HP's Reliable Transaction Router
May 2010

HP's Reliable Transaction Router (RTR) provides reliable transaction-messaging services between multivendor clients and servers. The clients and servers in an RTR application network can be any mix of HP's OpenVMS, HP-UX, Linux, and Windows servers. Therefore, RTR allows heterogeneous systems to be consolidated into a single, highly-redundant, reliable, and scalable application network.

Transaction integrity is provided via the two-phase commit protocol. Recovery from a node failure is immediate and transparent to the users. In-flight transactions are preserved; and no data is lost, allowing RPOs (Recovery Point Objectives) of zero to be met.

Planned downtime can be eliminated by rolling upgrades through the network. The network can be scaled by adding nodes with no application changes.

To a large extent, RTR is based on the same technology as that of HP's OpenVMS active/active split-site clusters.[1]
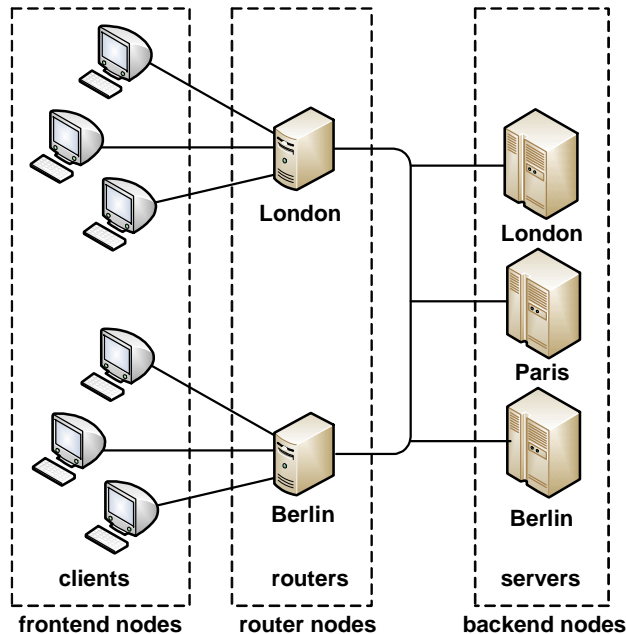
The RTR Architecture

RTR is based on a three-tier software model. The three tiers are the client applications, the RTR router, and server applications. RTR distinguishes between hardware and software components in its terminology, which we will use in this article. *Clients* are applications that run on physical *frontend* nodes. *Servers* are applications that run on *backend* nodes. *Routers* are software service modules that run on *router* nodes.

There may be any number of each kind of node in an RTR network. In addition, a node may host any of the software tiers. For instance, a router can be hosted on a frontend node, on a dedicated router node, or on a backend node. A single node can host the entire RTR network for development purposes, though this is not a recommended production configuration for reliability reasons.

Nodes are interconnected via TCP/IP links and may be geographically distributed for disaster tolerance.

---

[1] The reader is referred to the June, 2008, *Availability Digest* article OpenVMS Active/Active Split-Site Clusters, which can be found at http://www.availabilitydigest.com/public_articles/0306/openvms.pdf, for a description of the cluster distributed lock manager and disk shadow sets that are used by RTR.
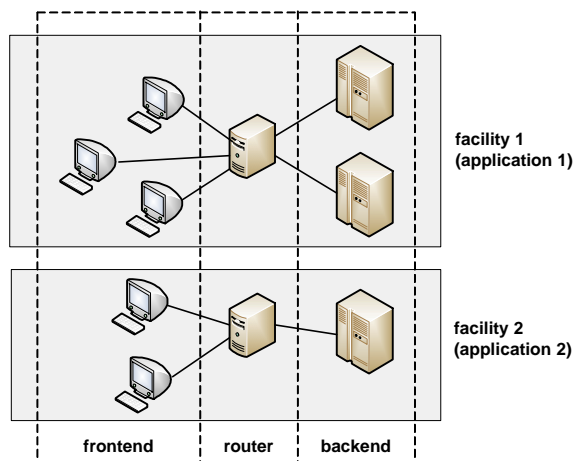
RTR Three Tier Architecture

Clients and servers are provided with an API (C, C++, or Java) that they can use to communicate with a router. Typically, a client will send a transaction message to a router. Based on the content of the message, the router will send the transaction's updates to one or more servers as appropriate under the two-phase commit protocol. Only if all involved servers vote to accept the transaction is the transaction committed. Otherwise, it is aborted.

