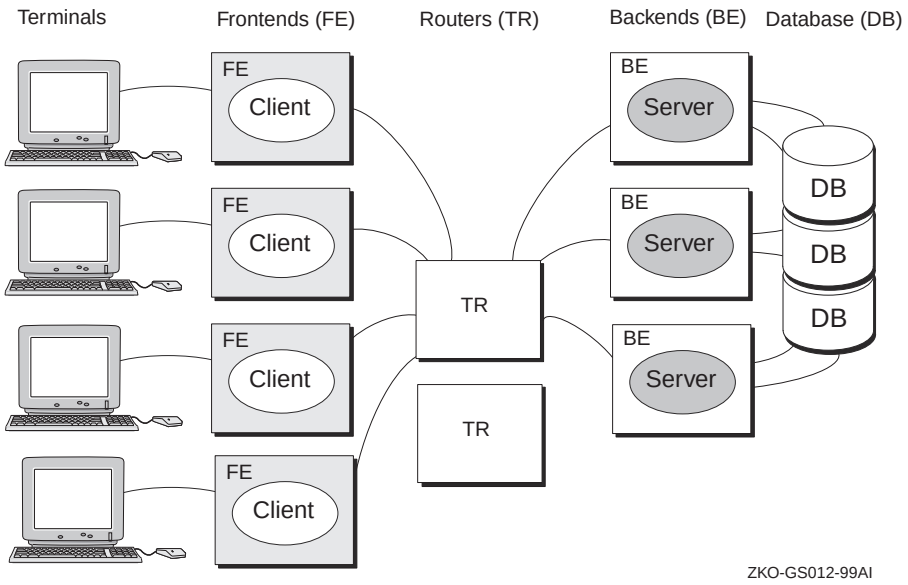


Table 2–1 Functional and Object-Oriented Programming Compared

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.
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.

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 C++ Foundation Classes manual. 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;
}
```

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.

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.

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.

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.

XA Support

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 not used by the application programs; it is only used by TM-to-RM exchanges to coordinate a transaction.

For details on using XA, see the RTR *C Application Programmer's Reference Manual* and the RTR *Application Design Guide*.

3
Reliability Features

Reliability in RTR is enhanced by the use of:

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

Servers

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.
- Shadow servers handle the *same* transactions and run on different nodes.
- Standby servers provide a node that can take over processing on a data partition when the primary server or node fails.
- 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.

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

Failover and Recovery

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

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.

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.

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.

Link Failure Recovery

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

Recovery Scenarios

This section describes how RTR recovers from different hardware and software failure. For additional information on failure and recovery scenarios, see the RTR *Application Design Guide*.

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.

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.

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.

4
RTR Interfaces

RTR provides interfaces for management and application programming.

You manage RTR with a management interface from the RTR management station. The management interfaces are:

- The command line interface or CLI
- The browser interface

The application programming interfaces (APIs) are:

- The object-oriented API for C++ programming, available with RTR Version 4.0. Use this API for all new development and, where appropriate, for new work on existing applications. An application can contain both object-oriented classes and portable API calls. This API can be used to implement applications on all platforms supported by RTR.
- The RTR API for C programming, available with RTR V3. This interface is also called the Portable API because it can be used to implement applications on all platforms supported by RTR.
- The OpenVMS API containing OpenVMS calls, available with RTR V2. This API, supported on OpenVMS only, is obsolete for new development. While applications using this API should be rewritten in the new object-oriented API to take advantage of new RTR features and capabilities, they can be changed by adding object-oriented classes rather than being rewritten.

You can use the command line interface to the API to write simple RTR applications for testing and experimentation. The CLI is described in the Reliable Transaction Router *System Manager's Manual*. Its use is illustrated in this chapter.

The RTR application programming interfaces are identical on all hardware and operating system platforms that support RTR. The object-oriented API is fully described in the manual Reliable Transaction Router *C++ Foundation Classes*. The C-programming API is fully described in the Reliable Transaction Router *C Application Programmer's Reference Manual*. Both APIs are used in designs in the RTR *Application Design Guide*.

RTR Management Station

You can manage RTR from a node on which RTR is running, from a remote node from which you send RTR commands to a node running RTR, or from a browser. The node where you enter commands, interact with the browser, or view results is your management station.

RTR Command Line Interface

The command line interface (CLI) to the RTR API enables the programmer to write short RTR applications from the RTR command line. This can be useful for testing short program segments and exploring how RTR works. For example, the following sequence of commands starts RTR and exchanges a message between a client and a server. To use these examples, you execute RTR commands simulating your RTR client application on the frontend and commands simulating your server application on the backend.

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.

The following example shows 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 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, see the 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.

[The user issues the following commands on the server application where RTR is running on the backend.]

