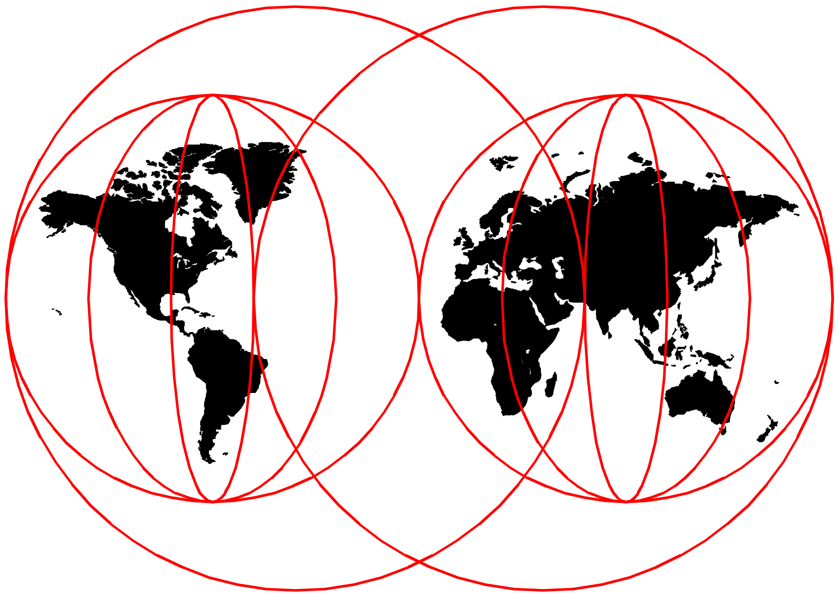
IBM

RSCT Group Services: Programming Cluster Applications

Yoshimichi Kosuge, Christoph Krafft



International Technical Support Organization

www.redbooks.ibm.com

SG24-5523-00

SG24-5523-00

International Technical Support Organization

RSCT Group Services: Programming Cluster Applications

April 2000

Take Note!

Before using this information and the product it supports, be sure to read the general information in Appendix E, “Special notices” on page 281.

First Edition (April 2000)

This edition applies to Version 3 Release 1 of the IBM Parallel System Support Programs for AIX (PSSP) Licensed Program, program number 5765-D51, and to all subsequent releases and modifications, until otherwise indicated in new editions, and The Enhanced Scalability feature of Version 4 Release 3 of the IBM High Availability Cluster Multi-Processing for AIX (HACMP) Licensed Program, program number 5765-D28, and to all subsequent releases and modifications.

This document created or updated on April 27, 2000.

Comments may be addressed to:
IBM Corporation, International Technical Support Organization
Dept. JN9B Mail Station P099
2455 South Road
Poughkeepsie, NY 12601-5400

When you send information to IBM, you grant IBM a non-exclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

Contents

Figures

Tables

Preface

Group Services is a distributed subsystem of the IBM Reliable Scalable Cluster Technology (RSCT) on the RS/6000 system. If you are writing a new application or considering updating an existing application, Group Services may help your application improve its availability in a number of ways.

Group Services encapsulates a collection of software abstractions that are commonly used in the design of highly-available systems. By using Group Services abstractions through the Group Services shared library, you do not have to develop your own mechanism for a highly-available application. This is important because such mechanisms tend to be complex, error-prone, and expensive to duplicate.

This redbook provides a sample application program to help you understand how to program Group Services applications.

The team that wrote this redbook

This redbook was produced by a team of specialists from around the world working at the International Technical Support Organization, Poughkeepsie Center.

Yoshimichi Kosuge is an IBM RS/6000 SP project leader at the International Technical Support Organization, Poughkeepsie Center. Since joining the ITSO in 1998, he has been involved in writing redbooks and teaching IBM classes on all areas of the RS/6000 SP system.

Christoph Krafft is an SP specialist at the IBM Global Services division in Germany. He has four years of experience working with the SP system. Christoph holds a degree in Technical Computer Science from the Berufsakademie in Stuttgart, Germany.

Special thanks to the following people for their invaluable contributions to this project:

Peter Badovinatz, Myung Bae, Marcos Novaes, Ji-Fang Zhang
IBM Poughkeepsie

Milos Radosavljevic
IBM ITSO, Austin Center

Comments welcome

Your comments are important to us!

We want our Redbooks to be as helpful as possible. Please send us your comments about this or other Redbooks in one of the following ways:

- Fax the evaluation form found in “IBM Redbooks review” on page 301 to the fax number shown on the form.
- Use the online evaluation form found at `http://www.redbooks.ibm.com/`
- Send your comments in an Internet note to `redbook@us.ibm.com`

Part 1. Group Services concepts

Chapter 1. Introduction

If you are writing a new application or are considering updating an existing application, Group Services may help the application improve its availability in a number of ways. Group Services provides the following services:

- Coordinate among peer processes.
- Create a message board to display the state of your application to other applications.
- Exchange a message among applications.
- Subscribe to changes in the state of other applications.

1.1 What Group Services provides

There are several advantages to using Group Services with your applications.

Peer process synchronization

An application can be considered distributed when it deploys multiple processes that may run in more than one computer and that share some level of cooperation. Usually, it is very difficult to write such applications due to the high level of complexity in carrying out synchronizations among distributed applications. This complexity is increased because, at any given time, synchronization among the applications can be interrupted, or some applications could become unavailable leaving the synchronization process in an inconsistent state. The recovery associated with this type of situation is typically very complex.

The Group Services provides distributed applications with a facility with which the application can execute atomic actions. An action is considered atomic if the action is done through a coordination mechanism provided by the Group Services.

Message board

Another complexity is sharing information. Distributed applications must share information to cooperate with one another. The information must be identical for all the applications and can be changed at any time.

The Group Services provides distributed applications with a general purpose message board facility for coordinating information of an application. All the distributed applications can change the contents and share the identical contents.

Message broadcast

The message board is non-volatile information, that is, the information can be retrieved later. However, there can be information that must be shared but does not necessarily have to be stored somewhere. This also becomes one of the complexities because it is not just a matter of sending messages. An application needs to check other applications that are currently running. It also needs their destination address.

The Group Services provides broadcast capability. It automatically broadcasts messages to all the applications currently running. Therefore, it becomes possible to write a distributed application without using any other communications protocol, such as TCP/IP. An application can be developed relying only on the Group Services communications facility.

