TIBCO Rendezvous®

Concepts

Software Release 8.1
April 2008

TIBCO®
The Power of Now®

Important Information

Contents

Figures

Tables

Preface

This manual explains TIBCO Rendezvous® software and the concepts needed to understand it and use it effectively. It is part of the documentation set for Rendezvous Software Release 8.1.

Topics

Manual Organization

The first group of chapters introduce the basics of Rendezvous software:

- Chapter 1, Product Overview, page 1 is a high-level introduction to Rendezvous software.
- Chapter 2, Architecture, page 15 introduces architecture concepts, and explains specific architectural elements.
- Chapter 3, Fundamentals, page 39 discusses the details of messages and subject-based addressing.™
- Chapter 4, The Rendezvous Daemon, page 55 introduces `rvd`—the daemon process.
- Chapter 5, Subject Names, page 61 presents details of subject name syntax and semantics.

The second group of chapters discuss important concepts of the Rendezvous programming architecture:

- Chapter 6, Data, page 69 presents details of datatypes, formats and storage issues.
- Chapter 7, Events, page 79 presents events, queues, dispatch, callback functions, and event semantics.
- Chapter 8, Transport, page 99 presents transports and their parameters.
- Chapter 9, Virtual Circuits, page 119 presents virtual circuits.
- Chapter 10, Guidelines for Programming, page 127 is a collection of rules of thumb for Rendezvous programmers.

The third group of chapters present quality of service features:

- Chapter 11, Certified Message Delivery, page 139
- Chapter 12, Distributed Queue, page 181
- Chapter 13, Fault Tolerance Concepts, page 195
- Chapter 14, Fault Tolerance Programming, page 213
- Chapter 15, Developing Fault-Tolerant Programs, page 227

Related Documentation

This section lists documentation resources you may find useful.

TIBCO Product Documentation

The following documents form the Rendezvous documentation set:

- *TIBCO Rendezvous Concepts*

  Read this book first. It contains basic information about Rendezvous components, principles of operation, programming constructs and techniques, advisory messages, and a glossary. All other books in the documentation set refer to concepts explained in this book.

- *TIBCO Rendezvous C Reference*

  Detailed descriptions of each datatype and function in the Rendezvous C API. Readers should already be familiar with the C programming language, as well as the material in *TIBCO Rendezvous Concepts*.

- *TIBCO Rendezvous C++ Reference*

  Detailed descriptions of each class and method in the Rendezvous C++ API. The C++ API uses some datatypes and functions from the C API, so we recommend the *TIBCO Rendezvous C Reference* as an additional resource. Readers should already be familiar with the C++ programming language, as well as the material in *TIBCO Rendezvous Concepts*.

- *TIBCO Rendezvous Java Reference*

  Detailed descriptions of each class and method in the Rendezvous Java language interface. Readers should already be familiar with the Java programming language, as well as the material in *TIBCO Rendezvous Concepts*.

- *TIBCO Rendezvous .NET Reference*

  Detailed descriptions of each class and method in the Rendezvous .NET interface. Readers should already be familiar with either C# or Visual Basic .NET, as well as the material in *TIBCO Rendezvous Concepts*.

- *TIBCO Rendezvous COM Reference*

  Detailed descriptions of each class and method in the Rendezvous COM component. Readers should already be familiar with the programming environment that uses COM and OLE automation interfaces, as well as the material in *TIBCO Rendezvous Concepts*.

- *TIBCO Rendezvous Administration*

  Begins with a checklist of action items for system and network administrators. This book describes the mechanics of Rendezvous licensing, network details, plus a chapter for each component of the Rendezvous software suite. Readers should have *TIBCO Rendezvous Concepts* at hand for reference.

- *TIBCO Rendezvous Configuration Tools*

  Detailed descriptions of each Java class and method in the Rendezvous configuration API, plus a command line tool that can generate and apply XML documents representing component configurations. Readers should already be familiar with the Java programming language, as well as the material in *TIBCO Rendezvous Administration*.

- *TIBCO Rendezvous Installation*

  Includes step-by-step instructions for installing Rendezvous software on various operating system platforms.

- *TIBCO Rendezvous Release Notes*

  Lists new features, changes in functionality, deprecated features, migration and compatibility information, closed issues and known issues.

How to Contact TIBCO Customer Support

For comments or problems with this manual or the software it addresses, please contact TIBCO Support as follows.

- For an overview of TIBCO Support, and information about getting started with TIBCO Product Support, visit this site:

  http://www.tibco.com/services/support

- If you already have a valid maintenance or support contract, visit this site:

  http://support.tibco.com

Entry to this site requires a username and password. If you do not have a username, you can request one.

Chapter 1 Product Overview

This chapter introduces TIBCO Rendezvous® software.

Topics

Introduction to Rendezvous Software

Rendezvous software makes it easy to create distributed applications that exchange data across a network. You get software support for network data transport and network data representation. Rendezvous software supports many hardware and software platforms, so programs running on many different kinds of computers on a network can communicate seamlessly.

From the programmer's perspective, the Rendezvous software suite includes two main components—a Rendezvous programming language interface (API) and the Rendezvous daemon.

Rendezvous software includes several programming language interfaces, which are efficient, easy to use, and compatible with most other libraries (including window systems). Other books in the documentation set describe the Rendezvous API for various programming languages.

The Rendezvous *daemon* runs on each participating computer on your network. All information that travels between program processes passes through the Rendezvous daemon as the information enters and exits host computers. The daemon also passes information between program processes running on the same host.

Rendezvous *programs* are programs that use Rendezvous software to communicate over a network.

A Rendezvous *distributed application system* is a set of Rendezvous programs that cooperate to fulfill a mission.

Benefits of Programming with Rendezvous Software

You gain numerous advantages when you program with Rendezvous APIs instead of other network APIs (such as TCP/IP sockets).

Benefits During Development

- Rendezvous software eliminates the need for programs to locate clients or determine network addresses.
- Rendezvous software simplifies the development of distributed application systems by hiding the networking details.
- Rendezvous software makes it easy to develop resilient systems because redundant data producers are transparent to consumers. (A *producer* is any program that sends data. A *consumer* is any program that receives data.)
- Rendezvous software is thread-safe, so you can use it with multi-threaded programs.

Benefits During Use

- Rendezvous programs can publish multicast messages to distribute information quickly and reliably to many consumers.
- Rendezvous programs can use request/reply interactions, such as queries.
- Rendezvous programs are location independent, and port easily between platforms.

Benefits as Programs Evolve

- Rendezvous distributed application systems are easier to maintain than traditional networked application systems.
- Rendezvous distributed application systems scale smoothly as you add new component processes.
- Rendezvous distributed application systems have longer useful lifetimes than traditional networked application systems.
- Rendezvous software supports many hardware and software platforms, so programs port smoothly as your environment evolves and expands.

Simplifying Distributed System Development

Rendezvous software eases distributed system development in these ways:

- Decoupling and Data Independence
- Location Transparency
- Architectural Emphasis on Information Sources and Destinations
- Reliable Delivery of Whole Messages
- Certified Message Delivery
- Distributed Queues
- Fault Tolerance (see Fault Tolerance Concepts on page 195)

Decoupling and Data Independence

Distributed systems can be difficult to develop, maintain and port. One reason for this difficulty is that components running on networked hosts are often tightly coupled—components must agree on network connections, the low-level format for data transfer, and other details. Rendezvous software allows looser coupling between the components of a distributed system. Loose coupling decreases costs for development, operation and maintenance, and increases system longevity.

Rendezvous self-describing data messages promote data independence; producers and consumers of data can communicate even if they do not share the same internal representations for data. Communicating programs can run on different hardware architectures, even though they use different bit order, byte alignment or numeric representations.

Data independence also eases program evolution. Producers can gracefully add new content fields to their messages without invalidating legacy receivers.

Location Transparency

Rendezvous software uses *subject-based addressing*™ technology to direct messages to their destinations, so program processes can communicate without knowing the details of network addresses or connections. Subject-based addressing conventions define a uniform name space for messages and their destinations.

The locations of component processes become entirely transparent; any application component can run on any network host without modification, recompilation or reconfiguration. Application programs migrate easily among host computers. You can dynamically add, remove and modify components of a distributed system without affecting other components.

Architectural Emphasis on Information Sources and Destinations

Decoupling distributed components eliminates much of the complexity traditionally associated with network programming. Rendezvous software frees you to devote more resources to solving application problems.

In the past, network programming was so complex that programmers often structured systems and individual component programs to minimize that complexity—even if the resulting architecture did not fit the application domain as well as it could. Rendezvous software lets you think about distributed system architecture in new ways. You can divide the system into modules along natural boundaries implied by the application's information content.

The first step in developing a distributed system is to identify sources and destinations of information. For example, sources of information include news-wire services, data entry stations, point-of-sale stations, sensors and measuring devices. Destinations of information include data displays and visualization stations, device controllers, statistical analyzers, and personal wireless devices. Components such as databases, schedulers, materials trackers and decision support interfaces can often be both sources and destinations of information within a larger system. Analyzing a distributed application problem in these terms very often suggests the most natural, efficient and flexible solution.

Reliable Delivery of Whole Messages

Rendezvous software provides reliable communications between programs, while hiding the burdensome details of network communication and packet transfer from the programmer. Rendezvous software takes care of segmenting and recombining large messages, acknowledging packet receipt, retransmitting lost packets, and arranging packets in the correct order. You can concentrate on whole messages, rather than packets.

While some conventional network APIs guarantee reliable delivery of point-to-point messages, most do not guarantee reliable receipt of multicast (or broadcast) messages. Multicast messages can often be lost when some of the intended recipients experience transient network failures. Rendezvous software uses proprietary reliable multicast protocols to deliver messages despite brief network glitches.

For programs that require even stronger guarantees, see Certified Message Delivery on page 139.

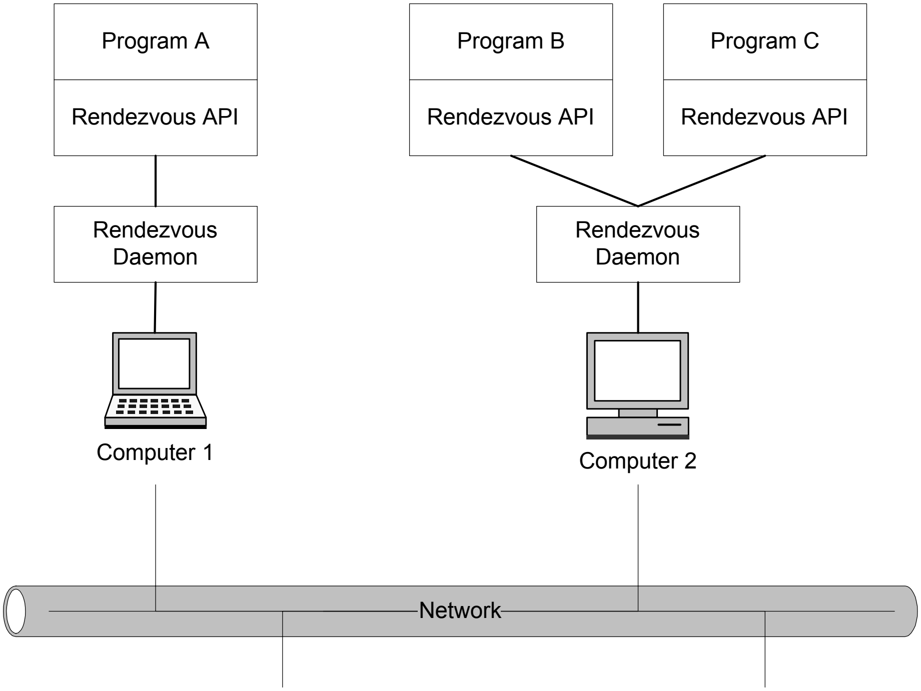
Rendezvous Components

Rendezvous software adds two parts to your current operating environment:

- An API library. Each program process links a version of the Rendezvous API library. Other books in the Rendezvous documentation set describe versions of the Rendezvous API for use with several different programming languages.
- The Rendezvous communications daemon—in most environments, one daemon process runs on each host computer.

Figure 1 illustrates the interaction of these two parts in the operating environment. Computer 1 runs program A and a daemon process. Computer 2 runs two programs, B and C, which connect to the network using a single Rendezvous daemon process. All three programs can communicate with one another.

Figure 1 Rendezvous Operating Environment



Any computer can run any number of Rendezvous programs. Usually all the programs on one computer share the same Rendezvous daemon.

Rendezvous Daemon

Programs depend on the Rendezvous *daemon*, a background process (`rvd`), for reliable and efficient network communication. The Rendezvous daemon completes the information pathway between Rendezvous program processes across the network. (Usually the daemon runs on the same computer as the program; however, it is possible to connect to a remote daemon.)

Programs attempt to connect to a Rendezvous daemon process. If a local daemon process is not yet running, the program starts one automatically and connects to it.

The Rendezvous daemon arranges the details of data transport, packet ordering, receipt acknowledgment, retransmission requests, and dispatching information to the correct program processes. The daemon hides all these details from Rendezvous programs.

The Rendezvous daemon is nearly invisible to the programs that depend upon it. Programs send and receive information using Rendezvous communications calls, and the Rendezvous daemon does the work of getting information to the right place.

For details of the daemon process and its command line, see `rvd` on page 36 in *TIBCO Rendezvous Administration*.

Multiple Daemons

In most situations, each computer runs one Rendezvous daemon process, which serves all the programs running on the same computer. In general, adding multiple daemons does *not* increase performance.

Rendezvous Language Interfaces

Rendezvous programmers can choose from several languages. Rendezvous programs communicate seamlessly, no matter which languages you choose. The language barrier disappears.

Table 1 Programming Languages

C	The C API supports the widest array of hardware platforms.
C++	The C++ API facilitates object-oriented design while preserving complete compatibility with the ANSI C API. (C++ programs can also call Rendezvous C API functions.)
Java	Programs can be stand-alone Java applications or browser-based Java applets. Java programs can communicate across the Internet to distant Rendezvous programs.
COM	The COM interface brings Rendezvous communications to programming systems based on COM, such as Visual Basic.
Perl 5	A Perl 5 loadable module presents a Rendezvous API that is parallel to the ANSI C API. Use Perl for rapid prototyping of C programs, or for remote system administration tasks. See Perl 5 Interface on page 371 in *TIBCO Rendezvous C Reference*.

Rendezvous Functionality

Table 2 summarizes the areas of functionality in Rendezvous software. Each language interface provides a subset of these facilities.

Table 2 Overview of Functionality

Facility	Description
Message	• Translate between universal wire format and local data formats • Manipulate messages and fields
Event	• Listen for messages by subject name • Register interest in timers and I/O events
Transport	• Define delivery scope, delivery mechanism and protocols • Connect to the network • Send messages
Queue	• Create and manipulate event queues • Dispatch events
Queue Group	• Customize event dispatch by combining queues
Certified Message Delivery	• Confirm delivery of each message to each registered recipient • Deliver messages despite process termination and restart
Distributed Queue	• Distribute a service over several processes
Fault Tolerance	• Coordinate redundant processes to achieve application-level fault tolerance

Developing Distributed Systems

You can use the Rendezvous libraries with any popular software development methodology. Add these steps to the development process:

- Identify information producers and consumers during the analysis phase. Consider whether the information flow is best characterized as request/reply or as publish/subscribe (or both). Use this analysis to guide architectural design.

  For example, if your program behaves as a request/reply application, you can divide it into client and server components. If your program behaves as a publish/subscribe application, you can divide it into sending components and listening components. In some cases programs communicate in both modes—request/reply and publish/subscribe.

- Identify the kinds of information that components exchange, the events that generate the information, and the information that triggers other events. Use this analysis to establish subject naming conventions and to design the content of data messages.
- Write your Rendezvous programs in one of the supported languages, using Rendezvous API calls as appropriate.
- Compile with the appropriate Rendezvous header files. Link your program code with the Rendezvous library. Link additional libraries for extended functionality. For details, see the Programmer’s Checklist section in each programming language reference manual.

Programming Examples

Programming examples for the supported languages are included on the installation media. When you install Rendezvous software, these examples appear in the `src/` directory.

We encourage you to examine these programs before writing your own programs.

Platform Support

Platforms are no longer listed in this book; instead, see the installation directories tables for each operating system in *TIBCO Rendezvous Installation*.

Chapter 2 Architecture

This chapter describes the architecture of the Rendezvous API.

Topics

Rendezvous API Architecture

Table 3 outlines the core architectural elements of Rendezvous programming interfaces.

Table 3 Architecture Summary

Element	Description
Message	Messages carry data among program processes or threads. Messages contain self-describing data fields. Programs can manipulate message fields, send messages, and receive messages.
Event	Programs create event objects to register interest in significant conditions. For example, dispatching a listener event notifies the program that a message has arrived; dispatching a timer event notifies the program that its interval has elapsed. Programs define event callback functions to process events.
Event Queue	Programs create event queues to organize events. A queue holds a sequence of event objects that are ready for dispatch.
Event Queue Group	Programs create event queue groups to prioritize event processing.
Event Dispatch	Programs dispatch events from queues or queue groups, processing each event with the corresponding callback function.
Transport	Programs use transport objects to send messages and listen for messages. A transport determines three aspects of message delivery: • Delivery scope—the potential range of its messages • Delivery mechanism—the path that its messages travel • Delivery protocol—the ways in which programs cooperate and share information concerning message delivery Transport objects of various types combine these aspects to yield different qualities of service—for example, intra-process delivery, network delivery, reliable delivery, certified delivery, distributed queue delivery.
Event Driver	Rendezvous software includes an event driver that places events in event queues. (Program code cannot access the event driver.)

New for Release 7

Release 7 is compatible with release 6 in all respects (except for new features).

The following sections of this chapter summarize the most important changes in release 7.

⚠ *We recommend that readers familiar with earlier releases read these sections.*

Topics

PGM and TRDP

Rendezvous release 7 introduces support for PGM and RPTP protocols, so sites with PGM networks can now use Rendezvous software. (Previous Rendezvous releases supported only TRDP protocols.)

When installing Rendezvous software, you can choose either of two variants—TRDP or PGM. To determine which variant to deploy, consult your site administrator and network administrator.

- Both variants support the same API.
- Programs developed using one variant transfer instantly to the other variant. It is not necessary to relink nor recompile. No programming changes are required. (See also, Make Transport Parameters Flexible on page 136.)
- The two variants differ only in the daemon executables and the protocols they use to communicate with one another.
- To bridge between a PGM network and a TRDP network, use two process instances of the Rendezvous routing daemon (`rvrd`)—one from each variant. For details, see Connecting PGM and TRDP Networks with Routing Daemons on page 99 in *TIBCO Rendezvous Administration*.
- Some hardware and operating system platforms do not support PGM; for availability, see *TIBCO Rendezvous Installation*.
- The PGM variant requires privileged access to raw sockets on each daemon host computer. In some environments, raw access is considered a security risk. If this privilege is not available, use the TRDP variant. (For example, the Solaris Zones feature precludes using raw sockets. On most UNIX variants, raw access requires that the daemon run as `root`. To determine whether this privilege is available in your environment, consult your local system and network administrators.)

Protocols and Services

RPTP Rendezvous release 7 adds a new reliable point-to-point protocol, which we call RPTP. RPTP is similar to TCP, but operates over a UDP channel.

Table 4 PGM and TRDP Communications (Sheet 1 of 2)

Aspect	TRDP	PGM
Multicast Communication	TRDP broadcast or multicast protocol over UDP channel	PGM multicast protocol over PGM channel

Table 4 PGM and TRDP Communications (Sheet 2 of 2)

Aspect	TRDP	PGM
Point-to-Point Communication	TRDP unicast protocol over UDP channel	RPTP over UDP channel
Direct Communication	RPTP over UDP channel	RPTP over UDP channel

Identifying the Daemon Variant

Daemon executables and processes have the same name in either variant. For example, the Rendezvous daemon is named `rvd`. You can use the start banner, the log file, or the browser administration interface to distinguish between the variants:

- TRDP daemons do *not* indicate their variant in banners or browser interfaces. This absence implies a TRDP variant.
- PGM daemons explicitly indicate PGM in their banners and browser interfaces.

See Also Specifying Direct Communication on page 105

Connecting PGM and TRDP Networks with Routing Daemons on page 99 in *TIBCO Rendezvous Administration*

Security Features

Rendezvous release 7 adds security features based on secure socket layer (SSL) protocols. These features affect three areas of communication:

- Client to daemon—see Secure Client Connections on page 21
- Between routing daemon neighbors—both `rvrd` and `rvsrd` offer this feature.
- Browser to daemon—based on HTTPS protocols.

Additional Security Tools

These three areas do not include access control, nor program-level encryption. For flexible access control facilities and an encryption API, use the companion product, TIBCO Rendezvous DataSecurity.

Certificates

X.509 certificates are crucial to these security features. For details, see Certificates and Security on page 52.

Performance

In all three of the communication areas listed above, security features require encryption and decryption computations. In general, these cryptographic computations detract from performance (compared to ordinary, non-secure daemons) for the sake of security.

Cryptographic computations require a pool of random data. Some operating systems maintain a random pool, and others do not. On operating systems that do not, each process must initialize its own random pool, which can cause delays as the program starts. You might observe such a delay when starting a secure daemon, or any application program that connects to a secure daemon.

Secure Client Connections

Rendezvous release 7 introduces two new daemons—`rvsd` and `rvsrd`—featuring SSL for secure connections to client program transports:

- `rvsd`, the Rendezvous secure communications daemon, corresponds to `rvd`
- `rvsrd`, the Rendezvous secure routing daemon, corresponds to `rvrd`

Administrators can deploy these secure daemons in situations where clients must connect securely over a non-secure network.

For an API summary, see Secure Daemon on page 60.

For administrative details, see Chapter 6, Secure Daemons (rvsd and rvsrd), on page 157 in *TIBCO Rendezvous Administration*.

Virtual Circuits

Release 7 introduces virtual circuits as a new quality of service. Virtual circuits feature Rendezvous communication between two terminals over an exclusive, continuous, monitored connection.

For a complete description, see Chapter 9, Virtual Circuits, on page 119

Direct Communication

Release 7 introduces direct communication capabilities. Now two application programs can conduct point-to-point communications without intermediary Rendezvous daemon (`rvd`) processes.

This arrangement can improve overall performance when a program sends many point-to-point messages to the same inbox.

For more information, see Direct Communication on page 116.

Routing Daemon Subject Weights and Path Costs

Routing path costs and import subject weights increase administrator control over Rendezvous routing daemons. Network administrators can use these parameters to arrange load balancing and declare preferred routes in a fault-tolerant routing configuration.

Path costs guide routers to select the best route. Subject weights divide messages among routers by subject name, while retaining fault tolerance.

For a detailed description of these parameters, see Load Balancing on page 86 in *TIBCO Rendezvous Administration*.

Batch Mode for Transports

Release 7 introduces a timer-based batch mode for network transport objects.

For more information, see Batch Modes for Transports on page 118.

Architectural Changes for Release 6

Release 6 introduced many architectural changes from earlier releases. The remainder of this chapter summarizes the most important changes in release 6.

> ⚠ *We recommend that readers familiar with earlier releases read the remainder of this chapter.*

Topics

API Name Changes

Rendezvous release 6 introduced important changes with respect to earlier releases. To reflect the sweeping nature of these changes, every API entry point has a new name.

In most instances, the new name reflects a new way of thinking about an object or function. In other cases, the name changes mostly in its prefix, and does not reflect an architectural change.

In general, experienced Rendezvous programmers and administrators will find the new names intuitive.

Despite these changes, releases 6 and later can interoperate with release 5 in most cases. For details, see Compatibility with Earlier Releases on page 37.

Code Migration

Release 6 (and later) includes a superset of the functionality of release 5—even though the API has a different structural organization. So one might erroneously think that the fastest way to migrate programs to release 6 would be to traverse the code, making the necessary changes in each line. We discourage this approach.

Instead, we recommend that you first understand the new architecture. Resource material includes this book, API reference books, and example source code. With this understanding, restructuring existing programs becomes straightforward.

Replacement for Sessions

In release 5 (and earlier releases) the *session* was a key architectural element, encapsulating a Rendezvous daemon connection, an event manager context, an event queue, and an event dispatcher—all in a single object. Release 6 (and later) separates these roles for greater flexibility and ease of use:

- Programs create *transport* objects to send outbound messages and receive inbound messages. Some types of transport connect to the Rendezvous daemon; others do not.
- Programs create *event queues* for inbound messages and other events.
- The Rendezvous *event driver* places events on the correct event queues. The event driver is not accessible from program code.
- Programs *dispatch* events from their event queues.

This division of labor gives programs more control. For example, programs control the assignment of events to event queues, queue priorities, and all aspects of event dispatch.

See Also

Transport, page 99
Events, page 81
Event Driver, page 83
Event Queues, page 84

Hollow Session

Obsolete

In releases 5 and earlier, Rendezvous messages and their operations relied on resources stored with the session object. Consequently, programs that used messages but did not use communications still required a *hollow* session—with the required resources but without a connection to the Rendezvous daemon.

In release 6 and later, Rendezvous messages are independent of such resources, so the concept of a hollow session is now obsolete (along with any kind of session).

Synchronous Session

Obsolete

In release 5 and earlier, each session included an event manager and message pump to dispatch messages and other events. Programs that required a different control mechanism could create *synchronous* sessions, and use their own message pump.

In release 6 and later, the concept of transport replaces the obsolete concept of a session. Transport objects do not include a message pump; instead, programs have complete control over event queues and event dispatch, while a separate event driver places events in queues.

Consequently, synchronous sessions are now obsolete (along with any kind of session).

All programs arrange all the details of event dispatch, which can be synchronous or asynchronous, or a combination of the two.

Multi-Threading Changes

The session-centered architecture of release 5 (and earlier releases) was an essentially single-threaded program model. Multi-threading was possible, but difficult.

Release 6 (and later) uses a multi-threaded programming model from the start. It is both thread-safe and thread-aware. (However, messages are not thread-aware objects; a program that accesses the same message simultaneously in separate threads must protect the message with an explicit locking mechanism. We expect this situation to occur only rarely.)

Release 6 (and later) requires operating system support for multi-threading.

Nonetheless, release 6 (and later) does support single-threaded programs. For example, a program can run as a single thread that dispatches events.

Event Manager Contexts and Related Restrictions

Obsolete

In eliminating sessions, Release 6 also eliminated the concept of event manager contexts, along with all the related restrictions on their use in multi-threaded programs.

Any program thread can create events that use any queue; any program thread can dispatch any queue—even queues created in other threads. Several threads can all dispatch events from the same queues; the callback function runs in the thread that dispatches the event.

Listeners and Senders (Conceptual Change)

In release 5, the Rendezvous object model included both sender and listener objects. This model encouraged programmers to think of a sender object as the actor in sending a message, and to think of a listening object as the actor in listening for messages and presenting the inbound messages to the callback function.

In release 6 (and later) the object model recasts these concepts:

- Sender objects no longer exist. Instead we encourage programmers to think of the transport object as the actor in sending a message.
- Listener objects are now *event* objects. We encourage programmers to think of the transport object and the listener event object as co-actors in listening for messages; each one has a critical role, and listening stops if either one is invalid.

Blocking Changes

Sleep and Select

Obsolete

In releases 5 and earlier, we warned programmers against using calls that block, such as the operating system calls `sleep()` and `select()`. Release 6 (and later) removes this restriction. Programmers can use these calls as needed, without adverse effects to Rendezvous software.

Because the event driver of release 6 (and later) is independent of program threads, operating system calls that block cannot obstruct the flow of events to queues. Nonetheless, programs must continue to dispatch queues appropriately, even when some program threads block.

Prompt Return

Obsolete

In releases 5 and earlier, we warned programmers to ensure that callback functions return promptly and do not block. Callback functions that did not observe these precautions would obstruct important internal mechanisms of Rendezvous software.

Because the event driver of release 6 (and later) is independent of program threads, blocking and long-running callback functions cannot interfere with Rendezvous internal mechanisms.

However, programmers must recognize that blocking and longing-running callback functions can affect program responsiveness. For most programs, we still recommend avoiding these behaviors. When callback functions must block, or must run for long periods of time, we recommend dispatching from several threads to ensure prompt processing of all events.

Signal Events Obsolete

Obsolete

In release 5 and earlier, Rendezvous software supported signal events in some operating systems.

Release 6 (and later) no longer supports signals.

Message Changes

Unitary Messages

Obsolete

The concept of a unitary message (that is, a message containing a single datum and no fields) is obsolete in release 6 (and later).

When a program from release 5 or earlier sends a unitary message and a program using release 6 (or later) receives it, the receiver reformats the inbound message, placing the single datum in a field named `_data_` with no field identifier.

See also Message Structure on page 73.

Encrypted Fields

Obsolete

The field type `TIBRVMSG_ENCRYPTED` is obsolete in release 6 (and later). Inbound messages from older programs that contain fields of this type produce an error when the receiving program attempts to extract them from the message.

Certified Message Delivery Changes

Certified Delivery Confirmation Order

In release 5 and earlier, confirming certified delivery of a message implicitly confirmed all previously unconfirmed messages with lower sequence numbers as well. For example, if message number `57` arrived, and messages `55` and `56` remained unconfirmed, then confirming message `57` automatically confirmed messages `55` and `56`.

In release 6 and later, programs can confirm individual messages without implicitly confirming any unconfirmed gaps in the sequence. Nonetheless, the listening CM transport holds the confirmation protocol message until all gaps are filled. Replaying the example, confirming message `57` does not confirm messages `55` and `56`. Instead, the listening CM transport records the confirmation, but does not send it back to the sending transport; when the program fills the gap by confirming messages `55` and `56`, the listening transport sends the confirmations for messages `55` through `57`.

Unsupported and Obsolete Advisory Messages

These advisory messages are no longer supported, and are no longer documented. Although daemons might still present some of these advisories, their semantics are proprietary and can change at any time:

- `HOST.START`
- `HOST.STOP`
- `LISTEN.START`
- `LISTEN.STOP`
- `SESSION.START`
- `SESSION.STOP`
- `LICENSE.NODES`
- `QUEUE.LISTENER.SUBSCRIPTION_MISMATCH`

Compatibility with Earlier Releases

Release 7 is compatible with release 6 in all respects (except for new features).

The remainder of this section details compatibility issues with release 5, which were introduced in release 6.

Datatypes

In release 6 (and later) messages support several new datatypes in message fields—for example, array types. Release 5 programs cannot interpret these new types, and treat them as *bad data*.

Release 6 (and later) supports a different set of numeric types than release 5. When a release 6 (or later) program receives a message with old numeric types from a release 5 program, it automatically converts the old numeric types to the corresponding new numeric types.

The datatype `RVMSG_ENCRYPTED` is obsolete.

rvd

In release 6 (and later) `rvd` accepts connections from programs compiled and linked with the Rendezvous release 5 API.

Release 5 `rvd` accepts connections from programs compiled and linked with the API of release 6 (or later)—however, for best performance we recommend `rvd` from the most recent release.

`rvd` processes from releases 6 (and later) can interoperate on the same network as `rvd` from release 5.

Programs compiled and linked with releases 5 and 6 (and later) can interoperate on the same network, except when they use new features or obsolete features.

rvrd

- An `rvrd` process from release 5 *cannot* establish a neighbor connection with `rvrd` processes from release 6 (or later).
- `rvrd` processes from release 5 can coexist in the same network as `rvrd` processes from release 6 (or later).

  However, you must be careful to avoid duplicate service. For more information, see Common Topology Errors on page 94 in *TIBCO Rendezvous Administration*.

- `rvrd` from release 6 (or later) interoperates with `rvrd` from release 5.
- `rvrd` from release 5 interoperates with `rvrd` from release 6 (or later).

rva

- `rva` from release 6 (or later) interoperates with release 5 `rvd`.
- `rva` from release 6 (or later) interoperates with release 5 Java programs.
- `rva` from release 5 interoperates with `rvd` from release 6 (or later).
- `rva` from release 5 *cannot* interoperate with Java programs from release 6 (or later).

rvcache

- `rvcache` from release 6 (or later) interoperates with release 5 programs or `rvd`.
- `rvcache` from release 5 interoperates with programs or `rvd` from release 6 (or later).

Chapter 3 Fundamentals

This chapter describes the fundamental concepts of Rendezvous software—messages, self-describing data, subject-based addressing, and interactions.

Topics

Messages and Data

Computer hardware and operating system platforms use different conventions for data representation. Data from one platform can be unintelligible on another platform. Rendezvous software uses a unified data representation to exchange messages among all supported platforms.

Messages are the common currency that Rendezvous programs use to exchange data. Rendezvous messages contain fields of *self-describing data*. Every message has a *subject name*, which describes its destination.

All data that enters or leaves a program through the Rendezvous daemon must be encapsulated in the fields of a message. As an abstraction, a message is a collection of self-describing data fields, which travel together between programs, processes or threads.

Programs can manipulate messages even before opening the Rendezvous environment (see The Rendezvous Environment on page 51).

Fields

Each field contains one data item of a specific datatype. Programs can identify and access the individual fields of a message either by *name* or by *numeric identifiers*.

From the programmer’s point of view, a message is a set of fields. Programs manipulate messages using API calls. A program can *create* a message, *add* fields to it, *remove* fields from it, *get* a field from a message, *update* the data value in a field, and *destroy* a message.

Wire Format

At a lower level, beyond these abstract operations, each message exists as a byte sequence in Rendezvous *wire format*—a uniform representation suitable for network communication among diverse hardware, operating system, and programming language platforms. Programs never access this representation, yet it is the foundation for all Rendezvous communication.

See Also

Data on page 69.

Supplementary Information for Messages

In addition to its fields, a message also carries other information.

Address Information

Before sending a message, a program adds these two pieces of address information to the message:

- The *subject* (also called the *send subject*) of a message directs it to its destinations.
- In contrast, the optional *reply subject* of a message is a return address, to which recipients can send reply messages.

Encoding Tag

In release 7.2 and later, sending programs can tag outbound messages to indicate the encoding of strings within the messages. Receivers can use these tags to translate strings correctly.

Certified Delivery Information

Certified message delivery (CM) features add more information to a message:

- Before sending a message using a certified delivery (CM) transport, a program can set a *time limit* for certified delivery guarantees on the message.

When a program sends a message using a CM transport, the transport automatically labels the outbound message with two additional pieces of CM information:

- The *sender name*—that is, the correspondent name of the CM transport.
- The *sequence number* of the message.

See Also

Labeled Messages on page 152

Self-Describing Data

Self-describing data is data that has been annotated by the producer, so all consumers can interpret and use the data properly. Every message and every message field consists of self-describing data. Table 5 presents the annotations that augment data so it is self-describing.

Table 5 Self-Describing Data

Element	Description
Field Annotations	
type	Producers must designate the type of every message field. Rendezvous software uses a set of wire-format datatype designations to characterize data by type and size. For example, the type `TIBRVMSG_I32ARRAY` denotes an array of 32-bit integers.
count (number of elements)	Producers must specify the length of all array data—that is, the number of elements in an array.
field name	Producers can label the fields of a message with names. Consumers use field names to select specific fields from messages.
field identifier	Producers can label the fields of a message with numeric identifiers. Consumers can use field identifiers to select specific fields from messages. All field identifiers in a message must be unique within that message.
Message Annotations	
subject name	Producers must label every outbound message with a send subject name (also called the *send subject name)*, which describes its content and destination set.
reply subject name	Producers can label an outbound message with a reply subject name, to which consumers can send reply messages.

See Also

Data on page 69
Subject Names on page 61

Names and Subject-Based Addressing

Subject-based addressing technology helps messages reach their destinations without involving programmers in the details of network addresses, protocols, hardware and operating system differences, ports and sockets. Subject-based addressing conventions define a simple, uniform name space for messages and their destinations.

Programs that produce data arrange that data into messages, label each outbound message with a subject name, and *send* those messages. Programs that consume data receive it by *listening* to subject names; a consumer listening to a subject name receives all messages labeled with that name, from the time it begins listening until it stops listening.

A subject name is a character string that specifies the destination of a message, and can also describe the message content. For programs to communicate, they must agree upon a subject name at which to *rendezvous* (hence the name of this product). Subject-based addressing technology enables *anonymous* rendezvous, an important breakthrough, which decouples programs from the network addresses of specific computers.

Communication between programs is also anonymous—consumers need not know where or how data is produced, and producers need not know where data is consumed, nor how it is used. Producers and consumers only need to agree to label data items with the same set of subject names, and that the actual data be in a form that both can manipulate and interpret.

Anonymous communication decouples data consumers from data producers. Consumers are insulated from most changes in data producing software, including the replacement of producer processes, and the shifting of responsibilities among a collection of producer processes.

Subject-based addressing technology places few restrictions on the syntax and interpretation of subject names. System designers and developers have the freedom (and responsibility) to establish conventions for using subject names. For more information, see Subject Name Syntax on page 62.

Programs can listen for wildcard subject names to access a collection of related data through a single subscription. For more information about wildcard subjects, see Using Wildcards to Receive Related Subjects on page 66.

Subject Names

Subject names consist of one or more *elements* separated by dot characters (periods). The elements can be used to implement a *subject name hierarchy* that reflects the structure of information in an application system.

These strings are examples of valid subject names:

```
RUN.HOME
RUN.for.Elected_office.President
```

See Also Subject Names, page 61.

Wildcard Subject Names

A program can receive a group of related subjects by listening for a *wildcard subject*. The following examples illustrate wildcard syntax and matching semantics. (For further examples, see Subject Name Syntax on page 62.)

Wildcard Subject	Matching Subjects	Non-Matching Subjects	Reason
RUN.*	RUN.AWAY	RUN.Run.run	extra element
	RUN.away	Run.away	case sensitivity
	RUN.Home	RUN	missing element
RUN.>	RUN.AWAY	HOME.RUN	position mismatch
	RUN.Run.run	Run.away	case sensitivity
	RUN.SWIM.BIKE.SKI	RUN	missing element

Inbox Names

An *inbox name* specifies a destination that is unique to a particular process. Rendezvous software uses point-to-point techniques to deliver messages with inbox subject names. See also:

- Subject-Based Addressing and Message Destinations on page 45.
- Multicast and Point-to-Point Messages on page 46.
- Inbox Names on page 115.

Subject-Based Addressing and Message Destinations

Rendezvous programs communicate by sending messages. Each message bears a subject name, which simultaneously specifies a delivery mode and the destination of the message:

- An *inbox name* specifies *point-to-point* delivery (also called *unicast* delivery) and a unique destination process.

  The process can receive several copies of the message by creating several listener events that listen to the same inbox name. Two processes cannot share an inbox name.