Facilities

Many applications can use an RTR network simultaneously. To keep applications separate, RTR defines a *facility* for each application. The facility includes the clients, routers, and servers currently being used by the application. In effect, a facility is an application network running in the RTR network environment. It encompasses software components – clients, routers, and servers, not hardware components – frontend nodes, router nodes, backend nodes.

Facilities become important, for instance, in the scope of the callout server, described later.

RTR Facilities

RTR Clients

Client applications run on frontend nodes. Frontend nodes may be Windows, Linux x86_32 or x86_64, HP-UX, or HP OpenVMS systems. Clients typically provide presentation services to the users and may include some business logic.

Clients are given an API for use in communicating with the routers. A client normally connects to one router instance for the duration of a session. However, if a client's router fails, RTR automatically fails over the client to a surviving router that may also be currently serving other clients. Failover is transparent to the client; in-flight transactions are replayed by the new router.

RTR Routers

The routers are the heart of RTR. They accept transactions from their clients and route the transactions to one or more servers as appropriate. Routing of transaction updates are based on transaction-message content. For instance, consider a banking application in which savings accounts are managed by one server and checking accounts are managed by another. If a router receives a transaction message instructing that an amount be moved from a savings account to a checking account, the router will route the debit amount to the savings account server and the credit amount to the checking account server.

Transactions are executed according to the two-phase commit protocol. All involved servers must vote to commit the transaction, or else the transaction is aborted. RTR guarantees the ACID properties of a transaction – atomicity, consistency, isolation, and durability.[2]

Routers may run on any node – frontend nodes, router nodes, or backup nodes. However, multiple routers are typically deployed on multiple router nodes to provide independence, scalability and redundancy. Routing nodes may run on OpenVMS, HP-UX, Linux x86_64, Windows Server 2003, Windows XP Service Pack 2, Windows 2008 R2, or Windows 7 nodes.

Routing nodes can be easily added if more capacity is needed. Should a router fail, its clients are seamlessly reconnected to a surviving router if there is one. The routers hide the network from the clients – the clients are unaware that they are dealing with a distributed application network supporting multiple servers.

A router involves no application software unless it is running callout servers, described later.

RTR Servers

Servers in an RTR network manage the application databases. They may also include some or all of the business logic. Servers run on backend nodes comprising OpenVMS, HP-UX, Linux x86_64, Windows Server 2003, Windows XP Service Pack 2, Windows 2008 R2, or Windows 7. Each server can be managing the database of its choice, such as Oracle, Microsoft SQL Server, Sybase, Informix, or Microsoft Access. Any number of servers of any mix may be included in an RTR network.
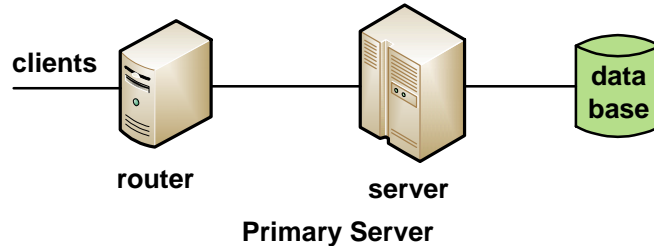
RTR provides an API for servers to communicate with their routers. Servers are voting participants in the transactional two-phase commit protocol. They can vote to commit or abort a transaction in which they are involved. If any server votes to abort the transaction, the transaction is aborted and has no effect on the databases.

Servers can be configured in several ways for reliability and scalability. In addition to acting in the primary role, servers can be configured as primary/standby pairs, shadow sets, or concurrent servers. In addition, callout servers can be employed to provide transaction authentication.

---

[2] Jim Gray, Andreas Reuter, pg. 6, Transaction Processing: Concepts and Techniques, Morgan Kaufmann Publishers, Inc.; 1993.
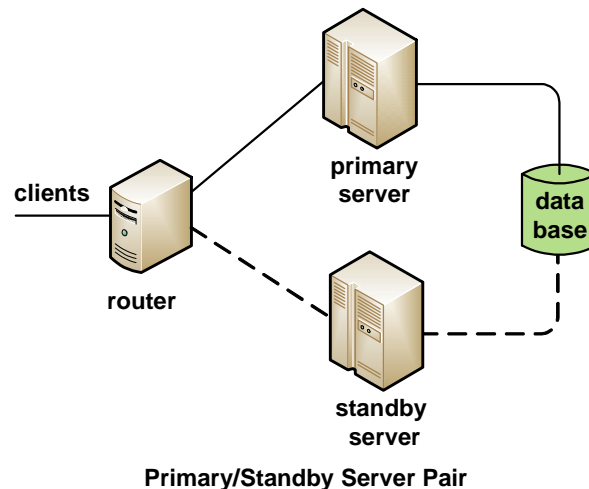
Primary Server

The fundamental configuration for a server is as the primary server for an application. If there is only a primary server configured for a database, there is no possibility of failover should the server fail. Any transactions in which it would have been a participant will abort.