Monitoring application

The Group Services offers very powerful synchronization, message board, and message broadcast mechanisms among distributed applications. However, some applications may not need these mechanisms. Instead, they want to monitor ongoing activities among distributed applications.

If this is the case, Group Services provides a monitoring mechanism. Using this mechanism, an application can easily implement its monitoring facility without complexity.

1.2 Solutions

Group Services encapsulates a collection of software abstractions that are commonly used in the design of highly-available systems.

By using the Group Services abstractions through the Group Services shared library, application developers do not have to develop their own synchronization, message board, or message broadcast mechanisms. This is important because such mechanisms tend to be complex, error-prone, and expensive to duplicate.

The Group Services facility greatly simplifies the writing of a distributed application. The utilization of the Group Services facility has cut months of development work, which would otherwise be spent reinventing the facilities that Group Services has.

Chapter 2. Boundaries and components

When your application uses the services provided by the Group Services, you need to consider boundaries and components of the Group Services. Services are available within boundaries. Components are required to conform to the Group Services environment.

2.1 Boundaries

Group Services provides applications with its services for a certain scope. An application may consist of multiple processes that run on multiple RS/6000 hardware. Therefore, when the application uses the services provided by the Group Services, it must consider boundaries in which the application can use them.

2.1.1 Node

A node is a piece of RS/6000 hardware on which the AIX operating system executes exclusively. For example, a control workstation or each SP node in an RS/6000 SP system is considered a node.

2.1.2 Domain

A domain is the collection of nodes on which the RS/6000 Cluster Technology is executing. A domain may not be exclusive. A node may be contained in multiple domains.

There are two types of domains for the Group Services: Group Services PSSP domains and Group Services HACMP/ES domains.

2.1.2.1 Group Services PSSP domain

A *Group Services PSSP domain* includes a control workstation and the set of SP nodes defined to be within an SP partition. This means that a control workstation can be within multiple Group Services PSSP domains. An application wishing to use Group Services on the control workstation or on an SP node must set (or not set) the following environment variables to ensure that it can initialize with the proper Group Services domain:

- HA_DOMAIN_NAME
- HA_GS_SUBSYS

This must be done on an SP node (which can only be in one Group Services PSSP domain) as well as on the control workstation (which will be in multiple Group Services PSSP domains if there are multiple defined SP partitions).

2.1.2.2 Group Services HACMP/ES domain

If HACMP/ES is installed on a node, that node will be part of *a Group Services HACMP/ES domain*. The Group Services HACMP/ES domain consists of all nodes that are part of the HACMP/ES cluster. An application wishing to use the Group Services must set the following environment variables to ensure that it can initialize with the proper Group Services domain:

- HA_DOMAIN_NAME
- HA_GS_SUBSYS

This must be done on all nodes of the HACMP/ES cluster.

For more information about setting the environment variables, refer to Section 4.1, “Choosing a domain” on page 45.

2.2 Components

An application that utilizes the Group Services subsystem shares groups. Each group that is maintained by the Group Services subsystem is uniquely named.

Any authorized process in a Group Services domain may create a new group or ask to become a member of a group. This request is called a *join request* or *joining the group*. If the join request is successful, the process becomes *a provider* for the group.

Any authorized process in a Group Services domain can ask to monitor a group. This request is called *a subscribe request* or *subscribing the group*. If the subscribe request is successful, the process becomes *a subscriber* for the group.

2.2.1 Group Services subsystem

The Group Services subsystem consists of the Group Services daemons. Each node requires at least one Group Services daemon executing. If a node is contained in multiple domains, the same number of Group Services daemons must be executing on a node.

2.2.2 Groups

A Group Services application defines one or more group names that are known to all of the processes that are part of the application. During initialization of the application, as each process in the application starts up, it asks to join the group. On receipt of the first join request, the Group Services creates the group. The subsequent join requests result in new providers joining the group.

A group has the following information that is unique to the group:

- The group attributes
- The membership list
- The group state value

In addition to these three piece of information, a group can share volatile information, such as the provider-broadcast message.

When the last provider in a group leaves the group voluntarily or involuntarily, the Group Services deletes the group. If a group is deleted, the group attributes, the membership list, and the group state value are lost.

For more information on joining or leaving a group, refer to Chapter 5, “Proposing protocols” on page 51.

2.2.2.1 The group attributes

The first join request creates a group and defines its *group attributes*. All subsequent requests to join this group also include the group attribute, and their group attributes must match the group’s established attributes. Otherwise, the subsequent join request is rejected.

The group attributes represent characteristics and control behaviors of a group as follows:

- The Group Services library version
- The application-specified group name and version
- The source group name for the group
- The number of voting phases
- The voting time limit
- The default voting value
- The batching protocols control
- The deactivate-on-failure control

All of the attributes can be changed dynamically except the following two attributes:

- The application-specified group name

- The source group name for the group

If an application needs to change one or both of these two attributes, it must dissolve the group and create the group with new attributes.

The group attributes are maintained by the group attributes block, which has the definition shown in Figure 1.

```
typedef struct {
    short                   gs_version;
    short                   gs_sizeof_group_attributes;
    short                   gs_client_version;
    ha_gs_batch_ctrl_t      gs_batch_control;
    ha_gs_num_phases_t      gs_num_phases;
    ha_gs_num_phases_t      gs_source_reflection_num_phases;
    ha_gs_vote_value_t      gs_group_default_vote;
    ha_gs_merge_ctrl_t      gs_merge_control;
    ha_gs_time_limit_t      gs_time_limit;
    ha_gs_time_limit_t      gs_source_reflection_time_limit;
    ha_gs_group_name_t      gs_group_name;
    ha_gs_group_name_t      gs_source_group_name;
} ha_gs_group_attributes_t;
```

Figure 1. The group attributes block

Each field contains the following information:

gs_version
This field contains the version level of the Group Services shared library. It is set by the Group Services subsystem.

gs_sizeof_group_attributes
This field contains the size of the group attributes block.

gs_client_version
This field contains a user-defined version code.

gs_batch_control
This field controls the batching of multiple join, failure leave, or cast-out protocols. In addition, this field controls a deactivate-on-failure facility.
For more information on batching protocols, refer to Section 3.1.6, “Batching protocols” on page 28. For more information about the deactivate-on-failure facility, refer to Section 3.3, “Deactivate-on-failure facility” on page 33.

gs_num_phases
This field specifies whether join, failure leave, and cast-out protocols are to be one-phase or n-phase protocols.