- Any other subject name is called a *public subject name*, and specifies *multicast* (or broadcast) delivery to a destination set. The destination set includes all programs that listen to that subject name; the set can be large, small, or empty, depending upon the number of listeners.

  Public subject names usually describe the content or subject of the message. The sender need not know which programs are listening when it sends a message—just as a radio disc jockey in a studio does not know who is listening at a particular moment in time.

Whether point-to-point or multicast, Rendezvous communication is always *anonymous*. Senders publish messages addressed to subject names, rather than specific computers, programs or sockets. Receivers subscribe to subject names (and receive all messages addressed to those names), rather than establishing unique communications pathways with senders.

Subject naming conventions are a key part of distributed system design in most Rendezvous applications. Subject naming conventions define a uniform name space for messages and their destinations. Subject-based addressing technology helps messages reach their destinations without involving programmers in the details of network addresses, subscription lists, ports and sockets.

Both multicast and point-to-point messages can flow over several types of transport mechanisms. For details, see Transport Scope on page 101, and Constructing the Network Parameter on page 107.

Virtual Circuit

Virtual circuits are special, and behave contrary to the general rule. All messages on a virtual circuit travel point-to-point, whether the subject is an inbox name or a public subject name.

See Also

Subject Names, page 61

Multicast and Point-to-Point Messages

Rendezvous software uses subject-based addressing to support both *reliable multicast* communications and *point-to-point* communications. These two kinds of messages differ slightly in the syntax of their subject names, but dramatically in their behavior.

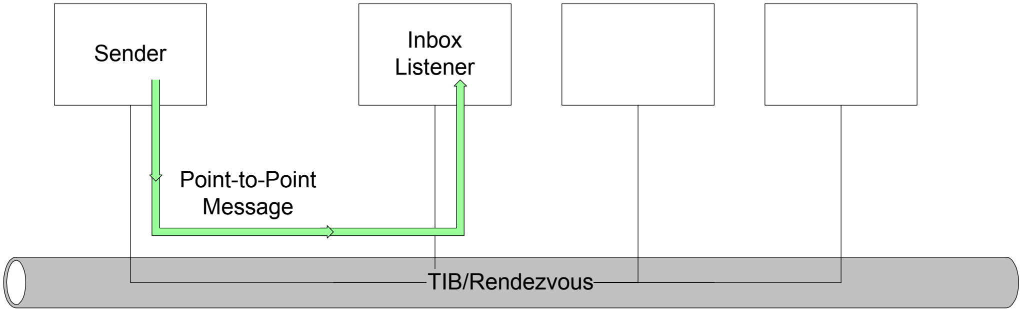
Both kinds of messages are efficient; in either multicast or point-to-point mode, the message itself traverses the network only once.

Point-to-Point Messages

A *point-to-point message* has at most one recipient program; its destination is an *inbox*—a subject name created dynamically by a specific program process. Figure 2 illustrates this model of message delivery. (One process can receive several copies of the message by listening several times to the same inbox name, but two processes cannot share an inbox name.)

All inbox names begin with `_INBOX` as their first element. A Rendezvous function creates inbox names dynamically; programs may not invent inbox names (in contrast to public subject names).

Figure 2 Point-to-Point Message



A point-to-point message is like a telegram sent to one specific person—no other person can receive it. The sender must know the name of the intended recipient. An inbox name is analogous to the address on a telegram. Creating an inbox name establishes a unique address for receiving point-to-point messages. To send a point-to-point message, the sending program must know the inbox name of the destination. (A recipient makes its inbox name known by multicasting it to potential senders using a prearranged subject name.)

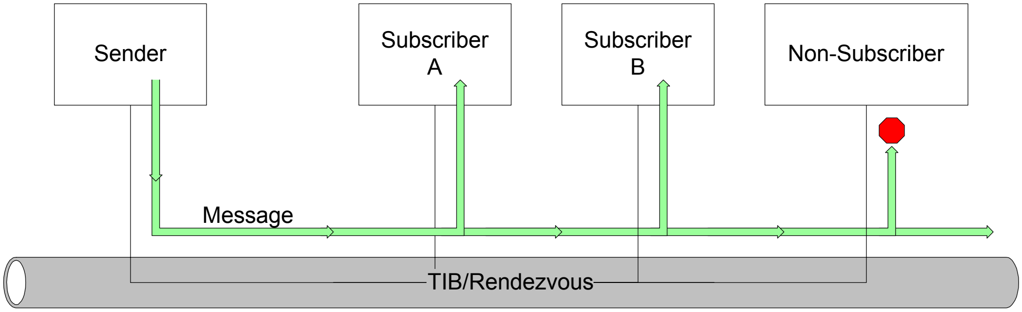
Multicast Messages

A *multicast message* is any message with many potential recipients. Potential recipients are called *subscribers*.

The *subject name* of the message indirectly determines the message's *destination*—the set of subscribers that receive the message. Every subscriber to that subject name receives the message; non-subscribers do not receive it. The set of subscribers can change dynamically, depending on which programs are listening for the subject name. If no subscribers exist, then none receive the message (even though it still travels the network). Figure 3 illustrates this model of message delivery.

Rendezvous applications are free to invent public subject names, constrained only by the syntactic and semantic rules in Subject Names on page 61.

Figure 3 Reliable Multicast Message



Multicast messages are like radio broadcasts; the sender picks a frequency, and any listener who tunes to that frequency receives the broadcast. The public subject name is analogous to a radio frequency; any program that listens for a subject receives all messages bearing that subject name.

A multicast message does not imply multicast packet protocols. A multicast message can reach its destination using multicast protocols, broadcast protocols, or even intra-process communications—depending on the transport object that the program uses to send the message. (In contrast, Rendezvous documentation uses phrases such as *multicast group* and *multicast addressing* to indicate multicast network protocols.)

Messages Mediate Interactions Between Programs

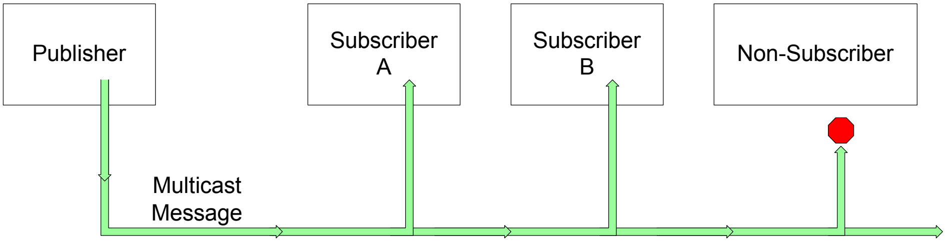
Three distinct kinds of interactions occur between programs in the Rendezvous environment:

- Publish/subscribe interactions, such as distribution of information to multiple recipients.
- Request/reply interactions, such as queries or transactions to individual services.
- Multicast request/reply interactions, such as queries to one or more anonymous services.

Publish/Subscribe Interactions

Publish/subscribe interactions are driven by events (usually the arrival or creation of data)—a publisher makes information available for general distribution. Communication is in one direction (publisher to subscribers), and often one-to-many as shown in Figure 4. The complete interaction consists of one multicast message, published once, and received by all subscribers.

Figure 4 Event-Driven Publish/Subscribe Interaction



Example applications:

- Securities data feed handlers publish the latest stock prices to hundreds of traders on a trading floor simultaneously.
- Materials movement systems distribute data to various materials handlers, controllers and tracking systems on a factory floor.
- Inventory levels flow continuously to accounting, purchasing, marketing and other departments in a retail store.
- A bug-tracking database immediately sends bug reports and updates to all personnel interested in a particular project.
- A master database publishes updates to a set of internet mirrors.

In publish/subscribe interactions, data producers are decoupled from data consumers—they do not coordinate data transmission with each other, except by using the same subject name. Producers publish data to the network at large.

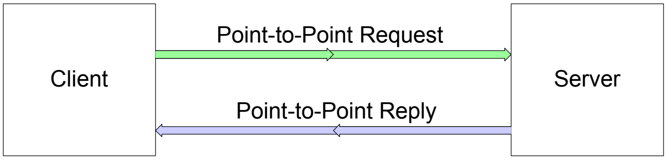
Consumers place a standing request for data by subscribing. Consumers can listen for messages on any subject(s) on the network; a subscription is a request for messages.

Rendezvous software supports publish/subscribe interactions with multicast communication.

Request/Reply Interactions

Demand for data drives *request/reply* interactions. A client requests data from a server; the server computes an individual response and returns it to the client. Communication flows in both directions, as in Figure 5. The complete interaction consist of two point-to-point messages—a request and a reply.

Figure 5 Demand-Driven Request/Reply Interaction



Demand driven computing is well-suited for distributed applications such as these examples:

- Transaction processing (as in ATM banking).
- Database query (with a remote DBMS).
- Factory equipment control.

In request/reply interactions, data producers coordinate closely with data consumers. A producer does not send data until a consumer makes a request. Each program sends its message to a specific inbox name within the other program.

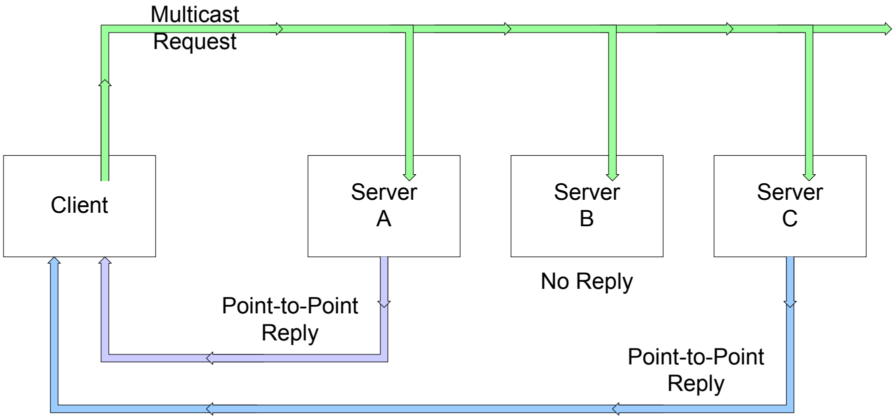
The server sends replies specifically to the client that requested the data. The requesting client listens until it receives the reply, and then stops listening (unless it expects further installments of information).

Rendezvous software supports request/reply interactions with point-to-point communication.

Multicast Request/Reply Interactions

Like request/reply interactions, *multicast request/reply* interactions are driven by demand for data. While traditional request/reply interactions involve one requestor and one server, in multicast request/reply interactions multiple servers can receive the request and respond as appropriate. Communication flows in both directions, and only some servers respond to the client request, as in Figure 6. The complete interaction consists of one multicast request message, and any number of point-to-point reply messages.

Figure 6 Multicast Request/Reply Interaction



Example applications:

- Database query with multiple servers.
- Distribution of computing sub-tasks to the first available server.
- Network management applications that multicast requests for test information.

In multicast request/reply interactions, data producers coordinate closely with data consumers. A server does not send data until a client requests it. Servers send point-to-point replies to the specific client program.

If a server has the information the consumer requested, it sends a reply. The requesting client listens until it receives one or more replies. The client stops listening by destroying the reply inbox listener.

Rendezvous software supports multicast request/reply interactions with a combination of multicast and point-to-point communication.

The Rendezvous Environment

Before using Rendezvous communications, programs must open the Rendezvous environment. Opening the environment initializes crucial global resources. All communication and event operations depend on this step; the only operations that do not require it as a precondition are message operations.

The open call creates a default event queue, an intra-process transport, an event driver, and other resources within the program. Closing the environment destroys these resources.

Table 6 Environment Calls

C	`tibrv_Open()`
	`tibrv_Close()`

C++	`Tibrv::open()`
	`Tibrv::close()`

Java	`Tibrv.open()`
	`Tibrv.close()`

COM	`Tibrv.open` `Tibrv.close`
.NET	`Environment.Open` `Environment.Close`

Certificates and Security

A *certificate* is a structured string of bytes that uniquely represents a specific identity. The structure of those bytes is determined by the X.509 identity certificate specification. A trusted certificate authority (CA) issues a certificate only to entities that meet its identification criteria (which may vary).

Certificates also play an important role in SSL and HTTPS secure communication protocols.

Rendezvous software uses certificates in four ways:

- Secure daemon components authenticate user programs.
- User programs authenticate secure daemons.
- Routing daemons authenticate other routing daemons (neighbors) when they communicate using SSL protocols.
- Web browsers authenticate Rendezvous daemon components when they communicate using HTTPS protocols.

See Also For more details, see Security Factors on page 165 in *TIBCO Rendezvous Administration*.

Certificate Encodings and Formats

When a certificate contains a private key, programs and file standards safeguard the key using password encryption. To use a certificate with a private key, a user must also have the private key password.

PEM Encoding

Rendezvous software supports PEM encoding of X.509 certificates. PEM encoding is a text format, so you can distribute certificates easily.

PKCS #12 Format

Rendezvous software supports PKCS #12 format for X.509 certificates. PKCS #12 encoding is a binary format. PKCS #12 data files usually bear the `.p12` filename extension.

Obtaining Certificates

Automatic Certificates

For ease of use, Rendezvous daemon components automatically generate their own self-signed certificates. You may use these certificates by copying them from the daemon’s browser administration interface, and distributing them to client programs, and to routing daemon neighbors as appropriate.

Self-signed certificates usually trigger a warning from web browsers. When connecting to the browser administration interface of a Rendezvous daemon component, you may safely proceed beyond these warnings when you have independently verified the identity of the daemon. To avoid these warnings, replace the automatic certificate with a certificate signed by a reliable commercial CA.

Using a Commercial CA Service

Each CA has its own policies and procedures for investigating identities and issuing certificates. Details are readily available through CA web sites, such as `www.verisign.com`.

When applying for a certificate, specify that you need these two files in either PKCS #12 binary format, or PEM text encoding:

- Public certificate
- Certificate with private key

CA Software

In some situations, application administrators prefer to act as an in-house CA, issuing identity certificates to users. CA software is available from companies such as Netscape and Microsoft. Shareware is also available, including the OpenSSL package.

Browsers

Many CA services and CA software programs can automatically install certificates in your browser. Many browsers can export certificates in PKCS #12 format (as `.p12` files).

Distributing Certificates

An X.509 certificate actually comprises two separate but interrelated certificate texts—a *public certificate* and a *certificate with private key*. To effectively use certificates, you must securely distribute the two texts to the appropriate places.

- A certificate with a private key proves the identity of the key holder. Each Rendezvous daemon process proves its identity this way. In some deployments, end users of client programs also identify themselves this way (while in other deployments, they prove their identities with user names and passwords).

- Public certificates denote the identity of a trusted party. You must distribute the public certificate text to all other programs that interact with that party.

Chapter 4 The Rendezvous Daemon

This chapter describes the Rendezvous daemon—the central communications component of Rendezvous software.

Topics

See Also

`rvd` on page 36 in *TIBCO Rendezvous Administration*

Role of the Rendezvous Daemon

The Rendezvous daemon is a background process that supports all Rendezvous communications. Distributed processes depend on it for reliable and efficient network communication. All information that travels between processes passes through a Rendezvous daemon as it enters a host computer or exits a sending process.

The Rendezvous daemon:

- Transmits outbound messages from program processes to the network.
- Delivers inbound messages from the network to program processes.
- Filters subject-addressed messages.
- Shields programs from operating system idiosyncrasies, such as low-level sockets.

The Rendezvous daemon process, `rvd`, starts automatically when needed, runs continuously and may exit after a period of inactivity. For more information, see `rvd` on page 36 in *TIBCO Rendezvous Administration*.

The Daemon and its Client Programs

The Rendezvous daemon completes the information pathway between a Rendezvous program process and the network. Each computer that runs a Rendezvous program usually runs a Rendezvous daemon process as well.

Each network transport object attempts to connect to a Rendezvous daemon process. If the daemon process is not yet running, the program starts one automatically and connects to it. If the transport cannot connect with a Rendezvous daemon, the Rendezvous transport creation call fails.

If the Rendezvous daemon exits, programs automatically restart it. Programs can monitor these events by listening for the system advisory messages `RVD.DISCONNECTED` and `RVD.CONNECTED`.

See Also Rendezvous Daemon, page 8.
Advisory message `RVD` on page 265.

Reliable Message Delivery

Rendezvous programs send and receive messages. The Rendezvous daemon at the sending computer divides messages into a stream of packets and sends them across the network. The Rendezvous daemon on each receiving computer reassembles the packets into messages and presents them to the listening program.

Standard multicast and broadcast protocols are not reliable and are unable to detect lost messages. Under normal conditions, Rendezvous reliable multicast protocols ensure that all operational hosts either receive each multicast message or detect the loss of a message. (For details and limitations, see `DATALOSS` on page 252.)

Reliability

Reliable delivery compensates for brief network failures. The receiving Rendezvous daemon detects missing packets and requests that the sending daemon retransmit them. The sending daemon stores outbound messages for a limited period of time (60 seconds), so it can retransmit the information upon request. It discards old messages after the time period elapses, and cannot retransmit after that time.

The Rendezvous daemon requires that the physical network and packet recipients are working. The Rendezvous daemon does not guarantee delivery to components that fail and do not recover for periods exceeding 60 seconds. (For stronger assurances of delivery, see Certified Message Delivery on page 139.)

When a sending daemon receives a retransmission request for data it has already discarded, it notifies the requesting daemon that it cannot retransmit it. One or both daemons present error advisories to indicate that this situation has occurred (see `DATALOSS` on page 252).

Other Features of Reliable Message Delivery

The reliable multicast protocol delivers messages *once and only once* to each subscription, despite multiple transient network failures.

It delivers all point-to-point messages from each sending transport in the order they are sent. It also delivers all multicast messages from each sending transport in the order they are sent. However, if a sending program interleaves multicast and point-to-point messages, Rendezvous software does not necessarily preserve the sending order between the two types of messages.

Rendezvous software does not preserve the absolute chronological order of messages sent by two or more different transports.

Secure Daemon

Before connecting to secure Rendezvous daemons, programs must register user information (such as a name and password, or a certificate), and the identities of one or more trusted secure daemons (such as names or certificates).

Table 7 Secure Daemon Calls

C	`tibrvSecureDaemon_SetDaemonCert()`
	`tibrvSecureDaemon_SetUserCertWithKey()`
	`tibrvSecureDaemon_SetUserNameWithPassword()`
C++	`TibrvSdContext:setDaemonCert()`
	`TibrvSdContext:setUserCertWithKey()`
	`TibrvSdContext:setUserNameWithPassword()`
Java	`TibrvSdContext.setDaemonCert()`
	`TibrvSdContext.setUserCertWithKey()`
	`TibrvSdContext.setUserNameWithPassword()`
COM	`TibrvSdContext.setDaemonCert()`
	`TibrvSdContext.setUserCertWithKey()`
	`TibrvSdContext.setUserNameWithPassword()`
.NET	`SDContext.SetDaemonCertificate`
	`SDContext.SetUserCertificateWithKey`
	`SDContext.SetUserNameWithPassword`

See Also Secure Daemons (rvsd and rvsrd) on page 157 in *TIBCO Rendezvous Administration*

Chapter 5 Subject Names

Each Rendezvous message bears a *subject* name.

Data-producing programs generate new data messages, label them with subject names, and send them using Rendezvous software. Data consumers receive data by listening to subject names. A consumer listening to a name receives all data labeled with a matching name, from the time it begins to listen until it stops listening.

Topics

Subject Name Syntax

Subject-based addressing™ technology places few restrictions on the syntax and interpretation of subject names. System designers and developers have the freedom (and responsibility) to establish conventions for using subject names. The best subject names reflect the structure of data in the application itself.

Structure

Each subject name is a string of characters that is divided into elements by the dot (.) character. It is illegal to incorporate the dot character into an element by using an escape sequence.

Length

Rendezvous limits subject names to a total length of 255 characters (including dot separators). However, some of that length is reserved for internal use.

The longest subject that (most) programs can receive is 196 characters.

The maximum total length (in characters) of subjects that your programs may send is 196 *minus* the number of elements. Include dot separators in this total.

The maximum element length is 127 characters (dot separators are not included in element length).

Typical subject names are shorter and use fewer elements. For maximum speed and throughput rates, use short subject names. For details, see Subject Name Performance Considerations on page 65.

Empty String Illegal

The empty string (“ “) is not a legal subject name.

Examples

These examples illustrate the syntax for subject names.

Table 8 Valid Subject Name Examples

`NEWS.LOCAL.POLITICS.CITY_COUNCIL`
`NEWS.NATIONAL.ARTS.MOVIES.REVIEWS`
`CHAT.MRKTG.NEW_PRODUCTS`
`CHAT.DEVELOPMENT.BIG_PROJECT.DESIGN`
`News.Sports.Baseball`
`finance`
`This.long.subject_name.is.valid.even.though.quite.uninformative`

Table 9 Invalid Subject Name Examples

`News..Natural_Disasters.Flood` (null element)
`WRONG.` (null element)
`.TRIPLE.WRONG..` (three null elements)

Special Characters in Subject Names

Table 10 Characters with Special Meaning in Subject Names

Char	Char Name	Special Meaning
_	Underscore	**Reserved Subject Names** ⚠ Subject names beginning with underscore are reserved. It is illegal for application programs to send to subjects with underscore as the first character of the first element, except `_INBOX` and `_LOCAL`. It is legal to use underscore elsewhere in subject names.
.	Dot	Separates elements within a subject name.
>	Greater-than	Wildcard character, matches one or more trailing elements.
*	Asterisk	Wildcard character, matches one element.

We recommend that you do not use tabs, spaces, or any unprintable character in a subject name.

For information about wildcard characters and matching, see Using Wildcards to Receive Related Subjects on page 66.

Subject Name Performance Considerations

When designing subject name conventions, remember these performance considerations:

- Shorter subjects perform better than long subjects.
- Subjects with several short elements perform better than one long element.
- A set of subjects that differ early in their element lists perform better than subjects that differ only in the last element.

Using Wildcards to Receive Related Subjects

Programs can listen for wildcard subject names to access a collection of related data through a single subscription.

The asterisk (*) is a wildcard character that matches any one element. The asterisk substitutes for whole elements only, not for partial substrings of characters within an element.

Greater-than (>) is a wildcard character that matches all the elements remaining to the right.

A listener for a wildcard subject name receives any message whose subject name matches the wildcard.

The following examples illustrate wildcard syntax and matching semantics.

Table 11 Semantics of Listening to Wildcard Subjects

Listening to this wildcard name	Matches messages with names like these:	But does not match messages with names like these (reason):
`RUN.*`	`RUN.AWAY` `RUN.away`	`RUN.Run.run` (extra element) `Run.away` (case) `RUN` (missing element)
`Yankees.vs.*`	`Yankees.vs.Red_Sox` `Yankees.vs.Orioles`	`Giants.vs.Yankees` (position) `Yankees.beat.Sox` (vs≠beat) `Yankees.vs` (missing element)

`*.your.*`	`Amaze.your.friends`
	`Raise.your.salary`
	`Darn.your.socks`
		`your` (missing elements)
		`Pick.up.your.foot` (position)

`RUN.>`	`RUN.DMC`
	`RUN.RUN.RUN`
	`RUN.SWIM.BIKE.SKATE`
		`HOME.RUN` (position)
		`Run.away` (case)
		`RUN` (missing element)

Table 12 Invalid Wildcard Subject Examples

Invalid Wildcards	Reason
`abc*xyz`	Asterisk (*) must take the place of one whole element, not a substring within a element.
`Foo.>.baz`	Greater-than (>) can only appear as the right-most character. Rendezvous software interprets this as a specific subject name.

Wildcard Sending

⚠ We do not recommend sending messages to wildcard subject names. Although transports do not prevent you from sending to wildcard subjects, doing so can trigger unexpected behavior in other programs that share the network.

For example, wildcard subjects can often be broader than intended, so that unrelated applications might receive messages that they cannot parse.

It is illegal for certified delivery transports to send to wildcard subjects.

Distinguished Subject Names with Special Meaning

Names that begin with an underscore character (_) are called *distinguished subject names*. Distinguished names indicate special meaning, special handling, or restricted use.

Table 13 Distinguished Subject Names

Prefix	Description
`_INBOX.`	All inbox names begin with this prefix. Programs may not create inbox names except with inbox creation calls. Programs cannot combine inbox names with wildcards. Programs must treat inbox names as opaque, not modify them, and refrain from making inferences based on the form of inbox names.
`_LOCAL.`	Messages with subject names that have this prefix are only visible and distributed to transports connected to the same Rendezvous daemon as the sender. For example, a program listening to the subject `_LOCAL.A.B.C` receives all messages sent on subject `_LOCAL.A.B.C` from *any* transport connected to the *same* daemon. A Rendezvous daemon does not transmit messages with `_LOCAL` subjects beyond that daemon.
`_RV.`	Subject names with this prefix indicate advisory messages, including informational messages, warnings and errors. Programs must not send to subjects with this prefix. For a description of advisory messages, see Advisory Messages on page 242.
`_`	All other subject names that begin with an underscore character indicate internal administrative messages. Programs must not send to subjects with this prefix.

Chapter 6 Data

This chapter describes datatypes and formats.

Topics

Self-Describing Data

Rendezvous programs exchange self-describing data. Each item of *self-describing data* consists of up to six parts:

- *Data* The data itself.
- *Type* An indicator to manipulate and interpret the data as an integer, a string, a composite message, or other datatype.
- *Size* The number of bytes that the data occupies.
- *Name* The subject name of the message, or the name of a field within a message.
- *Identifier* An optional integer that uniquely identifies a field within a message.
- *Count* The number of elements in an array datatype.

Between sender and listener, Rendezvous software uses this descriptive information to automatically convert messages between the *local data format* and Rendezvous *wire format*—a universal format that is independent of hardware, operating system, and programming language architectures. The universal wire format provides a common language to connect diverse programs. For details see Rendezvous Datatypes on page 71.

Rendezvous Datatypes

Rendezvous software supports the datatypes listed in Table 14. (The table lists names from the C and C++ language interfaces; other languages uses analogous names, with variations for language syntax.)

Table 14 Wire Format Datatypes (Sheet 1 of 2)

Wire Format Type	Type Description	Notes
Special Types		
`TIBRVMSG_`**`MSG`**	Rendezvous message	Composite of fields
`TIBRVMSG_`**`DATETIME`**	Rendezvous datetime	
`TIBRVMSG_`**`OPAQUE`**	opaque byte sequence	
`TIBRVMSG_`**`STRING`**	character string	`NULL`-terminated.
`TIBRVMSG_`**`XML`**	XML data (byte sequence)	
Scalar Types		

`TIBRVMSG_`**`BOOL`**	boolean	`TIBRV_`**`FALSE`**
		`TIBRV_`**`TRUE`**

`TIBRVMSG_`**`I8`**	8-bit integer	
`TIBRVMSG_`**`I16`**	16-bit integer	
`TIBRVMSG_`**`I32`**	32-bit integer	
`TIBRVMSG_`**`I64`**	64-bit integer	
`TIBRVMSG_`**`U8`**	8-bit unsigned integer	
`TIBRVMSG_`**`U16`**	16-bit unsigned integer	
`TIBRVMSG_`**`U32`**	32-bit unsigned integer	
`TIBRVMSG_`**`U64`**	64-bit unsigned integer	
`TIBRVMSG_`**`F32`**	32-bit floating point	
`TIBRVMSG_`**`F64`**	64-bit floating point	

Table 14 Wire Format Datatypes (Sheet 2 of 2)

Wire Format Type	Type Description	Notes
`TIBRVMSG_`**`IPADDR32`**	4-byte IP address	Network byte order. String representation is four-part dot-delimited notation.
`TIBRVMSG_`**`IPPORT16`**	2-byte IP port	Network byte order. String representation is a 16-bit decimal integer.
Array Types		
`TIBRVMSG_`**`I8ARRAY`**	8-bit integer array	
`TIBRVMSG_`**`I16ARRAY`**	16-bit integer array	
`TIBRVMSG_`**`I32ARRAY`**	32-bit integer array	
`TIBRVMSG_`**`I64ARRAY`**	64-bit integer array	
`TIBRVMSG_`**`U8ARRAY`**	8-bit unsigned integer array	
`TIBRVMSG_`**`U16ARRAY`**	16-bit unsigned integer array	
`TIBRVMSG_`**`U32ARRAY`**	32-bit unsigned integer array	
`TIBRVMSG_`**`U64ARRAY`**	64-bit unsigned integer array	
`TIBRVMSG_`**`F32ARRAY`**	32-bit floating point array	
`TIBRVMSG_`**`F64ARRAY`**	64-bit floating point array	
`TIBRVMSG_`**`MESSAGEARRAY`**	message array	Only C, Java and .NET language interfaces support these types.
`TIBRVMSG_`**`STRINGARRAY`**	string array	

Messages

A Rendezvous self-describing message is a composite containing zero or more fields. All messages have type TIBRVMSG_MSG (including nested submessages).

Each field in turn contains self-describing data. That is, each field consists of several parts—name, identifier (optional), data, type, size and count (number of array elements). Fields can be of any type (see Table 14 on page 71). Wire format permits arbitrarily deep nesting of submessages.

Messages are flexible. Unlike C structs, the actual fields present in a message need not be fixed at compile time.

Consider this schematic illustration of the fields in a message.

Field Name	Id	Type	Size	Count	Data
Name	1	TIBRVMSG_STRING	11	1	Jane Smith
Address	2	TIBRVMSG_STRING	30	1	1234 Home Street Anytown, USA
Phone	3	TIBRVMSG_STRING	13	1	415-123-4567
Age	4	TIBRVMSG_U8	1	1	46
Scores	5	TIBRVMSG_U8ARRAY	1	5	91, 99, 97, 99, 92

Sending programs build self-describing messages by creating an empty message, and then adding fields—one by one. You can add fields in any order.

Receiving programs can access individual fields by name or by identifier, or iterate through all the fields in a single-pass loop.

Message Structure

Rendezvous 5 and earlier releases supported two kinds of messages—unitary and composite. Rendezvous 6 and later releases support only composite messages.

That is, all messages contain data within fields. It is no longer possible to send a message consisting of a pure numeric value; instead use a message with a single field that contains a numeric value.

When a legacy program sends a unitary message to a new program, Rendezvous 6 automatically converts it to a (composite) message, with the unitary value stored in a field named `_data_`.

The field name `_data_` is reserved. More generally, all fields that begin with the underscore character are reserved.

Message Representation in Programs

Rendezvous software uses a proprietary wire format for transmitting data across networks, but programs must represent and manipulate data at either end of the transmission. Programming languages offer different approaches to internal data representation and manipulation. Within programs, Rendezvous software represents data in a way that makes sense for each supported language.

Messages are not Thread-Safe

⚠ Messages are not thread-safe objects; a program that accesses the same message simultaneously in separate threads must protect the message with an explicit locking mechanism. We expect this situation to occur only rarely.

Field Names and Field Identifiers

In Rendezvous 5 and earlier releases, programs would specify fields within a message using a field name. In Rendezvous 6 and later releases, programs can specify fields in two ways:

- A *field name* is a character string. Each field can have at most one name. Several fields can have the same name.
- A *field identifier* is a 16-bit unsigned integer, which must be unique within the message. That is, two fields in the same message cannot have the same identifier. However, a nested submessage is considered a separate identifier space from its enclosing parent message and any sibling submessages.

  (Java and COM store field identifiers as 32-bit signed integers, but the range is still only 16 unsigned bits.)

Message calls specify fields using a field name, or a combination of a field name and a unique field identifier. Field names are appropriate for *most* application programs; most programs do not require field identifiers.

Rules and Restrictions

`NULL` is a legal field name *only* when the identifier is zero. It is *illegal* for a field to have *both* a non-zero identifier *and* a `NULL` field name.

Search Characteristics

In general, an identifier search completes in constant time. In contrast, a name search completes in linear time proportional to the number of fields in the message. Name search is quite fast for messages with 16 fields or fewer; for messages with more than 16 fields, identifier search is faster.

Space Characteristics

The smallest field name is a one-character string, which occupies three bytes in Rendezvous wire format. That one ASCII character yields a name space of 127 possible field names; a larger range requires additional characters.

Field identifiers are 16 bits, which also occupy three bytes in Rendezvous wire format. However, those 16 bits yield a space of 65535 possible field identifiers; that range is fixed, and cannot be extended.

Strings and Character Encodings

Rendezvous software uses strings in several roles:

- String data inside message fields
- Field names
- Subject names (and other *associated* strings that are not strictly *inside* the message)
- Certified delivery (CM) correspondent names
- Group names (fault tolerance)

All these strings (in wire format) use the character encoding appropriate to the ISO locale of the *sender*. For example, the United States is locale `en_US`, and uses the Latin-1 character encoding (also called ISO 8859-1); Japan is locale `ja_JP`, and uses the Shift-JIS character encoding.

When two programs exchange messages within the same locale, strings are always correct. However, when a message sender and receiver use different character encodings, the receiving program must translate between encodings as needed. Rendezvous software does not translate automatically.

DateTime Format

The Rendezvous datatype `TIBRVMSG_DATETIME` indicates a value in DateTime format.

Rendezvous DateTime format encodes the time with microsecond accuracy. The total time range is approximately 17432 years on each side of the epoch (midnight, zero seconds into January 1, 1970).

In wire format, DateTime values are 8 bytes:

- 40 signed bits representing seconds (zero denotes the UNIX epoch, namely, 12:00am, January 1, 1970).
- 24 unsigned bits representing microseconds *after* the seconds value.

Although the program representations of DateTime values can be significantly larger than these 64 bits, the excess is unused at this time.

Chapter 7 Events

Rendezvous software encourages an event-driven programming paradigm. Program *callback functions* respond to asynchronous *events*, such as an inbound message or a timer event.

This chapter presents events and related concepts.

> The terms *event* and *events* are reserved words in some programming languages (for example, Visual Basic). In this book, these terms refer to Rendezvous events, and not to constructs specific to any programming language.

Topics

Event System Overview

The event system consists of several components.

Table 15 Event System Components

Component	Description
Event Object	Represents program interest in a set of events, and the occurrence of a matching event. See Events on page 81.
Event Driver	The event driver recognizes the occurrence of events, and places them in the appropriate event queues for dispatch. Rendezvous software starts the event driver as part of its process initialization sequence (the *open* call). See Event Driver on page 83.
Event Queue	A program creates event queues to hold event objects in order until the program can process them. See Event Queues on page 84.
Event Queue Group	Programs can organize event queues into groups for fine-grained control of dispatch priority. See Queue Groups and Priority on page 85.
Event Dispatch Call	A Rendezvous function call that removes an event from an event queue or queue group, and runs the appropriate callback function to process the event. See Dispatch on page 84.
Callback Function	A program defines callback functions to process events asynchronously. See Callback Functions on page 89.
Dispatcher Thread	Programs usually dedicate one or more threads to the task of dispatching events. Callback functions run in these threads.

Events

As a data object, an event has two roles:

- Programs create event objects to register interest in the set of matching event occurrences. Each event object represents the program's interest in a set of events, and the event's parameters specify that set.
- Rendezvous presents the event object to the appropriate callback function whenever an event occurs. In this context, the event object signifies the actual event that occurred.

Event Parameters

These parameters are common to *all* event creation calls. Additional parameters are specific to each type of event.

Parameter	Description
`queue`	For each occurrence of the event, place the event object on this event queue. See Event Queues on page 84.
`callback`	Upon dispatch, process the event with this callback function. See Callback Functions on page 89.
`closure`	Store this closure data in the event object. See Closure Data on page 89.

Event Classes

Rendezvous software recognizes three classes of events.

Table 16 Event Classes

Message	An inbound message has arrived. Additional creation parameters specify the subject name and transport. See also, Listener Event Semantics on page 91.
Timer	A timer interval has elapsed. An additional creation parameter specifies the interval. See also, Timer Event Semantics on page 96. The Rendezvous .NET API does *not* support this kind of event.
I/O	An I/O socket is ready. Additional creation parameters specify the socket, and the I/O condition. See also, I/O Event Semantics on page 94. The Rendezvous Java, COM and .NET APIs do *not* support this kind of event.

Signals

Obsolete

Earlier releases supported UNIX operating system signals as Rendezvous events. This feature is obsolete starting in release 6.

⚠

Programmers may use `signal()` calls, if the operating system supports them. We recommend caution and extensive testing, since the semantics of `signal()` in a multi-threaded environment can vary.

Event Driver

The *event driver* is the interface and conduit between the Rendezvous API and the operating system. It runs indefinitely, waiting for timer and I/O events from the operating system. It processes those events, matching them with Rendezvous event objects, and placing the appropriate event objects in their event queues.

The Rendezvous *open* call automatically starts the event driver, and the final *close* call terminates it. Your program does not control the event driver.

See Also The Rendezvous Environment, page 51

Event Queues

When an event occurs, the *event driver* places the event object on an *event queue*, where it awaits dispatch to its callback function.

Programs create event queues to influence the dispatch order, and to distribute events among several program threads. For example, a program could assign messages to a set of queues based on subjects. Program threads can dispatch events from separate queues.

Maximum Events and Limit Policy

A creation parameter limits the maximum number of events that a queue can contain. Another creation parameter selects the queue's policy for situations in which a new event would overflow the queue's maximum event limit.

Limit Policy	Description
Discard None	Never discard events; use this policy when a queue has no limit on the number of events it can contain.
Discard First	Discard the first event in the queue (which would otherwise be the next event to dispatch).
Discard Last	Discard the last event in the queue.
Discard New	Discard the new event (which would otherwise cause the queue to overflow its maximum event limit).

Dispatch

Queue dispatch calls remove the event at the head of a queue, and run its callback function.

Three types of dispatch calls behave differently in situations where the queue is empty:

- *Timed dispatch* blocks waiting for an event, but returns without dispatching anything if a waiting time limit is exceeded.
- Ordinary *dispatch* blocks indefinitely, until the queue contains an event.
- *Polling dispatch* does not block; if the queue is empty, it returns immediately.

We discourage programmers from depending on a one-to-one correspondence between dispatch and callback.

Intuitively, one might expect that each successful dispatch call triggers an event callback. However, it is possible for a dispatch call to return successfully without a callback. For example, it might dispatch an event that is internal to Rendezvous software, rather than an event defined in your program.

Dispatcher Threads

Most programs dispatch events in a loop that includes one of the dispatch calls. For convenience, the Rendezvous API includes a call that creates a separate dispatcher thread. The dispatcher thread runs a loop to repeatedly dispatch events. Programs can use this convenience feature, or dispatch events in any other appropriate way. However, every program *must* dispatch its events.