Primary Server

Standby Server

A standby server may be configured to back up the primary server. This is typically done via a standard cluster configuration. Should the primary server fail, the router will direct further transaction to the backup server.

Both servers must have a connection to the database. In standard clusters such as Microsoft's Windows Server Failover Cluster (WSFC)[3] – formerly known as the Microsoft Cluster Server (MSCS) – or HP's ServiceGuard cluster,[4] only the primary server can have a set of application partitions mounted. The standby server can be handling application processing for different partitions, or it can be running other applications.

Primary/Standby Server Pair

The primary and standby servers in standard clusters must be collocated in the same facility. In these clusters, failover requires applications to be started in the standby server, disk volumes to be mounted by the standby applications, and the database checked for corruption. Failover is typically measured in minutes. During this time, the application is paused; and transactions will time out. In-flight transactions are not protected and must be resubmitted to the standby server when it comes online.

OpenVMS clusters, on the other hand, provide distributed lock and cache management. Therefore, if the servers are running on OpenVMS nodes, both the primary and standby servers can have the database open for reads and writes (see footnote 1) and can actively share in the transaction load. This leads to instantaneous failover times transparent to the users. In-flight transactions are protected. In addition, the primary and standby servers can be located in different data centers, though degraded performance of a standby node using a remote database must be considered.

---

[3] Windows Server Failover Clustering, *Availability Digest*; April 2010.
   http://www.availabilitydigest.com/public_articles/0504/microsoft_cluster.pdf
[4] HP's ServiceGuard Clustering Facility, *Availability Digest*; May 2007.
   http://www.availabilitydigest.com/public_articles/0205/serviceguard.pdf

There can be many instances of a standby server. RTR will cascade failover to other standby servers should there be a standby-server failure. Furthermore, a primary server can also act as a standby server for other primary nodes.

In an active/standby configuration, there is only one copy of the database. Therefore, the database should use a redundant technology such as RAID so that no single disk failure will take down the system. The primary/standby configuration will survive any single node failure but will not survive a site failure. Even if an OpenVMS cluster with geographically-distributed nodes is used, a failure of the site containing the database will take out the application. To get protection against site failures, shadow servers, described next, should be used.
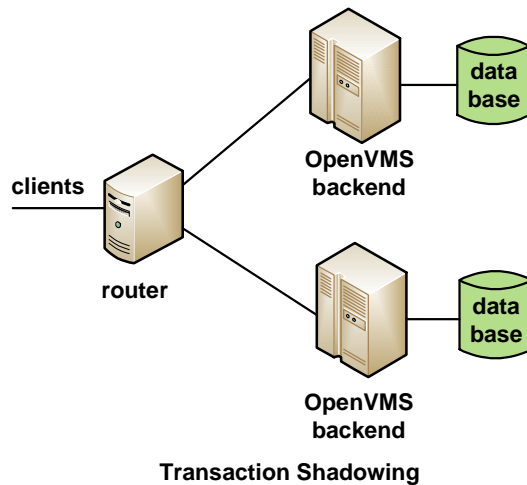
Shadow Server

The application database can be shadowed at the server level with shadow servers. In this configuration, a remote standby shadow server is provided and has its own copy of the application database. The router submits transactions to both the active shadow and the standby shadow, which independently process them. Using the facilities provided by OpenVMS clusters, RTR ensures that both database copies are kept in synchronism. This method is called *transaction shadowing*.

Furthermore, the active and standby servers can each be configured as collocated active/standby pairs as described earlier.

Should there be a node failure, database failure, or even an entire site failure, the RTR network and all of its applications survive since the other site continues processing. Failover is transparent to the users since in-flight transactions are protected.
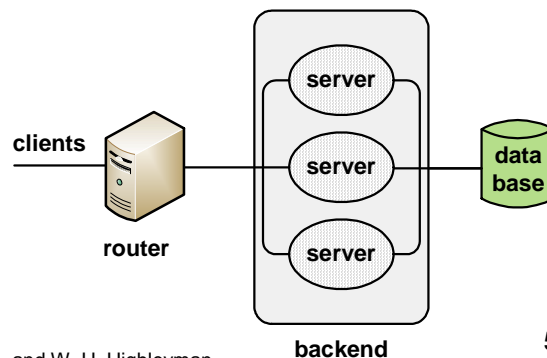
There can only be two members in a shadow server set, and the systems implementing the two members must be identical. Reads are distributed between the shadow-set members for load balancing. The shadow-set members can be geographically separated, though distance may be limited for performance considerations.