For more information on the number of phases, refer to Section 3.1.1, “Proposal phase” on page 19.

gs_source_reflection_num_phases
This field specifies whether the source-state reflection protocol is to be a one-phase or n-phase protocol. If no gs_source_group_name is given, this field is ignored.
For more information on the number of phases, refer to Section 3.1.1, “Proposal phase” on page 19. For the source-state reflection protocol, refer to Section 5.11, “Source-state reflection protocol” on page 103.

gs_group_default_vote
This field contains the default vote value to be used for the providers in this group. It can take on a value of HA_GS_VOTE_APPROVE to approve or HA_GS_VOTE_REJECT to reject.
For more information on the default vote value, refer to Section 3.1.2.3, “Default vote value” on page 22.

gs_merge_control
This field must be set to a value of HA_GS_DISSOLVE_MERGE.

gs_time_limit
This field contains the voting time limit in seconds. This is the number of seconds within which each provider must register its vote for each voting phase of an n-phase join, failure leave, and cast-out protocol. If the field is set to a value of 0, no limit is enforced.
For more information on the voting time limit, refer to Section 3.1.2.2, “Voting time limit” on page 21.

gs_source_reflection_time_limit
This field contains the voting time limit in seconds. This is the number of seconds within which each provider must register its vote for each voting phase of an n-phase source-state reflection protocol. If no gs_source_group_name is specified, or if it is specified and the gs_source_reflection_phases field contains a value of HA_GS_1_PHASE, this field is ignored.
For more information on the voting time limit, refer to Section 3.1.2.2, “Voting time limit” on page 21. For information about the source-state reflection protocol, refer to Section 5.11, “Source-state reflection protocol” on page 103.

gs_group_name
This field points to a string that contains the name of the group. Its

maximum length is 32 bytes, as defined by the HA_GS_MAX_GROUP_NAME_LENGTH constant.

gs_source_group_name

This field points to a string that contains the name of the source-group for this group. If there is no source-group, this field should be null.
For more information on source-target facility, refer to Section 3.4, "Source-target facility" on page 36.

2.2.2.2 The membership list

The membership list of a group is the list of providers in the group. Each provider is identified by a provider identifier. The Group Services subsystem maintains the list in the following order: The oldest provider is at the head of the list, and the youngest is at the end. All of the group's providers and subscribers see the same ordering of the list.

The Group Services subsystem updates the membership list when a provider joins or leaves a group.

The membership list is maintained by the membership information block. It has the definition shown in Figure 2.

```
typedef struct {
   unsigned int           gs_count;
   ha_gs_provider_t       *gs_providers;
} ha_gs_membership_t;
```

Figure 2. The membership information block

Each field contains the following information:

gs_count

This field contains the number of providers in the list.

gs_providers

This field contains a pointer to the first provider in the membership list. Each provider is described by a provider information block. For the provider information block, refer to Figure 8 on page 14.

2.2.2.3 The group state value

The group state value is defined by the application that is using the Group Services and is controlled by the providers in a way that is meaningful to the application. It is a byte field whose length may vary between 1 and 256 bytes. The state value is not interpreted by the Group Services subsystem.

The group state value is maintained by the group state value information block. It has the definition shown in Figure 3 on page 11.

```
typedef struct {
   int                    gs_length;
   char                   *gs_state;
} ha_gs_state_value_t;
```

Figure 3. The group state value information block

Each field contains the following information:

gs_length
: This field contains the length, in bytes, of the group state value. It must be a value between 1 and 256.

gs_state
: This field points to a buffer that contains the actual group state value bytes.

2.2.2.4 Provider-broadcast message

The Group Services provides currently joining providers with a volatile message as shared information. Providers in a group are allowed to send a message to all the providers in the group. This information is not stored in any place; therefore, it is a responsibility of an application to store it for later reference.

Provider-broadcast messages are defined by the application that is using the Group Services and are controlled by the providers in a way that is meaningful to the application. Provider-broadcast messages are not interpreted by the Group Services.

The provider-broadcast message is maintained by the provider-broadcast message block. It has the definition shown in Figure 4.

```
typedef struct {
   int                    gs_length;
   char                   *gs_message;
} ha_gs_provider_message_t;
```

Figure 4. The provider-broadcast message block

Each field contains the following information:

gs_length

This field contains the length, in bytes, of the message to be broadcast to providers. It must be a value between 1 and 2048.

gs_message

This field points to a buffer that contains the message.

2.2.3 Group Services client

When an AIX process initializes itself with the Group Services, it becomes *a Group Services client* (*GS client*). When a GS client joins a group, a GS client becomes *a provider*. When a GS client wishes to stop being a provider, it *leaves a group*. When a GS client *subscribes* a group, a GS client becomes *a subscriber*. When a GS client wishes to stop being a subscriber, it *unsubscribes a group*.

The term GS client is used to refer to both providers and subscribers. A process that has initialized with the Group Services but has not yet become a provider or subscriber is also referred to as a GS client.

There is not just one choice - between provider or subscriber; a GS client can become a group's provider or subscriber, and a GS client can also become multiple groups' provider or subscriber. These relationships are illustrated in Figure 5 on page 13.