```
$ RTR
Copyright Compaq Computer Corporation 1994.
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
```

[When the next command is issued, RTR waits for a message from the client.]

```
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.]
```

[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
```

[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
```

Browser Interface

With the RTR browser interface, your management station has a network-browser-like display from which you can view RTR status and issue RTR certain commands with a point-and-click operation, rather than by entering commands. Figure 4–1 shows one screen of the browser interface as you may view it from your management station running Microsoft Internet Explorer. Not all RTR CLI commands are accessible to the browser interface, only the most commonly used commands are available with the RTR browser. The browser interface provides help (for forms input) and logging windows, and navigational aids between displays.

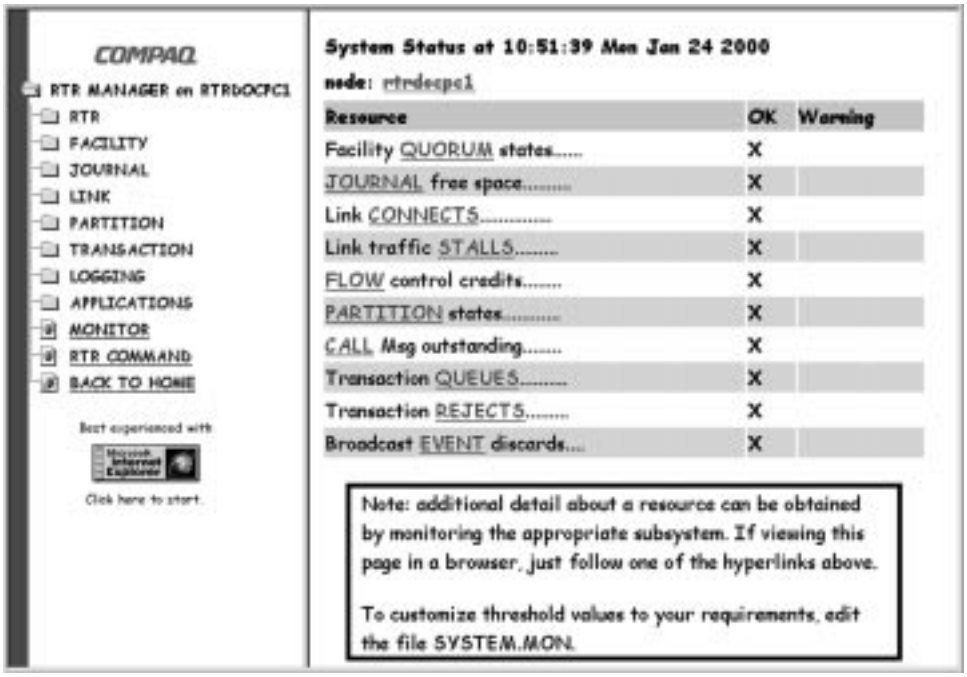
Application Programming Interfaces

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

RTR Object-Oriented Programming Interface

You can use the object-oriented programming interface to write C++ applications that use RTR. For more information on the object-oriented programming interface, see the RTR *C++ Foundation Classes* manual and the RTR *Application Design Guide*.

Figure 4–1 RTR Browser Interface



Sample C++ client code Example of object creation in an RTR client program.

```
//
// 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

Example of object creation in an RTR server program.

```
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);
}
```

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, see the RTR *C Application Programmer's Reference Manual* and the RTR *Application Design Guide.*

Snippets from client and server programs using the RTR C-programing API follow and are more fully shown in the RTR *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.