Table 17 Dispatcher Threads

C	`tibrvDispatcher`
C++	`TibrvDispatcher`
Java	`TibrvDispatcher`
.NET	`Dispatcher`

COM Automatic Dispatch

COM programs dispatch events in a different way. A special queue group automatically dispatches whenever an event is ready in any of its queues. Programs can add queues to this queue group to automatically dispatch them. For details, see the COM method `Tibrv.getAutoDispatchQueueGroup()`.

Queue Groups and Priority

Queue groups allow fine-grained control over dispatch order from a single blocking point. A group can contain any number of event queues; the relative priorities of the queues determine the order in which dispatch calls dispatch their events. A queue can belong to any number of groups, or none at all.

Queue group dispatch calls search the group’s queues in order of priority, and dispatch the head event from the first non-empty queue. If two or more queues have identical priorities, subsequent dispatch calls rotate through them in round-robin fashion.

Parallel to queue dispatch calls, two types of queue group dispatch calls behave differently in situations where the queue is empty:

- Ordinary *group dispatch* blocks indefinitely, until any queue in the group contains an event.
- *Timed group dispatch* blocks waiting for an event, but returns without dispatching anything if a waiting time limit is exceeded.
- *Polling group dispatch* does not block; if all the queues in the group are empty, it returns immediately.

Default Queue

The Rendezvous open call automatically creates a default queue. Programs can use this queue for simplicity, or create their own queues, or do both.

The default queue can contain an unlimited number of events (and never discards an event). It has priority `1` (last priority), so in any queue group, it is always among the last queues to dispatch.

When any queue discards an event (which would otherwise have caused the queue to exceed its event limit), Rendezvous software presents a `QUEUE.LIMIT_EXCEEDED` advisory message.

Strategies for Using Queues and Groups

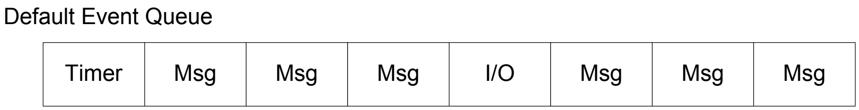
This section presents examples of several common uses of event queues and groups.

Default Queue Only

Use the default queue as the only event queue.

Programs migrating from Rendezvous 5 (and earlier) to release 6 (and later) can use this technique to emulate the event dispatch system of earlier releases.

Figure 7 Default Event Queue

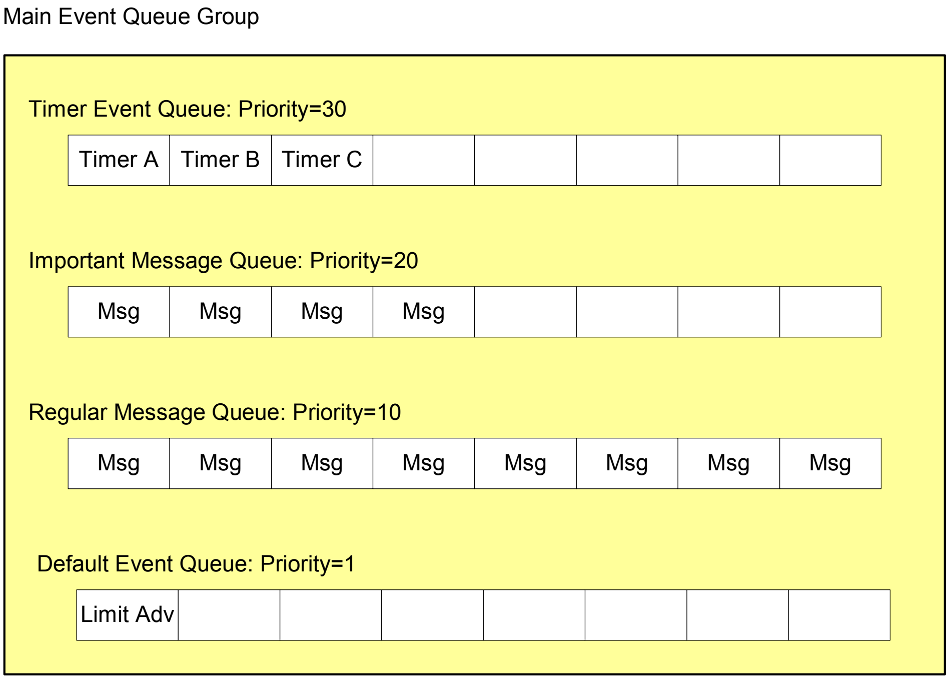


Prioritize Queues within a Group

Dispatch the group, rather than individual queues. Assign queue priorities to reflect the relative priorities of their events (1 indicates the lowest, or last, priority; larger integers indicate higher priority).

This technique ensures that important events dispatch before less important events. For example, you can always dispatch a timer queue before inbound message queues, or give priority to messages with specific subjects.

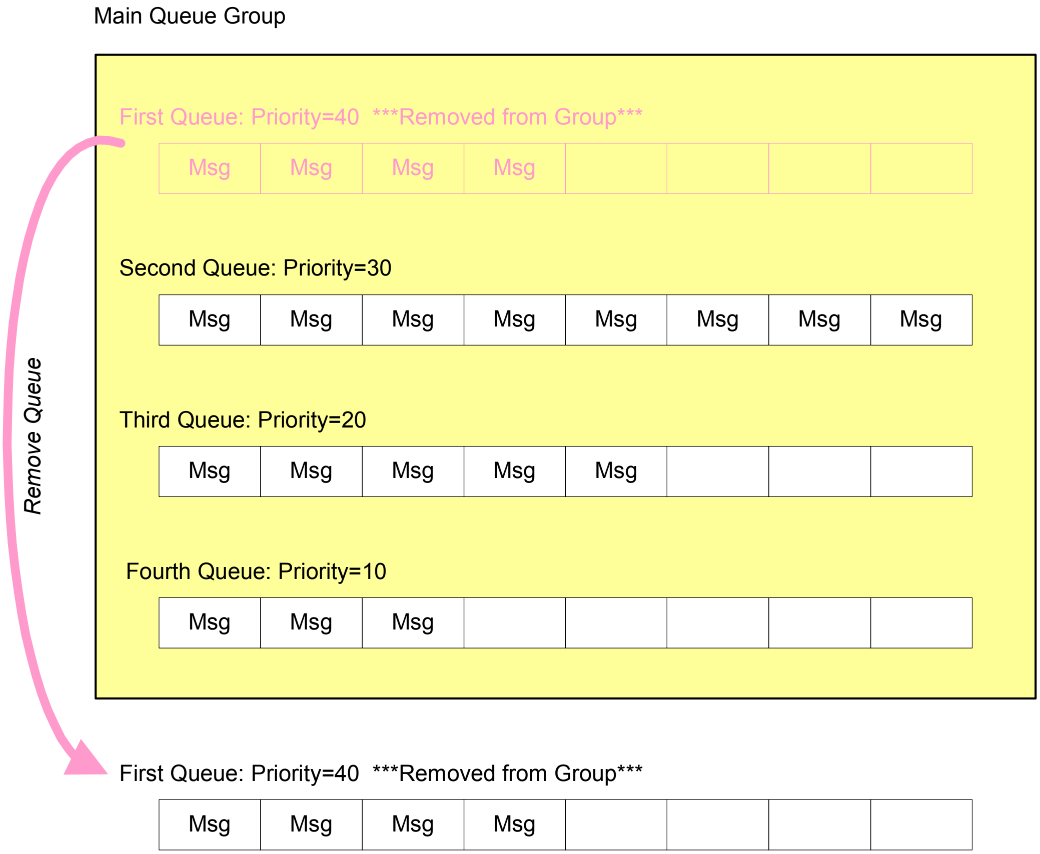
Figure 8 Prioritize Event Queues in a Group


Main Event Queue Group
Timer Event Queue: Priority=30
Timer A
Timer B
Timer C
Important Message Queue: Priority=20
Msg
Msg
Msg
Msg
Regular Message Queue: Priority=10
Msg
Msg
Msg
Msg
Msg
Msg
Msg
Msg
Default Event Queue: Priority=1
Limit Adv

Selectively Suspend Dispatch

Remove selected queues from a group to suspend dispatch of their events. Even when a queue has been removed from its group, events continue to accumulate. Normal dispatch resumes when you add the queue back into the group.

Figure 9 Remove Event Queues from a Group to Suspend Dispatch


Main Queue Group
First Queue: Priority=40 ***Removed from Group***
Msg Msg Msg Msg
Second Queue: Priority=30
Msg Msg Msg Msg Msg Msg Msg Msg
Third Queue: Priority=20
Msg Msg Msg Msg Msg
Fourth Queue: Priority=10
Msg Msg Msg
Remove Queue
First Queue: Priority=40 ***Removed from Group***
Msg Msg Msg Msg

Callback Functions

Callback functions are an essential component of your program's code. They do the actual application-specific work of responding to events—processing inbound messages, timer events and I/O conditions.

For example, inbound messages carry information from other program processes. As each message arrives, the receiving program must take appropriate actions, such as these:

- Display the message to an end-user.
- Store the message in a database.
- Use message elements in calculations.
- Forward the message across a gateway to another network.
- Compose and send a reply message.

In essence, an event object represents a request to run program-specific code whenever a matching event occurs. Event interest persists until the program explicitly cancels its interest by destroying the event object.

While an event object exists, the event that it specifies can occur many times; consequently the event object reappears in its event queue as matching events occur and recur. The callback function runs once for each occurrence of the event (that is, each time the event appears in its queue, and the program dispatches it).

Dispatch Thread

A callback function always runs in the thread that dispatched its event.

For example, the dispatcher convenience feature (see Dispatcher Threads on page 85) runs callback functions outside of program control (usually, in a separate dispatcher thread). In a contrasting example, when a program explicitly calls a dispatch function, the callback function runs in the thread that called the dispatch function.

Closure Data

When creating an event, a program can supply closure data. Rendezvous software neither examines nor modifies the closure. Instead, the event creation call stores the data with the event, and presents it to the event callback function.

A similar mechanism passes closure data to ledger review callback functions.

In C and C++ programs, the closure argument is a pointer (of type `void*`); it can point to any type of data.

In Java programs, the closure argument is a reference to any object.

In COM, the closure argument is a `Variant`.

Return Promptly

Obsolete

In releases 5 and earlier, we warned programmers to ensure that callback functions return promptly and do not block. Callback functions that did not observe these precautions would obstruct important internal mechanisms of Rendezvous software.

Starting in release 6, the event driver is entirely separate from the event dispatch mechanism. Consequently, blocking and long-running callback functions cannot interfere with Rendezvous internal mechanisms.

⚠

However, programmers must recognize that blocking and longing-running callback functions can delay dispatch of other events. For most programs, we still recommend avoiding these behaviors. When callback functions must block, or must run for long periods of time, we recommend dispatching from several threads to ensure prompt processing of all events.

Listener Event Semantics

The arrival of an inbound message is an important event for most Rendezvous programs. It is also an instructive exemplar of an asynchronous event—the receiving program cannot know in advance when a particular message might arrive in the queue. To receive messages, programs create listener events, define callback functions to process the inbound messages, and dispatch events in a loop.

While a program is listening for messages, the event driver queues the listener event each time a message arrives with a matching subject name. Each appearance of the event in the queue leads to a separate invocation of its callback function. At any moment in time, an event queue can contain several references to the same listener event object—each reference paired with a different inbound message.

Each time the callback function runs, it receives an inbound message as an argument. The callback function must process the message in an appropriate application-specific fashion.

Listening for Messages

Programs *listen* for messages by creating *listener events*. Each listener event object signifies that a specific transport is listening for messages that match a specific subject name (which may contain wildcards). The transport continues listening until the program destroys the listener object.

Receiving programs must define callback functions to process inbound messages. When a message arrives, Rendezvous software places the listening event on an event queue. The program dispatches the event to the listener's callback function.

One listener can cause many invocations of its callback function, since many inbound messages can match the desired subject name.

Programs can listen several times on the same subject and transport. Each listener can specify the same callback function or different callback functions. If a program creates two listeners with the same subject and transport, each matching inbound message causes two events (and two callback invocations to process them).

Table 18 Listener Creation (Sheet 1 of 2)

C	`tibrvEvent_CreateListener()`
	`tibrvcmEvent_CreateListener()`

Table 18 Listener Creation (Sheet 2 of 2)

C++	`TibrvListener::create()`
	`TibrvCmListener::create()`

Java	`TibrvListener()`
	`TibrvCmListener()`

COM	`TibrvListener.create` `TibrvCmListener.create`
.NET	`Listener` `Listener.Destroy`

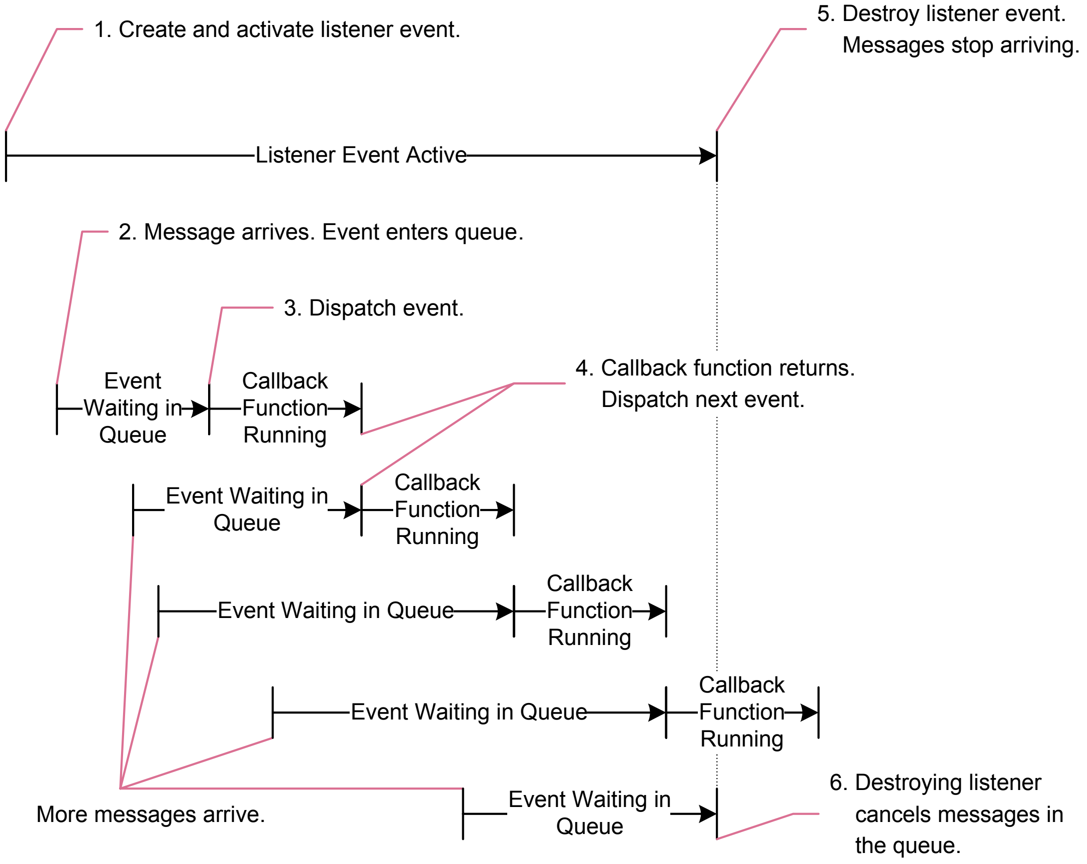
Activation and Dispatch

Inbound messages on the transport that match the subject trigger the event.

Creating a listener event object automatically *activates* the event—that is, the transport begins listening for all inbound messages with matching subjects. When a message arrives, Rendezvous software places the event object and message on the event queue. Dispatch removes the event object from the queue, and runs the callback function to process the message. (To stop receiving inbound messages on the subject, destroy the event object; this action cancels all messages already queued for the listener event.)

Figure 10 on page 93 illustrates that Rendezvous software does *not* deactivate the listener when it places new message events on the queue (in contrast to timer and I/O events, which are temporarily deactivated). Consequently, several messages can accumulate in the queue while the callback function is processing.

Figure 10 Listener Activation and Dispatch


1. Create and activate listener event.
5. Destroy listener event.
Messages stop arriving.
Listener Event Active
2. Message arrives. Event enters queue.
3. Dispatch event.
4. Callback function returns.
Dispatch next event.
Event Waiting in Queue
Callback Function Running
Event Waiting in Queue
Callback Function Running
Event Waiting in Queue
Callback Function Running
Event Waiting in Queue
Callback Function Running
Event Waiting in Queue
More messages arrive.
6. Destroying listener cancels messages in the queue.

When the callback function is I/O-bound, messages can arrive faster than the callback function can process them, and the queue can grow unacceptably long. In applications where a delay in processing messages is unacceptable, consider dispatching from several threads to process messages concurrently.

Destroying a Listener

A listening transport continues receiving inbound messages indefinitely, until the program destroys the listener event (or its supporting objects).

Destroying a listener's transport or event queue invalidates the listener event; inbound messages specified by the event no longer arrive. An invalid listener cannot be repaired; the program must destroy and re-create it.

I/O Event Semantics

Socket I/O can proceed only when certain I/O conditions exist. To receive notification of those conditions, programs can create I/O events, and define callback functions to read or write when sockets are ready.

When a program registers event interest on an I/O socket, the specified condition may occur repeatedly; each time the program can read or write one or more bytes on that socket, the event driver queues the I/O event.

When a program creates an I/O event, the creation call *activates* the event—that is, it requests notification from the operating system when the corresponding I/O situation occurs. When the specified condition occurs, Rendezvous software *deactivates* the event, and places the event object on its event queue. When the callback function returns, Rendezvous software automatically *reactivates* the event. In this way, only one reference to the I/O event can appear in a queue at any moment in time. (References to several I/O events can appear simultaneously, but two references to the same event cannot appear at the same time.) Figure 11 illustrates that Rendezvous software temporarily deactivates the I/O event from the time it enters the queue until its callback function returns.

Figure 11 I/O Event Activation and Dispatch

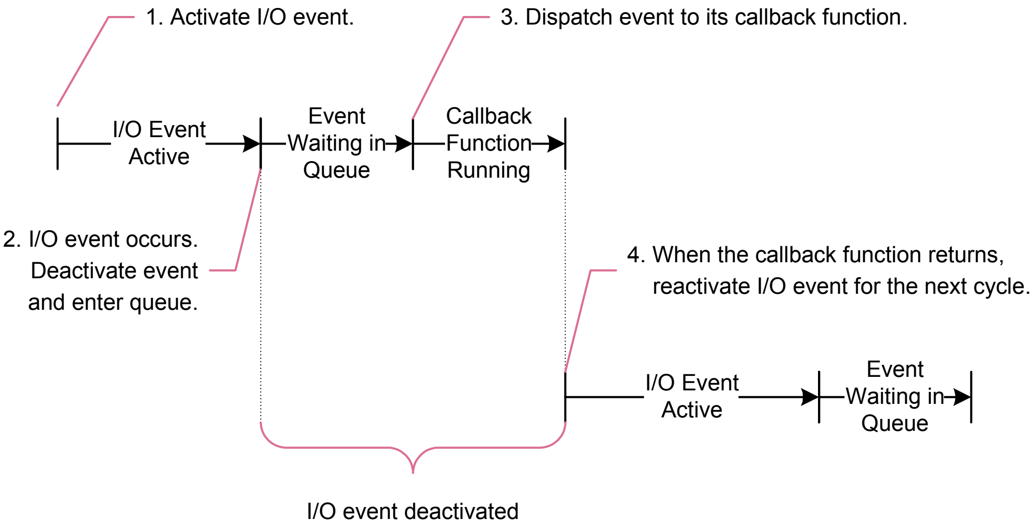

1. Activate I/O event.
3. Dispatch event to its callback function.
I/O Event Active
Event Waiting in Queue
Callback Function Running
2. I/O event occurs. Deactivate event and enter queue.
4. When the callback function returns, reactivate I/O event for the next cycle.
I/O Event Active
Event Waiting in Queue
I/O event deactivated

Table 19 I/O Event Creation

C	`tibrvEvent_CreateIO()`
C++	`TibrvIOEvent::create()`
Java	I/O events are not available with Java.
COM	I/O events are not available with COM.
.NET	I/O events are not available with .NET.

Operating System I/O Semantics

The semantics of all I/O conditions depend on the I/O semantics of the underlying operating system and its event manager. Rendezvous software does not change those semantics.

I/O events trigger when the operating system *reports* that an I/O condition on a monitored socket would succeed (transfer at least one byte without blocking).

For example, write events depend on the *state* of the socket—it must be write-available. That is, on each pass through the event driver's loop, the socket can accept one or more bytes.

Blocking I/O Calls

Rendezvous software *cannot guarantee* that a subsequent I/O call will not block.

I/O conditions are not necessarily static. *Event stealing* or *write overload* can change the I/O condition before the callback function runs.

As an example of event stealing, consider an I/O event indicating that a particular socket has data available for reading. Another thread or process could read the data from that socket before the I/O event callback function can read the socket. In that case, a read call could indeed block.

Availability

Socket I/O is available in the C and C++ interfaces, but not in the Java and COM interfaces.

Timer Event Semantics

Timer events are perhaps the most well-known type of asynchronous event. To do an operation after a specific time interval has elapsed, a program creates a timer event, and defines a callback function to do the desired operation.

When a program creates a timer event object, the creation call *activates* the timer event—that is, it requests notification from the operating system when the timer's interval elapses. When the interval elapses, Rendezvous software places the event object on its event queue. Dispatch removes the event object from the queue, and runs the callback function to process the timer event. On dispatch Rendezvous software also determines whether the next interval has already elapsed, and requeues the timer event if appropriate.

Notice that time waiting in the event queue until dispatch can increase the effective interval of the timer. It is the programmer's responsibility to ensure timely dispatch of events.

Figure 12 on page 97 illustrates a sequence of timer intervals. The number of elapsed timer intervals directly determines the number of event callbacks.

At any moment the timer object appears on the event queue at most once—not several times as multiple copies. Nonetheless, Rendezvous software arranges for the appropriate number of timer event callbacks based the number of intervals that have elapsed since the timer became active or reset its interval.

Destroying or invalidating the timer object *immediately* halts the sequence of timer events. The timer object ceases to queue new events, and an event already in the queue does not result in a callback. (However, callback functions that are already running in other threads continue to completion.)

A timer repeats indefinitely—until the program destroys the timer event object.

Resetting the timer interval *immediately* interrupts the sequence of timer events and begins a new sequence, counting the new interval from that moment. The reset operation is equivalent to destroying the timer and creating a new object in its place.

Figure 12 Timer Activation and Dispatch

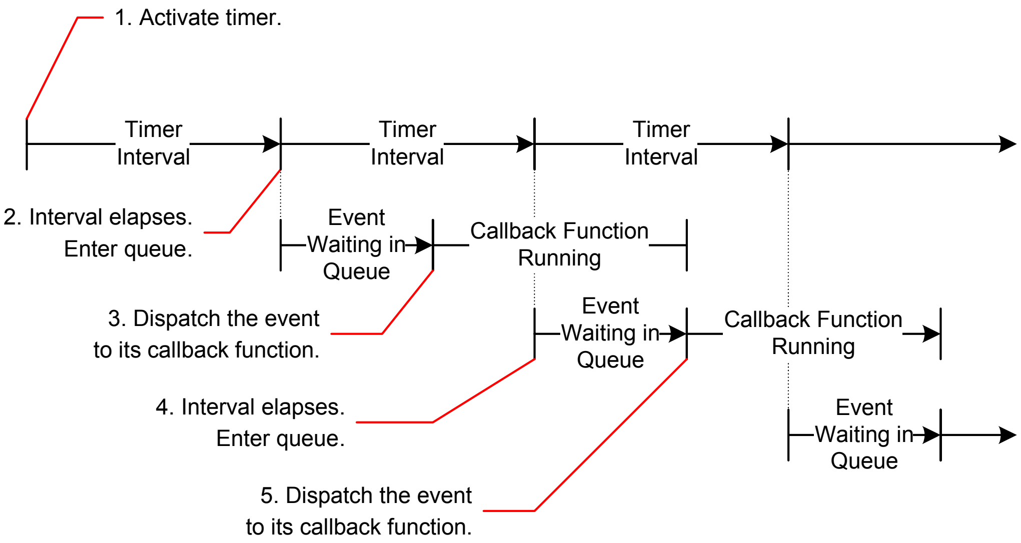

1. Activate timer.
Timer Interval
Timer Interval
Timer Interval
2. Interval elapses. Enter queue.
Event Waiting in Queue
Callback Function Running
3. Dispatch the event to its callback function.
Event Waiting in Queue
Callback Function Running
4. Interval elapses. Enter queue.
Event Waiting in Queue
5. Dispatch the event to its callback function.

Table 20 Timer Creation

C	`tibrvEvent_CreateTimer()`
C++	`TibrvTimer::create()`
Java	`TibrvTimer()`
COM	`TibrvTimer.create`
.NET	Timer events are not available with .NET.

Chapter 8 Transport

The software mechanism for sending and delivering messages is called a *transport*. A transport defines the delivery scope—that is, the set of *possible* destinations for the messages it sends. (In contrast, listeners filter messages by subject name, collectively defining the *actual* destination set.)

Topics

Transport Overview

Programs use transport objects to send messages and listen for messages. A transport determines three aspects of message delivery:

- Delivery scope—the potential range of its messages
- Delivery mechanism—the path (including software, hardware and network aspects) that its messages travel
- Delivery protocol—the ways in which programs cooperate and share information concerning message delivery

Various types of transport object combine these aspects to yield different qualities of service—for example, intra-process delivery, network delivery, reliable delivery, virtual circuit, certified delivery, and distributed queue delivery.

Transport Scope

Rendezvous software distinguishes between two broad classes of transports, each with a different potential scope and a different delivery mechanism.

- A *network transport* delivers messages across a network, to processes on one or more hosts. It also can deliver messages to several processes on a single host.

  All messages pass through a Rendezvous daemon process.

  The service, network and daemon parameters of a network transport specify its scope within the network (see Network Transport Parameters on page 102).

  Java programs can also use `rva` transports.

- An *intra-process transport* delivers messages between program threads within a single process.

  Intra-process messages do not pass through a Rendezvous daemon process; instead, they remain local within the program process. Intra-process messages are significantly faster than network messages.

  Each program contains exactly one intra-process transport; the Rendezvous open call creates it automatically. Its scope is the program process. Any thread can dispatch intra-process events from an appropriate queue.

  Programs can use intra-process messages to implement user events. For more information, see Intra-Process Transport and User Events on page 114.

Network Transport Parameters

Transport creation calls accept three parameters that govern the behavior of the transport: `service`, `network` and `daemon`. In simple networking environments, the default values of these parameters are sufficient.

However, some environments require special treatment, for any of these reasons:

- Several independent distributed applications run on the same network, and you must isolate them from one another (`service` parameter).
- Programs use the Rendezvous routing daemon, `rvrd`, to cooperate across a WAN with programs that belong to a particular service group, and the local programs must join the same service group (`service` parameter).
- A program runs on a computer with more than one network interface, and you must select a specific network for outbound multicast Rendezvous communications (`network` parameter).
- Computers on the network use multicast addressing to achieve even higher efficiency, and you must specify which multicast groups to join (`network` parameter).
- A program runs on one computer, but connects with a Rendezvous daemon process running on a different computer, and you must specify the remote daemon to support network communications (`daemon` parameter).
- Two programs use direct communication. Both programs must enable this feature and specify its service (`service` parameter).

If none of these conditions apply, then you can use default values for the transport parameters. If any of these conditions do apply, then choose appropriate parameter values.

Service Parameter

Rendezvous daemon (`rvd`) processes communicate using UDP or PGM services. The `service` parameter instructs the Rendezvous daemon to use this service whenever it conveys messages on this transport.

As a direct result, services divide the network into logical partitions. Each transport communicates on a single service; a transport can communicate only with other transports on the same service. To communicate on more than one service, a program must create a separate transport object for each service.

Service Groups

A *service group* is a group of Rendezvous transport objects that communicate using the same UDP or PGM service. Rendezvous daemon processes connect transports within a service group on the same network, so they can share messages with one another.

Interaction between Service and Network Parameters

Within each `rvd` process, all the transports that specify a given service must specify the same network parameter. That is, if the `service` parameters resolve to the same UDP or PGM port, then the `network` parameters must also be identical. (This restriction extends also to routing daemons.)

For example, suppose that the program `foo`, on the computer named `orange`, has a transport that communicates on the service `svc1` over the network `lan1`. It is illegal for *any* program to subsequently create a transport (connecting to the same daemon process on `orange`) to communicate on `svc1` over any other network—such as `lan2`. Once `rvd` binds `svc1` to `lan1`, that service cannot send outbound broadcast messages to any other network. Attempting to illegally rebind a service to a new network fails; transport creation calls produce an error status (`TIBRV_INIT_FAILURE`).

To work around this limitation, use a separate service for each network.

The limitation is not as severe as it might seem at first, because it only affects outbound broadcast messages:

- Point-to-point messages on the transport’s service travel on the appropriate network (as determined by the operating system) irrespective of the transport’s network parameter.
- Inbound broadcast messages on the transport’s service can arrive from any network, irrespective of the transport’s network parameter.

Specifying the Service

Rendezvous programs specify services in one of three ways, shown below in order of preference:

- Service name
- Port number
- Default

By Number

When you specify a port number, it must be a string representing a decimal integer (for example, `"7890"`).

By Name

When you specify a service name, the transport creation function calls `getservbyname()`, which searches a network database (such as NIS) or a flat file (such as `services` in the system directory).

Defaults

If you specify null, the transport creation function searches for the service name `rendezvous`.

If `getservbyname()` does not find `rendezvous`, the Rendezvous daemon instructs the transport creation function to use a hard default:

- The TRDP daemon offers the default service `7500`.
- The PGM daemon offers the default service `7550`.

We strongly recommend that administrators define `rendezvous` as a service, especially if either of the ports `7500` or `7550` is already in use.

For example, network administrators might add the following service entry to the network database (where `7500` is the port number):

```
rendezvous 7500/udp
```

Once this entry is in the network database, programmers can conveniently specify `NULL` or the empty string as the `service` argument to create a transport that uses the default Rendezvous service.

PGM and TRDP

The TRDP and PGM variants of `rvd` interpret this service specification differently:

- The TRDP variant interprets it as a UDP service.
- The PGM variant interprets it as a *pair* of services with the same port number—a PGM service for multicast communication, and a UDP service for point-to-point communication. Even though these twin services share the same port number, data does not cross from one to the other.

Specifying Direct Communication

To enable *direct communication*, specify a two-part service parameter, separating the parts with a colon. You may specify either part by service name, by port number, or by default. For example:

```
"7706:7707"
"rendezvous:5238"
":"
":0"
```

- The first part specifies the service that `rvd` uses for regular communication.

  The TRDP variant interprets this part as a UDP service.

  The PGM variant interprets this part as a *pair* of services with the same port number—a PGM service for multicast communication, and a UDP service for *ineligible* point-to-point communication.

- The second part specifies the UDP service that the Rendezvous transport object uses for direct communication. This part remains within the program; the transport object never passes it to `rvd`.

  Both variants interpret this part in the same way—as a UDP service for eligible point-to-point communication (RPTP).

Defaults

To use the `rendezvous` service for regular communication (or if `rendezvous` is not defined, the default regular service, 7500), omit the first part of the parameter.

To use an ephemeral service for direct communication, either omit or specify zero for the second part (but include the separating colon). The operating system assigns an available service number.

To disable direct communication, specify a one-part parameter, omitting the separating colon and the second part.

Restriction

On each host computer, programs can *bind* a UDP or PGM service at most once. Consider these consequences:

- On host computer `foo`, no two transport objects (whether in the same process or different processes) can bind the same UDP or PGM service for direct communication.
- If a transport object on host computer `foo` specifies a UDP or PGM service for regular communication, no other transport object on `foo` can bind that service for direct communication.
- The opposite is also prohibited. If a transport object on host computer `foo` binds a UDP service for direct communication, no other transport object on `foo` can bind that service for regular communication.

- The two parts of any service parameter must specify two *different* port numbers.
- However, any number of transport objects on host computer `foo`—in any number of processes—can specify the same UDP or PGM service for regular communication. Those transports communicate through `rvd`, which *binds* the service only once—in accord with the restriction.

See Also Direct Communication on page 116

Network Parameter

Every network transport object communicates with other transport objects over a network. On computers with only one network interface, the Rendezvous daemon communicates on that network without further instruction from the program.

On computers with more than one network interface, the `network` parameter instructs the Rendezvous daemon to use a particular network for all communications involving this transport. To communicate over more than one network, a program must create a separate transport object for each network.

The network parameter also specifies multicast addressing details (for a brief introduction, see Multicast Addressing on page 109).

To connect to a remote daemon, the `network` parameter must refer to the network from the perspective of the remote computer that hosts the daemon process.

Constructing the Network Parameter

The network parameter consists of up to three parts, separated by semicolons—network, multicast groups, send address—as in these examples:

`lan0`	network only
`lan0;225.1.1.1`	one multicast group
`lan0;225.1.1.1,225.1.1.5;225.1.1.6`	two multicast groups, send address
`lan0;;225.1.1.6`	no multicast group, send address

Part One—Network

Part one identifies the network, which you can specify in several ways:

(Sheet 1 of 2)

Host name	When a program specifies a host name, the transport creation function calls `gethostbyname()`, which searches a network database to obtain the IP address.
Host IP address	When a program specifies an IP address, it must be a string representing a multi-part address. For example: `"101.120.115.111"`

(Sheet 2 of 2)

Network name (where supported)	When an application specifies a network name, the transport creation function calls `getnetbyname()`, which searches a network database such as Network Information Services (NIS) or a flat file (such as `networks`) in the system directory.
Network IP number	If a program specifies a host IP address or a network IP number it must be in dotted-decimal notation. For example, `101.55.31`.
Interface name (where supported)	When an application specifies an interface name, the transport creation function searches the interface table for the specified interface name. For example, `lan0`. The interface name must be one that is known to `ifconfig` or `netstat`.
Default	If a program does not specify a network, the transport creation function uses the default network interface: • TRDP daemons use the network interface which corresponds to the host name of the system as determined by the C function `gethostname()`. • PGM daemons use the default PGM multicast interface, `224.0.1.78`.

The use of the UDP broadcast protocol has generally been superseded by IP multicast protocol. To use broadcast protocols without multicast addressing, specify only part one of the network parameter, and omit the remaining parts.

Part Two—Multicast Groups

Part two is a list of zero or more multicast groups to join, specified as IP addresses, separated by commas. Each address in part two must denote a valid multicast address. Joining a multicast group enables listeners on the resulting transport to receive data sent to that multicast group.

For a brief introduction to multicasting, see Multicast Addressing on page 109.

Part Three—Send Address

Part three is a single send address. When a program sends multicast data on the resulting transport, it is sent to this address. (Point-to-point data is not affected.) If present, this item must be an IP address—not a host name or network name. The send address *need not* be among the list of multicast groups joined in part two.

If you join one or more multicast groups in part two, but do not specify a send address in part three, the send address defaults to the first multicast group listed in part two.

Multicast Addressing

Multicast addressing is a focused broadcast capability implemented at the hardware and operating system level. In the same way that the Rendezvous daemon filters out unwanted messages based on service groups, multicast hardware and operating system features filter out unwanted messages based on multicast addresses.

When no broadcast messages are present on the service, multicast filtering (implemented in network interface hardware) can be more efficient than service group filtering (implemented in software). However, transports that specify multicast addressing still receive broadcast messages, so combining broadcast and multicast traffic on the same service can defeat the efficiency gain of multicast addressing.

Rendezvous software supports multicast addressing only when the operating system supports it. If the operating system does not support it, and you specify a multicast address in the network argument, then transport creation calls produce an error status (`TIBRV_NETWORK_NOT_FOUND`).

Daemon Parameter

The `daemon` parameter instructs the transport creation function about how and where to find the Rendezvous daemon and establish communication.

Each Rendezvous transport establishes a communication conduit with the Rendezvous daemon, as the following steps describe:

1. The daemon process opens a (TCP) *client socket*, and waits for a client to request a connection.

   The `-listen` option of the Rendezvous daemon (`rvd`) specifies the socket where the Rendezvous daemon should listen for new client program connections.

2. The program calls the transport creation function, which contacts the daemon at the client socket specified in its `daemon` parameter.

   The `daemon` parameter of the transport creation function must correspond to the `-listen` option of daemon process; that is, they must specify the same communication type and socket number.

   If no daemon process is listening on the specified client socket, then the transport creation call automatically starts a new daemon process (which listens on the specified client socket), and then attempts to connect to it.

3. The daemon process opens a conduit for private communication with the new transport object in the program. All future communication uses that conduit.

   The request socket is now free for additional requests from other client transports.

Specifying a Local Daemon

Specify the daemon's client socket as a character string with components separated by colons.

For *local* daemons, specify the transport creation function's `daemon` parameter and the `-listen` option to the daemon process as a (TCP) socket number; for example: `"6555"`

To use the default client socket, supply `NULL` as the `daemon` argument to the transport creation function, and omit the `-listen` option to the daemon process.

Remote Daemon

In most cases, programs connect to a local daemon, running on the same host as the program. Certain situations require a remote daemon, for example:

- The program runs on a laptop computer that is not directly connected to the network. Instead, the laptop connects to a workstation on the network, and the daemon runs on that workstation.
- The program connects to a network at a remote site.

For *remote* daemons, specify two parts (introducing the remote host name as the first part):

- Remote host name
- Port number

For example:

```
    "purple_host:6555"
```

Direct communication is not available when connecting to a remote daemon (see Direct Communication on page 116).

Suppress Daemon Auto-Start

The policy that a transport cannot automatically start a remote daemon also results in a convenient way to suppress the auto-start feature of a local daemon. To do so, specify the local daemon with a two-part parameter, as if it were a remote daemon. For the first part (the host) supply either the local computer's loopback address, `127.0.0.1`, or the local host name (if the host does not support a loopback address). For example:

```
    127.0.0.1:7500
    my_host_name:7500
```

Secure Daemon

To connect to a *secure* daemon, specify three parts:

`ssl:`*host*`:`*port_number*

For example:

```
    ssl:myhost.net:8344
    ssl:102.24.12.3:8344
```

Colon characters (`:`) separate the three parts.

`ssl` indicates the protocol to use when attempting to connect to the daemon.

host indicates the host computer of the secure daemon. You can specify this host either as a network IP address, or a hostname. Omitting this part specifies the local host.

port_number specifies the port number where the secure daemon listens for SSL connections. This part is required; you may not omit it.