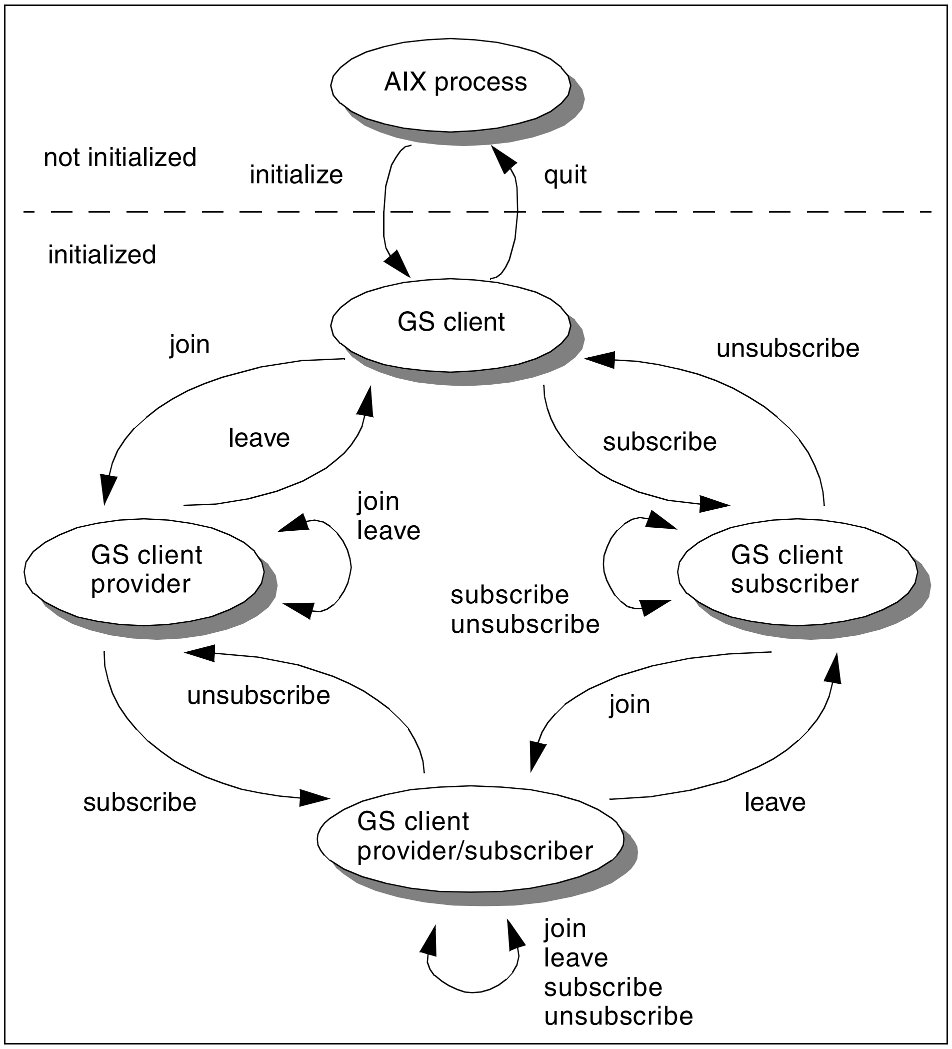


Figure 5. GS client, provider, and subscriber

2.2.3.1 Descriptor

When a process initializes itself with the Group Services, it receives a descriptor that is a socket file descriptor to communicate with the Group Services daemon. The descriptor is defined as shown in Figure 6.

```
typedef int ha_gs_descriptor_t;
```

Figure 6. Descriptor

2.2.4 Providers

A provider has the following information to identify itself:

- Provider token
- Provider ID

  The provider ID includes the following information:

  - An application-defined instance number
  - The node number of the node on which the provider is executing

2.2.4.1 Provider token

When a GS client joins a group, it receives a token that identifies the membership of the GS client in the group as a provider. The provider token is defined as shown in Figure 7.

```
typedef int ha_gs_token_t
```

Figure 7. The provider token

2.2.4.2 Provider information block

The provider information block identifies each provider to the other providers in a group. It contains an application-defined instance number and the number of the node on which the provider is executing. The provider information block is defined as shown in Figure 8.

```
typedef union {
    struct {
        short                   _gs_instance_number;
        short                   _gs_node_number;
    } _gs_provider_info;
    int                         gs_provider_id;
} ha_gs_provider_t;
```

Figure 8. The provider information block

Each field contains the following information:

_gs_instance_number

This field contains the instance number of the provider. This instance number is specified by the provider when it joins a group and must be unique for each provider on a single node within the group.

When Group Services itself acts as a provider, this field contains a value of HA_GS_instance_number. Its value is -1.

_gs_node_number

This field contains the node number of the provider. This value is assigned by Group Services.

In the case of Group Services PSSP domain, 0 is assigned to a control workstation, 1 is assigned to node 1, and so on.

In the case of Group Services HACMP/ES domain, a node number is assigned during cluster configuration. The value is assigned automatically in alphanumeric order during cluster configuration. If you add a node to an existing running cluster, it gets the lowest free node number. You cannot assign a specific value. HACMP/ES calls this node number the *handle value* and stores it in the handle attribute of the HACMPcluster GODM class. For more information, refer to Chapter 3, “Component Design”, of the *HACMP Enhanced Scalability Handbook*, SG24-5328.

When the Group Services subsystem itself is acting as a provider, this field contains a value of HA_GS_node_number. Its value is -1.

gs_provider_id

This field contains the _gs_instance_number and the _gs_node_number in a single word.

Note

The following definitions are available for convenience of programming:

```
#define  gs_node_number       _gs_provider_info._gs_node_number
#define  gs_instance_number   _gs_provider_info._gs_instance_number
```

2.2.5 Subscriber

A subscriber has only a subscriber token to identify itself.

2.2.5.1 Subscriber token

When a GS client subscribes a group, it receives a token that identifies the membership of the GS client in the group as a subscriber. The subscriber token is defined as the same as the provider token shown in Figure 7 on page 14.

2.3 Relationship between boundaries and components

The previous sections introduced the boundaries and components of Group Services. The following sections describe their relationship by using a sample Group Services configuration shown in Figure 9 on page 16.