Transaction Shadowing

Should there be a failure of one member of the shadow set, the surviving member will journal further transactions until the failed member is restored. At this time, the surviving member will use the journal to recover the database of the member being returned to service. There is no impact on ongoing operations during the recovery process.

Concurrent Servers

Concurrent servers provide a pool of servers in a single backend node. All servers can be updating the same partitions in the database connected to that node. As a transaction is received, RTR routes it to one of the concurrent servers, which processes it against the application database. If a server fails, the

Concurrent Servers

transaction is transparently resubmitted to another server in the pool.

Concurrent servers can increase the performance of a backend node by introducing an element of parallel processing. The system administrator can adjust the number of servers in the pool according to current or projected volume. Thus, concurrent servers also provide a measure of scalability.

One problem with concurrent servers is that a programming error can cascade through the entire pool of servers. If a transaction causes a server to fail because of a programming error, the transaction will be submitted to another server. That server may fail as well, and the process will continue until all servers in the pool have failed. To prevent this from happening, a transaction will be aborted if it causes three instances of a server to crash.

Concurrent servers can be configured in any backend, whether the backend is acting as a primary server, a standby server, or a shadow server. The implementation of concurrent servers varies by operating system. They may be implemented as threads within a single process, as lightweight processes, or as multiple processes.

Callout Server

A callout server provides authentication services for a facility. It is associated with a particular facility and receives all transactions directed to or passing through that facility. It may run on a router node or on a backend node in the facility.

Though it makes no database updates, a callout server is a voting member of each transaction presented to it. It is typically used for authorization and validation checks. If it finds that a transaction is inappropriate, it votes not to commit; and the transaction is aborted. Consequently, a callout server can provide transaction authentication with no application changes.

Callout servers may also be used for other purposes, such as for audit-trail logging.

Broadcast

RTR provides a broadcast capability for clients and servers to send mass notifications to each other. A client or server may establish a class of broadcast. Clients and servers may subscribe to one or more broadcast classes to receive notifications.

One caveat is that if the client network does not provide a broadcast capability, and if there are a large number of clients, broadcast messages to clients will have to be sent to clients on a point-to-point basis. This could create a large network load. Broadcasts to servers do not share this problem as there are generally a limited number of servers.

Fault Recovery

Properly configured, RTR survives any single fault. Most recovery is transparent with in-flight transactions preserved.

Router Failures

Multiple instances of the router can be configured on router nodes and backend nodes. Clients are assigned to a particular router instance to route their transactions.

If a router fails either because it crashes or because its node fails, RTR will reconnect the clients served by the failed router to surviving routers in the RTR network. Recovery is transparent to the users since RTR reroutes transactions in progress to the new routers.

When a failed router is returned to service, the clients that it had been servicing are returned to it.

Server Failures

Concurrent servers provide multiple servers on the same node. Standby servers provide multiple servers on different nodes. Shadow servers provide multiple servers on different clusters. The recovery procedures are different for each. However, properly configured, RTR recovers transparently from any single server fault as long as there is another server to which to fail over.

Primary Server

If there is only a single primary server in the RTR facility, its failure is not recoverable. There is no backup to which to recover. The application is down until the server is returned to service.

Concurrent Servers

If one of the servers in a concurrent server pool fails, any transaction that it was in the process of handling is resubmitted transparently to another server in the pool. RTR will attempt to restart the failed server; and if successful, the server will be returned to the pool.

Standby Servers

Standby servers are paired with primary servers in standard cluster architectures. The primary server and one or more standby servers connect to a common (presumably redundant) database. In standard clusters, only the primary server may have the database mounted. In OpenVMS clusters, the primary and all standby servers may connect to the common database; and all may be active in processing transactions. The distinction between primary and standby servers is blurred in this case.

In standard clusters, the failure of the primary server causes a standby server to take over transaction processing. Failover is controlled by a cluster manager. Typically, applications on the standby server are started, the database is mounted by the new application instances, and the database is checked for corruption (a failure at the wrong time could corrupt the database, for instance by interrupting a block split). If the database is corrupted, it must be repaired. Only then can the standby server begin accepting transactions. Failover time for standard servers is typically measured in minutes. During this time, the application is down.