Two Identical Arguments

Programs that connect to a secure daemon must specify an identical three-part string to two API calls:

- The `daemonName` parameter of the call that registers the secure daemon's certificate—see Secure Daemon on page 60
- The `daemon` parameter of the call that creates the transport object that connects to the secure daemon

Sending Messages

Programs can *send* messages at any time to any other Rendezvous programs. All send operations are methods of transport objects:

- Send an outbound message.
- Send an outbound message in reply to an inbound message. This variation automatically extracts the reply subject from the inbound message.
- Send an outbound message and wait for an inbound reply. This variation blocks until a reply arrives.

Table 21 Sending Calls

C	`tibrvTransport_Send()`
	`tibrvTransport_SendReply()`
	`tibrvTransport_SendRequest()`

C++	`TibrvTransport::send()`
	`TibrvTransport::sendReply()`
	`TibrvTransport::sendRequest()`

Java	`TibrvTransport.send()`
	`TibrvTransport.sendReply()`
	`TibrvTransport.sendRequest()`

COM	`TibrvTransport.send`
	`TibrvTransport.sendReply`
	`TibrvTransport.sendRequest()`

.NET	`Transport.Send`
	`Transport.SendReply`
	`Transport.SendRequest`

Intra-Process Transport and User Events

In addition to the three types of Rendezvous events, programs can implement user events using the intra-process transport.

Consider these properties of a message that serves as a user event, sent on an intra-process transport:

- Process-local. The user event message remains within the process.
- Thread-to-thread. A user event message can be sent to a specific thread (or set of threads), because the message event appears in a specific event queue. (That is, only a thread that dispatches that queue can receive the message event.)

Programs can exploit these properties to build a thread-to-thread intra-process communication mechanism. Consider an example program that contains one thread that controls a fax hardware device. The other threads send messages to the fax thread, which processes them by sending faxes.

See Also Intra-Process Transport on page 192 in *TIBCO Rendezvous C Reference*
Intra-Process Transport on page 176 in *TIBCO Rendezvous Java Reference*

Inbox Names

Transport objects can create inbox names, designating a destination that is unique to that transport object and its process. Rendezvous software uses point-to-point techniques to deliver messages with inbox subject names.

One common use of inbox names is as reply subject names in request/reply interactions (see Request/Reply Interactions on page 49).

Request reply interactions that cross network boundaries depend on Rendezvous routing daemons (`rvrd`) in both directions. Routing daemons annotate inbox name reply subjects during the request phase, and interpret those annotations during the reply phase—ensuring correct delivery across network boundaries. (Programs that send reply inbox names within message data fields circumvent this mechanism, and cannot receive replies across network boundaries.)

Table 22 Inbox Calls

C	`tibrvTransport_CreateInbox()`
C++	`TibrvTransport::createInbox()`
Java	`TibrvTransport.createInbox()`
COM	`TibrvTransport.createInbox()`
.NET	`Transport.CreateInbox`

See Also

Subject-Based Addressing and Message Destinations on page 45.
Multicast and Point-to-Point Messages on page 46.

Direct Communication

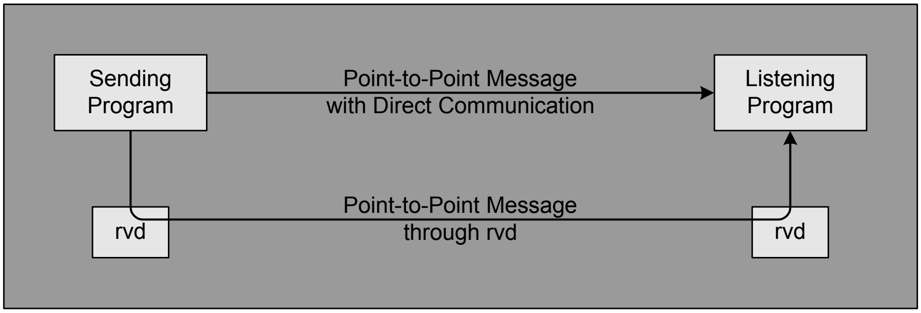
Release 7 introduces direct communication capabilities between two network transport objects.

Overview

With *direct communication*, two application programs can conduct eligible point-to-point communications without intermediary Rendezvous daemon (`rvd`) processes. This arrangement can decrease message latency and context switching for point-to-point messages.

Figure 13 contrasts the route of a point-to-point message with direct communication against the same message with regular communication (through `rvd`). In the path through `rvd`, each of the two daemons could add a small delay. The direct path avoids these sources of potential delay.

Figure 13 Direct Communication between Two Programs


Sending Program
Point-to-Point Message with Direct Communication
Listening Program
rvd
Point-to-Point Message through rvd
rvd

Direct communication uses RPTP over a UDP channel.

Usage

To enable direct communication, specify a two-part `service` parameter when creating the transport object:

- The first part controls regular communication—including messages to public subjects and ineligible point-to-point messages.
- The second part controls direct communication—messages to inbox subjects at eligible destination transports.

Eligibility

All *eligible* messages automatically use direct communication, traveling directly between the two programs. All *ineligible* messages flow through `rvd`.

A *message* is eligible for direct communication if it meets *all* of these conditions:

- The message has an *inbox* destination subject.
- Its sending transport object is eligible and enabled.

- Its receiving transport object is eligible and enabled.
- The network path between the sender and the receiver does not cross through Rendezvous routing daemon (`rvrd`).

A *transport* object is eligible for direct communication if it meets *all* of these conditions:

- The transport is *enabled* for direct communication (that is, it has a two-part `service` parameter).

  Note that a program can enable a transport only if program links release 7 (or later) of the Rendezvous API library.
- The transport connects to a *local* daemon.

Restrictions

Both the sending and receiving transport objects must enable direct communication. If only one of the two transports enables direct communication, then point-to-point messages between them flow through `rvd`.

Direct communication is *not* available for transport objects that connect to remote daemons.

Direct communication is *not* available for `rva` transport objects in Java programs.

When the path between two transports crosses a routing daemon (`rvrd`), direct communication is *not* available between those transports. Even if both transports enable direct communication, point-to-point messages still flow through `rvd` and `rvrd`.

Direct communication applies only to point-to-point messages (that is, messages with inbox destinations) between two enabled transports.

Nonetheless, messages on a virtual circuit *always* travel point-to-point—even messages with public subject names. The virtual circuit terminals wrap all messages within internal point-to-point messages. So a virtual circuit that employs enabled transports at both terminals always reaps the benefits of direct communication.

Cost

Each enabled transport consumes a UDP port.

See Also

Specifying Direct Communication on page 105
Remote Daemon on page 111

Batch Modes for Transports

Release 7 introduces a timer batch mode for network transport objects. In a narrowly limited set of situations, this feature can improve application performance. Most customers can ignore this feature.

Overview

With default batch behavior, a transport object transmits outbound messages to the Rendezvous communications daemon as soon as possible.

With timer batch behavior, the transport can delay transmitting small outbound messages to the daemon. For programs that send many small messages, this behavior can improve efficiency. The cost of that improvement is data latency.

Advantages

When used appropriately, it is possible that timer batching can promote one or both of these advantages:

- In some situations, it can reduce context switching on the sending computer, improving CPU utilization.
- In some situations, it can reduce the number of packets on the network, improving bandwidth utilization.

Indications

These conditions characterize situations where timer batch mode might be advantageous:

- Maximum data latency of approximately 25 milliseconds is acceptable.
- The program sends very small messages—generally 100 bytes or fewer.
- A correct program performs poorly with default batch behavior.

Contraindications

These conditions characterize situations in which we do *not* recommend timer batch mode:

- Data latency is *not* acceptable.
- Timer batch behavior does not produce measurable improvements in the performance of your application.

Usage

Programs control batch mode by setting a property of the transport object.

Chapter 9 Virtual Circuits

Virtual circuits feature Rendezvous communication between two terminals over an exclusive, continuous, monitored connection.

Topics

Virtual Circuits Overview

Conceptual Definition

In his classic textbook, *Computer Networks*, Andrew Tanenbaum describes virtual circuits by analogy with a public telephone network:

> The telephone customer must first set up the virtual circuit (dial the call), then transmit the data (talk), and finally close down the circuit (hang up). Although what happens inside the telephone system or subnet is undoubtedly very complicated, the two users are provided with the illusion of a dedicated point-to-point channel between themselves. In particular, information is delivered to the receiver in the same order in which it is transmitted by the sender.

Quality of Service

Rendezvous virtual circuits provide a similar quality of service. A virtual circuit is an exclusive, monitored connection between two *terminals*—each of which is a virtual circuit transport object.

- The two terminals communicate *exclusively* with each other. They do not communicate with any other transport object.
- Each terminal can send messages to the other terminal. Messages arrive in the same order as the opposite terminal sent them.
- Each terminal *monitors* the connection to ascertain continuous correct operation.
- A failure anywhere along the circuit causes the entire circuit to cease functioning.
  - Each terminal presents a `VC.DISCONNECTED` advisory message.
  - The terminals can no longer deliver inbound messages to listener objects.
  - Attempting to send outbound messages produces error status.
  - The terminals *cannot* reconnect. Programs must destroy them, and all listener objects that use them. To establish a new virtual circuit, programs may create new terminal objects.

Scope

The scope of a virtual circuit transport is limited to exactly one other transport—the terminal at the opposite end of the connection. No other transports receive messages sent on the virtual circuit—not even transports that communicate on the same network and UDP or PGM service as the virtual circuit terminals. Conversely, terminals of a virtual circuit do not receive messages sent by any other transports.

A program can create any number of listener objects that use a virtual circuit transport object. They can listen on inbox names or on public subject names. In either case, they can receive only those messages sent by the opposite terminal.

Similarly, a program can specify the destination of an outbound message using either an inbox name or a public subject name. In either case, the message travels point-to-point to the opposite terminal.

Mechanism

Every virtual circuit terminal employs an ordinary transport object as an underlying communication mechanism. The transport can be an (`rvd`) network transport or the intra-process transport. (Several virtual circuits can employ the same transport. The transport can carry other messages as well.)

The transport carries both data and protocol communications for the virtual circuit. All such communication travels point-to-point between the two terminals. The terminals multiplex inbound messages to appropriate listener objects.

Protocol

A hidden protocol establishes and monitors the connection between terminals.

Terminals present advisory messages to programs to report changes in the connection's status.

Direct Communication

Because virtual circuits rely on point-to-point communication between the two terminals, they can use direct communication to good advantage. To do so, both terminals must employ network transports that enable direct communication. For an overview, see Direct Communication on page 116.

See Also

Andrew S. Tanenbaum, *Computer Networks*, 1981, Prentice-Hall, Englewood Cliffs, New Jersey.

Properties of Virtual Circuits

Properties

Virtual circuits guarantee these properties:

- All inbound messages come only from the opposite terminal. Senders outside of a virtual circuit cannot insert data into its message stream.
- All outbound messages go only to the opposite terminal. Listeners outside of a virtual circuit cannot receive its message stream.
- Terminals receive explicit feedback if the connection is interrupted. Programs can promptly detect failure of mission-critical components or communications, and act to restore or replace them.

Application Ideas

The first and second properties suggest virtual circuits for applications that must insulate its message streams against subject name interference.

The third property suggests virtual circuits for applications that require continuous communication, such as monitoring and control of manufacturing processes, no-loss data transmission, or data replication.

Migration Path

Rendezvous implements virtual circuit terminals as transport objects. When appropriate, you can easily convert most existing programs to use virtual circuits (instead of ordinary network transports) with only minor code changes.

Programming Paradigm

These steps illustrate the programming paradigm for Rendezvous virtual circuits.

1. Program code in process A sends a series of application-level messages to process B requesting a virtual circuit.
2. Program code in process B responds to one of those requests by creating a virtual circuit transport *accept* object. Creating the accept object also produces a *connect subject*—an inbox where the accept object listens for protocol messages.
3. Program code in B sends an application-level reply message to A, inviting A to connect to the accept object. The reply subject of the invitation message is the connect subject of the accept object.
4. Program code in process A creates a virtual circuit transport *connect* object—supplying the connect subject from the invitation as an argument to the create call.
5. The connect object in A automatically initiates a protocol to establish the virtual circuit connection between the two terminal objects.
6. When the connection is *complete* (that is, ready to use) both terminals present `VC.CONNECTED` advisories. From this time forward, either process can send messages on the virtual circuit.

Connect Subject

B must send the connect subject as the *reply subject* of the invitation message—not as an application-level field of the message. This detail is especially important when routing hardware or software intervenes between the two processes.

Request and Invitation

Notice that programs must send the request and invitation messages on transports that are already operational. They cannot send them on the virtual circuit transport, because the circuit is not yet complete.

Listening and Sending

Programs may create listener objects on virtual circuit transports at any time after the create call returns.

Attempting to send messages before the connection is complete produces error status.

Testing the New Connection

Code in both processes (A and B) must test the new connection before using it. We recommend a two-part test. Immediately after creating a virtual circuit terminal object, do these two steps, in this order:

1. Listen for the `VC.CONNECTED` advisory on the terminal transport object. If the terminal presents this advisory, it is ready to use.

   It is possible to miss this advisory. That is, the terminal might present it before the program creates the listener to intercept it. In this situation, the program could wait indefinitely for the advisory, which has already come and gone. To avoid this situation, do the following step as well.

2. Poll the connection to test whether it is operational, supplying zero as the timeout parameter to the wait-for-connection call. If this call returns without error, the terminal is ready to use.

Virtual Circuit API

The virtual circuit API consists of three calls (see Table 23) and two advisory messages.

Create Terminals

Two calls create the terminal objects—one call for the terminal that *accepts* connections, and one for the terminal that subsequently *connects* to it.

The two types of terminal play complementary protocol roles as they attempt to establish a connection. However, this difference soon evaporates. After the connection is complete, the two terminals behave identically.

Testing the Connection

A third call tests the current status of the connection. Programmers can arrange for this call to return immediately, or block until the connection is complete (wait for connection).

Other Transport Calls

Programs send messages, create inbox names, and create listeners using the same calls as for ordinary transports.

Table 23 Virtual Circuit Calls (Sheet 1 of 2)

C	`tibrvTransport_CreateAcceptVc()`
	`tibrvTransport_CreateConnectVc()`
	`tibrvTransport_WaitForVcConnection()`

C++	`TibrvVcTransport`
	`TibrvVcTransport::createAcceptVc()`
	`TibrvVcTransport::createConnectVc()`
	`TibrvVcTransport::waitForVcConnection()`

Java	`TibrvVcTransport`
	`TibrvVcTransport.createAcceptVc()`
	`TibrvVcTransport.createConnectVc()`
	`TibrvVcTransport.waitForVcConnection()`

Table 23 Virtual Circuit Calls (Sheet 2 of 2)

COM	`TibrvTransport.createAcceptVc()`
	`TibrvTransport.createConnectVc()`
	`TibrvTransport.waitForVcConnection`

.NET	`VCTransport.CreateAcceptVC`
	`VCTransport.CreateConnectVC`
	`VCTransport.WaitForVCConnection`

Advisories

Advisory messages report connection status changes asynchronously; see `VC.CONNECTED` on page 267, and `VC.DISCONNECTED` on page 268.

Chapter 10 Guidelines for Programming

This chapter describes techniques and guidelines to help you create programs that use Rendezvous software to the fullest advantage.

Topics

Avoid Sending Binary Data Buffers or Internal Structs

Rendezvous programs can exchange binary data buffers using datatype `TIBRVMSG_OPAQUE`. The program is free to use any format and content within opaque data. However, we recommend against extensive use of opaque data.

For example, opaque buffers can contain data structures mapped by C language structs—but *beware,* this technique couples your programs rather tightly to the data structure. If you change the struct definition in the sender, you must also change it in the listener, and vice versa. Exchanging structs also makes it more difficult to introduce new, interoperating programs in the future. Furthermore, exchanging internal structs makes it difficult to for your program to interoperate with programs developed in other languages.

Binary data and internal structs are also *platform dependent*—you cannot exchange raw, binary data between programs running on machines that represent numbers or character strings with different formats.

Instead of binary buffers or structs, we recommend using Rendezvous self-describing data to ease data exchange. Rendezvous datatypes span the most common atomic and array datatypes of most programming languages, and Rendezvous messages can emulate any struct or composite datatype.

Do Not Pass Local Values

If you exchange structs or binary buffers, remember that many data types could be meaningless at the receiving end. For example, a pointer is a memory address *inside a particular computer*—it has no meaning to any other program running on other computers. You must always send actual data *by value* rather than referencing it with a pointer.

Many opaque data structures or quantities are similarly meaningless outside of a particular program (for example, UNIX file descriptors). Do not send this kind of data to other programs.

Use Self-Describing Data

Use self-describing data to exchange information whenever possible. *Self-describing data* contains not only the values of interest to the program, but also descriptive names and type indicators. Rendezvous software uses its universal wire format for self-describing data to insulate your programs from the data representation differences across hardware and operating system platforms.

Several packages are available for managing self-describing data. Some implement industry standards such as XML, X.409, IDL or ASN.1. Others, like Rich Text Format (RTF) meet special needs of an industry or software tool. If your company or group has adopted one of these packages or standards, you can use it with Rendezvous software by packaging the data as opaque bytes (see `TIBRVMSG_OPAQUE` at Rendezvous Datatypes on page 71).

Like Rendezvous software, most of those packages include functions that map data between the formats used inside your program and a normalized format for network interchange—handling the details of format conversion, alignment and structure.

Establish Subject Naming Conventions

We recommend that you carefully plan the subject naming conventions for programs, and document them clearly for reference. Follow these guidelines:

- Plan naming conventions to reflect the logical structure of the data in the application domain.
- Study the programming examples in the `src/examples/` subdirectory.
- As you design naming conventions, think about the kinds of information that your programs will receive. Also think about the kinds of information that your program will ignore.
- Use a reasonably small number of levels in subject names—four or five is usually sufficient. (Rendezvous software permits many levels in subject names, but we recommend limiting the number of levels you actually use.)
- Avoid the use of spaces and special characters even where permitted by the Rendezvous API. They could cause trouble later with various editors, browsers and other tools.
- Keep subject names manageable and readable.
- Keep subject names short for maximum speed and message throughput.
- Allocate the maximum storage for subject names. Subject name length is artificially limited to 255 bytes so that programs can allocate name buffers with a reasonable size. To maximize code reusability, allocate 255 bytes for buffers that receive subject names, even if your program does not use such long names.

  In C and C++, the 255 byte limit is defined by the constant `TIBRV_SUBJECT_MAX` in the Rendezvous header files.
- Structure subject names so that subscribing programs can use wildcards effectively. Using wildcards is a powerful technique to filter inbound messages. Wildcards also offer a convenient way to subscribe to groups of subjects with a single listening call.

Do Not Send to Wildcard Subjects

⚠ We do not recommend sending messages to wildcard subject names. Although transports do not prevent you from sending to wildcard subjects, doing so can trigger unexpected behavior in other programs that share the network.

It is illegal for certified delivery transports to send to wildcard subjects.

Control Message Sizes

Although the ability to exchange large data buffers is a feature of Rendezvous software, it is best not to make messages *too* large. For example, to exchange data up to 10,000 bytes, a single message is efficient. But to send files that could be many megabytes in length, we recommend using multiple send calls, perhaps one for each record, block or track. Empirically determine the most efficient size for the prevailing network conditions. (The actual size limit is 64 MB, which is rarely an appropriate size.)

Avoid Flooding the Network

Rendezvous software can support high throughput, but all computers and networks have limits. Do not write programs that might flood the network with message traffic. Other computers must filter all multicast messages, at least at the hardware level and sometimes at software levels (operating system or Rendezvous daemon).

Do not code loops that repeatedly send messages without pausing between iterations. Pausing between messages helps leave sufficient network resources for other programs on the network. For example, if your program reads data from a local disk between network operations, it is unlikely to affect any other machines on a reasonably scaled and loaded network; the disk I/O between messages is a large enough pause.

Publishing programs can achieve high throughput rates by sending short bursts of messages punctuated by brief intervals. For example, structure the program as a timer callback function that sends a burst of messages each time its timer triggers; adjust the timer interval and the number of messages per burst for optimal performance.

When a program sends messages faster than the network can accommodate them, its outbound message queue grows. When any outbound message waits in the outbound message queue for more than 5 seconds, Rendezvous software presents a `CLIENT.FASTPRODUCER` warning advisory message. A program that receives this warning advisory message should slow the rate at which it sends messages.

Beware of Network Boundaries

Rendezvous software can use a combination of multicast, broadcast and point-to-point network messages. If your network includes bridges and routers, some of these messages may not cross segment boundaries.

Sometimes the network equipment can be configured to pass and block exactly the right traffic. In other situations, you may need Rendezvous routing daemon to pass broadcast and multicast traffic across wide-area network boundaries. Consult your system administrator or network administrator.

See Also Routing Daemon (rvrd), page 67 in *TIBCO Rendezvous Administration*

Make Transport Parameters Flexible

Ensure that system administrators and end-users can alter the program's service, network, and daemon parameters at each site. While it is proper and convenient for a program to use default values for these transport parameters, it is dangerous to assume that the default values you choose will work at every installation. Check that your program documentation explains how to change these parameters, and when it is appropriate to do so.

Verify Each Inbound Message

Programs must verify each inbound message field to assure integrity and robustness (so that inappropriate or unexpected messages do not cause errors within the program).

For example, always verify that a field has the expected datatype. Consider a suite of programs that uses a field named `SCORES` to carry an array of integer values. When a new program begins sending messages in which `SCORES` contains a string value, the existing programs must exhibit robust behavior. To ensure robustness, always check the datatype of a field before operating on its data value.

Understand Sockets

Transport creation calls accept two parameters that direct the transport to open *two different* kinds of sockets:

- The `service` parameter specifies a UDP or PGM service (also commonly called a UDP or PGM *port*); the transport opens a UDP or PGM socket to that network service.

  Rendezvous daemon processes uses the UDP or PGM service for communication with other Rendezvous daemon processes across the network.

- The `daemon` parameter specifies a TCP port number; the transport opens a TCP socket to that port.

  Transport objects use the TCP port for communication between a client program and its Rendezvous daemon (usually on the same host computer).

  This parameter corresponds to the `-listen` parameter of `rvd`.

These two types of socket are *not* interchangeable; confusing the two leads to programming errors that are difficult to diagnose and repair.

One source of this confusion is that the default `rendezvous` service (for TRPD daemons) is UDP service `7500`, and the default `daemon` parameter is TCP socket `7500`. Although these two numbers are the same, they specify different items.

Chapter 11 Certified Message Delivery

Although Rendezvous communications are highly reliable, some programs require even stronger assurances of delivery. Certified delivery features offer greater certainty of delivery—even in situations where processes and their network connections are unstable.

This chapter explains the conceptual foundations of Rendezvous certified message delivery, the way it works, and ways to use it.

Topics

- *Ledger Storage, page 167*
- *Relay Agent, page 169*

See Also

Certified delivery software uses advisory messages extensively. For example, advisories inform sending and receiving programs of the delivery status of each message. For complete details, see Appendix B, Certified Message Delivery (RVCM) Advisory Messages, on page 269.

With programs that send or receive certified messages across network boundaries, you must configure the Rendezvous routing daemons to exchange `_RVCM` administrative messages. Discuss this detail with your network administrator.

Table 24 Certified Message Delivery Programming Details

C	Certified Message Delivery on page 249 in *TIBCO Rendezvous C Reference*
C++	Certified Message Delivery on page 281 in *TIBCO Rendezvous C++ Reference*
Java	Certified Message Delivery on page 249 in *TIBCO Rendezvous Java Reference*
COM	Certified Message Delivery on page 217 in *TIBCO Rendezvous COM Reference*
.NET	Certified Message Delivery on page 155 in *TIBCO Rendezvous .NET Reference*

Certified Delivery Features

Rendezvous *certified* message delivery software offers stronger delivery assurances than standard Rendezvous *reliable* delivery.

- **Certainty**

  Certified delivery assures programs that every certified message reaches each intended recipient—in the order sent. When delivery is not possible, both sending and listening programs receive explicit information about each undelivered message.

- **Control**

  Programs determine an explicit time limit for each message.

  Sending programs can disallow certified delivery to specific listener transports.

- **Convenience**

  Once a program sends a certified message, Rendezvous software continues delivery attempts until delivery succeeds, or until the message's time limit expires.

- **Detail**

  Rendezvous certified delivery software presents advisory messages to inform programs of every significant event relating to delivery.

- **Ledger Recording: Process-Based or File-Based**

  Rendezvous certified delivery software records the status of each message in a ledger. For programs that require certification only for the duration of the program process, choose a process-based ledger. For programs that require certification that transcends process termination and program restart, choose a file-based ledger.

- **Graceful Degradation**

  Certified delivery meshes smoothly with standard Rendezvous communications. When certified delivery is disallowed, delivery conditions degrade gracefully to the standard Rendezvous reliable delivery semantics.

Reliable versus Certified Message Delivery

Standard Rendezvous communications software features *reliable message delivery*, which works well for many programs. *Certified message delivery* protocols offer even stronger assurances of delivery, along with tighter control, greater flexibility and fine-grained reporting. Table 25 compares the two delivery protocols.

Table 25 Comparing Reliable and Certified Message Delivery (Sheet 1 of 2)

Aspect	Reliable Delivery	Certified Delivery
Location of Protocols	Reliable message delivery protocols are implemented in the Rendezvous daemon (`rvd`).	Certified message delivery protocols are implemented in a separate library layer (`tibrvcm`). This library uses `rvd` for message transport.
Protocol Visibility	Reliable delivery protocols are invisible to programmers.	Certified delivery calls automatically adhere to certified delivery protocols, yet the protocols give programmers abundant status information and limited control.
Protocol Information	Rendezvous daemons inform programs when data is lost. No information about the lost data is available.	The library presents advisory messages to inform programs of every significant event related to certified delivery. Advisories identify specific messages by correspondent name, subject name and sequence number.
Ledger	None.	The certified delivery library records outbound messages in a ledger, either within the program process storage, or in file storage.
Time Limit	`rvd` retains outbound messages for 60 seconds.	The certified delivery library retains outbound messages in the ledger until either delivery is complete or the time limit (set by the program) expires.
Effective Range	60 seconds, or `rvd` process termination—whichever is first.	With persistent correspondents, certified delivery can extend beyond program process restart. It is not affected by `rvd` process termination.
Network Bandwidth	Minimal network overhead beyond the message itself.	Additional network overhead to confirm delivery of each certified message.

Table 25 Comparing Reliable and Certified Message Delivery (Sheet 2 of 2)

Aspect	Reliable Delivery	Certified Delivery
File Storage	No file storage overhead.	Optional file-based ledgers consume file storage for each message until delivery is complete (or the time limit expires).
Routing Daemons	Both protocols work across Rendezvous routing daemons (`rvrd`).	

Example Applications

Certified delivery is appropriate when a sending program requires individual confirmation of delivery for each message it sends. For example, a traveling sales representative enters sales orders on a laptop computer, and sends them to a central office. The representative must know for certain that the order processing system has received the data.

Certified delivery is also appropriate when a receiving program cannot afford to miss any messages. For example, in an application that processes orders to buy and sell inventory items, each order is important. If any orders are omitted, then inventory records are incorrect.

Certified delivery is appropriate when each message on a specific subject builds upon information in the previous message with that subject. For example, a sending program updates a receiving database, contributing part of the data in a record, but leaving other parts of the data unchanged. The database is correct only if all updates arrive in the order they are sent.

Certified delivery is appropriate in situations of intermittent physical connectivity—such as discontinuous network connections. For example, consider an application in which several mobile laptop computers must communicate with one another. Connectivity between mobile units is sporadic, requiring persistent storage of messages until the appropriate connections are reestablished.

Inappropriate Situations

High Data Rates

We do not recommend certified message delivery in situations that require high data rates. In comparison with reliable delivery, certified delivery requires additional processing time, exchanges more control messages, and consumes more process memory (and optionally disk storage as well). Usage of these resources grows in proportion to three quantities:

- The number of certified messages sent.
- The size of the data in those messages.
- The number of listeners that receive certified delivery.

Therefore, for optimal performance, we encourage sparing use of certified message delivery.

Decentralization

Rendezvous certified message delivery has a decentralized, stream-oriented, peer-to-peer architecture. Table 26 outlines the differences between centralized, server-based architectures (such as message queuing products), and decentralized architectures (such as Rendezvous certified message delivery).

Table 26 Centralized versus Decentralized Architecture (Sheet 1 of 3)

Aspect	Centralized	Decentralized
Example	JMS Message queuing products.	Rendezvous Certified Message Delivery
Components	Message producers. Message consumers. Centralized server as intermediary.	Message producers. Message consumers.
Basic Operating Principle	A producer sends a message to the central server. The server stores the message until it has delivered it to each consumer.	A producer sends a message to consumers. The producer stores the message until each consumer has acknowledged receipt.
Communication Pattern	Producer to server; server to consumers.	Peer-to-peer.
Protocol	Store and forward queue protocol.	Stream-oriented protocol.
Administration	An intermediary server is required between producers and consumers.	No intermediary is required. Producers communicate directly with consumers.

Table 26 Centralized versus Decentralized Architecture (Sheet 2 of 3)

Aspect	Centralized	Decentralized
Resources		
Network Bandwidth	Each message traverses the network at least twice—once from the producer to the server, and again from the server to the consumers. Some servers multicast to all consumers simultaneously; others send to each consumer individually. Control and protocol messages use additional bandwidth.	Each message traverses the network once from producer to all consumers. Control and protocol messages use additional bandwidth.
Storage Resources	The central server stores all messages and delivery state for all its clients; it requires disk resources in proportion to total throughput volume.	Each producer stores its outbound messages and some delivery state; it requires disk resources in proportion to its outbound volume. Each consumer stores its inbound delivery state; it requires minimal disk resources.
Storage Integrity	Disk failure on a server host computer can be catastrophic, affecting all messages from every client. Many installations protect against disk failure using safeguards such as disk mirroring.	Disk failure on a peer host computer affects only the messages that its programs produce or consume. However, disk mirroring for each individual peer is often impractical.

Table 26 Centralized versus Decentralized Architecture (Sheet 3 of 3)

Aspect	Centralized	Decentralized
State		
State Information	All information about message delivery state resides with the central server.	Information about message delivery state is distributed, residing in part with each individual producer, and in part with each individual consumer.
State Master	The central server is the master of overall delivery state.	Since delivery state information is distributed, no entity can be the single master of the overall state. Rather, individual peers are masters of their own parts of the state. Relay agents are not masters of any state.
Monitoring Delivery State	Programs can query the state master (server) about delivery state.	Delivery state monitoring requires application-level code in each producer and consumer.
Changing Delivery State	In some centralized architectures, the state master (server) can make administrative changes to delivery state—for example, it might delete, reorder, or replay messages.	No central component can make administrative changes to overall delivery state.

Certified Message Delivery in Action

Certified message delivery is a protocol with several steps, each described in a subsequent section:

- Creating a CM Transport, page 150.
- Discovery and Registration for Certified Delivery, page 154.
- Delivering a Certified Message, page 156.

Creating a CM Transport

To send or receive messages using certified delivery features, a program must first create a *CM transport* (also called a delivery-tracking transport). Each CM transport employs an ordinary transport for network communications. The CM transport adds information so that it can participate in certified delivery protocols; the additional information includes a name and a ledger.

Table 27 CM Transport Creation Calls

C	`tibrvcmTransport_Create()`
C++	`TibrvCmTransport::create()`
Java	`TibrvCmTransport()`
COM	`TibrvCmTransport.create`
.NET	`CMTransport`

CM Correspondent Name

Each CM transport has a name—which may be reusable, or non-reusable. The name identifies the CM transport to other CM transports, and is part of the CM label that identifies outbound messages from the CM transport.

A name is *reusable* when a program supplies it explicitly to the CM transport creation call. When a CM transport with a reusable name also has a file-based ledger, it operates as an instance of a *persistent correspondent*—which allows continuity of certified delivery beyond transport invalidation and program restarts (for more information, see Persistent Correspondents on page 159).

Two CM transports must not bind the same reusable name—that is, at any moment in time, each reusable name must be unique throughout the network. CM transports may reuse a name *sequentially*, but *not simultaneously*. Violating this rule can significantly obstruct certified delivery.

Programs may omit the name from the CM transport creation call—in which case the call generates a unique, *non-reusable* name for the CM transport. No other CM transport on any computer can *ever* have the same name. As a result, a CM transport with a non-reusable name operates as a *transient correspondent*—no subsequent CM transport can continue the certified delivery behavior of a transient CM transport.

Correspondent names have the same syntax as Rendezvous subject names. For more information about the syntax of reusable names, and practical advice on selecting a reusable name, see Reusable Names on page 166. For further details about the syntax of Rendezvous subject names, see Subject Names on page 61.

Ledger

Each CM transport keeps a *ledger*, in which it records information about every unresolved outbound certified message, every subject for which this CM transport receives (inbound) certified messages, and other cooperating CM transports.

Programs may store the ledger in a *ledger file*, or in process-based storage within the running program. (Even when a CM transport uses a ledger file, it may sometimes replicate parts of the ledger in process-based storage for efficiency; however, programmers cannot rely on this replication.)

Ledger files must be unique. That is, two CM transports must not use the same ledger file (concurrently).

A CM transport with a file-based ledger and a reusable name qualifies as a *persistent correspondent*, with certified delivery behavior that can extend beyond CM transport destruction.

Labeled Messages

A *labeled message* is like an ordinary Rendezvous message, except that it includes supplementary information, which CM transports can use for certified message delivery:

- The correspondent name of the CM transport that sent the message.
- A sequence number assigned by the sending CM transport.
- A time limit, after which the sending program no longer expects its CM transport to certify delivery of the message.

Sending a Labeled Message

Any CM transport can send a labeled message by using the sending calls in the certified message delivery library layer (see Table 28).

Table 28 CM Send Calls

C	`tibrvcmTransport_Send()`
	`tibrvcmTransport_SendReply()`
	`tibrvcmTransport_SendRequest()`

C++	`TibrvCmTransport::send()`
	`TibrvCmTransport::sendReply()`
	`TibrvCmTransport::sendRequest()`

Java	`TibrvCmTransport.send()`
	`TibrvCmTransport.sendReply()`
	`TibrvCmTransport.sendRequest()`

COM	`TibrvCmTransport.send`
	`TibrvCmTransport.sendReply`
	`TibrvCmTransport.sendRequest()`

.NET	`CMTransport.Send`
	`CMTransport.SendReply`
	`CMTransport.SendRequest`

Receiving a Labeled Message

Two kinds of listening transport can receive labeled messages:

- An ordinary transport listens by creating an ordinary listening event.
- A CM transport listens by creating a CM listening event.

Ordinary Listener Transport

When an ordinary transport receives a labeled message, it presents it to the appropriate callback function as if it were an ordinary message. That is, it ignores the supplementary information that distinguishes a labeled message (the sender's correspondent name and sequence number).

CM Listener Transport

When a CM transport receives a labeled message, its behavior depends on context:

- If the CM listener transport is registered for certified delivery, it presents the supplementary information to the callback function.
- If a CM listener transport is *not* registered for certified delivery with the sending CM transport, it presents the sending transport's correspondent name to the callback function, but omits the sequence number.

  In addition, if appropriate, the CM listener transport automatically requests that the sending transport register the listener for certified delivery. (See Discovery and Registration for Certified Delivery on page 154.)

Discovery and Registration for Certified Delivery

Discovery

When a CM listening transport receives a labeled message from a sending CM transport that is not listed in the listener's ledger, we say that the listener *discovers* the sender on the message subject.

Four actions follow discovery:

- The CM listener transport adds the sender's correspondent name to the listener's ledger, as a source of messages on the subject.
- The CM listener transport contacts the CM sending transport to *request registration* for certified delivery of the subject.
- The CM listener transport presents a `REGISTRATION.DISCOVERY` advisory.
- The CM listener transport stores inbound messages on the newly discovered subject from the CM sender.

Registration

When a sending CM transport receives a registration request from a CM listener transport, the sender automatically *accepts* the request (but see Disallowing Certified Delivery on page 164, and No Response to Registration Requests on page 165). Acceptance consists of these four actions:

- The CM sender transport *registers* the listener for certified delivery of the subject— recording that fact in the sender's ledger.
- The CM sender transport notifies the CM listener transport that the registration requested is accepted—the sender accepts responsibility for certified delivery on the subject.
- The CM sender transport presents a `REGISTRATION.REQUEST` advisory, to announce the new registered listener to the sending program.
- When the CM listener transport receives the acceptance reply, it presents a `REGISTRATION.CERTIFIED` advisory.

After registration completes successfully, the CM listener transport queues the stored inbound messages in the correct sequence.

Certified Delivery Agreement

Following registration and acceptance, the sending and listening CM transports have a *certified delivery agreement* on the subject.

- The sending CM transport is responsible to record each outbound message on that subject, and to retain the message in its ledger until it receives confirmation of delivery from the listener (or until the time limit of the message elapses).
- In return, the listening CM transport is responsible for confirming delivery of each message.

Rendezvous certified delivery software arranges all of this accounting automatically. The sending and listening programs do not participate directly in these protocols—only indirectly, by sending and listening with certified delivery library calls.

Notice that although both transports participate in a certified delivery agreement, the agreement is asymmetric—it certifies messages from a sender to a listener. A two-way conversation requires *two* separate certified delivery agreements to certify messages in both directions.

We refer to the two CM transports that participate in a certified delivery agreement as a *certified sender* and a *certified listener*, and the labeled messages that flow between them are *certified messages*. Notice the subtle difference in terminology—before establishing a certified delivery agreement, the participating transports are *CM senders and CM listeners*; afterward, they are *certified senders and certified listeners.* Similarly, a labeled message becomes a *certified message* only when the sender and receiver maintain a certified delivery agreement.

Delivering a Certified Message

Once a delivery agreement is in place, all subsequent messages on the subject (from the certified sender to the certified listener) are *certified messages*. Each certified message generates a series of protocol actions:

- When a certified listening transport queues a certified message for the listener's callback function, it includes the sender's correspondent name and the message sequence number.
- When the callback function returns, the certified listening transport automatically confirms delivery to the sender. (Programs can override this behavior and confirm delivery explicitly.)
- When confirmation reaches the certified sender, the sending transport records delivery in its ledger, and presents a `DELIVERY.CONFIRM` advisory.
- When confirmation has arrived from every certified listener for this message, the sending transport deletes the message from its ledger, and presents a `DELIVERY.COMPLETE` advisory (to the transport it employs for network communication).

Automatic Confirmation of Delivery

The default behavior of certified listener transports is to automatically confirm message delivery upon return from the callback function. Programs can selectively override this behavior for specific CM listener event objects (without affecting other listener event objects).

By overriding automatic confirmation, the listening program assumes responsibility for explicitly confirming each inbound certified message.