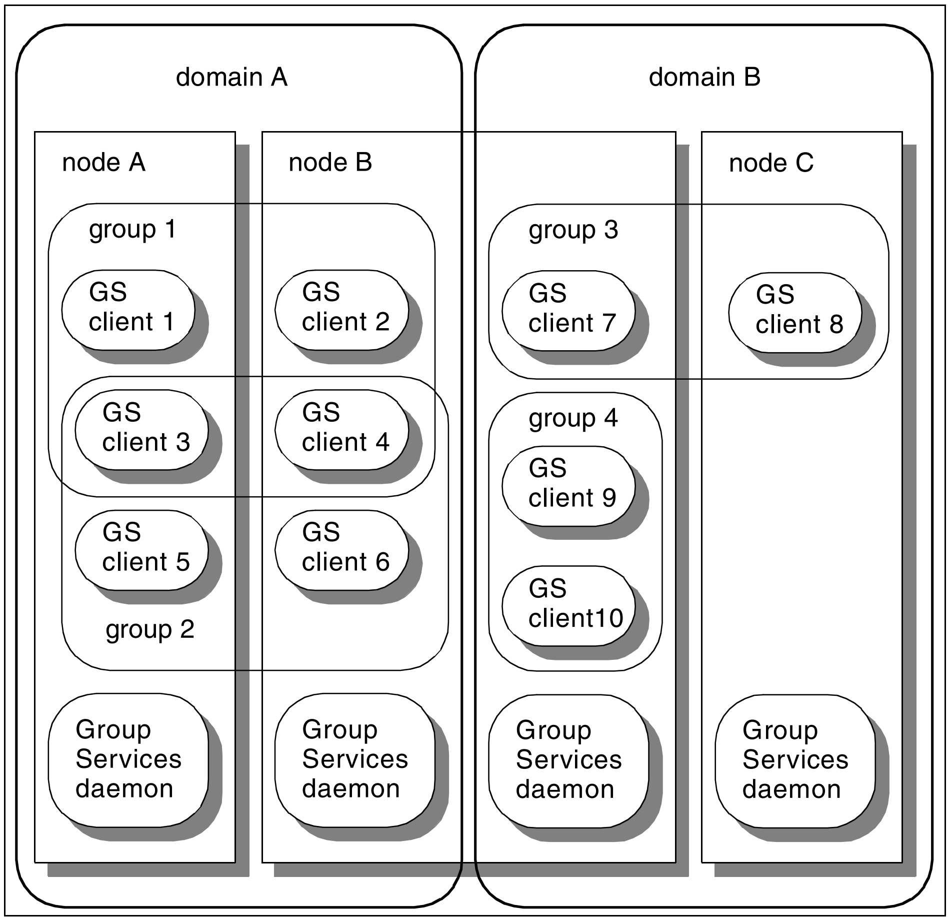


Figure 9. Boundaries and components

2.3.1 Domains, nodes, and Group Services daemons

The following relationship exists between domains, nodes, and Group Services daemons:

- One domain can contain multiple nodes. For example, domain A contains nodes A and B; similarly, domain B contains nodes B and C.
- One node can be contained in multiple domains. For example, node B is contained in domains A and B.
- The number of Group Services daemons running on a node must be the same as the number of domains in which the node is contained. For

example, node B is contained in two domains: A and B. Therefore, it requires two Group Services daemons running on node B.

2.3.2 Domains and groups

The following relationship exists between domains and groups:

- One domain can contain multiple groups. For example, domain A contains the groups 1 and 2; similarly, domain B contains groups 3 and 4.
- One group cannot be contained in multiple domains. For example, groups 1 or 2 cannot be contained in domain B; similarly, groups 3 or 4 cannot be contained in domain A.

2.3.3 Groups and GS clients

There is the following relationship between groups and GS clients:

- One GS client can be contained in multiple groups if all of them are contained in the same domain. For example, the GS client 3 or 4 is contained in both group 1 and group 2 because both group 1 and group 2 are contained in one domain domain A. However, clients 1 through 6 cannot be contained in groups 3 or 4 because groups 3 and 4 are contained in another domain, domain B.
- One GS client can be contained in multiple groups as a provider. For example, GS client 3 or 4 can be contained in group 1 as a provider and in group 2 as a provider also.
- One GS client can be contained in multiple groups as a subscriber. For example, GS clients 3 or 4 can be contained in group 1 as subscribers and in group 2 as a subscriber also.
- One GS client can be contained in some multiple groups as a provider and in other multiple groups as a subscriber. For example, GS clients 3 or 4 can be contained in group 1 as providers and in group 2 as subscribers.

Chapter 3. Protocols and facilities

A group is formed by the Group Services subsystem and providers. In addition, if necessary, subscribers are included. As long as Group Services itself and providers do not want to change the group attributes, the membership list, or the group state value or broadcast messages, there is no activity in a group.

However, when they do, Group Services provides a mechanism that coordinates these activities between them. This mechanism is called a *protocol*.

To make a protocol more powerful and flexible, the Group Services provides some additional facilities.

3.1 Protocols

A protocol has one or more phases, and each phase is categorized as one or two of the following phase types:

- Proposal phase
- Voting phase
- Commit phase

A protocol starts its first phase as a proposal phase to propose a protocol. Then, it may or may not have voting phases to approve or reject the proposed protocol. Finally, the protocol ends its last phase as a commit phase to report the result of the proposed protocol.

3.1.1 Proposal phase

In a proposal phase, a protocol is proposed by either one of the following:

- The Group Services
- A provider including a GS client that is being initialized with the Group Services; however, it has not been a provider yet.

Each protocol proposal indicates whether it is one of the following protocols:

- A one-phase protocol
- An n-phase protocol

A one-phase protocol is approved automatically, and a result of the protocol is notified immediately. Therefore, both the proposal phase and the commit phase are handled by the same phase. Voting phases are not included for this protocol.

An n-phase protocol has one or more voting phases. A result of the protocol is notified in the last voting phase. Therefore, both the voting phase and the commit phase are handled by the same phase.

The number of phases is defined by the ha_gs_num_phases_t type as shown in Figure 10.

```
typedef enum {
    HA_GS_1_PHASE            = 0x0001,
    HA_GS_N_PHASE            = 0x0002
} ha_gs_num_phases_t;
```

Figure 10. The ha_gs_num_phases_t type

Each value indicates the following meanings:

HA_GS_1_PHASE	This value indicates that the protocols are to be one-phase protocols.
HA_GS_N_PHASE	This value indicates that the protocols are to be n-phase protocols.

3.1.2 Voting phase

If a protocol is an n-phase protocol, it has voting phases. The providers in the group are required to vote on one of the following values:

Approve	The provider approves the proposed protocol.
Continue	The provider neither approves nor rejects the proposed protocol at this time; however it wants to continue to another voting phase.
Reject	The provider rejects the proposed protocol.

The Group Services itself could be involved in voting the proposed protocol, if necessary.

The voting values are defined by the ha_gs_vote_value_t type as shown in Figure 11 on page 21.

```
typedef enum {
    HA_GS_NULL_VOTE,
    HA_GS_VOTE_APPROVE,
    HA_GS_VOTE_CONTINUE,
    HA_GS_VOTE_REJECT
} ha_gs_vote_value_t;
```

Figure 11. The ha_gs_vote_value_t type

The values have the following meanings:

HA_GS_NULL_VOTE
: This value indicates a null vote. It keeps the default vote at its previous value. For more information, refer to Section 5.12, “Voting on proposed protocol” on page 106.