5
The RTR Environment

The RTR environment has two parts:

- The system management environment
- The runtime environment

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 managment 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 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:

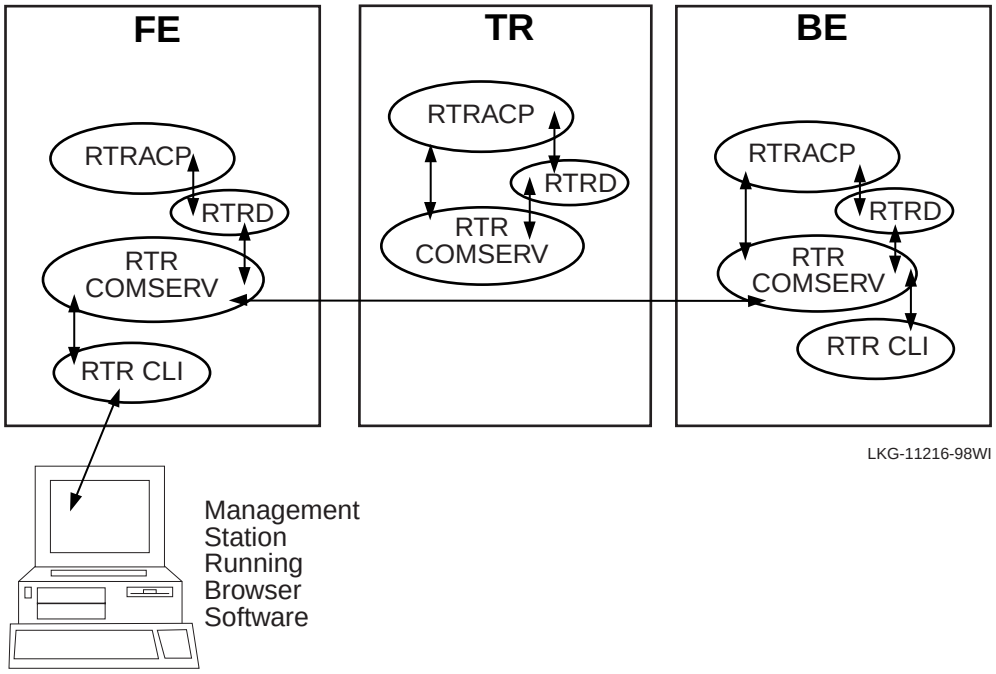
- Receives commands from RTR
- Remains temporarily waiting for another command
- Exits automatically when idle for some time

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)

The RTR system management environment is illustrated in Figure 5–1.

Figure 5–1 RTR System Management Environment



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 RTR *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.

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 RTR *System Manager's Manual*.

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 RTR *System Manager's Manual* chapter "Partition Management."

The RTR Runtime Environment

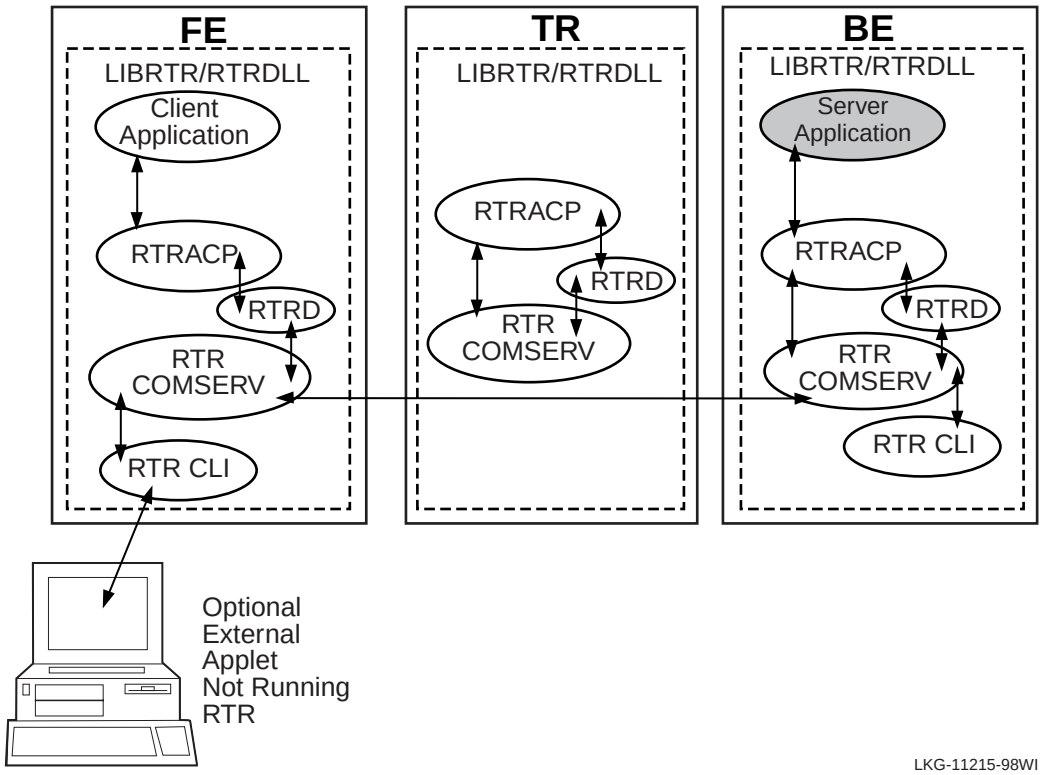
When all RTR and application components are running, the RTR runtime environment contains:

- Client application

- Server application
- RTR shareable image, LIBRTR
- RTR control process, RTRACP
- RTR daemon, RTRD

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



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. RTR *Release Notes* 2. RTR *Installation Guide* 3. RTR *Migration Guide* (if upgrading from RTR V2 to V3) 4. RTR *System Manager's Manual*
an applications or system management developer, programmer, or software engineer	RTR *Application Design Guide* RTR *C++ Foundation Classes* RTR *C Application Programmer's Reference Manual*

Glossary

A few additional terms are defined in the Glossary to the Reliable Transaction Router *Application Design Guide*.

ACID

Transaction properties supported by RTR: atomicity, consistency, isolation, durability.

ACP

The RTR Application Control Process.

API

Application Programming Interface.

applet

A small application designed for running on a browser.

application

User-written software that uses employs RTR.

application classes

The C++ API classes used for implementing client and server applications.

backend

BE, the physical node in an RTR facility where the server application runs.

bank

An establishment for the custody of money, which it pays out on a customer's request.

branch

A subdivision of a bank; perhaps in another town.

broadcast

A nontransactional message.

callout server

A server process used for transactional authentication.

channel

A logical port opened by an application with an identifier to exchange messages with RTR.

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 browser, perhaps running an *applet*, could connect to a web application that acts as an RTR client, sending data to a server through RTR.

In other contexts, a client can be a physical system, but in the context of RTR and in this document, such a system is always called a frontend or a node.

client classes

C++ foundation classes used for implementing client applications.

commit process

The transactional process by which a transaction is prepared, accepted, committed, and hardened in the database.

commit sequence number (CSN)

A sequence number assigned to an RTR commit group, established by the vote window, the time interval during which transaction response is returned from the backend to the router. All transactions in the commit group have the same CSN and lock the database.

common classes

C++ foundation classes that can be used in both client and server applications.

concurrent server

A server process identical to other server processes running on the same node.

CPU

Central processing unit.

data marshalling

The capability of using systems of different architectures (big endian, little endian) within one application.

data object

See RTRData object.

deadlock

Deadly embrace, a situation that occurs when two transactions or parts of transactions conflict with each other, which could violate the consistency ACID property when committing them to the database.

disk shadowing

A process by which identical data are written to multiple disks to increase data availability in the event of a disk failure. Used in a cluster environment to replicate entire disks or disk volumes. See also transactional shadowing.

dispatch

A method in the C++ API RTRData class which, when called, interprets the contents on the RTRData object and calls an appropriate handler to process the data. The handler chosen to process the data is the handler registered with the transaction controller. This method is used with the event-driven receive model.

DTC

Microsoft Distributed Transaction Coordinator.

endian

The byte-ordering of multibyte values. Big endian: high-order byte at starting address; little endian: low-order byte at starting address.

event

RTR or application-generated information about an application or RTR.

event driven

A processing model in which the application receives messages and events by registering handlers with the transaction controller. These handlers are derived from the C++ foundation class message and event-handler classes.

event handler

A C++ API-derived object used in event-driven processing that processes events.

facility

The mapping between nodes and roles used by RTR and established when the facility is created.

facility manager

A C++ API management class that creates and deletes facilities.

facility member

A defined entity within a facility. A facility member is a role and node combined. Can be a client, router or server.

failover

The ability to continue operation on a second system when the first has failed or become disconnected.

failure tolerant

Software that enables an application to continue when failures such as node or site outages occur. Failover is automatic.

fault tolerant

Hardware built with redundant components to ensure that processing survives component failure.

frontend

FE, the physical node in an RTR facility where the client application runs.

FTP

File transfer protocol.

inquorate

Nodes/roles that cannot participate in a facility's transactions are inquorate.

journal

A file containing transactional messages used for recovery.

key range

An attribute of a key segment, for example a range A to E or F to K.

key segment

An attribute of a partition that defines the type and range of values that the partition handles.