Consider overriding automatic confirmation when processing inbound messages involves activity that is asynchronous with respect to the message callback function, such as computations in other threads or additional network communications.

Table 29 Confirmation of Delivery Calls (Sheet 1 of 2)

C	`tibrvcmEvent_SetExplicitConfirm()`
	`tibrvcmEvent_ConfirmMsg()`

C++	`TibrvCmListener::confirmMsg()`
	`TibrvCmListener::setExplicitConfirm()`

Table 29 Confirmation of Delivery Calls (Sheet 2 of 2)

Java	`TibrvCmListener.confirmMsg()`
	`TibrvCmListener.setExplicitConfirm()`

COM	`TibrvCmListener.confirmMsg` `TibrvCmListener.setExplicitConfirm`
.NET	`CMListener.ConfirmMessage` `CMListener.SetExplicitConfirmation`

Requesting Confirmation

If a certified sender transport does not receive prompt confirmation of delivery from a certified listener transport (for example, because of network glitches), it automatically requests confirmation. After each request, it presents a `DELIVERY.NO_RESPONSE` advisory.

When a certified listening transport receives a request for confirmation, it checks its ledger, and reconfirms receipt of the messages that it has already confirmed. (This behavior is identical, whether the program uses automatic confirmation, or overrides it.)

Sequencing and Retransmission

Each sending CM transport assigns sequence numbers serially for each outbound subject, so the sequence numbers reflect the order of messages from the sending transport on a specific subject.

Certified messages always dispatch from the event queue in order by sequence number.

For example, a certified listening transport is receiving certified delivery of the subject `FOO` from a certified sender named `BAZ`. After receiving and queuing message number `32`, the next message to arrive is message `35`. Certified delivery software holds message `35` until it can first queue messages `33` and `34`; once these messages arrive, the listening transport queues events for each of the three messages in the proper order.

Meanwhile, the certified listening transport automatically requests retransmission of messages `33` and `34` from `BAZ`. In a case where the time limit on those messages has expired—so `BAZ` no longer has them in its ledger—the certified listener transport presents a `DELIVERY.UNAVAILABLE` advisory, indicating that messages `33` and `34` are no longer available. Then it queues an event for message `35`.

Notice that although certified messages always dispatch from the queue in order of sequence number, it is still possible that a program might process them out of order. For example, if a program dispatches the queue from several threads, the thread processing number `43` might return from its callback function before the thread processing number `42`.

Persistent Correspondents

We introduced the concept of persistent correspondents in the section CM Correspondent Name on page 150. A reusable name and a file-based ledger allow a persistent correspondent to continue certified delivery beyond the invalidation of a CM transport or the restart of a process.

Example

Consider an example application system, in which program JOE generates important information, and sends it to program SUE in certified messages on the subject REMEMBER.THIS. Upon receipt, SUE stores the information in a database.

If either JOE or SUE terminate unexpectedly, it is crucial that certified messages still arrive for entry into the database. To ensure this result, both programs must represent persistent correspondents—that is, both programs create CM transports with reusable names (JOE_PER and SUE_PER), and each of these CM transports keeps a file-based ledger. In addition, SUE requires old messages by setting a parameter when creating the CM transport SUE_PER.

During operation, JOE has sent message number 57 on the subject REMEMBER.THIS, but has not yet received delivery confirmation for messages 53-56. SUE is processing message 53, when a sudden hardware failure causes SUE to terminate. Meanwhile, JOE continues to send messages 58-77.

When the computer restarts, SUE restarts and recreates SUE_PER. The ledger file for SUE_PER indicates that message 52 was received and confirmed for the subject REMEMBER.THIS. As soon as SUE_PER discovers that JOE_PER sends labeled messages on the required subject, SUE_PER requests a certified delivery agreement for the subject REMEMBER.THIS. When JOE_PER accepts, JOE_PER retransmits the stored messages 53-77 on that subject.

Notice these details:

- SUE does not miss any REMEMBER.THIS messages. However, the new SUE_PER must gracefully fix any difficulties caused by partial processing of message 53 by the old SUE_PER.
- JOE and SUE communicate using a public subject name—not an inbox. Inbox names are unique, so they cannot continue beyond transport invalidation.
- SUE explicitly requires old messages by setting a parameter when creating the persistent correspondent named SUE_PER; this parameter ensures that JOE_PER retransmits certified messages that the previous instance of SUE_PER had not confirmed. If the value of the requireOldMsgs argument were false,

JOE_PER would delete stored outbound messages for SUE_PER from its ledger, instead of retransmitting them.

Anticipating a Listener

In some situations, a sending CM transport can anticipate the request for certified delivery from a (listener) persistent correspondent that has not yet registered.

Consider an example in which a database program (`DB`) records all messages with the subject `STORE.THIS`. The program `DB` creates a CM transport that instantiates a persistent correspondent named `DB_PER`. All programs that send messages with the subject `STORE.THIS` depend on this storage mechanism.

One such sending program is `JAN`. Whenever `JAN` starts, it can anticipate that `DB_PER` will request certified delivery of the subject `STORE.THIS`. Suppose that `JAN` starts, but `DB` is not running, or a network disconnect has isolated `JAN` from `DB`. Anticipating that it will eventually receive a registration request for `STORE.THIS` from `DB_PER`, `JAN` makes an add listener call. The effect is that the sending CM transport in `JAN` behaves as if it has a certified delivery agreement with `DB_PER` for the subject `STORE.THIS`; it stores outbound messages (on that subject) in its ledger. When `DB` restarts, or the network reconnects, the sender CM transport in `JAN` automatically retransmits all the stored messages to `DB_PER`.

It is not sufficient for a sender to anticipate listeners; the anticipated listening programs must also require old messages when they create their CM transports.

The sending transport must be available to process the registration request and redeliver stored messages (if necessary).

Table 30 Add Listener Calls

C	`tibrvcmTransport_AddListener()`
C++	`TibrvCmTransport::addListener()`
Java	`TibrvCmTransport.addListener()`
COM	`TibrvCmTransport.addListener`
.NET	`CMTransport.AddListener`

Canceling Certified Delivery

Either a listening or a sending program can cancel a certified delivery agreement.

A listening program can cancel agreements when destroying the CM listener *event* object, using a cancel agreements parameter of the calls in Table 31. All sending CM transports that had certified delivery agreements to the destroyed listener present `REGISTRATION.CLOSED` advisories.

Notice that destroying the CM transport object in the listening program implicitly invalidates all its listener events, but does not cancel their certified delivery agreements; the sender continues to store outbound messages in its ledger.

Table 31 Close Listener Calls

C	`tibrvcmEvent_Destroy()`
C++	`TibrvCmListener::destroy()`
Java	`TibrvCmListener.destroy()`
COM	Delete the CM listener event object (or `TibrvCmListener.destroy`)
.NET	`CMListener.Destroy`

A certified sender transport can cancel certified delivery of a specific subject to a specific CM listening transport, using calls in Table 32. The sender transport deletes from its ledger all information about delivery of the subject to the listening transport. The sender presents a `REGISTRATION.CLOSED` advisory. If the listening correspondent is available (that is, its program is running and reachable), it presents a `REGISTRATION.NOT_CERTIFIED` advisory. (Unlike the disallow listener calls in Table 33 on page 164, these calls do not cause denial of subsequent registration requests.)

Table 32 Remove Listener Calls (Sheet 1 of 2)

C	`tibrvcmTransport_RemoveListener()`
C++	`TibrvCmTransport::removeListener()`
Java	`TibrvCmTransport.removeListener()`
COM	`TibrvCmTransport.removeListener`

Table 32 Remove Listener Calls (Sheet 2 of 2)

.NET	`CMTransport.RemoveListener`

Disallowing Certified Delivery

As described in Registration on page 154, CM transports automatically accept all registration requests. This is true *except* when a CM sender transport explicitly *disallows* certified delivery to a listening correspondent.

Calls in Table 33 disallow a listening correspondent; these calls cancel existing certified delivery agreements with the listening correspondent (on all subjects), and cause the CM transport to automatically deny subsequent registration requests from the listening correspondent.

When a CM sender transport has disallowed a listening correspondent, the events connected with registration do not occur. Instead, the CM sender transport notifies the listener transport that the request is disallowed. When the listening transport receives the rejection notice, it presents a `REGISTRATION.NOT_CERTIFIED` advisory.

Allow listener calls supersede the effect of a previous disallow listener call, allowing subsequent registration requests from the listener transport to succeed.

Table 33 Disallow Listener Calls

C	`tibrvcmTransport_DisallowListener()`
C++	`TibrvCmTransport::disallowListener()`
Java	`TibrvCmTransport.disallowListener()`
COM	`TibrvCmTransport.disallowListener`
.NET	`CMTransport.DisallowListener`

No Response to Registration Requests

It is possible that a registration request never reaches the CM sender transport, or the acceptance notice never reaches the CM listener transport (for example, because of network glitches, or termination of the sending program). After repeated attempts to register without response from the sender, the listening CM transport presents a `REGISTRATION.NO_RESPONSE` advisory. After several attempts to register with no response, the listening transport stops sending requests.

Reusable Names

CM transports that represent persistent correspondents require reusable names. Reusable names must obey the syntax for Rendezvous subject names.

Reusable names must not contain wildcard characters. Reusable names may not begin with reserved elements (such as `_INBOX`, `_RV` or `_LOCAL`).

For best performance, keep reusable names short—only a few characters. At maximum, use no more than three or four elements, and no more than 50 characters total.

The empty string (“ “) is not a legal correspondent name.

For syntactic details of subject names, which also apply to reusable names, see Subject Names on page 61.

Ledger Storage

Each CM transport records information in a ledger, which occupies storage space—whether within the program process, in a ledger file, or both.

A CM transport that represents a persistent correspondent must keep a copy of the ledger in a file. The file-based ledger preserves certified delivery information beyond transport invalidation, or process termination and restart.

A file-based ledger has two associated costs:

- The ledger file consumes disk space.
- The program pauses to update the ledger file (synchronously or asynchronously).

Transient correspondents need not pay these costs, because they do not use ledger files. However, keeping the ledger in process-based storage consumes process memory.

Rendezvous software neither clears nor deletes ledger files.

See Also Persistent Correspondents, page 159.

Ledger Size

The size of the ledger depends on several factors—the most important of which is the retention rate of stored data. That is, the ledger grows fastest in response to the cumulative length of incompletely delivered messages.

Program developers can estimate the expected size of the ledger, and must ensure that the process can allocate sufficient memory to contain it. For a file-based ledger, ensure that sufficient disk space is available as well.

Ledger Storage Allocation

Ledger files use a general storage allocation scheme:

- Each ledger acquires a pool of storage, and manages that pool. The ledger reuses existing storage whenever possible. Ledger files grow as needed.
- Ledger files grow by adding storage in integer multiples of a minimum allocation size.
- Because storage allocation involves *expensive* operating system calls, ledger files pre-allocate large blocks of storage to reduce the number and frequency of such calls.

- Rendezvous software automatically reclaims ledger file storage on operating systems that support this feature.

Ledger File Location

The ledger file must reside on the same host computer as the program that uses it. Do *not* use network-mounted storage for ledger files.

Remember that certified message delivery protects against component or network failure. Placing ledger files across a network (for example, on a separate file server) introduces a new dependency on the network, leaving components vulnerable to network failures.

Relay Agent

Relay agents support certified delivery in situations where persistent correspondents connect only intermittently to the network. This feature supports certified message delivery among laptop computers, and among persistent correspondents that run as ephemeral processes (for example, UNIX `cron` jobs).

For example, consider the situation in Figure 14 on page 170. Mobile employees use laptop computers, connecting to the network whenever time and communications access permit. They run programs that communicate using certified message delivery. Relay agents collect messages on behalf of disconnected client programs, and deliver them when the programs reconnect. Relay agents also store certified delivery protocol state on behalf of their client programs, and resynchronize that state whenever the client reconnects.

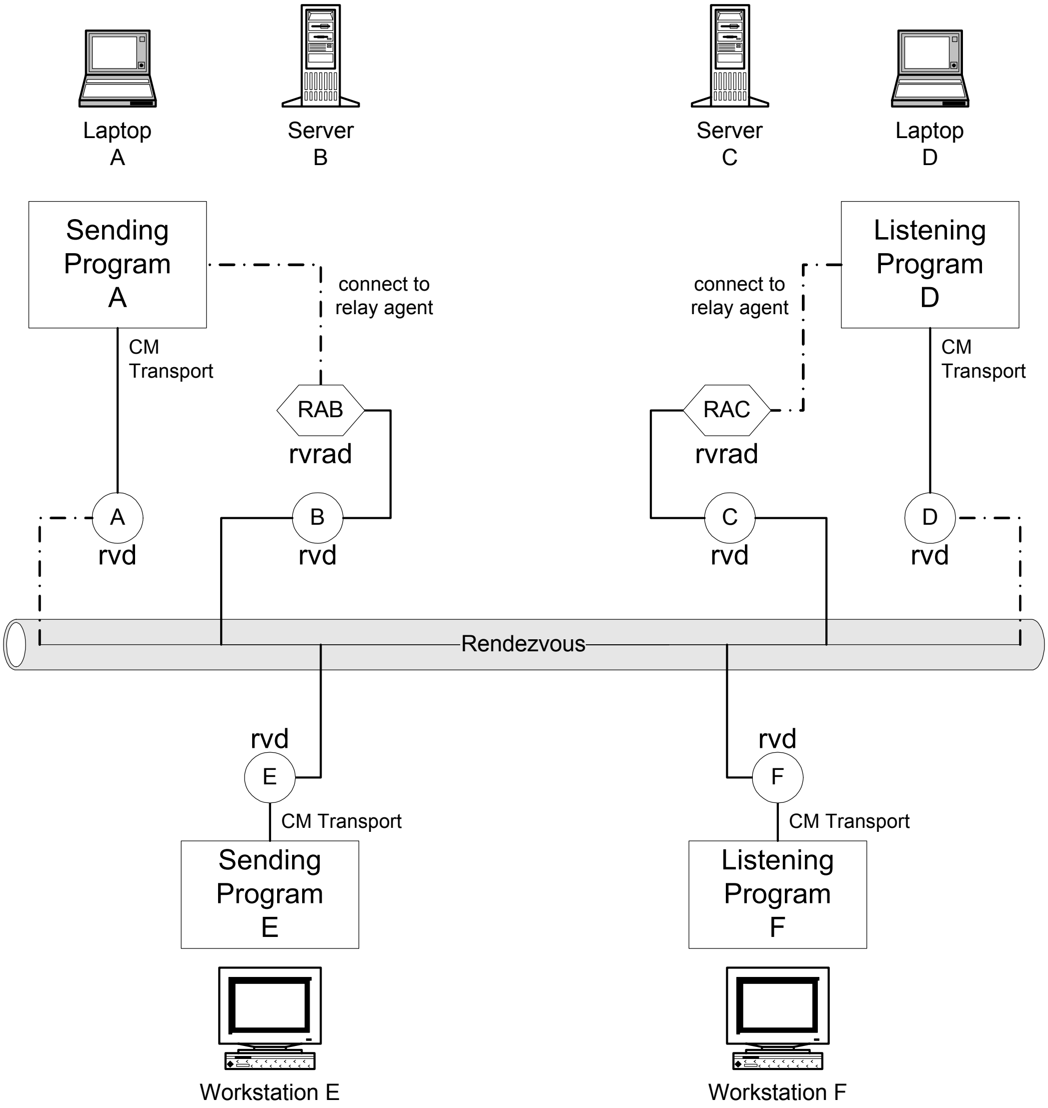
Operation

In Figure 14 on page 170, solid lines represent continuous connections, while broken lines represent intermittent connections.

The sending program on laptop A creates a CM transport, designating RAB as its relay agent. The listening program on laptop D creates a CM transport, designating RAC as its relay agent. (When both programs run in the same network, they could designate the same relay agent. In this example, the programs use separate relay agents, to illustrate that computers B and C could be in different networks.) Computers E and F remain continuously connected to the network; their programs do not require relay agents, but they can interact with RAB and RAC.

The relay agents store inbound certified messages and labeled messages (and other messages related to certified delivery features) on behalf of their disconnected client programs. When a client is connected, it receives inbound messages immediately.

Figure 14 Relay Agents



When laptop A is connected to the network, and connected to RAB, the sending program transfers its outbound certified messages to RAB, which in turn sends them to the network. When laptop A is disconnected, the program process stores its outbound certified messages; upon reconnection, it transfers the stored messages to RAB, which sends them to the network.

When laptop D is connected to the network, and connected to RAC, the listening program receives all inbound certified messages through RAC. When the connected listener confirms receipt, the confirmations flow through RAC.

In contrast, when laptop D is disconnected, the relay agent named RAC stores certified messages on behalf of the listening program; upon reconnection, RAC transfers the stored messages to the listening program.

When laptops A and D are both connected to the network, the sending program receives inbound confirmations as the listening program on D sends them (through intermediaries RAC and RAB). When laptop A is disconnected, the relay agent named RAB stores the confirmations from D; upon reconnection, it transfers the confirmations to the sending program on A. If the listener on D disconnects before confirming receipt, the listener stores the confirmations until it reconnects to RAC; upon reconnection, the listener transfers the stored confirmations to RAC.

Communication

Once a CM transport designates a relay agent, *all* its communications flow through that relay agent.

Each time a CM transport reconnects with the network, it attempts to contact its designated relay agent using multicast or broadcast protocol messages. After establishing contact, the program client and its relay agent use point-to-point communications to transfer messages and resynchronize protocol state information.

Protocol State

Relay agents mirror the protocol state associated with certified delivery—including registration state and message confirmation state. When the client is disconnected from the relay agent, their protocol states can diverge; they resynchronize when the client reconnects. The client CM transport is *always* the master of the protocol state, and the relay agent is the mirror.

Transparency

Relay agents are *transparent*; that is, the *semantics* of certified delivery, confirmation, discovery, and registration remain the same, whether or not relay agents are operating. Confirmation indicates that the certified listener has actually received the certified message (relay agents do not generate confirmations on behalf of listening clients).

Connecting and Disconnecting

A program can control the connections to its relay agent *implicitly* or *explicitly*:

- Implicit control works best for a program that creates a new CM transport each time it establishes network connectivity.

  — Creating a CM transport implicitly connects it to its designated relay agent.

  — Destroying a CM transport implicitly disconnects it from its relay agent.

  For example, consider a laptop program that the user terminates before disconnecting, and restarts after reconnecting. Even though this program embodies a persistent correspondent, it creates a new CM transport object each time it restarts.

  As another example, consider an ephemeral process (such as a UNIX `cron` job). Each time it starts, it creates a new CM transport object, even though each instance embodies the same persistent correspondent.

- Explicit control works best for a program that retains the same CM transport object through several cycles of intermittent network connectivity.

  — Explicitly connect each CM transport to its relay agent whenever network connectivity resumes.

  — Explicitly disconnect each CM transport from its relay agent before disconnecting from the network.

  For example, consider a laptop user who connects to the network whenever a telephone line is available, yet continues to use the same program process even when the computer is physically disconnected from the network. The program can use one CM transport object for its entire process lifetime, explicitly connecting to the relay agent when the computer connects to the network, and disconnecting from the relay agent when the computer disconnects from the network.

Table 34 summarizes the calls that control relay agent connections—implicitly and explicitly.

Table 34 Relay Agent Connect and Disconnect Calls (Sheet 1 of 2)

C	Implicit	`tibrvcmTransport_Create()`
		`tibrvcmTransport_Destroy()`

	Explicit	`tibrvcmTransport_ConnectToRelayAgent()`
		`tibrvcmTransport_DisconnectFromRelayAgent()`

Table 34 Relay Agent Connect and Disconnect Calls (Sheet 2 of 2)

C++	Implicit	`TibrvCmTransport::create()`
		`TibrvCmTransport::destroy()`
	Explicit	`TibrvCmTransport::connectToRelayAgent()`
		`TibrvCmTransport::disconnectFromRelayAgent()`
Java	Implicit	`TibrvCmTransport()`
		`TibrvCmTransport.destroy()`
	Explicit	`TibrvCmTransport.connectToRelayAgent()`
		`TibrvCmTransport.disconnectFromRelayAgent()`
COM	Implicit	`TibrvCmTransport.create`
		`TibrvCmTransport.destroy`
	Explicit	`TibrvCmTransport.connectToRelayAgent`
		`TibrvCmTransport.disconnectFromRelayAgent`
.NET	Implicit	`CMTransport`
		`CMTransport.Destroy`
	Explicit	`CMTransport.ConnectToRelayAgent`
		`CMTransport.DisconnectFromRelayAgent`

Connecting

Connect calls are non-blocking; they immediately return control to the program, and asynchronously attempt to connect to the relay agent (continuing until they succeed, or until the program makes a disconnect call).

When a CM transport attempts to connect to a relay agent, Rendezvous software automatically locates the relay agent process (if it exists). When the CM transport successfully connects to the relay agent, they synchronize:

- The CM transport receives a `RELAY.CONNECTED` advisory, informing it of successful contact with the relay agent.

  (When a CM transport cannot locate its relay agent, it presents a `DELIVERY.NO_RESPONSE` advisory; however, we recommend against designing programs to rely on this side effect.)

- If the client transport is a *CM listener*, the relay agent ensures that it is listening to the same set of subjects on behalf of the client. The relay agent also updates its confirmation state to reflect the state of the program transport.
- If the client transport is a *CM sender*, the relay agent updates its acceptance state to reflect the state of the program. The sending client updates its confirmation state to reflect the state of the relay agent.
- The CM transport and relay agent exchange the data messages that they have been storing during the time they were disconnected.

We recommend that programs remain connected for a minimum of two minutes, to allow time for synchronization to complete. (Two minutes is a generous estimate, which is sufficient for most situations. Actual time synchronization time can be much shorter, and varies with the number of stored messages and the degree to which protocol state has changed.)

Disconnecting

Disconnect calls are non-blocking; they immediately return control to the program, and asynchronously proceed with clean-up tasks:

- If the client transport is a *CM listener*, the relay agent attempts to synchronize its listening state with the transport (to assure that the relay agent adequately represents the CM listeners of the client).
- The CM transport stops communicating with the relay agent.

  The CM transport stores subsequent outbound events—including data messages and protocol state changes. If the CM transport is a certified *sender*, it stops requesting delivery confirmation for outstanding unconfirmed messages. (See also, Requesting Confirmation on page 157.)

  The relay agent stores subsequent inbound events for the CM transport—including data messages and protocol state changes.
- A CM transport that explicitly disconnects (and is not destroyed) presents a `RELAY.DISCONNECTED` advisory, informing the program that it is safe to sever the physical network connection. (A CM transport that implicitly disconnects during its destruction sequence will never present this advisory; instead, it is safe to sever the physical connection when the destroy call returns.)

Delays

Programs that connect only intermittently add delays to several phases of certified message delivery, including discovery and registration, delivery, and confirmation of receipt. Because of these delays, certified messages might require longer time limits than they would in the absence of relay agents.

To understand the possible sources of delays, consider this worst-case situation. Figure 15 on page 176 depicts the network arrangement; Figure 16 on page 177 illustrates the narrative:

1. The CM transport on laptop D listens to the subject `foo`. When the listener transport connects to its relay agent (RAC), RAC begins listening on its behalf. The listener transport then disconnects from RAC.

2. The CM transport on laptop A sends messages to the subject `foo`.

   Although the CM send call produces a labeled message, the CM transport on D is not yet registered as a certified listener—so when it receives this message, it does not confirm receipt. By anticipating the listener and pre-registering it, the CM sender can certify this outbound message even before the listener requests registration.

   When the sender transport connects to its relay agent (RAB), it transfers its stored outbound messages to RAB, which in turn sends them to the network. The sender then disconnects from RAB.

3. RAC receives the messages from RAB, stores them for its listening CM transport client on D, and automatically requests registration on behalf of that listening client.

   RAB receives the registration request, and stores it for its sending client on A.

4. When the CM sending transport on A connects, it transfers its waiting outbound messages to RAB, and accepts the registration request.

   Both RAB and RAC store the acceptance state.

   From this time forward, the sender and listener transports have a certified delivery agreement (even though the listener has not yet received any notice of acceptance). Subsequent messages from the sender are certified (but messages sent previously are not certified).

   Then the sender disconnects.

5. When the listening CM transport on D connects, it transfers the acceptance state from RAC, along with stored inbound messages. The listener confirms receipt of the certified messages. Both RAB and RAC store the confirmation state.

6. When the sending CM transport on A connects, it transfers the confirmation state from RAB.

Figure 15 Relay Agents and Delays: Network

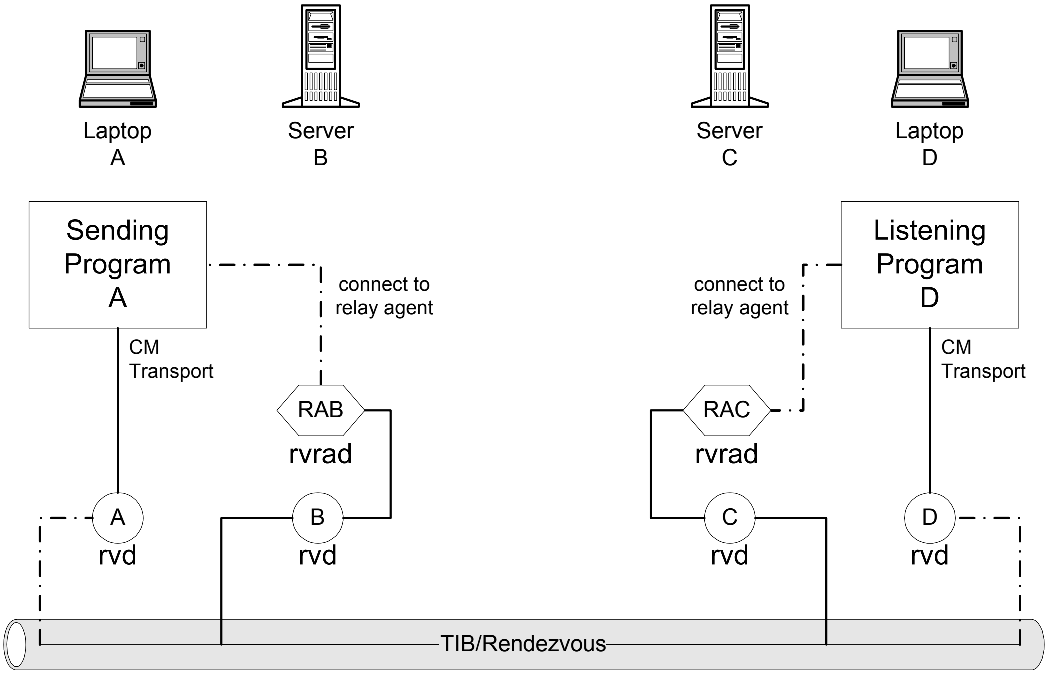
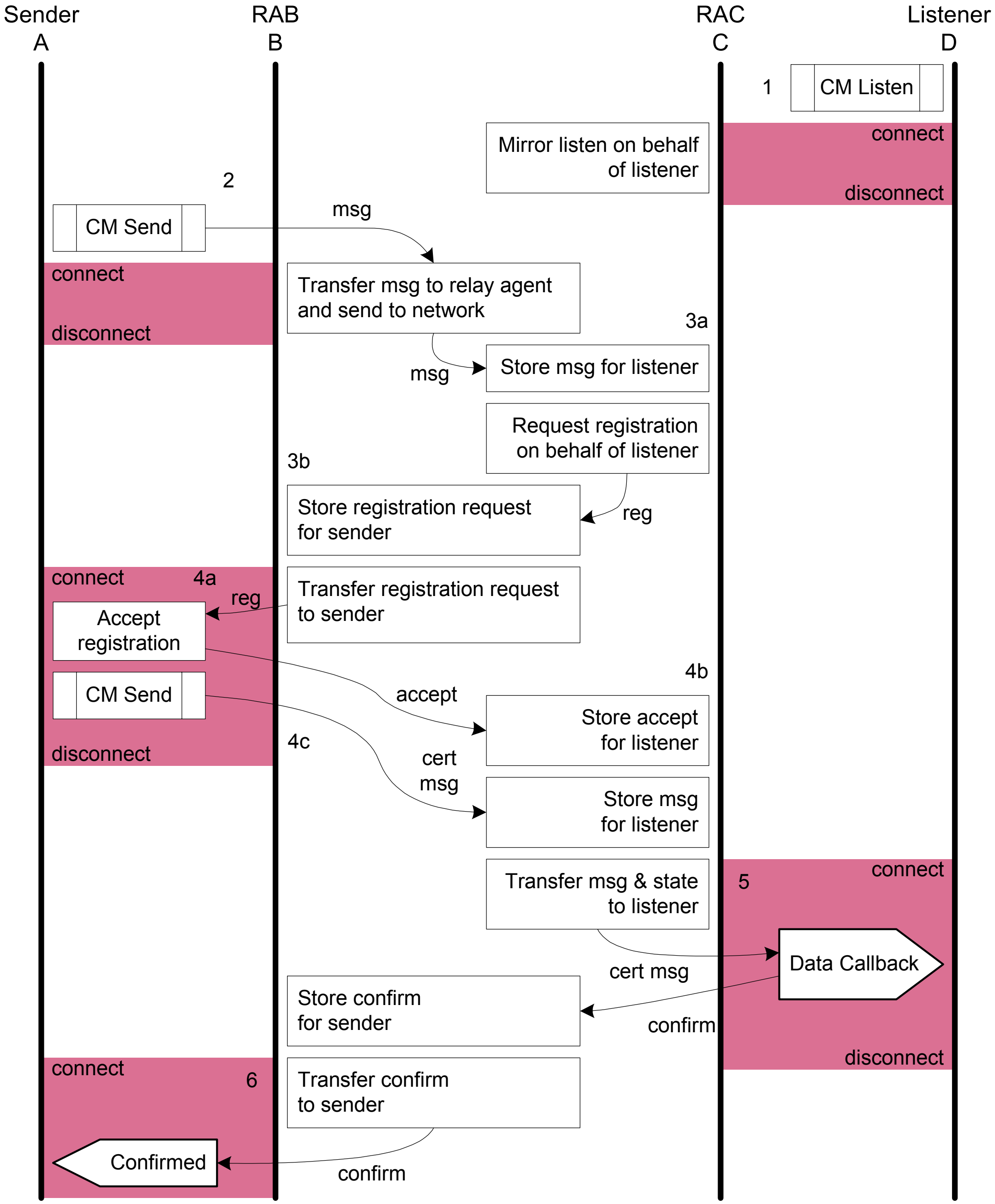


Figure 16 Relay Agents and Delays: Timing



Notice these delays:

- Although the listener transport begins listening at step 1, the sender transport does not receive a request for certified delivery until step 4. Messages sent before establishing the certified delivery agreement are not certified for that listener.
- Although the sender transport begins sending certified messages at step 4, it does not receive confirmation until step 6. The message time limit must be longer than this delay, otherwise the message expires before confirmation arrives, and the sender transport presents a `DELIVERY.FAILED` advisory.

Capacity

Each relay agent can serve CM transports in several client programs simultaneously, limited only by the relay agent host computer and its network interface. The client CM transports of a relay agent can both send and listen.

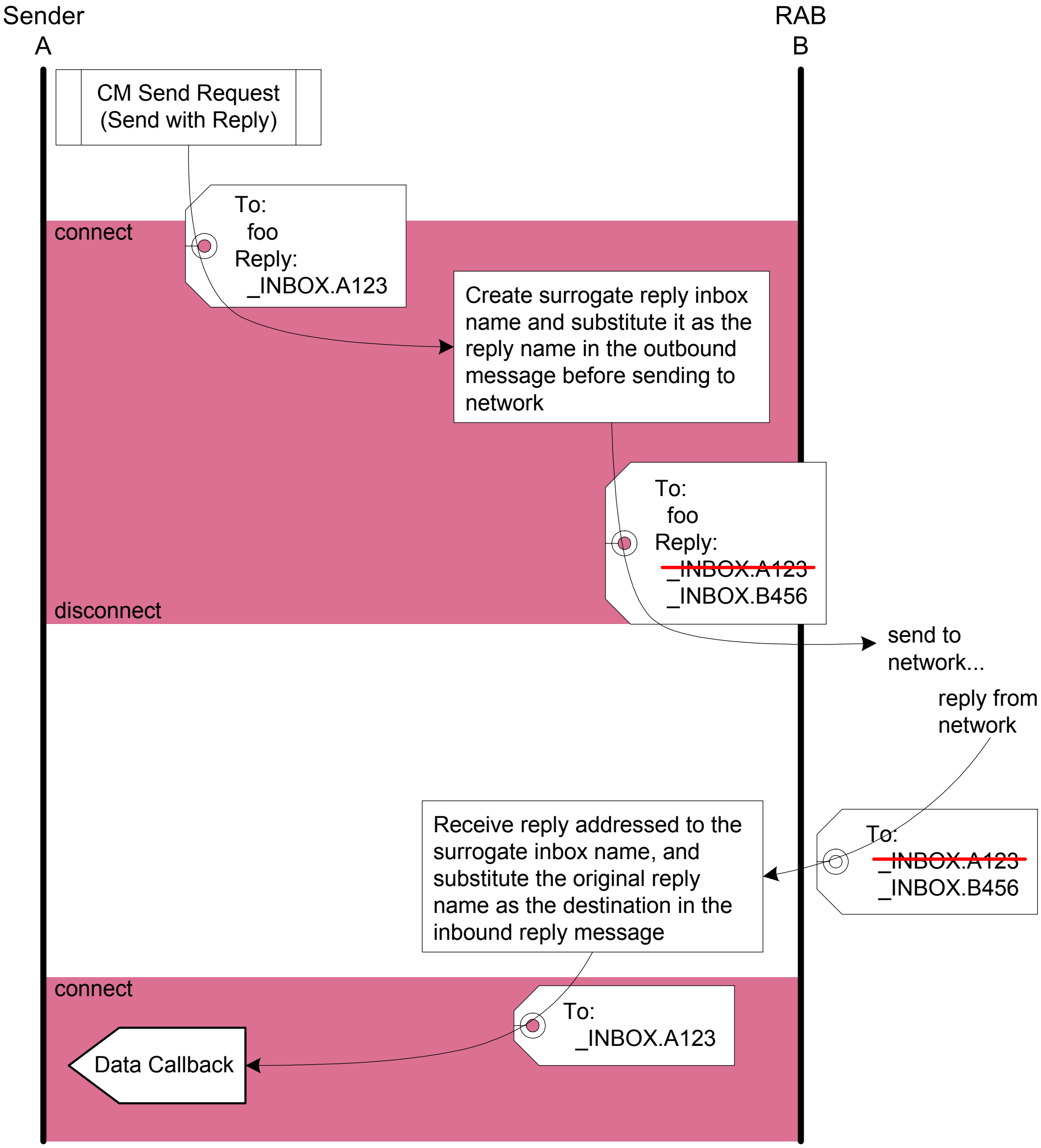
Reply Name Substitution

Consider the intermittently connected program (A) in Figure 17 on page 179. A sends a certified request message (for example, a query) and expects a reply message by listening to an inbox name (the reply can be certified or an ordinary message). When A disconnects, it becomes unavailable to receive the reply message.

To ensure that the reply arrives properly, the relay agent (RAB) establishes a surrogate inbox name, and substitutes that name as the reply name on the outbound message. When the reply arrives, RAB receives it, and substitutes the original reply inbox name before transferring the reply to A. Figure 17 on page 179 illustrates this sequence.

To see why reply name substitution is important, consider the result *without* substitution. Suppose instead that A places the reply inbox name in a field of the request message, and sends it without a proper reply name. In this erroneous situation, the relay agent cannot intercept the reply message—so if A is disconnected when the reply is sent, A does not receive the reply.

Figure 17 Relay Agents: Reply Name Substitution


Sender
A
RAB
B
CM Send Request
(Send with Reply)
connect
To:
foo
Reply:
_INBOX.A123
Create surrogate reply inbox name and substitute it as the reply name in the outbound message before sending to network
To:
foo
Reply:
~~_INBOX.A123~~
_INBOX.B456
disconnect
send to
network...
reply from
network
Receive reply addressed to the surrogate inbox name, and substitute the original reply name as the destination in the inbound reply message
To:
~~_INBOX.A123~~
_INBOX.B456
connect
To:
_INBOX.A123
Data Callback

Notice that if A destroys the original inbox listener event or invalidates its transport, then it can never receive the reply message from RAB.

Notice that when the reply name is a public subject (rather than an inbox), reply name substitution is unnecessary—since the relay agent can listen to the same reply subject as A. This arrangement works properly even when A terminates and restarts, or creates a new transport.

See Also For information about the relay agent process, see Relay Agent on page 249 in *TIBCO Rendezvous Administration*.

Chapter 12 Distributed Queue

A *distributed queue* is a group of CM transport objects, each in a separate process.

Programs can use distributed queues for *one-of-n* certified delivery to a group of worker processes.

Topics

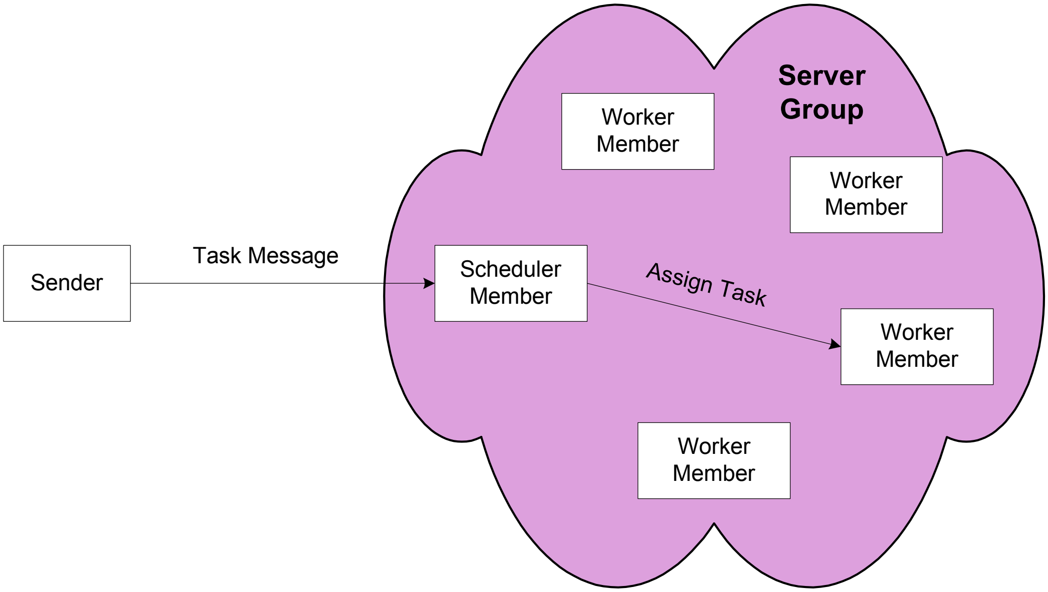
Distributed Queue Example

Programs can use distributed queues for *one-of-n* certified delivery to a group of servers, in order to balance the load among the servers.