HA_GS_VOTE_APPROVE
: This value approves the proposal.

HA_GS_VOTE_CONTINUE
: This value continues to another voting phase.

HA_GS_VOTE_REJECT
: This value rejects the proposal.

Each voting phase can have one of the following outcomes:

- The proposed protocol is approved if every provider votes to approve the proposal. This approval terminates the protocol.
- The proposed protocol is rejected if at least one provider votes to reject the proposal. This rejection terminates the protocol.
- The protocol continues for another round if no provider votes to reject, and at least one provider votes to continue.

3.1.2.1 Barrier synchronization

An n-phase protocol is a mechanism that allows barrier synchronization. All providers in the group involved in the protocol proposal must arrive at the barrier (that is, they must submit a vote) before the protocol can proceed to the next phase. This guarantees that the group remains synchronized during the protocol. A provider’s arrival at a barrier is signalled by its submission of a vote to approve, continue, or reject the proposal.

3.1.2.2 Voting time limit

The voting phase has a voting time limit. All the providers are required to vote until this time limit expires. This allows the providers to determine if their providers are not responding quickly enough during voting phases.

When the Group Services has sent a notification to providers for a vote, it sets a timer. If the Group Services has not received a voting response from the provider by the time the time limit expires, it assumes that the provider is not going to respond and applies the group's default vote for this provider. The default vote applies only to the currently-running protocol. If the provider votes later, the vote is ignored, and the provider is given an error code that indicates that the time limit was exceeded.

The Group Services notifies the providers that a default vote was applied because the time limit was exceeded; however, it does not, at this moment, notify them which providers exceeded the time limit. If the default vote value causes the protocol to be approved or rejected, the Group Services delivers an announcement notification to the providers that lists the providers that exceeded the time limit. Group Services takes no further action. However, a provider may propose a protocol to remove any providers that exceeded the time limit if appropriate.

The voting time limit is also used to time the execution of deactivate scripts during failure leave, expel, or cast-out protocols. For more information on these protocols, refer to Section 5.3, "Failure leave protocol" on page 60, Section 5.8, "Expel protocol" on page 81, or Section 5.10, "Cast-out protocol" on page 95.

The voting time limit is defined in seconds by the ha_gs_time_limit_t type as shown in Figure 12.

```
typedef unsigned short ha_gs_time_limit_t;
```

Figure 12. The ha_gs_time_limit_t type

3.1.2.3 Default vote value

By default, the default vote value is reject. However, a provider can set the default vote value to approve when it creates a group or after creating the group. The Group Services does not permit a default vote value to continue, because it could lead to a non-terminating protocol.

The default vote value is registered to the gs_group_default_vote field in the group attributes block as shown in Figure 1 on page 8. The default vote value is defined by the ha_gs_vote_value_t type shown in Figure 11 on page 21.

3.1.3 Commit phase

For the last phase, a protocol has the commit phase. In this phase, GS clients receive one of the following notifications:

Protocol approved notification

This notification is sent to the providers of a group to indicate that a proposed protocol has been approved. It is also sent to the subscribers of the group.
Note that a protocol approved notification is sent as the first and only notification for a one-phase protocol.

Protocol rejected notification

This notification is sent to the providers of a group to indicate that a proposed protocol has been rejected. Subscribers are not notified when proposed protocols are rejected.

For more information on these notifications, refer to Section 7.6, “Protocol approved notification” on page 143 or Section 7.7, “Protocol rejected notification” on page 145.

In addition to these notifications, a GS client might receive the following notification:

Announcement notification

This notification is sent to the providers of a group to announce an item of interest within the group. They include warnings that individual providers have not voted within the time limit or responded to a responsiveness check.

For more information on an announcement notification, refer to Section 7.8, “Announcement notification” on page 146.

3.1.4 Protocol flows

This section provides general protocol flows. They are categorized by the number of protocol phases and by what a protocol is proposed. To be precise, protocol flows depend on each case and condition. For more detailed information on each protocol flow, refer to Chapter 5, “Proposing protocols” on page 51.

3.1.4.1 One-phase protocol proposed by a provider

The protocol flow of a one-phase protocol proposed by a provider is illustrated in Figure 13 on page 24.

When a provider wants to take an action to the group, it proposes a protocol to the Group Services (①). The protocol proposal includes information about how it wants to take an action to the group. The protocol is approved automatically. Then, the Group Services submits a notification to providers and subscribers (②). The notification contains information about the approved protocol.

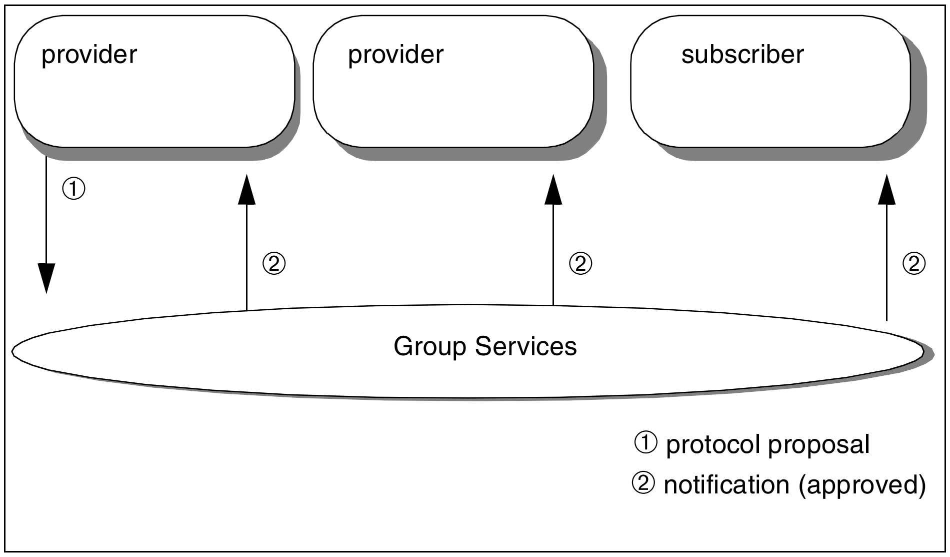


Figure 13. One-phase protocol proposed by a provider

3.1.4.2 N-phase protocol proposed by a provider

The protocol flow of an n-phase protocol proposed by a provider is illustrated in Figure 14 on page 25.