LAN

Local area network.

link

A communications path between two nodes in a network.

local node

The node on which a C++ API client or server application runs. The local node is the computer on which this instance of the RTR application is executing.

management classes

C++ API classes used by new or existing applications to manage RTR.

member

See facility member.

message

A logical grouping of information transmitted between software components, typically over network links.

message handler

A C++ API-derived object used in event-driven processing that processes messages.

multichannel

An application that uses more than one channel. A server is usually multichannel.

multithreaded

An application that uses more than one thread of execution in a single process.

MS DTC

Microsoft DTC; see DTC.

node

A physical system.

nontransactional message

A message containing data that does not contain any part of a transaction such as a broadcast or diagnostic message. See transactional message.

partition

RTR transactions can be sent to a specific database segment or partition. This is data content routing and handled by RTR when so programmed in the application and specified by the system administrator. A partition can be in one of three states: primary, standby, and shadow.

partition properties

Information about the attributes of a partition.

polling

A processing method where the application polls for incoming messages.

primary

The state of the partition servicing the original data store or database. A primary has a secondary or shadow counterpart.

process

The basic software entity, including address space, scheduled by system software, that provides the context in which an image executes.

properties

Application, transaction and system information.

property classes

Classes used for obtaining information about facilities, partitions, and transactions.

quorate

Nodes/roles in a facility that has quorum are quorate.

quorum

The minimum number of routers and backends in a facility, usually a majority, who must be active and connected for the valid completion of processing.

quorum node

A node, defined specified in a facility as a router, whose purpose is not to process transactions but to ensure that quorum negotiations are possible.

quorum threshold

The minimum number of routers and backends in a facility required to achieve quorum.

roles

Roles are defined for each node in an RTR configuration based on the requirements of a specific facility. Roles are frontend, router, or backend.

rollback

When a transaction has been committed on the primary database but cannot be committed on its shadow, the committed transaction must be removed or rolled back to restore the database to its pre-transaction state.

router

The RTR role that manages traffic between RTR clients and servers.

RTR configuration

The set of nodes, disk drives, and connections between them used by RTR.

RTR environment

The RTR run-time and system management areas.

RTRData object

An instance of the C++ API RTRData class. This object contains either a message or an event. It is used for both sending and receiving data between client and server applications.

secondary

See shadow.

server

A server is always a server application or process, one that reacts to a client application's units of work and carries them through to completion. This may involve updating persistent storage such as a database file, toggling the switch on a device, or performing another pre-defined task. In the context of RTR, a server must run on a node defined to have the backend role.

In other contexts, a server may be a physical node, but in RTR and in this document, physical servers are called backends or nodes.

server classes

C++ foundation classes used for implementing server applications.

shadow

The state of the server process that services a copy of the data store or primary database. In the context of RTR, the shadow method is transactional shadowing, not disk shadowing. Its counterpart is primary.

SMP

Symmetric MultiProcessing.

standby

The state of the partition that can take over if the process for which it is on standby is unavailable. It is held in reserve, ready for use.

TPS

Transactions per second.

transaction

An operation performed on a database, typically causing an update to the database. Analogous in many cases to a business transaction such as executing a stock trade or purchasing an item in a store. A business transaction may consist of one or more than one RTR transaction. A transaction is classified as original, replay, or recovery, depending on how it arrives at the backend:

Original—Transaction arrived on the first attempt from the client.
Replay—Transaction arrived after some failure as the result of a re-send from the client (that is, from the client transaction-replay buffers in the RTRACP).
Recovery—Transaction arrived as the result of a backend-to-backend recovery operation (recovery from the journal).

transaction controller

A transaction controller processes transactions. A transaction controller may have 0 or 1 transactions active at any moment in time. It is through the transaction controller that messages and events are sent and received.

transactional message

A message containing transactional data.

transactional shadowing

A process by which identical transactional data are written to separate disks often at separate sites to increase data availability in the event of site failure. See also disk shadowing.

two-phase commit

A database commit/rollback concept that works in two steps: 1. The coordinator asks each local recovery manager if it is able to commit the transaction. 2. If and only if all local recovery managers agree that they can commit the transaction, the coordinator commits the transaction. If one or more recovery managers cannot commit the transaction, then all are told to roll back the transaction. Two- phase commit is an all-or-nothing process: either all of a transaction is committed, or none of it is.

WAN

Wide area network.

Index