This example illustrates a distributed group of database servers that accept certified messages representing tasks (updates and queries). Rendezvous distributed queue software assigns each task to exactly one of the servers, while the group of servers and the distribution of tasks remains completely transparent to the client processes.

Figure 18 illustrates a server group as a cloud of distributed queue members. From outside, the group appears to be a single transport object; inside the cloud, the members act in concert to process inbound task messages. A program outside the group sends a task message to the group; notice that the sender is not a group member, and does not do anything special to send its message to a group; rather, it sends its message to an ordinary subject name. Inside the group, the member acting as scheduler assigns each task message to exactly one of the workers; only that worker processes the task message.

Figure 18 One-of-N Delivery to a Distributed Queue



Distributed Queue Members

A distributed queue is a group of cooperating transport objects, each in a separate process; each transport object is called a member. From the outside, a distributed queue appears as though a single transport object; inside, the group members act in concert to process inbound task messages. Ordinary transports and CM transports can send task messages to the group; notice that the senders are not group members, and do not do anything special to send messages to a group; rather, they send messages to ordinary subject names. Inside the group, the member acting as scheduler assigns each task message to exactly one of the other members (which act as workers); only that worker processes the task message.

The members of a distributed queue all share the same reusable correspondent name, indicating that they are members of the distributed queue with that name.

Each member of a distributed queue must listen for the same subjects using CM listener objects. Yet even when *n* members listen for each inbound message (or task), only one member processes the message.

Certified Delivery Behavior in Queue Members

Group members support a limited subset of certified delivery calls—members can listen to a subject, override automatic confirmation of delivery, and confirm delivery. Member transports do *not* support calls associated with sending messages.

Distributed queues do not use ledger files. Group members automatically require old messages from certified senders.

Scheduler recovery and task rescheduling are available only when the task message is a certified message (that is, a certified delivery agreement is in effect between the task sender and the distributed queue transport scheduler).

Member Roles—Worker and Scheduler

Each distributed queue member has two distinct roles—as a worker, and as a potential scheduler.

In the *worker role*, members listen for task messages, and process inbound task messages as assigned by the scheduler.

Rendezvous fault tolerance software maintains exactly one active scheduler in each distributed queue; if the scheduler process terminates, another member assumes the role of scheduler. The member in the *scheduler role* assigns inbound tasks to workers. (A scheduler can assign tasks to its own worker component, but only does so when all other workers are busy.)

Enforcing Identical Subscriptions

It is important that all members of a distributed queue listen to the same set of subjects, and we recommend that programs enforce this rule among the distributed queue members. The easiest technique for enforcing identical subscriptions is to fix the subscription list within the program code (for example, as a constant).

If one member removes a subscription (that is, it destroys a listener), then we recommend that all the members also close that subscription. The easiest technique for enforcing this rule is to avoid removing subscriptions at all.

Fault Tolerance versus Distributed Queues

Fault tolerance usually requires that every member of a fault tolerance group receive each message. In contrast, each message to a distributed queue group is received by exactly one worker in the group. These mutually exclusive semantics cannot co-exist in the same distributed application program. That is, a program cannot simultaneously be a member of a fault tolerance group and a member of a distributed queue.

At a lower level, however, distributed queues automatically use fault tolerance software to elect and maintain a scheduler (see Scheduler Parameters on page 185).

Scheduler Parameters

Although any group member has the potential to become the scheduler, the software maintains exactly one scheduler at all times. Parameters guide the software to select the most suited member as scheduler.

Scheduler Weight

Scheduler weight represents the ability of a member to fulfill the role of scheduler, relative to other members of the same distributed queue. The group members use relative scheduler weight values to elect one member as the scheduler; members with higher scheduler weight take precedence. For further details, see Rank and Weight on page 204.

Heartbeat Interval

The active scheduler sends heartbeat messages at the interval you specify (in seconds). Heartbeat messages inform other members that a member is acting as the scheduler. All members of a group must specify the same scheduler heartbeat interval. To determine the correct value, see Step 4: Choose the Intervals on page 235.

Scheduler Activation Interval

In addition, all members of a group must specify the same scheduler activation interval. When the heartbeat signal from the scheduler has been silent for this interval (in seconds), the worker with the greatest scheduler weight takes its place as the new scheduler. To determine the correct value, see Step 4: Choose the Intervals on page 235.

Assigning Tasks to Workers

The scheduler assigns each inbound task to a worker. That worker alone processes the task message in a data callback function.

Worker Weight

Relative worker weights assist the scheduler in assigning tasks. When the scheduler receives a task, it assigns the task to the available worker with the greatest worker weight. The default worker weight is 1.

The scheduler applies a round-robin ordering to distribute tasks among several workers equivalent with equal weight.

Availability

When the scheduler receives a task, it assigns the task to an *available* worker with the greatest worker weight.

A worker is considered available unless either of these conditions are true:

- The pending tasks assigned to the worker exceed its task capacity.
- The worker is also the scheduler. (The scheduler assigns tasks to its own worker only when all other workers are busy.)

Task Capacity

Task capacity is the maximum number of tasks that a worker can accept. When the number of accepted tasks reaches this maximum, the worker cannot accept additional tasks until it completes one or more of them.

When the scheduler receives a task, it assigns the task to the worker with the greatest worker weight—unless the pending tasks assigned to that worker exceed its task capacity. When the preferred worker has too many tasks, the scheduler assigns the new inbound task to the worker with the next greatest worker weight.

The main task of a scheduler is to distribute tasks to workers; therefore a scheduler always set its task capacity to 1.

The default worker task capacity is 1. Programmers can tune task capacity based on two factors:

- Multi-tasking program on multiprocessing hardware.
- Communication time lag.

Tuning for Multiprocessing Hardware

Multiprocessing can raise the task capacity of a worker program.

On a multi-processing computer, a multi-threaded program that devotes *n* threads on *n* processors to inbound tasks can have task capacity *n*.

When programming a multi-threaded worker, ensure that the listener object that receives the tasks is set for explicit confirmation of certified messages, and that each thread explicitly confirms each task message when it finishes processing the task.

Tuning for Communication Time Lag

In most distributed queue applications, the communication time is an insignificant fraction of the task turnaround time. That is, the time required to *assign* a task and signal its completion is very small compared to the time required to *process* the task. For example, when average task turnaround time is 2 seconds, of which communication time contributes only 10 milliseconds to the total, then task capacity is the same as the number of processors or threads.

However, in some situations communication time can be significant—for example, when the group members are distributed at distant sites connected by a WAN. When communication time is significant, the meaning of task capacity changes; instead of signifying the number of tasks that a worker can process concurrently, it signifies the number of tasks that can fill the worker's capacity despite the communication time lag.

In most situations, a simple procedure computes a reasonable task capacity. For each worker, do these steps:

1. Measure the average round-trip communication time between scheduler and worker—that is, the time to send an assignment message and return a result message, without any task processing time intervening.
2. Measure the average task processing time, without any communication time.
3. Divide the average round-trip communication time by the average task processing time; round up to the nearest integer; add 1. The result is the theoretical task capacity that minimizes idle time for the worker.

For example, when the average round-trip time is 500 milliseconds, and the average task processing time is 1 second, then setting the task capacity to 2 minimizes the worker's idle time between tasks.

When tuning task capacity to compensate for communication time lag, balance is critical. Underloading a worker (by setting its tasks capacity too low) can cause the worker to remain idle while it waits for the scheduler to assign its next task. Conversely, overloading a worker (by setting its task capacity too high) can cause some assigned tasks to wait, while other workers that might have accepted those tasks remain idle. Tune performance by empirical testing.

Task Capacity

⚠ Tuning task capacity to compensate for communication time lag is more complicated than it might seem. For this purpose, use caution when setting task capacities greater than 1; task capacities greater than 3 are rarely correct. Instead of enhancing performance, incorrect settings can significantly degrade performance, and can even cause reassignment of tasks and duplicated work.

Complete Time

The complete time parameter of the scheduler affects the reassignment of tasks:

If the complete time is non-zero, the scheduler waits for a worker to complete an assigned task. If the complete time elapses before the scheduler receives completion from the worker, the scheduler reassigns the task to another worker.

Zero is a special value, which specifies no limit on the completion time—that is, the scheduler does not set a timer, and does not reassign tasks when task completion is lacking. All members implicitly begin with a default complete time, which is zero; programs can change this parameter.

See Also Case Studies—Complete Time, page 192.

Reassigning Tasks in Exceptional Situations

Under normal operating conditions, distributed queue software arranges for exactly one worker to process each task. This section describes three *exceptional conditions* that require different semantics:

- A worker exits or loses network communication before completing an assigned task. The scheduler reassigns the task to another worker.
- A worker processes tasks more slowly than expected. The scheduler uses its complete time parameter to determine whether to reassign the task to another worker. Duplicate processing can occur.
- Scheduler replacement—the scheduler exits or loses network communication, so another member replaces it as the active scheduler. The new scheduler reassigns incomplete tasks, guided by its complete time parameter. Duplicate processing can occur.

Two factors can affect behavior in exceptional situations:

- When the sender and scheduler have a certified delivery agreement, behavior differs from when they do not.
- Behavior differs depending on the scheduler's complete time parameter.

Worker Exit

When a worker exits or loses network communication, the scheduler detects its absence and reassigns all of that worker's incomplete tasks to other workers.

This behavior applies when the task source (sender) and the scheduler have a certified delivery agreement. This behavior applies for any setting of the scheduler's complete time parameter.

Slow Worker

When a worker processes tasks more slowly than expected, the scheduler detects slow operation using timers controlled by the scheduler's complete time parameter.

Complete Time = 0 Scheduler reassigns a task when the assigned worker does not accept it. Once a worker accepts, the scheduler waits indefinitely for completion.

Potential non-completion of tasks.

Complete Time > 0 Scheduler reassigns a task when the assigned worker does not accept it, or does not complete it in time.

Potential duplication of tasks.

This behavior applies *only* when the sender and scheduler have a certified delivery agreement. When no certified delivery agreement is in effect, the scheduler does not reassign tasks based on delayed completion.

The main concern in this situation is to ensure that all certified tasks complete in a timely fashion. When a slow worker hinders this goal, then the scheduler reassigns its task—selecting speed over unduplicated processing. However, if the complete time parameter is zero, then the scheduler selects unduplicated processing over speed (but see also, Scheduler Replacement).

Scheduler Replacement

When the scheduler exits or loses network communication, another member replaces it as the active scheduler.

Complete Time = 0 The new scheduler immediately reassigns all incomplete tasks.

Complete Time > 0 The new scheduler immediately reassigns all unaccepted tasks. It also sets a timer to elapse after the complete time; when the timer expires, the new scheduler reassigns all incomplete tasks.

This case presents the lowest probability of task duplication, but duplication is still possible for slow workers.

This behavior applies *only* when the sender and scheduler have a certified delivery agreement. When no certified delivery agreement is in effect, the new scheduler does not reassign tasks.

The main concern in this situation is to ensure that all certified tasks complete *at least once* (that is, no certified task remains unprocessed). The new scheduler reassigns all tasks that are at risk (for example, the assigned worker might have exited during scheduler replacement). In achieving this goal, the probability of duplicate processing is high. However, the lowest probability of duplicates is the case in which the complete time parameter is non-zero; in this case, the scheduler uses the extra information to reduce duplication.

Case Studies—Complete Time

The appropriate value for complete time depends upon the goals of the program, and on its operating environment. These case studies illustrate the criteria for selecting the value for this parameter.

Mandelbrot Set

Because it generates beautiful displays, the Mandelbrot set is a popular visual symbol of fractal phenomena. Consider an example application that computes a Mandelbrot display, using a distributed queue of servers to compute the display data.

A display component divides the display region into small chunks (tasks), and sends each chunk to the server group for concurrent processing. Within the group, the scheduler assigns each chunk to an available worker. The worker computes the data for its chunk, and sends the results to the display component, which reassembles and displays them.

Mandelbrot computations are characterized by many small chunks of processing, each of which completes in a short time—estimate under 0.1 seconds. Estimate the network travel time for each task at a few milliseconds. Duplicate processing does no harm (other than wasted effort). The main priority is fast response, since the user is waiting for the display.

For programs with these criteria, consider a complete time of 0.5 seconds. Any task that is still incomplete after 0.5 seconds is reassigned, so the display user perceives only minimal delay, even in exceptional situations. Yet 0.5 seconds is sufficiently high to prevent thrashing in the event of several slow workers; use empirical data to fine tune this parameter.

Fax Confirmation

Consider a business that relies on telephone transactions between customers and service representatives. At the conclusion of each call, the customer receives a one-page summary and confirmation by fax.

A large number of service representatives all send certified tasks to a distributed queue of fax servers. When a fax server receives a task, it transmits the fax and notes the result in the customer’s database record.

Each fax task could take several minutes—including redials and other delays. Estimate the network travel time for each task at a few milliseconds. Speed is not critical; the customer is satisfied as long as the fax arrives within 10 minutes. The highest priority is that each task completes (that is, no task is lost). However, duplicated tasks are undesirable—customers prefer to receive only one fax summary.

For programs with these criteria, consider a complete time of 300 seconds (5 minutes). If a task remains incomplete for 5 minutes, the scheduler reassigns the task to another server.

Distributed Queues and Certified Listener Advisory Messages

Table 42 on page 270 lists the certified delivery and distributed queue advisory messages, and indicates which ones can be received by listening correspondents. Members of a distributed queue act in concert as a single listening correspondent; when one member of a distributed queue receives a listener advisory, all the members receive it. (`QUEUE.SCHEDULER.OVERFLOW` is an exception to this rule; only the scheduler receives it.)

Chapter 13 Fault Tolerance Concepts

This chapter explains the conceptual foundations of Rendezvous fault tolerance software, the way it works, and ways to use it.

Topics

See Also

- Chapter 14, Fault Tolerance Programming, on page 213
- Chapter 15, Developing Fault-Tolerant Programs, on page 227

Fault Tolerance

In nearly every enterprise, mission-critical programs must continue to function properly despite sudden difficulties such as process termination, hardware failure and network disconnect. *Fault tolerance* in a network environment is characterized by rapid recovery from such failures.

Some fault-tolerant distributed programs keep service interruptions to a minimum by using redundant processes that cooperate across the network. Rendezvous fault tolerance software facilitates the development of distributed programs that use redundant processes for fault tolerance.

Rendezvous fault tolerance software helps your program achieve fault tolerance by coordinating a group of redundant processes. Some processes actively fulfill the tasks of the program, while other processes wait in readiness. When one of the active processes fails, another process rapidly assumes active duty.

Rendezvous fault tolerance software supports any number of cooperating processes connected by a local or wide-area network. Rendezvous fault tolerance software monitors the health of cooperating processes, determines when a key process is no longer in service, and instructs another process to take its place.

Rendezvous fault tolerance software is fast, compact, and adds little overhead to programs.

You can use Rendezvous fault tolerance software to design fault-tolerant behavior into programs from the start, or to retrofit existing programs for fault tolerance.

Programs can passively monitor the number of active members in a fault-tolerance group (whether or not the monitoring program is itself fault-tolerant).

Fault Tolerance versus Distributed Queues

Fault tolerance usually requires that every member of a fault tolerance group receive each message. In contrast, each message to a distributed queue group is received by exactly one worker in the group. These mutually exclusive semantics cannot co-exist in the same distributed application program. That is, a program cannot simultaneously be a member of a fault tolerance group and a member of a distributed queue.

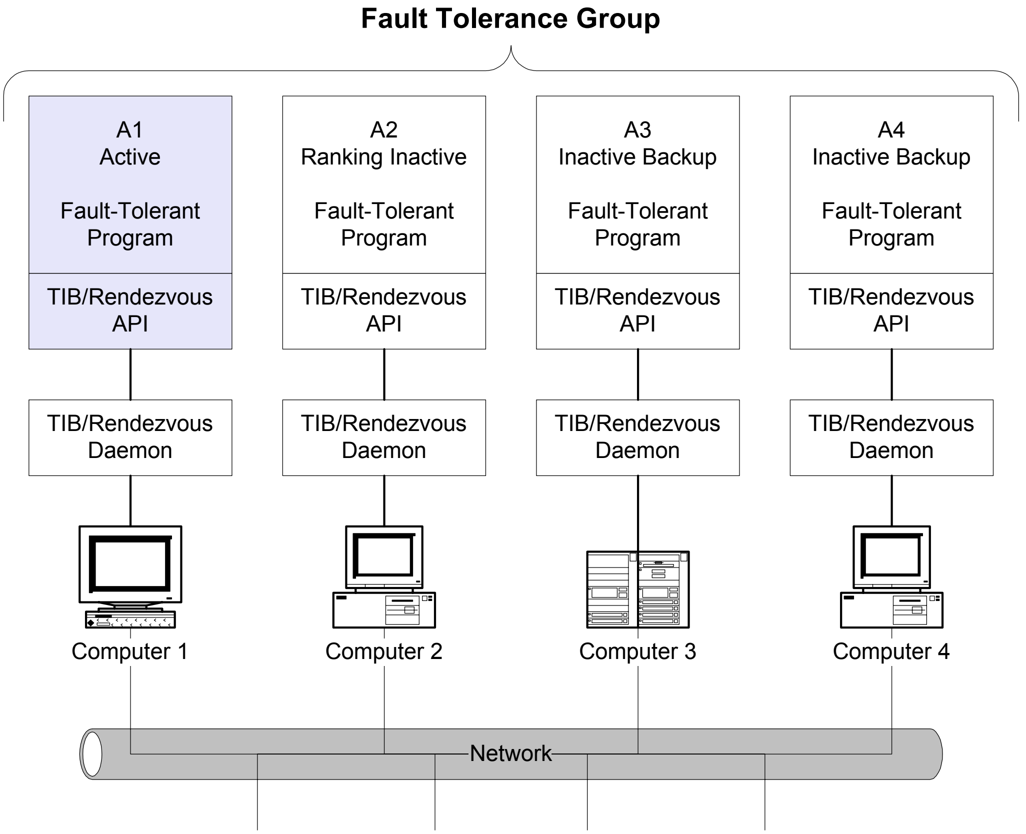
Fault Tolerance in Action

Rendezvous fault tolerance software uses Rendezvous software to communicate between processes.

Components and Operating Environment

Figure 19 on page 197 illustrates the components of a fault-tolerant distributed program in a typical network operating environment. Four identical program processes (A1, A2, A3, A4) run on four separate computers, connected by a network. The program incorporates the Rendezvous API library (including fault tolerance and communications software); a Rendezvous daemon process mediates between each program process and the network.

Figure 19   Fault Tolerance Operating Environment



Example Fault-Tolerant Multicast Producer

In one scenario, a program produces a stream of multicast messages to the network (for example, stock market prices, news stories). Other programs on the network listen to those messages and consume the information.

Only one of the four producer processes (A1) actively sends information. The other three are backup processes; they each compute an information stream that is parallel to that of the active process, but they do not send the information. If the active process stops functioning, Rendezvous fault tolerance software directs one of the three backup processes to begin active multicasting in its place. In this way the redundant processes cooperate to provide a fault-tolerant service without flooding the network with redundant information.

Advantages of Rendezvous Fault Tolerance Software

Rendezvous fault tolerance software builds on the proven strengths of Rendezvous communications software.

- Location Independence

  Programs that use Rendezvous fault tolerance software are not bound to specific computers, network addresses, nor even to a specific network. TIBCO's subject-based addressing technology gives administrators deployment flexibility.

- Scalability

  Rendezvous fault tolerance software supports any number of active processes—one, two, twenty, or twenty thousand—up to the intrinsic limits of the network and host computers.

  Programs can scale smoothly to any degree of redundancy—from one backup process to hundreds—without additional programming effort.

- Network Non-Interference

  Several different fault-tolerant programs can share the same network without interference.

- Convenience

  Rendezvous fault tolerance API is a decision-making tool. It organizes and directs programs for fault-tolerant backup behavior. It simplifies development and use of fault-tolerant programs.

  Like all other Rendezvous software, it is compact and easy to use.

  You can develop new fault-tolerant programs or retrofit existing programs for fault-tolerant operation.

Groups and Membership

A *fault tolerance group* is a set of program processes that cooperate for fault-tolerant service. Each *member* of a fault tolerance group is an event object instance, in a process running on a computer attached to a network; by extension, we also say that the process is a *member* of the fault tolerance group. A fault tolerance group can have any number of members.

New processes can *join* the group (become members) at any time by creating a fault tolerance member object. Each process that joins a group remains a member until it either withdraws or terminates. Member processes can *withdraw* from the group (cease to be members) at any time by destroying their fault tolerance member object.

Many fault tolerance groups can coexist on the same network. Distinct group names prevent interference. A process can be a member of more than one fault tolerance group (by creating several fault tolerance member objects, one for each group).

Group Name

Each fault tolerance group requires a unique group name. All members of a group must share the same group name. The group name identifies the members of the group, and labels the messages they exchange with one another.

For information about the syntax of group names, and practical advice on selecting a group name, see Step 1: Choose a Group Name on page 228.

Active and Inactive

At any moment in time, each member of a fault tolerance group is either active or inactive. An *active* member directly fulfills the program's mission—for example, broadcasting information, responding to queries, or filling requests. An *inactive* member provides backup capacity; inactive members wait in the background, ready to become active immediately if an active member stops functioning.

When a new member joins a group, it is initially inactive.

Each member incorporates Rendezvous fault tolerance software, which operates behind the scenes. Rendezvous fault tolerance software maintains the correct number of active members by issuing instructions, called *actions*, to group members. Each action instructs a member to activate, deactivate, or prepare to activate. (For details, see Fault Tolerance Callback Actions on page 214.)

Rendezvous fault tolerance *monitor* software can passively monitor the members of a group, so other programs can determine the number of active members without actually joining the group. For a description, see Passive Monitor on page 211.

Alternate Terminology

Some approaches to fault tolerance use different terms to describe active and inactive members. In particular, an active member is called *primary*, while inactive members are called *secondary*.

This terminology implies that only one member can be active—the primary. However, Rendezvous fault tolerance software allows any number of active members. This book uses the more general terms, active and inactive.

Fault Tolerance Callback Function

When a process joins a fault tolerance group, it must specify a *fault tolerance callback function* as a parameter. The callback function must be defined by the program. When Rendezvous fault tolerance software detects any change that requires action by the program, it queues the callback function. It is not necessary that all members of a group have the same fault tolerance callback function.

Among its arguments, the callback function receives a token denoting an action—in this way Rendezvous fault tolerance software instructs the program how to behave in order to assure fault-tolerant operation. The callback function *must* comply with the action instruction by taking whatever steps are needed.

The action token may instruct an inactive member to activate; or it may instruct an active member to deactivate; or it may instruct an inactive member to prepare to activate (hinting that an instruction to activate *might* soon arrive).

For more information, see Fault Tolerance Callback Actions on page 214, and Program Callback Functions on page 215.

For information about callback closure arguments, see Closure Data on page 89.

Active Goal

You can design fault tolerance groups with any number of active members. That number is called the *active goal*.

When a member joins a group, it specifies the active goal as a parameter. Rendezvous fault tolerance software maintains that goal, regulating which members are active by issuing instructions to activate and deactivate. When too few members are active, it instructs inactive members to activate; when too many members are active, it instructs active members to deactivate.

Every member of a group must specify the same value for the active goal parameter. It is an error for members of the same group to specify different values for this parameter. When Rendezvous fault tolerance software detects values that do not agree, it delivers an error advisory message to each member of the group (see `PARAM_MISMATCH` on page 297).

Example: One Active Member

Broadcast producer programs generally function properly with only one member active at any moment in time; all other members are inactive backup processes. Example Fault-Tolerant Multicast Producer on page 198 illustrates this kind of program. If the number of active members drops below one, then no information flows to consumers. With more than one active member, duplicate messages flood the network.

As long as one or more member processes exist, Rendezvous fault tolerance software maintains a single active member.

Example: Several Active Members

Consider a data-mirroring program that stores data in several different locations; each process instance of the program stores the data on its host computer. It does not matter which of many possible locations store the copies, as long as the network always has at least 5 complete copies. Such a program satisfies its fault tolerance requirement by setting the active goal to 5.

Rank and Weight

Rendezvous fault tolerance software sorts the members of a group, assigning each member a unique *rank*. Rank determines which members are active.

A member with rank *n* takes precedence over a member with rank *n+1*. In this sense, *n* represents a *higher* rank than *n+1*.

If the active goal of a group is *n*, then the members with rank *1* through *n* are active. The member with rank *n+1* is known as the *ranking inactive member*. If one of those active members fails, then Rendezvous fault tolerance software instructs the ranking inactive member to activate.

The most important factor in assigning rank is *weight*. When a process joins a fault tolerance group, it specifies its weight as a parameter. Weight represents the ability of a member to fulfill its function—relative to other members of the same group.

To rank the members of a group, Rendezvous fault tolerance software sorts the members by weight. The member with the highest weight receives rank 1 (so it outranks all other members); the member with the next highest weight receives rank 2; and so on. When two or more members have the same weight, Rendezvous fault tolerance software ranks them in way that is opaque to programs.

Weight Values

Each member specifies its weight as a positive integer.

Zero is a special, reserved value; Rendezvous fault tolerance software assigns zero weight to processes with resource errors, so they activate only as a last resort when no other members are available. Programs must always assign weights greater than zero.

(For further details, see Disabling a Member on page 224, and `DISABLING_MEMBER` on page 299.)

Assigning Weight

Weight lets you influence the ranking of member processes based on external knowledge of the operating environment. Assign weight after considering properties such as hardware speed, hardware reliability, and load factors.

For example, if member A runs on a computer that is much faster than member B, then assign higher weight to A than B. Greater weight expresses your opinion that A fulfills its task more effectively than B. As a result, A is ranked before B, and takes precedence.

If members C, D and E all run on equally fast computers with approximately equal load factors, then assign them equal weight. Equal weight expresses no preference for any process over the others. Rendezvous fault tolerance software ranks them in a way that is opaque to programs.

Rank among Members with Different Weight

Members of greater weight always outrank members of lower weight. For example, if member A has weight 200, and member B has weight 100, then A always outranks B.

Inactive members of greater weight preempt active members of lower weight. For example, if B (weight 100) is already active when A (weight 200) starts, then Rendezvous fault tolerance software instructs B to deactivate, and instructs A to activate in its place.

Ranking Members with Equal Weight

If members C and D have equal weight, their relative rank is opaque to programmers. That is, their relative rank does *not necessarily* depend on the order in which two processes start. Consider these (possibly surprising) consequences:

- If member process C starts before member D, you cannot deduce from this order that C outranks D. Nor can you deduce the reverse, that D outranks C.
- If the active goal for the group is 1, and C starts first, and D starts immediately after C—then you cannot assume that either process will be the first to become the active member.

Status Quo among Members with Equal Weight

A ranking inactive member never preempts an active member with the same weight—despite its precedence in rank.

For example, if members E and F have equal weight, with E outranking F, and F already active, then E does not preempt F to become active in its place.

Contrast that example with a situation in which neither E nor F is active, and a new active member is needed to complete the active goal—in this case E activates (rather than F) because E outranks F.

Adjusting Weight

In addition to specifying weight when a process joins a fault tolerance group, sophisticated programs can adjust their weight at any time to reflect changing conditions.

For example, a member might track the changing load factor of its host computer, and adjust its weight accordingly. Rendezvous fault tolerance software automatically recomputes the ranking of members whenever a member changes its weight.

Adjusting weights causes each member to recompute their relative weights of all the members of the group. For large groups this recomputation can affect performance.

For examples, see Adjusting Member Weights on page 225.

Heartbeats

Each active member of a fault tolerance group broadcasts a *heartbeat signal* to the other group members. The heartbeat signal is a regular, periodic stream of *heartbeat messages*. The heartbeat signal indicates that the member is still active.

When a process joins a fault tolerance group, it specifies its *heartbeat interval* as a parameter. This interval governs a repeating timer; each time the heartbeat interval elapses, Rendezvous fault tolerance software publishes a heartbeat message.

To determine the correct value for the heartbeat interval, see Step 4: Choose the Intervals on page 235.

Detecting Member Failure

Members can fail for several reasons (this list is not exhaustive):

- Process termination.
- Process suspension (for example, UNIX Control-Z).
- Software errors.
- Hardware failure.
- Network disconnect.

Rendezvous fault tolerance software does not distinguish between these failures. In each case, the failed member cannot fulfill its mission (or can fulfill it only locally), and another member must take its place.

Rendezvous fault tolerance software detects failure of an active member in two ways—heartbeat tracking and independent confirmation.

Heartbeat Tracking

The inactive members of a group listen for heartbeat messages from all of the active members. A steady stream of heartbeat messages is an important indicator of process health. While the heartbeat continues, no action is needed. If the heartbeat messages from an active member cease to arrive, then Rendezvous fault tolerance software considers that member to be *lost*, and instructs the ranking inactive member to activate, replacing the lost member.

Independent Confirmation

Rendezvous fault tolerance software also detects a set of events that indicate the loss of an active member. Consider these example events:

- An active member withdraws from the fault tolerance group.
- An active member process terminates, or disconnects from its Rendezvous daemon.
- A Rendezvous daemon process terminates; an active member that relies on that daemon is unable to function.
- A network hardware failure separates the network into two or more disconnected parts.

When Rendezvous fault tolerance software detects such events, it restores the active goal by directing member processes to activate.

Activation Interval

Inactive members track the heartbeats of active members, detecting the loss of an active member when its heartbeat signal is silent for a duration called the *activation interval*. (The name *activation interval* refers to the use of this parameter as the time that an inactive member waits before becoming active when a heartbeat signal is lost.)

When a process joins a fault tolerance group, it specifies the activation interval as a parameter. All members of a group must specify the same activation interval. To determine the correct value, see Step 4: Choose the Intervals on page 235.

Prepare-to-Activate Hints

Before becoming active, some programs might need to do one or more time-consuming steps, such as opening an ISDN line or opening a database connection. Such programs need time to prepare before they can activate. They can request that Rendezvous fault tolerance software issue a *prepare-to-activate* hint—an early warning that an activate instruction might soon arrive.

Requesting Hints

When a process joins a fault tolerance group, it may specify a *preparation interval* as a parameter.

- A zero value indicates that the program does not need advance warning before it can activate. Rendezvous fault tolerance software does not issue prepare-to-activate hints.
- Any non-zero value is a request for advance warning. Rendezvous fault tolerance software issues a prepare-to-activate hint before an instruction to activate.

Timing of Hints

When a member requests advance warning, Rendezvous fault tolerance software *always* issues a prepare-to-activate hint before each instruction to activate. The ranking inactive member always receives them in the correct order, but cannot rely on the time between them. The intervening time may equal the difference between the activation interval and the preparation interval, or it may be less; however, it is never greater than this difference.

Hints Do Not Imply Subsequent Activation

A prepare-to-activate hint is just that—a hint. This hint does not necessarily imply that the member will definitely need to activate. It is possible for a member to receive several such hints without an instruction to activate.

Passive Monitor

A program can passively track the number of active members of a group using a fault tolerance *monitor*. That program need not be a member of the fault tolerance group it monitors.

Monitors are *passive* in that they do not affect the members of the monitored group in any way. Members do not detect that a monitor exists.

Programs that passively monitor a group detect the same number of active members as do members of the group they monitor.

Monitor Callback Function

When a program starts monitoring (by creating a monitor event), it must specify a *monitor callback function* as a parameter. The callback function must be defined by the program. When Rendezvous fault tolerance software detects any change in the number of active members, it calls the callback function—which receives the number of active members as an argument.

Monitors in Action

Monitors give Rendezvous programs limited capability to determine the health of the fault-tolerant programs upon which they depend. In the most common scenario, a client program monitors a critical service, and adapts its own behavior accordingly. Consider these examples.

Example: Monitor for Data Quality

A data display program receives many items of time-critical information from several groups of fault-tolerant broadcast producers (one group for each kind of information). The display updates every time new information arrives. The end user must be confident that the displayed information is current. The display program monitors the health of each producer group, and displays each information item with a color code indicating the quality of the information (based on the health of the corresponding producer).

For example, if a producer group has an active member (normal operation), then all information from that producer appears on a white background. If the producer has no active member (a catastrophic failure), then the display marks all information from that producer with a yellow background, to signal the end user that the information might be obsolete. When the producer group once again has an active member, the display changes the background of each new item to white, to show that it represents current information.

Example: Monitor for Available Service

A group of query servers responds to requests from numerous client programs. Before submitting a query, a client program checks the number of active servers. If no servers are active, the client program informs the end user that it cannot submit the query. If many servers are active, the client submits the query. If only a few servers are active, the client submits the query, and informs the end user that the response may be delayed.

Example: Monitor to Ascertain Complete Response

Some programs use redundancy to cross-check results. Each member in a fault tolerance group computes the same information—but each uses a different program, coded by a different programming team, running on a different kind of computer hardware platform. A client program receives, collates and compares the results of their computations, and reports to an end user.

The collator must report as soon as it receives a response from all the active members; it must not delay while waiting for a response from a member that has terminated unexpectedly. The collator monitors the group to determine the number of active members. When the number of responses equals the number of active members, the collator reports the combined results.

Chapter 14 Fault Tolerance Programming

This chapter describes the practical issues that arise when developing and deploying fault-tolerant distributed programs.

Topics

See Also For details of programming in specific languages, see the documents in Table 35.

Table 35 Fault Tolerance

C	Fault Tolerance on page 219 in *TIBCO Rendezvous C Reference*
C++	Fault Tolerance on page 243 in *TIBCO Rendezvous C++ Reference*
Java	Fault Tolerance on page 217 in *TIBCO Rendezvous Java Reference*
COM	Fault Tolerance on page 187 in *TIBCO Rendezvous COM Reference*
.NET	Fault Tolerance on page 133 in *TIBCO Rendezvous .NET Reference*

Fault Tolerance Callback Actions

Fault tolerance callback functions receive an `action` argument—one of three tokens, which instruct the callback function to behave in one of three ways:

- `ACTIVATE`

  This action token instructs the callback function to switch the member process into active mode.
- `DEACTIVATE`

  This action token instructs the callback function to switch the member process into inactive mode.
- `PREPARE_TO_ACTIVATE`

  This action token is a hint that Rendezvous fault tolerance software might soon issue an instruction to activate. It instructs the callback function to prepare for possible activation by doing any time-consuming steps that can be done before the actual order to activate.

  Remember that the prepare-to-activate hint is exactly that—a hint. Several circumstances might later render activation unnecessary.

The names of the tokens vary slightly among the different programming languages.

Table 36 Fault Tolerance Actions

C	`tibrvftAction`
C++	`tibrvftAction`
Java	Tokens are defined as constant fields of `TibrvFtMember`.
COM	Tokens are defined as in the interface definition file.
.NET	`ActionToken` enumerates the tokens.

Program Callback Functions

Fault tolerance callback functions must do the required action, as specified by the action token.

The basic structure of a fault tolerance callback function is a C `switch` statement that branches on the action token. The specific actions of the callback function within the `case` clauses of that `switch` statement depend on application semantics. (In other programming languages, use analogous constructs.)

Table 37 Fault Tolerance Callback Function

C	`tibrvftMemberCallback`
C++	`TibrvFtMemberCallback::onFtAction()`
Java	`TibrvFtMemberCallback.onFtAction()`
COM	`TibrvFtMember_onFtAction`
.NET	`ActionTokenReceivedEventHandler`

See Also Fault Tolerance Callback Actions, page 214
Ensure Timely Event Processing, page 216

Ensure Timely Event Processing

The fault tolerance callback function is the only channel of information from Rendezvous fault tolerance software to your program. Rendezvous fault tolerance software can call the callback function at virtually any time—whether active or inactive, a member program must be ready for a callback event.

To ensure timely processing of fault tolerance events, follow these guidelines:

- Associate fault tolerance member events with a high priority queue, so events on other queues do not delay fault tolerance callback processing.
- Ensure that callback functions return promptly. This rule applies to *all* callback functions that dispatch in the same thread as the fault tolerance callback function.

If a callback function monopolizes the dispatch thread (for example, by blocking, or with a lengthy computation), then the fault tolerance callback function might remain in its queue waiting for dispatch. Such backlog could cause problems such as these:

- Delay a program from activating, resulting in interrupted service.
- Delay a program from deactivating, resulting in redundant service.
- Delay a timer, or the arrival of a fault tolerance control message, interfering with fault tolerance software.
- Delay the arrival of regular messages, interfering with the program performance.

Multiple Groups

In some situations a process joins more than one fault tolerance group. Each group protects a specific role that the process plays within a larger distributed application system.

Example: Mutual Backup across a WAN

Figure 20 on page 218 illustrates a situation with two levels of fault tolerance. Network sites in Tokyo and Seattle are connected by a WAN link, and Rendezvous routing daemons forward messages on demand between the two sites.

At each site, a pair of computation servers listens for client requests, processes each request, and sends the results to the client.

Local Fault Tolerance Coverage

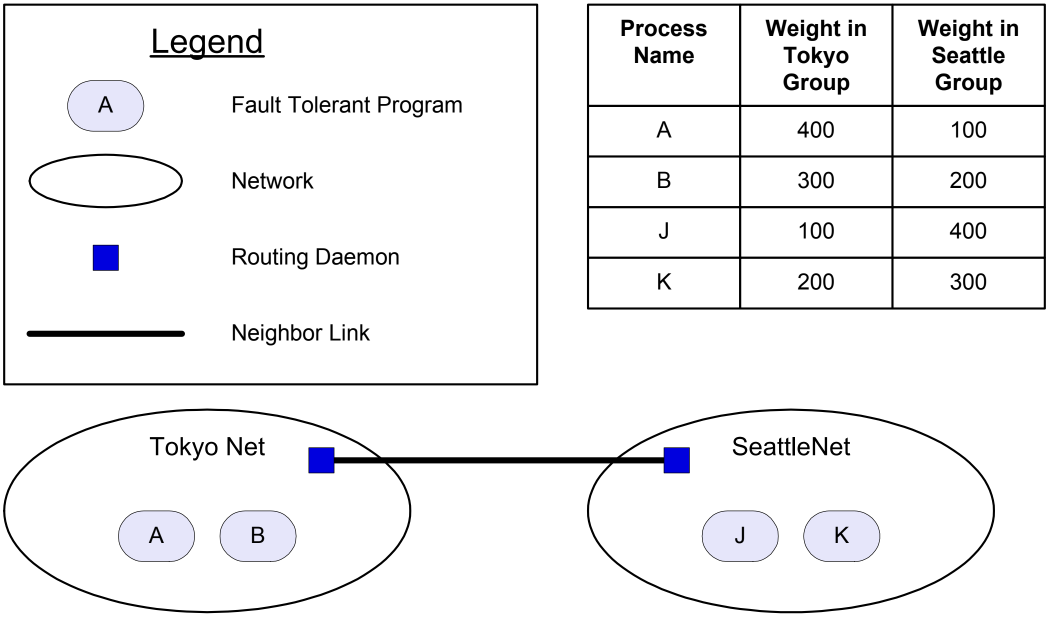
The volume of requests is low, and one process can accommodate them. However, the query service is critical to the enterprise, so each site runs two process instances, which cooperate for fault tolerance. In Tokyo the processes are A and B; in Seattle, J and K.