In the case of a one-phase protocol, a proposed protocol is approved automatically. This may not be appropriate for some situations. There could be a provider that does not want to approve the proposed protocol or wants to delay the decision. If this is the case, a provider can propose a protocol as an n-phase protocol (①).

When the Group Services subsystem receives the protocol proposal, it submits a notification to providers (②). Upon receiving this notification, providers are required to vote to approve, reject, or continue the proposed protocol (③). Subscribers do not receive this notification.

The Group Services does not proceed with the protocol until all providers have complete their voting or the voting time limit expires. This phase is called a *barrier synchronization voting phase*.

The voting phase will be repeated until the proposed protocol is approved or rejected.

If the proposed protocol is approved, the Group Services submits a notification to providers and subscribers (④). This notification contains information about the approved protocol. If the proposed protocol is rejected, the Group Services submits a notification to providers only (⑤). This notification contains information about the rejected protocol.

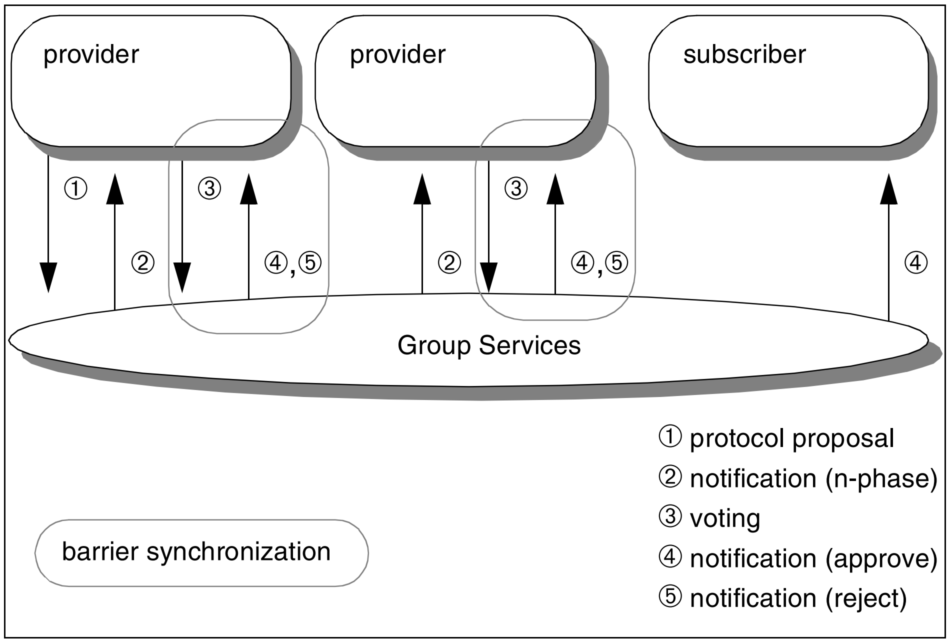


Figure 14. N-phase protocol proposed by a provider

3.1.4.3 One-phase protocol proposed by the Group Services

It is not only providers that can propose protocols; Group Services can propose protocols also.

The protocol flow of a one-phase protocol proposed by Group Services is illustrated in Figure 15 on page 26.

When the Group Services wants to take an action to the group, it proposes a protocol to itself and the protocol is approved automatically. Then, the Group Services submits a notification to providers and subscribers (①). The notification contains information about the approved protocol.

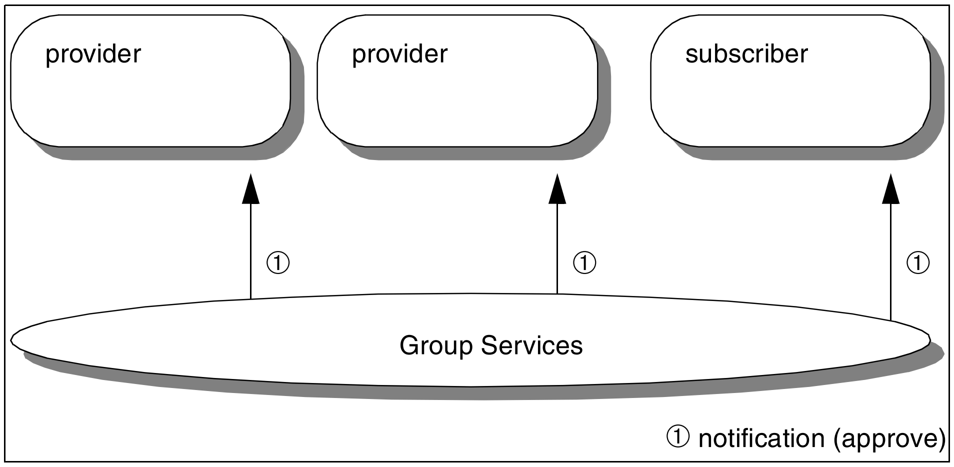


Figure 15. One-phase protocol proposed by Group Services

3.1.4.4 N-phase protocol proposed by the Group Services

The protocol flow of an n-phase protocol proposed by Group Services is illustrated in Figure 16 on page 27.

When Group Services wants to take an action to the group, it proposes a protocol as an n-phase protocol and submits a notification to providers (①). Upon receiving this notification, providers are required to vote to approve, reject, or continue on the proposed protocol (②). Subscribers do not receive this notification.

Group Services does not proceed with the protocol until all providers have completed their voting or the voting time limit has expired.

The voting phase will be repeated until the proposed protocol is approved or rejected.

If the proposed protocol is approved, Group Services submits a notification to providers and subscribers (③). This notification contains the information about the approved protocol. If the proposed protocol is rejected, the Group Services submits a notification to providers only (④). This notification contains information about the rejected protocol.