In OpenVMS clusters, all servers are actively processing transactions. Therefore, the failure of a server simply means that further transactions are routed to surviving servers. Any transactions that were being processed by the failed server are simply routed to surviving servers. Failover is instantaneous and transparent to the users.

Shadow Servers

With concurrent servers and standby servers, there is no database recovery required since there is only one database. It continues in use following failover and fallback.

However, with shadow servers, there are two databases that must be kept in synchronism. During normal operation, synchronization is the responsibility of RTR. This is accomplished by sending each transaction to both shadow servers.

However, if one server (or database) fails, all transactions are processed only by the surviving shadow server. There must be a mechanism to resynchronize the database of the failed server when it is returned to service.

Journaling solves this problem. When one member of a shadow set fails, the other member records all subsequent transactions in a journal. When the failed member is returned to service, its database is brought up to date by draining to it the queue of transactions stored in the journal. When its database is synchronized, it is returned to service.

Facilities within the OpenVMS cluster also allow the two databases to be compared and the standby database to be corrected if it differs in any way from the primary database. Furthermore, facilities exist to copy the contents of a shadow database to a recovering database if journaling is not sufficient.

Network Failures

The failure of a network link is generally detected by timeout. If a link between a client and a router fails, the failure is treated as a router failure; and the client is connected to another router. If the failure is in a link connecting a router to a server, the failure is treated as a server failure; and the router is reconnected to a standby server or to a shadow server if there is one.

Scalability

RTR supports several levels of scalability. Additional frontend, router, and backend nodes can be added to an existing RTR network with no application changes and without impacting current operations. Thus, if a node is becoming too heavily loaded, one or more additional nodes of the same type can be added; and clients, routers, or servers can be relocated to reduce the load on the affected node. Likewise, nodes can be removed if desired.

Different partitions of an application database can run on different nodes to allow parallel updating of the database.

The size of a concurrent server pool can be expanded or reduced by the system administrator.

Eliminating Planned Downtime

RTR eliminates planned downtime for configurations comprising standby or shadow servers. A server can be taken down, upgraded, and returned to service without affecting operations. The upgrade can be rolled through the RTR network in this way node-by-node.

If a database in a cluster needs to be upgraded, for instance, to change its schema, the cluster must be taken down since there is only one database instance in the cluster. In this case, planned downtime is required. However, if a database in a shadow set must be upgraded, this can be accomplished by performing the upgrade on the shadow set databases one at a time; and planned downtime is not required.

Programming Interfaces

The client and server APIs provided by RTR support several programming languages, including Java, C++, and C. The XA interface is supported for Oracle, Microsoft SQL Server, and any conforming X/Open DTP resource manager.

RTR supports object-oriented programming. RTR functions can be mapped into Java objects and C++ foundation classes.

System Management

RTR provides several system-management options. A CLI (command line interpreter) allows a system administrator to enter any of the RTR configuration and management commands.

The RTR Manager provides a web-browser interface to set up and manage an RTR configuration and individual nodes. Likewise, the RTR Explorer provides a web-browser interface to monitor the behavior of an RTR network. RTR Explorer generates warning, error, and fatal alarms. The administrator can zoom in on specific nodes to get a more detailed view of current operations. Between these two facilities, the most popular RTR commands can be executed by an intuitive GUI interface.

A smart plug-in (SPI) is available for the HP OpenView systems management facility. OpenView monitors, manages, and controls distributed applications running on a wide variety of systems.

An RTR network can be managed from a node on which RTR is running, on an independent remote node, or via a web browser on or accessing a node running RTR.

Summary

RTR allows the expansion of an existing application to incorporate previously incompatible systems. RTR allows heterogeneous systems to be consolidated into a single highly-redundant, reliable, and scalable application network.

Failover from software or hardware faults is automatic and is generally transparent to the users. Transaction integrity is guaranteed via the use of the two-phase commit protocol. Extensive management facilities are provided via intuitive web-browser interfaces. Though the use of RTR in an existing application is somewhat invasive because RTR API calls must be added to clients and servers, no other application changes need be made except to add additional business functions.

RTR is an important adjunct to OpenVMS clusters. These clusters execute commands at the database read/write/update level and do not support transaction processing. RTR running in an OpenVMS cluster brings transaction processing to these active/active clusters.