The active Tokyo process listens for requests that carry the subject name `TOKYO.REQUEST`. The active Seattle process listens for requests that carry the subject name `SEATTLE.REQUEST`.

To administer fault tolerance at the Tokyo site, processes A and B join a fault tolerant group named `TOKYO.APP1`. A has higher weight than B, so A is initially active. The group’s active goal is one, so only one member of the group actively processes requests. Similarly, processes J and K in Seattle join a group named `SEATTLE.APP1`. J has higher weight than K, so J is initially active.

If the active member at either site fails, the inactive member at the same site takes its place.

Figure 20 Mutual Backup across a WAN



Process Name	Weight in Tokyo Group	Weight in Seattle Group
A	400	100
B	300	200
J	100	400
K	200	300

Long-Distance Fault Tolerance Coverage

Although unlikely, it is distinctly possible that both request servers at a site might fail simultaneously. If the WAN link is still operative, the Seattle site can serve as a backup for the Tokyo site, and vice versa.

For long-distance fault tolerance coverage, Seattle processes J and K join the Tokyo fault tolerant group, `TOKYO.APP1`; and Tokyo processes A and B join the Seattle fault tolerant group, `SEATTLE.APP1`. The table in Figure 20 lists the relative weights of all four processes in each of the two fault tolerance groups. Notice that within each group, local members have higher weight than distant members, so a distant member activates only when both local members fail or withdraw from the group.

If both Tokyo processes A and B fail, Seattle process K takes their place. When K receives a prepare-to-activate hint, it begins listening to the subject `TOKYO.REQUEST`. When the Rendezvous routing daemon detects the new listening interest, it begins forwarding the messages with subject `TOKYO.REQUEST` from Tokyo to Seattle, where K receives them. When Rendezvous fault tolerance software instructs K to activate, K begins processing those request messages, sending the results back to clients in Tokyo.

To enable this example, the routing daemons on each side of the WAN link must exchange all messages with subjects that match `_RVFT.>`. For details, see Forward Fault Tolerance Messages across Network Boundaries on page 377 in *TIBCO Rendezvous Administration*.

Longest Service Interruption

In most situations, the longest service interruption is no longer than the activation interval.

Cascading Failure Situations

Cascading failure is an unusual situation in which several members fail in succession—either as they activate, or toward the end of their activation interval. As a result, several activation intervals may elapse before service is restored (or the supply of inactive members is exhausted).

Cascading failure can result from fatal program errors during activation, or from a rare coincidence of unrelated failures.

New Member Situations

When a new member joins a group, Rendezvous fault tolerance software identifies the new member to existing members (if any), and then waits for a period of one activation interval to receive identification from them in return. If, at the end of this interval, it determines that too few members are active, then it instructs the new member to activate.

Minimizing Response Time

You can minimize the response time in failure situations by separating fault tolerance messages from other Rendezvous messages. Two kinds of separation affect response time:

- Queue and dispatch.

  Use a separate queue object only for fault tolerance messages. Assign that queue high priority within its group queue. Ensure that the thread that dispatches that queue group does not delay or block.

  This strategy prevents heartbeat messages from waiting behind other events in the queue.

- Service.

  Use a separate Rendezvous service (UDP or PGM port) for fault tolerance messages.

  Designating separate services on the network separates fault tolerance messages from competing messages (from your program or other programs). An uncluttered service results in faster handling of fault tolerance messages by the Rendezvous daemons.

  A separate service implies a separate transport.

  Before you specify a service, always consult with your system administrator. Your system administrator assigns specific network services for specific purposes. Explain that you need a clear UDP or PGM port, free from any other messages.

See Also Service Selection on page 20 in *TIBCO Rendezvous Administration*

Distribute Members

When you deploy a fault-tolerant program, it is important to distribute the member processes appropriately across both computers and network segments. Independence increases the effectiveness of redundant processes.

Protect against Hardware Failure

To protect a fault-tolerant program against hardware-related failures, each member process must run on a separate computer. If two members were to run on the same computer, then both processes would be vulnerable to exactly the same failures; a single disconnected power cord or loose network cable would disable both processes simultaneously.

We recommend redundancy of special hardware. For example, if the program relies on a data feed line, install one line on each computer that runs a member process. If the program relies on a special board or card, install that hardware on each computer that runs a member process.

Protect against Network Disconnect

If the program serves a network consisting of several segments, distribute member processes appropriately across all the segments. To protect against disconnect from the rest of the network, run at least one member on each segment.

We recommend redundant copies of software and data files. If all the members of a group depend on network access to a single data file, then network disconnect would disable all the processes that can no longer access that file.

Member File Access

When members of a fault tolerance group depend on file access, it is crucial that each member use a private copy of every relevant file. In particular, consider these two guidelines.

- Do not share files.

  Ensure that two members never modify the same file. Sharing a file can corrupt the data in that file (even when the members do not write to it simultaneously). Using local files prevents this kind of corruption.

- Store files on the local host computer, not across the network on a file server—and especially not on a separate network or segment.

  In situations where network disconnect causes a member to activate, that same disconnect can separate the activating backup member from network-mounted files. Using local files keeps the data where it is needed.

Copying Context Files

In some programs, an activating member must establish its operating context by reading files left by a previously active member that has failed. When using this technique, be sure to make a local, private copy of the file.

When the content of the context files changes frequently, inactive members can periodically copy the context files, so that a network disconnect does not preclude access when it is needed.

Upgrading Versions

When upgrading a program to a new version, it is sometimes important that old and new versions run simultaneously. Be sure that the two process instances do not reference the same file. Instead, make a local copy of the file for the new version to use.

Disabling a Member

Rendezvous fault tolerance software routinely uses resources such as storage and timers. Resource allocation errors prevent it from functioning correctly.

When Rendezvous fault tolerance software requests a resource but receives an error (for example, the member process cannot allocate memory, or start a timer), it attempts to send the member process a DISABLING_MEMBER advisory message, and sets the member’s weight to zero, effectively disabling the member. Weight zero implies that this member is active only as a last resort—when no other members outrank it.

Rendezvous software never resets the weight of a member to anything other than zero.

Adjusting Member Weights

Member processes can change their weight using the set weight call.

Table 38 Set Weight Call

C	`tibrvftMember_SetWeight()`
C++	`TibrvFtMember::setWeight()`
Java	`TibrvFtMember.setWeight()`
COM	`TibrvFtMember.setWeight`
.NET	Weight is an instance property of `FTGroupMember`.

A program that can determine its own suitability for its task can adjust its weight accordingly, as in these examples. (Nonetheless, changing weight frequently can cause thrashing, so we do not recommend it.)

Example 1: Resources

Consider a situation in which program performance depends upon a resource, such as long distance communications lines. Such a program could vary its weight as a function of available lines.

Example 2: Load

If a program can detect that its host computer has become heavily loaded, the program could lower its weight, allowing a member on a lightly loaded computer to become active in its place.

Example 3: System Administrator

In a third example, system administration staff track the availability of resources and the performance of various computers. In response to such information, an administrator could send a message instructing a member process to change its weight.

Chapter 15 Developing Fault-Tolerant Programs

This chapter describes development steps for constructing fault tolerant programs with Rendezvous software. These steps apply in all programming languages.

Topics

Step 1: Choose a Group Name

Each fault tolerance group requires a unique group name. The group name identifies the members of a group, and separates their fault-tolerance messages from messages belonging to other groups.

Number of Groups

If a program serves a single purpose, then its group name can be a single element. In many cases, that element can be a variation on the program's name, suitably modified to obey the syntax rules for group names.

However, if the program serves any of several non-interchangeable purposes (depending on configuration parameters), then each service requires a unique group name.

For example, consider a general database query server program—the service depends on the database it opens. In theory, the same program can service queries for airline flight information, airplane parts inventory information, airplane maintenance records, passenger ticket information, government regulations, or weather reports.

In this situation we recommend choosing group names with two elements, in which the first element identifies the application system, and the second element identifies its database service. The current example could use these names:

- `DBQ.FLIGHTS`
- `DBQ.PARTS`
- `DBQ.MAINTENANCE`
- `DBQ.TICKETS`
- `DBQ.REGULATIONS`
- `DBQ.WEATHER`

Some processes belong to several fault tolerance groups simultaneously (see Multiple Groups on page 217). Once again, this situation suggests a two-element group name—or more than two if necessary.

Group Name Syntax

Members of a group exchange messages that embed their group name, so group names must obey the syntax for Rendezvous subject names.

Group names must not contain wildcard characters.

For best performance, keep group names short. We recommend no more than three or four elements, and no more than 50 characters (total).

For an introduction to Rendezvous subject names, see Subject Names on page 61.

Step 2: Choose the Active Goal

Each member process of a fault tolerance group must specify its active goal when it joins the group. Rendezvous fault tolerance software manages the members so that the number of active members equals the active goal if possible.

Choose an active goal to fit the application:

- If it is important to avoid duplication of service, then consider an active goal equal to one.
- If it is important to share the service load across many members, then choose an active goal to match the expected need for service.

For details and examples, see Active Goal on page 203.

All the members of a group must specify the same active goal.

Step 3: Plan Program Behavior

Consider these issues early in the design phase of your program.

- *Parallel Data State, page 231*.
- *Continuity—Track Active Backlog, page 232*.
- *Activation, page 233*.
- *Preparing to Activate, page 233*.
- *Deactivation, page 233*.
- *Serve It Once, page 234*.
- *Send it Once, page 234*.

Parallel Data State

An inactive member must be ready to activate in the same data state as the formerly active member it replaces. In some situations data state is irrelevant. In other situations it is straightforward to duplicate the data state either by copying and reading a state file, or by completing a brief computation. However, in some situations the data state is complex, or the result of cumulative operations, so the best way to maintain readiness is to compute a parallel data state while inactive.

Example: Current Value Cache

The `rvcache` utility stores the most recent message for each subject name. Whenever a program queries for a cached subject, `rvcache` sends the program the current data corresponding to that subject. (For a more information, see Current Value Cache on page 267 in *TIBCO Rendezvous Administration*.)

Two or more `rvcache` processes can cooperate for fault-tolerant operation, with only one active process. All member processes (whether active or inactive) passively collect and store the same data—but only the active process responds by sending the current data when a program sends a query. Every inactive member always has all the cached data it needs to begin active duty; the data state of each inactive member is parallel to that of the active member.

Notice that in this application the inactive members are far from idle; they collect and store data just like the active member.

Furthermore, when starting a new `rvcache` process, the administrator can copy the store file from another fault-tolerant member, in order to initialize its database to contain the same data as existing member processes.

Continuity—Track Active Backlog

Some applications depend on a continuous stream of data. They must receive all the data—even if they receive it late. For the programs that produce that data, it is essential to maintain continuity of the outbound data stream.

Although Rendezvous fault tolerance software quickly restores service, a finite service interruption always exists between the failure of an active member and the activation of an inactive member. When continuity is essential, it is the responsibility of the inactive members to maintain continuity across the service interruption.

Inactive members maintain continuity by tracking the *backlog* from the active member. That is, the inactive member retains enough information to reproduce the expected output of the active member during the longest service interruption. When it activates, it produces that backlog output before processing any new data. Although the backlog output is delayed, no holes appear in the output stream.

Example: Data Distribution

Many enterprises require access to prodigious amounts of data, which must flow to decision makers in a timely fashion, without interruption. Many organizations use data distribution software that receives data from a serial port, processes it, and broadcasts it across a network to numerous computer workstations.

To ensure continuous service, data distribution software can operate in fault tolerance pairs, with one active member and one inactive member. Each member receives the same data, and each member processes the data, but only the active member broadcasts the data.

Once the active member has broadcast a data item, it can discard that data item.

However, the inactive member must hold the data until it receives the corresponding broadcast item from the active member. To see why, consider the service interruption between the time that the active member fails and the time that the inactive member activates. During the service interruption data continues to arrive, but neither member is broadcasting that data. When the inactive member activates, it must broadcast that backlog data—filling the gap in the data stream. To support this behavior, the inactive member can discard a data item only after confirming that the active member has broadcast it.

Notice that in this application the inactive member does work that the active member does not do; in addition to processing the same set of data items, the inactive member must also retain data, and discard it only at the proper time.

Activation

Consider the actions that your program does to switch from inactive to active.

In some programs the state change in the callback function is as straightforward as toggling a flag; functions throughout program code can branch on the flag to determine inactive or active behavior. Other programs must open data files, open communication lines, allocate resources, begin listening to Rendezvous subjects, or set timers to trigger computations.

Remember, each step delays activation. Whenever resources permit, we recommend minimizing the steps that wait until activation time; taking these steps at start time results in quicker activation.

If the program must maintain continuity after a service interruption, see Continuity—Track Active Backlog on page 232.

Arrange for any needed transition steps in the program's fault tolerance callback function.

Preparing to Activate

Consider whether any of the activation steps are time-consuming. For example, delays are common when opening an ISDN line or opening a database connection.

If such steps might cause unacceptable delays when the program activates, consider separating those preparations from the actual activation sequence. Instead, do them when the fault tolerance callback function receives a prepare-to-activate hint.

Consider setting a duration limit for preparations. For example, if the program allocates a large block of storage when preparing to activate, set a timer to expire after two or three activation intervals. If the timer expires before an actual instruction to activate, then deallocate the storage. If the call to activate arrives first, cancel the timer.

Deactivation

Consider the actions that the program does to deactivate. Usually these actions reverse the activation steps, but in some applications it might be more expedient to retain resources (anticipating the need to reactivate).

Arrange for any needed transition steps in the program's fault tolerance callback function.

Serve It Once

For request server applications, ensure that each request receives service from only one active member. Duplicate service wastes server resources, and could result in incorrect behavior.

Consider whether distributed queues might be a better fit for such applications. See Distributed Queue on page 181.

Send it Once

For broadcast producer applications, ensure that members of a fault tolerance group cooperate to send each data item only once. If several processes are can be active simultaneously, they must not send duplicate data.

Step 4: Choose the Intervals

Table 39 summarizes the four interval parameters that regulate the behavior of Rendezvous fault tolerance software. It is important that you choose appropriate values for these interval parameters.

Choosing the intervals requires a balance among several considerations:

- The need for uninterrupted service.

  Ideally, critical applications must run with only minimal interruptions in service. Realistically, it takes time to discover a service interruption. You can reduce this time to the minimum that your network can support, but at the cost of network capacity and computer time.

- Network transmission time.

  It takes time for heartbeat messages to traverse the network, and that time varies with distance and network load. This fact limits the minimum achievable heartbeat interval, which in turn limits the minimum achievable activation interval.

- Finite network capacity.

  The network that carries heartbeat messages also carries application data. Smaller heartbeat intervals imply more frequent heartbeats. Avoid cluttering the network with too-frequent heartbeat messages.

Table 39 Fault Tolerance Interval Parameters (Sheet 1 of 2)

Parameter	Description
`heartbeatInterval`	Each active member broadcasts a sequence of heartbeat messages to inform the other group members that it is still active. The heartbeat interval determines the time between heartbeat messages. Parameter to the member creation call.
`activationInterval`	Inactive members track heartbeat messages from each active member. When the time since the last heartbeat from an active member reaches this activation interval, Rendezvous fault tolerance software instructs the ranking inactive member to activate. Parameter to the member creation call.

Table 39 Fault Tolerance Interval Parameters (Sheet 2 of 2)

Parameter	Description
`preparationInterval`	Some programs require advance notice to prepare before activation. When the time since the last heartbeat from an active member equals this preparation interval, Rendezvous fault tolerance software issues a hint to the ranking inactive member, so it can prepare to activate. Parameter to the member creation call.
`lostInterval`	Monitor functions passively track heartbeat messages from active members of a fault tolerance group. When the time since the last heartbeat from an active member reaches this lost interval, Rendezvous fault tolerance software considers that member lost, and calls the monitor callback, passing it the current number of active members. Supply to the start monitor creation call.

First: Determine the Activation Interval

The activation interval influences the longest service interruption in two situations:

- When a new member joins a fault tolerance group, the initialization phase requires one activation interval before it can become active.
- In most failure situations the maximum service interruption is identical to the activation interval (assuming an inactive member exists).

In each case, you must determine the amount of time that can elapse before interrupted service becomes a problem. Use an activation interval equal to that time.

Recommended Lower Bound

We recommend an activation interval no less than 3 seconds, though Rendezvous fault tolerance software accepts lower values. However, if your application is distributed across a WAN, we recommend an activation interval no less than 10 seconds.

Second: Determine the Heartbeat Interval

Recommended Lower Bounds

We recommend a heartbeat interval no lower than 1 second, though Rendezvous fault tolerance software accepts lower values.

However, wide-area links transmit heartbeats more slowly (and at greater cost) than local networks. If your application is distributed across a WAN, we recommend a heartbeat interval no less than 2 seconds.

Recommended Relationship between Activation and Heartbeat Interval

The heartbeat interval must be *strictly less* than the activation interval.

Our experience indicates that in most situations, the optimal heartbeat interval is slightly less than one third of the activation interval. For example, an activation interval of 10 seconds implies a heartbeat interval of 3 seconds.

However, messages traversing wide-area links show greater variability in arrival time (compared with local networks). If your application is distributed across a WAN, we recommend a heartbeat interval that is less than one fifth of the activation interval. For example, an activation interval of 30 seconds implies a heartbeat interval of 6 seconds or less.

Conserving Network Capacity

It is important to conserve network capacity (bandwidth). Each heartbeat is a message. Each active member sends one message at every heartbeat interval. If the heartbeat interval is too small, then your program may overload the network with heartbeat messages.

Once you have established the activation and heartbeat intervals for your application, apply this reality check. Calculate the number of heartbeat messages that all the active members of your program will send. Does this figure still leave network capacity for other programs? If not, increase the heartbeat and activation intervals accordingly.

For example, if the heartbeat interval is 0.1 seconds, and an application requires one active member, then the network must carry 100 messages per second to sustain the heartbeat signal. If the application requires 50 active members, then the network must carry 5000 messages per second to sustain the heartbeat signals.

Third: Determine the Preparation Interval

The last step is to determine whether the program requires time to prepare before it can activate, and if so, the length of time it needs.

If the program needs no preparation time, then supply zero as the preparation interval.

If the program does need preparation time to complete set-up tasks, estimate the length of time needed. Subtract that time from the activation interval to obtain the preparation interval.

Recommended Relationship between Preparation and Activation Interval

If non-zero, the preparation interval must be strictly greater than the heartbeat interval, and strictly less than the activation interval. We recommend that you choose a preparation interval that is greater than twice the heartbeat interval.

For programs that require preparation time, we recommend a preparation interval no less than 75% of the activation interval. For example, an activation interval of 10 seconds implies a preparation interval of 7.5–9.5 seconds. Smaller preparation intervals may increase the rate of false-positive prepare-to-activate hints.

For Monitors: Determine the Lost Interval

When monitoring a fault tolerance group, the lost interval argument must equal the activation interval of the group.

Step 5: Program Start Sequence

Fault-tolerant programs follow a typical start sequence:

1. Begin in the program's inactive state.

   For example, set flags within the program that prevent the behavior of an active member.

2. Do set-up tasks that do not depend on Rendezvous software.
3. Open the Rendezvous environment.
4. Do set-up tasks that depend on Rendezvous software. (Requires the Rendezvous environment to be open.)
5. Create the fault tolerance member event. (Requires the Rendezvous environment to be open.)

Appendix A System Advisory Messages

Rendezvous software presents *advisory messages* to inform programs of exceptional situations that might affect them. Advisory messages report errors, warnings and other useful information. This appendix describes the *system advisory messages* generated by Rendezvous communications and Rendezvous daemon components.

Topics

See Also

- Certified Message Delivery (RVCM) Advisory Messages, page 269
- Fault Tolerance (RVFT) Advisory Messages, page 293

Advisory Messages

Rendezvous software presents asynchronous advisory messages to Rendezvous programs. Advisory messages indicate errors, warnings and other information.

In contrast with status codes (which indicate success or failure *within* a specific Rendezvous call), asynchronous advisory messages notify programs of events that occur *outside* of the program's direct flow of control—for example, the program is processing inbound messages too slowly, causing the daemon's message queue to overflow.

Advisory Summary

Table 40 Rendezvous System Advisories (Sheet 1 of 2)

Advisory Message	Class	Page
`CLIENT.DEFUNCT`	`ERROR`	246
`CLIENT.FASTPRODUCER`	`WARN`	247
`CLIENT.ILLEGAL_PUBLISH`	`ERROR`	248
`CLIENT.NOMEMORY`	`ERROR`	249
`CLIENT.SLOWCONSUMER`	`ERROR`	250
`DATALOSS`	`ERROR`	252
`DISPATCHER.THREAD_EXITED`	`INFO`	254
`HOST.STATUS`	`INFO`	255
`LICENSE.EXPIRE`	`ERROR, WARN`	258
`QUEUE.LIMIT_EXCEEDED`	`WARN`	259
`RETRANSMISSION.INBOUND.EXPECTED`	`INFO`	260
`RETRANSMISSION.INBOUND.REQUEST_NOT_SENT`	`INFO`	261
`RETRANSMISSION.OUTBOUND.SENT`	`INFO`	263
`RETRANSMISSION.OUTBOUND.SUPPRESSED`	`INFO`	264

Table 40 Rendezvous System Advisories (Sheet 2 of 2)

Advisory Message	Class	Page
RVD RVD.RECONNECT_FAILED RVD.DISCONNECTED RVD.CONNECTED	 ERROR WARN INFO	265
UNREACHABLE.TRANSPORT	INFO	266
VC.CONNECTED	INFO	267
VC.DISCONNECTED	ERROR, INFO	268

Receiving Advisory Messages

Rendezvous programs can receive advisory messages in the same way as any other messages—by listening to subject names.

For example, the subject _RV.*.SYSTEM.> matches all advisories related to communications. Programs can also listen more selectively for specific advisories, as appropriate.

Advisories related to a specific transport present on that transport. A program that creates several transports might need to listen for advisory messages on each of its transports.

Advisories not related to a specific transport present on the intra-process transport.

Advisory messages wait for dispatch in the queue that the program designates when creating the listener.

(Programs listen for advisory messages using ordinary Rendezvous listening calls, rather than certified listening calls.)

Redirecting Advisories to stderr

Rendezvous software informs programs of exceptional conditions by presenting advisory messages. Error and warning messages indicate situations that could have serious consequences. Rendezvous software protects programs from losing important messages by catching error and warning messages that would otherwise be lost.

If Rendezvous software detects an error or warning message, and the program is listening for a matching subject, then Rendezvous software queues it for the appropriate callback function within the program. However, if the program is *not* listening for a matching subject, Rendezvous software intercepts the error or warning message, and redirects it to `stderr` (or equivalent).

Informational messages are neither caught nor redirected; the only way to receive them is to listen for them explicitly.

Some platforms do not support the concept of `stderr`, or support it only in limited cases. When `stderr` is not supported, error and warning messages are lost unless the program explicitly listens for them. For example, Microsoft Windows operating systems do support `stderr`, but only in console-based applications; they do not support it in GUI (window-based) programs.

System Advisory Subject Names

Rendezvous software constructs the subject names of system advisory messages using this template:

```
_RV.class.SYSTEM.name
```

Table 41 SYSTEM Advisory Subject Name Elements

Element	Description
class	The *class* element denotes the severity of the situation: • The class `ERROR` indicates either a problem that requires immediate action, or a situation in which Rendezvous software could not complete its task properly. • The class `WARN` indicates an anomalous situation that is not yet critical. In many cases Rendezvous software can rectify the situation by itself. • The class `INFO` indicates an interesting event in the normal operation of Rendezvous software.
source	The *source* element is `SYSTEM` for all advisories from Rendezvous communications software and the Rendezvous daemon.
name	The *name* element describes the situation that the advisory reports. This element can actually consist of several elements, so the wildcard character > (rather than `*`) is the correct way to match all names in this position.

CLIENT.DEFUNCT

Advisory

Subject Name Syntax

`_RV.ERROR.SYSTEM.CLIENT.DEFUNCT`

Purpose

Client transports receive this advisory when they become unusable as a consequence of deleting an RVDM subject map.

Background

Limitation on Computers with Multiple Network Interfaces on page 25 in *TIBCO Rendezvous Administration* discusses a limitation—all the client transports of a daemon that use a particular service must also specify the same network parameter.

RVDM subject mapping discards application-specified network parameters, and replaces them with multicast groups, effectively transcending this limitation, and enabling transports to communicate that would otherwise be unable to co-exist.

In this situation, deleting the subject map causes the existing client transports to revert to their own network parameters. If these network parameters conflict with one another, one transport (T1) determines the fate of the others:

- Transports with the same network specification as T1 continue to function properly.
- Transports with conflicting network specifications become defunct. They receive this advisory, and stop functioning.

Deleting a subject map could cause this effect at several managed daemons throughout an enterprise. The surviving network specification could be different (and probably will be different) at each daemon. As a result, the surviving client transports at the various daemons will probably not be able to communicate.

Remarks

Only the defunct client transports receive this advisory.

This message is an error—the daemon stops communication for the affected transports.

Message Fields

Field Name	Description
`tport`	The transport ID.
`descr`	The client's description string.

CLIENT.FASTPRODUCER

Advisory

Subject Name Syntax

`_RV.WARN.SYSTEM.CLIENT.FASTPRODUCER`

Purpose

A sending client of the Rendezvous daemon receives this advisory when it produces data faster than the network can distribute it.

Remarks

Only the specific client transport receives this advisory.

This message is a warning—the program is sending messages faster than the physical network can accept them; the program's outbound message queue will grow until it exhausts available storage. (In contrast to `CLIENT.SLOWCONSUMER`, this advisory does not indicate an error, because data is never discarded.)

When a client program sends many outbound messages in rapid succession, they remain in the client's storage until `rvd` can accept them to place them on the network. When `rvd` leaves an outbound message waiting in the client program more than 5 seconds, Rendezvous software presents this advisory to the client transport.

Message Fields

Field Name	Description
`waiting`	The number of outbound messages waiting in the client program.

Diagnosis

`FASTPRODUCER` advisories can indicate any of several situations:

- The client program is sending messages in a tight loop—either the daemon or the physical network cannot absorb the volume.
- The Rendezvous daemon is starved for CPU cycles; either its host computer is too heavily loaded, or the priority of the daemon process is too low.

CLIENT.ILLEGAL_PUBLISH

Advisory

Subject Name Syntax

`_RV.ERROR.SYSTEM.CLIENT.ILLEGAL_PUBLISH`

Purpose

A sending client of the Rendezvous daemon receives this advisory when it publishes on an illegal subject.

Remarks

Only the specific client transport receives this advisory.

This message is an error—the daemon rejects and discards messages sent on illegal subjects, and reports that action with this advisory.

Message Fields

Field Name	Description
`sub`	The illegal subject of the message.
`tport`	The transport ID.
`descr`	The client’s description string.

Diagnosis

`ILLEGAL_PUBLISH` advisories can indicate sending to a wildcard subject when wildcards are prohibitted (for example, when RVDM disables wildcard publishing on a service).

CLIENT.NOMEMORY

Advisory

Subject Name Syntax `_RV.ERROR.SYSTEM.CLIENT.NOMEMORY`

Purpose A program receives this advisory when it cannot allocate sufficient process storage.

Remarks Only the specific client receives this advisory.

When a program cannot allocate storage within a Rendezvous API call, the call returns the status code `TIBRV_NO_MEMORY`. In asynchronous situations, the program presents this advisory instead.

This advisory always indicates an error. Either data is lost, or the client program cannot continue.

Diagnosis `NOMEMORY` advisories can indicate several asynchronous situations in which the program cannot allocate storage; for example:

- The client program cannot allocate memory to receive an inbound messages from the Rendezvous daemon.

  After presenting this advisory, the client program disconnects its transport from the daemon, discarding all queued data. The client then attempts to reconnect to the daemon.

- The client program cannot allocate memory to create the timer it needs to reconnect to the daemon.

  After presenting this advisory, the client program sleeps, and attempts to reconnect later.

In either of these situations, the system administrator or programmer must determine the reason that the program exhausted its process storage, and remedy the problem at its source.

CLIENT.SLOWCONSUMER

Advisory

Subject Name Syntax `_RV.ERROR.SYSTEM.CLIENT.SLOWCONSUMER`

Purpose A listening client of the Rendezvous daemon receives this advisory when data is arriving faster than the client is consuming it.

Remarks Only the specific client transport receives this advisory.

This message indicates an error—the Rendezvous daemon has discarded the oldest inbound messages to make room for new ones. (Clients cannot determine which messages are discarded.)

When many messages arrive in rapid succession, `rvd` buffers the messages until clients can consume them (that is, accept them from `rvd`). When unconsumed inbound messages either wait for more than 60 seconds or overflow the client's buffer (within the daemon), `rvd` discards them, and presents this advisory to the affected client transport to indicate that data has been lost. This advisory is the next message that the transport receives—ahead of all other inbound messages.

Message Fields

Field Name	Description
`waiting`	The number of messages waiting in `rvd`.
`dropped`	The number of messages that `rvd` has discarded.
`bytes_dropped`	The total number of bytes in the messages that `rvd` has discarded.
`reason`	This string indicates the reason that the daemon discarded messages: • `time limit` indicates that the data remained unconsumed in the daemon beyond the 60-second time limit. • `size limit` indicates that message data overflowed the explicit size limit specified in the daemon's `-max-consumer-buffer` parameter.

Diagnosis Several situations can cause a slow consumer:

- The client program is oversubscribed—it cannot process the volume of data for which it is listening.
- The client program is starved for CPU cycles—its host computer is too heavily loaded.

DATALOSS

Advisory

Subject Name Syntax

```
_RV.ERROR.SYSTEM.DATALOSS.OUTBOUND.PTP
_RV.ERROR.SYSTEM.DATALOSS.OUTBOUND.BCAST
_RV.ERROR.SYSTEM.DATALOSS.INBOUND.PTP
_RV.ERROR.SYSTEM.DATALOSS.INBOUND.BCAST
```

Purpose

Client programs receive this advisory when a sending daemon denies a retransmission request.

Remarks

`DATALOSS` advisories indicate network transmission problems.

Rendezvous daemons use a reliable delivery protocol, in which sending daemons retain outbound messages for 60 seconds. If a receiving daemon detects that it missed an inbound message, it requests retransmission from the sending daemon. If 60 seconds have already elapsed, the sending daemon has already discarded the message, so it cannot retransmit. Under normal operating conditions, both daemons notify all their client transports that data has been lost; in some situations, not all of the daemons report the loss.

Clients of the sending daemon present `DATALOSS.OUTBOUND` advisories. Clients of the receiving daemon present `DATALOSS.INBOUND` advisories.

`PTP` indicates that the lost data was a point-to-point message. `BCAST` indicates that the lost data was a multicast message.

Since the daemons cannot determine which client transports are affected by the loss, they present these advisories to all of their clients on the service that lost data (even though not every client on that service has actually lost data).

Message Fields

Field Name	Description
`host`	The IP address of the *other* computer.
`lost`	The number of packets requested by the host, but not retransmitted by the sending daemon (during the interval since the last advisory of this type for the receiving host and service).

Diagnosis

These advisories indicate situations that defeat the Rendezvous reliable delivery protocols:

- Some hardware component is experiencing intermittent failures; the component could be a faulty network card, a loose connection, or a frayed wire.

- The network is overloaded.
- A Rendezvous daemon process is starved for CPU cycles; either its host computer is too heavily loaded, or the priority of the daemon process is too low.
- The Rendezvous daemon is running with a `-reliability` parameter lower than 60 seconds. (See also, Reliability and Message Retention Time on page 43 in *TIBCO Rendezvous Administration*.)
- When the error description string includes the words `multicast destination`, this advisory can be a symptom of interference in a mixed environment with regard to RVDM subject maps. For more information, see Mixed Environment—Subject Maps on page 210 in *TIBCO Rendezvous Administration*.

Limitations

Rendezvous reliable delivery protocols implement fast and efficient delivery of messages under normal operating conditions. For diagnostic convenience, the Rendezvous daemon reports `DATALOSS` advisories when detection would not incur the cost of additional network traffic. However, `DATALOSS` advisories are not guaranteed in every situation.

Rendezvous software does not report `DATALOSS` advisories across routing daemon neighbor links—only to transports directly connected to the daemon that detects the loss.

If your program requires stronger confirmation of delivery, consider using the certified delivery feature (see Certified Message Delivery on page 139).

DISPATCHER.THREAD_EXITED

Advisory

Subject Name Syntax

`_RV.INFO.SYSTEM.DISPATCHER.THREAD_EXITED.`*thread_name*

Purpose

A dispatcher thread exited, producing an error status code.

Remarks

Dispatcher threads are a programmer convenience. Programs that do not use dispatcher threads never present this advisory.

When the dispatcher thread has a name, it appears as the *thread_name* element of the advisory subject; otherwise a default thread name appears in that position.

The process transport presents this advisory.

If a program is listening for this advisory, the event driver places it on the queue associated with the listener event. We recommend using the default queue for this advisory, since it never discards an event.

Message Fields

Field Name	Description
`status`	The status code. This field has datatype `TIBRVMSG_U32`.
`description`	A string corresponding to the status code. This field has datatype `TIBRVMSG_STRING`.

Diagnosis

This advisory reports these program situations:

- The dispatch thread exited because its dispatch timeout elapsed; that is, the thread waited for that time period, during which no events were ready for dispatch.
- An error occurred while attempting to dispatch the queue or queue group. This status usually indicates that the queue or queue group is destroyed or invalid.
- The dispatcher thread was destroyed.

See Also

Dispatcher Threads, page 85

HOST.STATUS

Advisory

Subject Name Syntax

`_RV.INFO.SYSTEM.HOST.STATUS.`*hostid*

Purpose

Reports the operating status of a daemon processes.

Remarks

Each communications daemon periodically broadcasts this advisory to its network.

Each advisory presents a snapshot of daemon statistics on one active service. If a daemon operates on more than one service, it sends a separate advisory for each one. Statistics within each snapshot are cumulative since the daemon began communicating on the service.

The *hostid* element of the advisory subject name is the IP address (in hex notation) of the daemon that sent the advisory.

Message Fields

(Sheet 1 of 3)

Field Name	Description
`hostaddr`	The IP address of the daemon’s host computer. This value has type `TIBRVMSG_IPADDR32`.
`sn`	Serial number from the daemon’s license key. This value has type `TIBRVMSG_U32`.
`os`	A code number denoting the operating system of the daemon’s host computer. This value has type `TIBRVMSG_U8`.
`ver`	The software release number (version) of the daemon. This value has type `TIBRVMSG_STRING`.
`httpaddr`	IP address where the daemon listens for HTTP connections. This value has type `TIBRVMSG_IPADDR32`.

(Sheet 2 of 3)

Field Name	Description
`httpport`	HTTP port where the daemon listens for HTTP connections. This value has type `TIBRVMSG_IPPORT16`.
`hsaddr`	IP address where the daemon listens for HTTPS secure connections. This value has type `TIBRVMSG_IPADDR32`.
`hsport`	HTTP port where the daemon listens for HTTPS secure connections. This value has type `TIBRVMSG_IPPORT16`.
`time`	Time of this snapshot (Zulu time). This value has type `TIBRVMSG_DATETIME`.
`up`	Elapsed time since the daemon began operating on this `service`. This value has type `TIBRVMSG_U32`.
`ms`	Messages sent by the daemon on this `service`. This value has type `TIBRVMSG_U64`.
`bs`	Bytes sent (summed over all messages tallied in `ms`). This value has type `TIBRVMSG_U64`.
`mr`	Messages received by the daemon on this `service`. This value has type `TIBRVMSG_U64`.
`br`	Bytes received (summed over all messages tallied in `mr`). This value has type `TIBRVMSG_U64`.
`ps`	Packets sent (outbound). This value has type `TIBRVMSG_U64`.
`pr`	Packets received (inbound). This value has type `TIBRVMSG_U64`.

(Sheet 3 of 3)

Field Name	Description
rx	Packets retransmitted (outbound). This value has type TIBRVMSG_U64.
pm	Packets missed (inbound). This value has type TIBRVMSG_U64.
idl	Inbound data loss (in packets). This value has type TIBRVMSG_U64.
odl	Outbound data loss (in packets). This value has type TIBRVMSG_U64.
irrs	Inbound data retransmission requests suppressed (that is, requests not sent) by RXC at the receiver (in packets). This value has type TIBRVMSG_U64.
orrs	Outbound data retransmission requests suppressed (that is, requests ignored) by RXC at the sender (in packets). This value has type TIBRVMSG_U64.
ipport	IP port where the daemon listens for client connections. This is identical to the transport daemon parameter. This value has type TIBRVMSG_IPPORT16.
service	Service for which this advisory presents a snapshot. This is identical to the transport service parameter. This value has type TIBRVMSG_STRING.
network	Network for which this advisory presents a snapshot. This is identical to the transport network parameter. This value has type TIBRVMSG_STRING.

LICENSE.EXPIRE

Advisory

Subject Name Syntax

```
_RV.ERROR.SYSTEM.LICENSE.EXPIRE
_RV.WARN.SYSTEM.LICENSE.EXPIRE
```

Purpose The Rendezvous daemon license has expired, or will expire soon.

Remarks Each Rendezvous daemon process requires a valid license. Some licenses are valid for only a limited time. These messages indicate expiration (or impending expiration) of a license.

Warning As a warning advisory, this message indicates that the license will expire soon. Every Rendezvous daemon that depends on that license presents this warning message to each of its client transports.

Error As an error advisory, this message indicates that the license has already expired. Every affected Rendezvous daemon stops servicing its client programs, and presents this error message to each client transport.

Message Fields

Field Name	Description
`expiretime`	The expiration time of the license. The value has type `TIBRVMSG_DATETIME`.

QUEUE.LIMIT_EXCEEDED

Advisory

Subject Name Syntax

`_RV.WARN.SYSTEM.QUEUE.LIMIT_EXCEEDED.`*queue_name*

Purpose

An event queue exceeded its event limit, discarding an event.

Remarks

When the queue has a name, it appears as the *queue_name* element of the advisory subject; otherwise a default queue name appears in that position.

The process transport presents this advisory.

If a program is listening for this advisory, the event driver places it on the queue associated with the listener event. We strongly recommend using a queue that never discards an event; the default queue is a good choice for this purpose.

RETRANSMISSION.INBOUND.EXPECTED

Advisory

Subject Name Syntax `_RV.INFO.SYSTEM.RETRANSMISSION.INBOUND.EXPECTED.`*hostid*

Purpose A receiving daemon produces this advisory when it detects missed inbound packets, and expects possible retransmission.

Remarks A communications daemon presents this advisory to subscribing clients.

We do not recommend subscribing to this advisory for general monitoring; rather, see `HOST.STATUS` on page 255. `RETRANSMISSION.INBOUND.EXPECTED` is better suited to identifying the specific sending host from which this daemon is missing inbound packets.

Message Fields

Field Name	Description
`host`	The IP address of the *sending* daemon's host computer (the source of the missed packets).
`lost`	The number of missed packets.

RETRANSMISSION.INBOUND.REQUEST_NOT_SENT

Advisory

Subject Name Syntax _RV.INFO.SYSTEM.RETRANSMISSION.INBOUND.REQUEST_NOT_SENT

Purpose A receiving daemon produces this advisory when, as a chronically-lossy receiver, it censors its own retransmission requests.

Remarks A communications daemon presents this advisory to subscribing clients.

Diagnosis This advisory indicates that a receiving daemon's retransmission control (RXC) feature has identified itself as a chronically-lossy receiver, and is protecting network bandwidth by censoring its retransmission requests (that is, not sending those requests). Each advisory corresponds to one censored request.

It is likely that the advisory DATALOSS.INBOUND.BCAST will accompany this advisory, since the receiving daemon does not request retransmission of the missed packets.

Check the receiver for the following possible problems:

- Some hardware component is experiencing intermittent failures; the component could be a faulty network card, a loose connection, or a frayed wire.
- A Rendezvous daemon process on the receiving host is starved for CPU cycles; either its host computer is too heavily loaded, or the priority of the daemon process is too low.
- The rate at which inbound data arrives overwhelms the capacity of the receiving host computer, which has insufficient CPU power, network bandwidth, memory, or some other limiting resource.
- If several hosts on the same network produce these advisories, the problem could be in the network routing or switching hardware.

Message Fields

Field Name	Description
host	The IP address of the host computer where the chronically-lossy receiver (daemon) is running.
lost	The number of missed data packets for which RXC (in the receiving daemon) suppressed a retransmission request.

See Also

`RETRANSMISSION.OUTBOUND.SUPPRESSED` on page 264
Retransmission Control on page 45 in *TIBCO Rendezvous Administration*

RETRANSMISSION.OUTBOUND.SENT

Advisory

Subject Name Syntax `_RV.INFO.SYSTEM.RETRANSMISSION.OUTBOUND.SENT`

Purpose A sending daemon produces this advisory when it retransmits requested packets.

Remarks A communications daemon presents this advisory to subscribing clients.

We do not recommend subscribing to this advisory for general monitoring; rather, see `HOST.STATUS` on page 255. `RETRANSMISSION.OUTBOUND.SENT` is better suited to identifying the specific receiving host which is missing packets from this sending daemon.

Message Fields

Field Name	Description
`host`	The IP address of the daemon’s host computer.
`lost`	The number of packets retransmitted.

RETRANSMISSION.OUTBOUND.SUPPRESSED

Advisory

Subject Name Syntax `_RV.INFO.SYSTEM.RETRANSMISSION.OUTBOUND.SUPPRESSED`

Purpose A sending daemon produces this advisory when it ignores retransmission requests from a chronically-lossy receiver.

Remarks A communications daemon presents this advisory to subscribing clients.

Diagnosis This advisory indicates that the retransmission control (RXC) feature has identified a chronically-lossy receiver, and is protecting network bandwidth by ignoring its retransmission requests. Each advisory corresponds to one suppressed request.

Check the indicated receiver for the following possible problems:

- Some hardware component is experiencing intermittent failures; the component could be a faulty network card, a loose connection, or a frayed wire.
- A Rendezvous daemon process on the receiving host is starved for CPU cycles; either its host computer is too heavily loaded, or the priority of the daemon process is too low.
- The rate at which inbound data arrives overwhelms the capacity of the receiving host computer, which has insufficient CPU power, network bandwidth, memory, or some other limiting resource.
- If these advisories indicate several hosts on the same network, the problem could be in the network routing or switching hardware.

Message Fields

Field Name	Description
`host`	The IP address of the host computer where the chronically-lossy receiver (daemon) is running.
`lost`	The number of requested data packets for which RXC (in the sending daemon) suppressed retransmission.

See Also `RETRANSMISSION.INBOUND.REQUEST_NOT_SENT` on page 261
Retransmission Control on page 45 in *TIBCO Rendezvous Administration*

RVD

Advisory

Subject Name Syntax

```
_RV.WARN.SYSTEM.RVD.DISCONNECTED
_RV.INFO.SYSTEM.RVD.CONNECTED
_RV.ERROR.SYSTEM.RVD.RECONNECT_FAILED
```

Purpose The transport's connection to the Rendezvous daemon has changed.

Remarks Every affected transport presents this advisory.

Disconnected The warning advisory, `_RV.WARN.SYSTEM.RVD.DISCONNECTED`, indicates that the transport is no longer connected to a Rendezvous daemon process. While the transport object and the daemon are disconnected, no data flows between them in either direction. If the daemon process was running on the local host computer, the transport object will automatically attempt to restart it and reconnect. In the interim, the daemon is not receiving inbound messages on behalf of the transport.

Several external situations can result in `RVD.DISCONNECTED`; for example:

- The `rvd` process was removed.
- The wire connecting the remote program host to the `rvd` host is broken.
- The remote `rvd` host experienced a power failure.

Connected The informational advisory, `_RV.INFO.SYSTEM.RVD.CONNECTED`, indicates that the transport is again connected to a Rendezvous daemon.

Reconnect Failed The error advisory, `_RV.ERROR.SYSTEM.RVD.RECONNECT_FAILED`, indicates that after becoming disconnected from its daemon, the transport cannot reconnect at the same IP address.

For example, this error can occur in a network architecture that uses virtual IP addresses, the redirector might assign one remote daemon when a transport first connects, but might assign a different daemon when the transport attempts to reconnect. In this situation, transport behavior is undefined.

This advisory indicates that the transport is no longer valid, and any objects that depend on the transport are no longer valid. The program must destroy the invalid transport and associated objects, and then recreate them.

UNREACHABLE.TRANSPORT

Advisory

Subject Name Syntax `_RV.INFO.SYSTEM.UNREACHABLE.TRANSPORT.`*transport_id*

Purpose A transport has terminated.

Remarks When a program terminates a transport object, its Rendezvous daemon sends this advisory to the network at large.

Diagnosis Either the program destroyed the transport as part of its normal operation, or the program terminated abruptly. The daemon does not distinguish between these two possibilities.

VC.CONNECTED

Advisory

Subject Name Syntax `_RV.INFO.SYSTEM.VC.CONNECTED`

Purpose A virtual circuit is established and ready to use.

Remarks Virtual circuit transport objects (that is, both terminals of the circuit) present this advisory.

Message Fields None.

Missed Advisory It is possible to miss this advisory. That is, the transport object can present the advisory before the program begins listening for it. To avoid this situation, combine listening for this advisory with a direct test of the connection. For details, see Testing the New Connection on page 123.

See Also Virtual Circuits on page 119

VC.DISCONNECTED

Advisory

Subject Name Syntax

```
_RV.ERROR.SYSTEM.VC.DISCONNECTED
_RV.INFO.SYSTEM.VC.DISCONNECTED
```

Purpose A virtual circuit connection has broken.

Remarks Virtual circuit transport objects (that is, both terminals of the circuit) present this advisory.

Error The error advisory, `_RV.ERROR.SYSTEM.VC.DISCONNECTED`, indicates virtual circuit failure:

- Either terminal detects `DATALOSS` (that is, missing messages) on the circuit.
- Either terminal detects loss of the heartbeat signal from the opposite terminal.

Info The informational advisory, `_RV.INFO.SYSTEM.VC.DISCONNECTED`, indicates graceful disconnection—that is, one of these cases:

- The process at the other end of the circuit exits.
- The process at the other end of the circuit explicitly destroys its virtual transport object.
- The process at *this* end of the circuit explicitly destroys its virtual transport object. (It presents this advisory before freeing its storage.)

Message Fields

Field Name	Description
`description`	A string that describes the reason for the broken connection. This field has datatype `TIBRVMSG_STRING`.

See Also Virtual Circuits on page 119

Appendix B Certified Message Delivery (RVCM) Advisory Messages

Rendezvous CM transports (and distributed queue members) present advisory messages on the transports they employ for network communications; they do not broadcast advisories on the network.

Topics

Certified Delivery and Distributed Queue Advisory Messages

Table 42 lists the certified delivery and distributed queue advisory messages, and indicates which ones listening CM transports can present.

Table 42 Rendezvous Certified Message Delivery Advisories

Advisory Message	Class	To	Page
`DELIVERY.CONFIRM`	`INFO`	Sender	273
`DELIVERY.COMPLETE`	`INFO`	Sender	274
`DELIVERY.NO_RESPONSE`	`WARN`	Sender	275
`DELIVERY.FAILED`	`ERROR`	Sender	276
`DELIVERY.UNAVAILABLE`	`ERROR`	Listener	278
`REGISTRATION.REQUEST`	`INFO`	Sender	280
`REGISTRATION.CERTIFIED`	`INFO`	Listener	281
`REGISTRATION.NOT_CERTIFIED`	`WARN`	Listener	282
`REGISTRATION.NO_RESPONSE`	`WARN`	Listener	283
`REGISTRATION.CLOSED`	`INFO`	Sender	284
`REGISTRATION.DISCOVERY`	`INFO`	Listener	285
`REGISTRATION.MOVED`	`INFO`	Listener, Sender	286
`REGISTRATION.COLLISION`	`ERROR`	Listener, Sender	287
`RELAY.CONNECTED`	`INFO`	Listener, Sender	288
`RELAY.DISCONNECTED`	`INFO`	Listener, Sender	289
`QUEUE.SCHEDULER.ACTIVE` `QUEUE.SCHEDULER.INACTIVE`	`INFO`	Queue Member	290
`QUEUE.SCHEDULER.OVERFLOW`	`WARN`	Queue Member	291

RVCM Advisory Subject Names

Rendezvous certified message delivery software constructs the subject names of advisory messages using these templates:

```
_RV.class.RVCM.category.condition.subject
_RV.class.RVCM.category.condition.name
```

Distributed queue software constructs the subject names of advisory messages using this template:

```
_RV.class.RVCM.category.role.condition.name
```

Table 43 RVCM Advisory Subject Name Elements (Sheet 1 of 2)

Element	Description
class	The *class* element denotes the severity of the situation: • `ERROR` indicates either a problem that precludes certified delivery, or a situation in which delivery did not complete. • `WARN` indicates an anomalous situation that might jeopardize certified message delivery. • `INFO` indicates an interesting event in the normal operation of certified message delivery.
source	The *source* element is `RVCM` for all advisories from certified message delivery software.
category	The *category* element indicates the general category of the situation that the advisory describes: • `DELIVERY` pertains to the tracking and delivery of specific messages. • `REGISTRATION` pertains to the administration of certified delivery service. • `RELAY` pertains to relay agents. • `QUEUE` pertains to distributed queues.
condition	The *condition* element indicates the specific situation that the advisory reports.
role	The *role* element of `QUEUE` advisories indicates whether the advisory pertains to the member in the `LISTENER` (worker) role or the `SCHEDULER` role.

Table 43 RVCM Advisory Subject Name Elements (Sheet 2 of 2)

Element	Description
subject	The *subject* element contains the subject name to which the advisory pertains. These names often consist of several elements, so the wildcard character "`>`" (rather than "`*`") is the correct way to match all names in this position.
name	The *name* element contains the reusable name to which the advisory pertains. These names may consist of several elements, so the wildcard character "`>`" (rather than "`*`") is the correct way to match all names in this position.

DELIVERY.CONFIRM

Advisory

Subject Name Syntax

`_RV.INFO.RVCM.DELIVERY.CONFIRM.`*subject*

Purpose

A sender presents this advisory whenever a registered listener confirms receipt of a certified message.

Remarks

`_RV.INFO.RVCM.DELIVERY.CONFIRM.`*subject* indicates confirmed delivery of a certified message to a particular listener.

The default behavior of listeners is automatic confirmation upon return of the listener's data callback function. Programs can override this behavior, and confirm delivery with an explicit call for each inbound certified message. (See Automatic Confirmation of Delivery on page 156.)

The *subject* element is the subject name of the certified message. The subject often consists of several elements, so the wildcard character `">"` (rather than `"*"`) is the correct way to match all subjects in this position.

Message Fields

Field Name	Description
`subject`	The subject name of the certified message. It is identical to the *subject* element. This field has datatype `TIBRVMSG_STRING`.
`listener`	The name of the listener that confirmed receipt. This field has datatype `TIBRVMSG_STRING`.
`seqno`	The sequence number of the confirmed message. This field has datatype `TIBRVMSG_U64`.

DELIVERY.COMPLETE

Advisory

Subject Name Syntax `_RV.INFO.RVCM.DELIVERY.COMPLETE.`*subject*

Purpose A sender presents this advisory when all registered listeners have either confirmed delivery of a certified message, or canceled interest in receiving it.

Remarks The sender presents this advisory after deleting the message from its ledger.

`DELIVERY.COMPLETE` means that no listener still requires certified delivery of the message. It is possible that some listeners were previously expecting certified delivery of the message, but have canceled interest in it—either by closing the subscription, or by moving without requiring old messages.

The *subject* element is the subject name of the certified message. The subject often consists of several elements, so the wildcard character `">"` (rather than `"*"`) is the correct way to match all subjects in this position.

Message Fields

Field Name	Description
`subject`	The subject name of the certified message. It is identical to the *subject* element. This field has datatype `TIBRVMSG_STRING`.
`seqno`	The sequence number of the complete message. This field has datatype `TIBRVMSG_U64`.

DELIVERY.NO_RESPONSE

Advisory

Subject Name Syntax

`_RV.WARN.RVCM.DELIVERY.NO_RESPONSE.`*name*

Purpose

A sender presents this advisory when a listener does not confirm certified messages.

Remarks

The advisory repeats at a regular interval until the sender receives a delivery confirmation from the listener.

When several certified listeners are unresponsive, the sender presents a separate advisory for each one.

The *name* element is the name of the unresponsive listener. The listener name can span several elements, so the wildcard character `">"` (rather than `"*"`) is the correct way to match all subjects in this position.

Message Fields

Field Name	Description
`listener`	The name of the listener that has not confirmed message delivery. It is identical to the *name* element. This field has datatype `TIBRVMSG_STRING`.

Diagnosis

`DELIVERY.NO_RESPONSE` advisories can indicate any of these difficulties:

- The physical network is unreliable.
- The receiving program has terminated or is unstable.
- A program cannot locate its relay agent process.

DELIVERY.FAILED

Advisory

Subject Name Syntax

`_RV.ERROR.RVCM.DELIVERY.FAILED.`*subject*

Purpose

A sender presents this advisory when the time limit of a message expires before all registered listeners have confirmed delivery.

Remarks

The time limit is fixed by the CM send call.

The sender presents this advisory after Rendezvous software deletes the message from the sender's ledger.

The *subject* element is the subject name of the certified message. The subject often consists of several elements, so the wildcard character ">" (rather than "*") is the correct way to match all subjects in this position.

Message Fields

Field Name	Description
`subject`	The subject name of the certified message. It is identical to the *subject* element. This field has datatype `TIBRVMSG_STRING`.
`seqno`	The sequence number of the failed message. This field has datatype `TIBRVMSG_U64`.
`data`	The data portion of the failed message. This field has datatype `TIBRVMSG_MSG`.
`listener`	A `DELIVERY.FAILED` advisory can contain one or more fields named `listener`. Each `listener` field contains the name of a listener that did not confirm delivery. This field has datatype `TIBRVMSG_STRING`.

Diagnosis

This advisory indicates an error, and reports unexpected and usually undesirable behavior.

`DELIVERY.FAILED` advisories can indicate any of several difficulties:

- The physical network is unreliable.
- One or more receiving programs have terminated or are unstable.

- The sending program is using a time-out value that is too low.
- The sending program explicitly marked the message as expired.

DELIVERY.UNAVAILABLE

Advisory

Subject Name Syntax

`_RV.ERROR.RVCM.DELIVERY.UNAVAILABLE.`*subject*

Purpose

A listener presents this advisory when one or more messages are no longer available for retransmission.

Remarks

Consider this example situation. Message number `49` arrives at the listener out of sequence—it has not yet received messages number `46`, `47` and `48`. Rendezvous software automatically requests that the sender retransmit messages `46`–`48`. However, the time-out has expired for those messages, and they are no longer available in the sender's ledger. The listening transport first presents a `DELIVERY.UNAVAILABLE` advisory, indicating that messages `46-48` are lost; then it queues message `49`.

In this example a range of messages are unavailable. The advisory indicates this range by noting the sequence numbers of the first unavailable message (`seqno_begin` is `46`) and the last unavailable message (`seqno_end` is `48`). In a situation where only one message is unavailable, the fields `seqno_begin` and `seqno_end` both contain the same value.

The *subject* element is the subject name of the unavailable certified messages. The subject often consists of several elements, so the wildcard character `">"` (rather than `"*"`) is the correct way to match all subjects in this position.

Message Fields

(Sheet 1 of 2)

Field Name	Description
`subject`	The subject name of the unavailable certified messages. It is identical to the *subject* element. This field has datatype `TIBRVMSG_STRING`.
`seqno_begin`	The sequence number of the first unavailable message in the range. This field has datatype `TIBRVMSG_U64`.
`seqno_end`	The sequence number of the last unavailable message in the range. This field has datatype `TIBRVMSG_U64`.

(Sheet 2 of 2)

Field Name	Description
`sender`	The name of the sender that cannot retransmit the message. This field has datatype `TIBRVMSG_STRING`.

Diagnosis This advisory indicates an error, and reports unexpected and usually undesirable behavior.

`DELIVERY.UNAVAILABLE` advisories can indicate either of these difficulties:

- The physical network is unreliable.
- The sending program is using a time-out value that is too low.

REGISTRATION.REQUEST

Advisory

Subject Name Syntax `_RV.INFO.RVCM.REGISTRATION.REQUEST.`*subject*

Purpose A sender presents this advisory when a registration request arrives from a listener.

Remarks Rendezvous software automatically accepts registration requests—unless the sender has previously disallowed the listener. After accepting a request, Rendezvous software presents this advisory to the sending program. The advisory callback function can disallow the listener (see page 164) to revoke certified delivery and deny subsequent requests.

The *subject* element is the subject name of the certified message. The subject often consists of several elements, so the wildcard character `">"` (rather than `"*"`) is the correct way to match all subjects in this position.

Message Fields

Field Name	Description
`subject`	The subject name for which the listener requested certified delivery. It is identical to the *subject* element. This field has datatype `TIBRVMSG_STRING`.
`listener`	The name of the listener requesting certified delivery. This field has datatype `TIBRVMSG_STRING`.

REGISTRATION.CERTIFIED

Advisory

Subject Name Syntax

`_RV.INFO.RVCM.REGISTRATION.CERTIFIED.`*subject*

Purpose

A listener presents this advisory when a sender accepts its registration request for certified delivery.

Remarks

The *subject* element is the subject name for which delivery is certified. The subject often consists of several elements, so the wildcard character `">"` (rather than `"*"`) is the correct way to match all subjects in this position.

Message Fields

Field Name	Description
`subject`	The subject name for which delivery is certified. It is identical to the *subject* element. This field has datatype `TIBRVMSG_STRING`.
`sender`	The name of the sender that accepted the request for certified delivery. This field has datatype `TIBRVMSG_STRING`.
`seqno_begin`	The first sequence number for which delivery is certified from the `sender`. This field has datatype `TIBRVMSG_U64`.

REGISTRATION.NOT_CERTIFIED

Advisory

Subject Name Syntax `_RV.WARN.RVCM.REGISTRATION.NOT_CERTIFIED.`*subject*

Purpose A listener presents this advisory when a sender disallows certified delivery to the listener.

Remarks Senders can either remove or disallow a listener to cancel certified delivery to that listener. The listener presents a separate advisory for each subject for which it previously expected certified delivery.

The listener continues to receive messages on the disallowed subject, even though delivery is not certified.

The *subject* element is the subject name of the certified message. The subject often consists of several elements, so the wildcard character `">"` (rather than `"*"`) is the correct way to match all subjects in this position.

Message Fields

Field Name	Description
`subject`	The subject name for which delivery is not certified. It is identical to the *subject* element. This field has datatype `TIBRVMSG_STRING`.
`sender`	The name of the sender that canceled certified delivery. This field has datatype `TIBRVMSG_STRING`.

REGISTRATION.NO_RESPONSE

Advisory

Subject Name Syntax

`_RV.WARN.RVCM.REGISTRATION.NO_RESPONSE.`*subject*

Purpose

A listener presents this advisory when Rendezvous software receives no response to a request for certified delivery.

Remarks

This advisory indicates a network communication failure in of one of two modes:

- The request never reached the sender.
- The sender received the request, but its response never reached the listener.

In either case, delivery is not certified.

The listener continues to receive messages on the subject, even though delivery is not certified.

After several attempts receive no response, the listener stops requesting registration.

The *subject* element is the subject name for which delivery is not certified. The subject often consists of several elements, so the wildcard character `">"` (rather than `"*"`) is the correct way to match all subjects in this position.

Message Fields

Field Name	Description
`subject`	The subject name for which delivery is not certified. It is identical to the *subject* element. This field has datatype `TIBRVMSG_STRING`.
`sender`	The name of the sender that did not respond. This field has datatype `TIBRVMSG_STRING`.

Diagnostics

This advisory can indicate either network glitches, or termination of the sender process.

REGISTRATION.CLOSED

Advisory

Subject Name Syntax

`_RV.INFO.RVCM.REGISTRATION.CLOSED.`*subject*

Purpose

Cooperating senders presents this advisory in two situations:

- A listener destroys a certified listener event object.
- A sender cancels certified delivery of *subject* to a listening correspondent.

Remarks

Destroying a listening event implies that the listener program no longer requires delivery of that subject. Senders that certify delivery to the listener need not continue to do so. Each sender removes from its ledger all items associated with the closed subject and listener, and then presents this advisory to inform the sending program.

When a sender cancels certified delivery, it removes from its ledger all items associated with the canceled subject and listener, and then generates this advisory to the sending program (to trigger any operations in the callback function for the advisory).

The *subject* element is the subject name of the certified message. The subject often consists of several elements, so the wildcard character `">"` (rather than `"*"`) is the correct way to match all subjects in this position.

Message Fields

Field Name	Description
`subject`	The subject name that closed. It is identical to the *subject* element. This field has datatype `TIBRVMSG_STRING`.
`listener`	The name of the listening correspondent that no longer receives the subject. This field has datatype `TIBRVMSG_STRING`.
`seqno_last_sent`	The sequence number of the last message sent to the closed listener. This field has datatype `TIBRVMSG_U64`.
`seqno_last_confirmed`	The sequence number of the last message for which the listener confirmed delivery before closing. This field has datatype `TIBRVMSG_U64`.

REGISTRATION.DISCOVERY

Advisory

Subject Name Syntax

`_RV.INFO.RVCM.REGISTRATION.DISCOVERY.`*subject*

Purpose

A listener presents this advisory when it discovers a new (previously unfamiliar) sending CM transport.

Remarks

Discovery occurs when a listening CM transport receives a labeled message from a CM sender that is not listed in the listener's ledger. Four things happen as a result (not necessarily in this order):

- Rendezvous software queues the inbound message (but this message is not itself certified).
- Certified delivery software adds the sender's name to the listener's ledger, as a source of messages on the subject.
- The listener automatically requests certified delivery of the subject from the sender.
- The listener presents this advisory message.

The *subject* element is the subject name of the certified message. The subject often consists of several elements, so the wildcard character `">"` (rather than `"*"`) is the correct way to match all subjects in this position.

Message Fields

Field Name	Description
`subject`	The subject name of the inbound message that caused discovery. It is identical to the *subject* element. This field has datatype `TIBRVMSG_STRING`.
`sender`	The name of the new sender. This field has datatype `TIBRVMSG_STRING`.

REGISTRATION.MOVED

Advisory

Subject Name Syntax

`_RV.INFO.RVCM.REGISTRATION.MOVED.`*name*

Purpose

A CM transport presents this advisory when it processes any communication from a persistent correspondent with a familiar name, but an unfamiliar address.

Remarks

The combination of a familiar name with an unfamiliar address implies that a cooperating persistent correspondent has moved—that is, the name has moved to a new transport object (possibly in a new process, or even on a different host computer). The receiving correspondent automatically updates its ledger to reflect the new address of the sending correspondent that moved.

The *name* element is the reusable name of the relocated correspondent. The name may consist of several elements, so the wildcard character `">"` (rather than `"*"`) is the correct way to match all subjects in this position.

Message Fields

Field Name	Description
`name`	The reusable name of the correspondent that moved. It is identical to the *name* element. This field has datatype `TIBRVMSG_STRING`.

Diagnostics

Several `REGISTRATION.MOVED` advisories in rapid succession might indicate a name collision (two or more correspondents that claim the same name). This situation can cause thrashing, which defeats the benefit of certified message delivery.

REGISTRATION.COLLISION

Advisory

Subject Name Syntax

`_RV.ERROR.RVCM.REGISTRATION.COLLISION.`*name*

Purpose

A CM transport receives this advisory when it discovers a newly enabled CM transport that claims the same name.

Remarks

Name collisions can result in thrashing, which defeats the benefit of certified message delivery. We recommend that programs include a policy for reporting and resolving collisions.

The *name* element is the reusable correspondent name that is the locus of the collision. The name may consist of several elements, so the wildcard character `">"` (rather than `"*"`) is the correct way to match all subjects in this position.

Message Fields

Field Name	Description
`name`	The name that caused collision. It is identical to the *name* element. This field has datatype `TIBRVMSG_STRING`.

RELAY.CONNECTED

Advisory

Subject Name Syntax `_RV.INFO.RVCM.RELAY.CONNECTED.`*name*

Purpose A CM transport presents this advisory when it connects (or reconnects) to its designated relay agent.

Remarks Connect calls are non-blocking; they immediately return control to the program, and asynchronously attempt to connect to the relay agent. Upon making successful contact with the relay agent process, the transport presents this advisory.

The *name* element is the name of the relay agent. The name may consist of several elements, so the wildcard character `">"` (rather than `"*"`) is the correct way to match all relay agent names in this position.

Message Fields

Field Name	Description
`name`	The name of the relay agent. It is identical to the *name* element. This field has datatype `TIBRVMSG_STRING`.

See Also Connecting and Disconnecting, page 172

RELAY.DISCONNECTED

Advisory

Subject Name Syntax `_RV.INFO.RVCM.RELAY.DISCONNECTED.`*name*

Purpose A CM transport presents this advisory when it disconnects from its relay agent.

Remarks Disconnect calls are non-blocking; they immediately return control to the program, and asynchronously proceed with clean-up tasks. When those tasks are complete, the transport presents this advisory, indicating that it is safe to sever the physical network connection.

The *name* element is the name of the relay agent. The name may consist of several elements, so the wildcard character ">" (rather than "*") is the correct way to match all relay agent names in this position.

Message Fields

Field Name	Description
`name`	The name of the relay agent. It is identical to the *name* element. This field has datatype `TIBRVMSG_STRING`.

See Also Connecting and Disconnecting, page 172

QUEUE.SCHEDULER.ACTIVE

Advisory

Subject Name Syntax

_RV.INFO.RVCM.QUEUE.SCHEDULER.ACTIVE.*name*
_RV.INFO.RVCM.QUEUE.SCHEDULER.INACTIVE.*name*

Purpose

A distributed queue member presents a SCHEDULER.ACTIVE advisory when it becomes the active scheduler.

A distributed queue presents a SCHEDULER.INACTIVE advisory when it relinquishes the scheduler role, becoming a worker.

Remarks

The *name* element is the distributed queue correspondent name. The name may consist of several elements, so the wildcard character ">" (rather than "*") is the correct way to match all subjects in this position.

Message Fields

Field Name	Description
name	The distributed queue correspondent name. It is identical to the *name* element. This field has datatype TIBRVMSG_STRING.

QUEUE.SCHEDULER.OVERFLOW

Advisory

Subject Name Syntax

`_RV.WARN.RVCM.QUEUE.SCHEDULER.OVERFLOW.`*name*

Purpose

A distributed queue scheduler presents a `SCHEDULER.OVERFLOW` advisory when it discards a task because its backlog of tasks is too great and no workers are available to accept a task.

Remarks

The *name* element is the distributed queue correspondent name. The name may consist of several elements, so the wildcard character `">"` (rather than `"*"`) is the correct way to match all subjects in this position.

The scope of this advisory is the transport that created the scheduler member.

Notice that if the scheduler has not set its task backlog limit, then the queue can never overflow (and this advisory is never presented).

Message Fields

Field Name	Description
`name`	The distributed queue correspondent name of the scheduler. It is identical to the *name* element. This field has datatype `TIBRVMSG_STRING`.
`subject`	The subject name of the discarded task message. This field has datatype `TIBRVMSG_STRING`.
`sender`	The distributed queue correspondent name of the task sender. This field has datatype `TIBRVMSG_STRING`.
`seqno`	The RVCM sequence number of the discarded task message. This field has datatype `TIBRVMSG_U64`.
`reason`	The string `"Task backlog limit reached."` This field has datatype `TIBRVMSG_STRING`.

Appendix C Fault Tolerance (RVFT) Advisory Messages

Advisory messages inform programs of problematic situations.

A fault tolerance member object presents advisory messages on its transport; it does not broadcast advisories on the network.

Topics

Fault Tolerance Advisory Messages

Table 44 Rendezvous Fault Tolerance Advisory Messages

Advisory Message	Class	Page
PARAM_MISMATCH	ERROR, WARN	297
DISABLING_MEMBER	ERROR	299
TOO_MANY_ACTIVE	WARN	300
TOO_FEW_ACTIVE	WARN	302

RVFT Advisory Subject Names

Rendezvous fault tolerance software constructs the subject names of advisory messages using this template:

`_RV.`*class*`.RVFT.`*name*`.`*group*

Table 45 RVFT Advisory Subject Name Elements

Element	Description
class	The *class* element denotes the severity of the situation. • `ERROR` indicates a problem that certainly precludes fault tolerance. • `WARN` indicates an anomalous situation that might jeopardize fault tolerance. (Rendezvous fault tolerance software does not generate advisory messages with class `INFO`.)
source	The *source* element is `RVFT` for all advisories from fault tolerance software.
name	The *name* element describes the situation that the advisory reports.
group	The *group* element is the name of the fault tolerance group to which the problematic situation applies. Group names may consist of several elements, so the wildcard character `">"` (rather than `"*"`) is the correct way to match all names in this position.

RVFT Advisory Description Field

Advisory messages from Rendezvous fault tolerance software may contain a field named RVADV_DESC. If present, this field contains a character string that describes further details of the problematic situation.

For example, PARAM_MISMATCH advisory messages could result from any of several different situations in which parameters do not match. The RVADV_DESC field contains a string explaining which parameters do not match.

PARAM_MISMATCH

Advisory

Subject Name Syntax

`_RV.ERROR.RVFT.PARAM_MISMATCH.`*group*
`_RV.WARN.RVFT.PARAM_MISMATCH.`*group*

Purpose

A fault tolerance member presents these advisory messages when other group members use parameters that do not match its own corresponding parameters.

Error Advisory

`_RV.ERROR.RVFT.PARAM_MISMATCH.`*group* indicates an error—a severe situation in which parameter mismatch precludes correct fault tolerance behavior. Each group member presents this advisory message.

The `RVADV_DESC` field describes the error in more detail:

- `Active goal differs`

  Another member has a different active goal parameter than this member. All members of the group must aim for the same number of active members. Find and correct the discrepancy immediately.

- `Activation interval differs`

  Another member has a different activation interval than this member. All members of the group must use the same activation interval. Find and correct the discrepancy immediately.

Warning Advisory

`_RV.WARN.RVFT.PARAM_MISMATCH.`*group* indicates a warning—an anomalous situation in which parameter mismatch casts doubt upon correct fault tolerance behavior. The `RVADV_DESC` field describes the error in more detail:

- `Heartbeat interval differs`

  Another member has a different heartbeat interval than this member. Although it is not required that all members of the group use the same heartbeat interval, using different heartbeat intervals can result in incorrect fault tolerance behavior that is difficult to diagnose. If you are certain that the different intervals are correct, you may ignore this warning; otherwise, find and correct the discrepancy.

- `Other prepares before this member's heartbeat`

  Another member has a preparation interval that is less than or equal to this member's heartbeat interval. When this member is active, the other member's fault tolerance callback function triggers repeatedly, receiving a series of false-positive `RVFT_PREPARE_TO_ACTIVATE` hints. While this behavior is not incorrect, it can be far from optimal. If you are certain that the intervals are correct, you may ignore this warning; otherwise, find and correct the discrepancy.

- `This member prepares before other's heartbeat`

The preparation interval of this member is less than or equal to the heartbeat interval of another (external) member. When that member is active, this member's fault tolerance callback function triggers repeatedly, receiving a series of false-positive RVFT_PREPARE_TO_ACTIVATE hints. While this behavior is not incorrect, it can be far from optimal. If you are certain that the intervals are correct, you may ignore this warning; otherwise, find and correct the discrepancy.

Message Fields

Field Name	Description
RVADV_DESC	A string describing the specific situation of this advisory. This field has datatype TIBRVMSG_STRING.

See Also Fault Tolerance Concepts, page 195.

DISABLING_MEMBER

Advisory

Subject Name Syntax `_RV.ERROR.RVFT.DISABLING_MEMBER.`*group*

Purpose Rendezvous fault tolerance software sometimes detects problems that make a fault tolerance member unable to function. It automatically deactivates the affected member, disables it (by setting its weight to zero), and generates this error advisory message.

Remarks Only the disabled member presents this advisory.

The `RVADV_DESC` field always contains `NULL`.

This advisory indicates that Rendezvous fault tolerance software cannot allocate memory, publish a message or set a timer. In most cases, the primary cause is insufficient memory.

In some cases, memory is so scarce that the disabled member is unable to process the advisory message, nor even relay the advisory message to `stderr`.

For a complete discussion, see Disabling a Member on page 224.

Remedy The most reliable remedy is to arrange for the process to exit gracefully, and then restart the member. In the meantime, the disabled process is replaced by another member (presuming that other members are running). Be certain to investigate and correct the cause of the resource problem.

Message Fields

Field Name	Description
`RVADV_DESC`	A string describing the specific situation of this advisory. This field has datatype `TIBRVMSG_STRING`.

See Also Fault Tolerance Concepts, page 195
Disabling a Member, page 224

TOO_MANY_ACTIVE

Advisory

Subject Name Syntax

`_RV.WARN.RVFT.TOO_MANY_ACTIVE.`*group*

Purpose

A fault tolerance member presents this warning advisory message when it detects that too many group members are active. This situation is usually transient, and resolves itself quickly without intervention. However, if the situation persists, it might indicate problems that require attention.

Remarks

Rendezvous fault tolerance software detects the situation when it receives a heartbeat message from a member that was not already known to be active.

This warning indicates that the following conditions *all* hold simultaneously:

- The number of members broadcasting heartbeat messages is greater than the active goal parameter.
- This member is active.
- This member will not deactivate (that is, its rank indicates it should remain active).

The conclusion is that one or more other members are active that should not be active. In most cases those members quickly detect the anomaly, and deactivate. Normally the situation resolves itself within one activation interval.

Notice that a member does not receive this advisory if it is either inactive or about to deactivate.

Diagnosis

This warning can indicate any of several situations:

- A network separates into two or more disconnected parts, and then reconnects.

  Rendezvous fault tolerance software arranges for the correct number of active members on each disconnected part of the network. When the parts reconnect, the active members with the lowest rank become extraneous, and quickly deactivate. This warning indicates that a network problem occurred.

- Members have different active goal parameters.

  If member A has an active goal of one member, and B has an active goal of two members, then A and B will both become active, and A receives this complaint that too many members are active. (Both A and B also receive the `PARAM_MISMATCH` error advisory, with `Active goal differs` in the `RVADV_DESC` field.)

- Interval parameters to Rendezvous fault tolerance software are too short compared to the speed of the hardware clock and operating system services.

  See Step 4: Choose the Intervals on page 235.

- The active member did not send timely heartbeat messages. For example, a callback function blocked, or did not return promptly, delaying the heartbeat messages.

  See Ensure Timely Event Processing on page 216.

Message Fields

Field Name	Description
`RVADV_DESC`	A string describing the specific situation of this advisory. This field has datatype `TIBRVMSG_STRING`.

See Also Fault Tolerance Concepts, page 195.

TOO_FEW_ACTIVE

Advisory

Subject Name Syntax

`_RV.WARN.RVFT.TOO_FEW_ACTIVE.`*group*

Purpose

A fault tolerance member presents this warning advisory message when it detects that too few group members are active.

This situation is usually transient, and resolves itself quickly without intervention. However, if the situation persists, it might indicate problems that require attention.

Remarks

This warning indicates that the following conditions *all* hold simultaneously:

- This member is inactive.
- This member will not activate (that is, its rank indicates it should remain inactive).
- Nevertheless, the number of members broadcasting heartbeat messages is still less than the active goal parameter.

Notice that a member does not receive this advisory if it is either active or about to activate.

Diagnosis

This warning can indicate any of several situations:

- Network connectivity is erratic. The network repeatedly separates into two or more disconnected parts, and then reconnects.

  Notify your network administrator immediately.
- Member processes terminate immediately upon activation. New members activate to replace them, resulting in a cascade of failures.

  Possible causes include errors in program code, and transient network overload.

Message Fields

Field Name	Description
`RVADV_DESC`	A string describing the specific situation of this advisory. This field has datatype `TIBRVMSG_STRING`.

See Also

Fault Tolerance Concepts, page 195.

Index