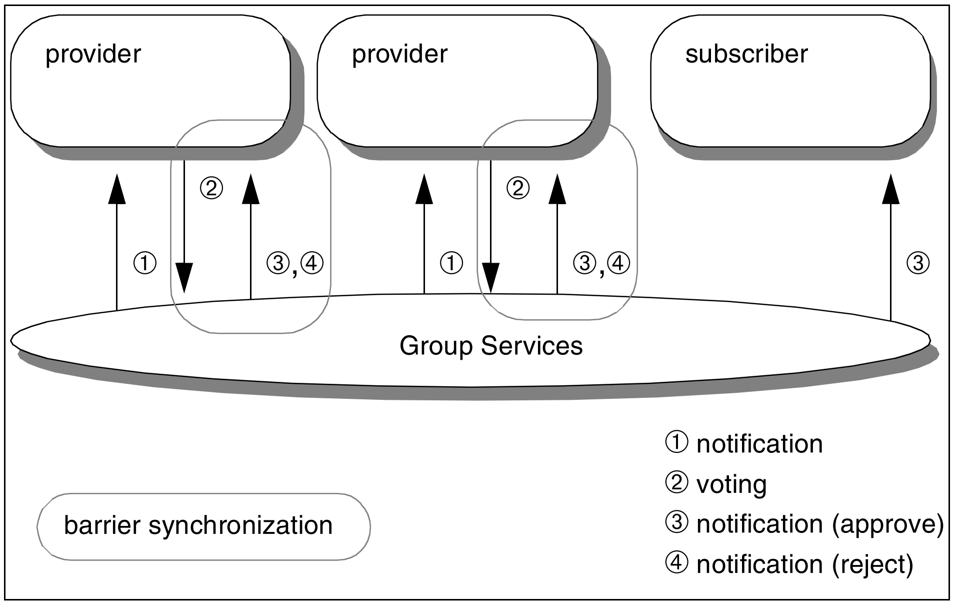


Figure 16. N-phase protocol proposed by Group Services

3.1.5 Serializing protocols

It is possible for a protocol to be proposed while another protocol is being executed or for multiple providers to propose protocols at the same time. In either case, protocol proposals are serialized, and only one proposal is allowed to be executed for a group. The other protocol proposals are returned with synchronous or asynchronous errors to the providers that proposed the protocols. It is the responsibility of a provider that receives a returned proposal to resubmit it for execution if appropriate. For more information on synchronous or asynchronous errors, refer to Section 8.1, “Synchronous/asynchronous errors” on page 153.

As an exception, the following protocols are queued to be proposed later when the running protocol completes:

- A join protocol
- A protocol proposed by the Group Services (a failure leave protocol, a cast-out protocol, and a source-state reflection protocol)

A protocol serialization is illustrated in Figure 17 on page 28. If no protocol is currently running, a proposed protocol is executed immediately (①). If another protocol is currently running, a protocol proposal is returned (②). However, exceptional protocol proposals are queued to be proposed later (③). If these protocol proposals are queued, even if a currently-running protocol has completed, another protocol proposal is returned (④).

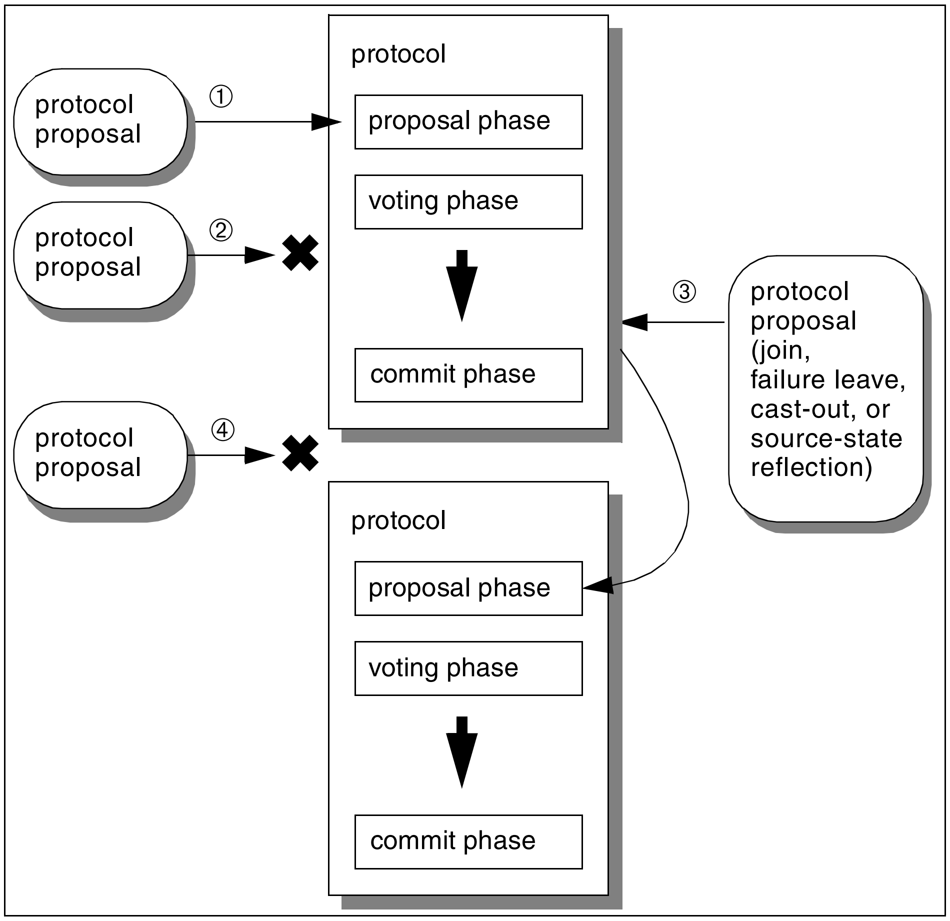


Figure 17. Protocol serialization

3.1.6 Batching protocols

During group initialization, when all of the providers are joining their groups, each join proposal requires the execution of a separate protocol. Similarly, during system shutdown, when all of the providers are leaving their groups, each leave proposal requires the execution of a separate protocol. To decrease the load on the system, Group Services provides a mechanism for batching protocols.

There is always a lag time between a protocol proposal and the actual execution of that proposed protocol. The lag time allows Group Services to batch multiple failure leave protocol proposals. In this case, Group Services collects all of the failure leave protocol proposals and issues a single failure leave protocol proposal that handles multiple providers. Similarly, the Group Services batches together multiple cast-out or join protocol proposals into a single protocol proposal. In all other cases, it deals with proposals one at a time.

To control batching protocols, an application needs to set the gs_batch_control field in the group attributes block shown in Figure 1 on page 8. The field can take one of the values defined by the ha_gs_batch_ctrl_t type shown in Figure 18 (except HA_GS_DEACTIVATE_ON_FAILURE, which is used by a deactivate-on-failure facility).

```
typedef enum {
   HA_GS_NO_BATCHING                 = 0x0000,
   HA_GS_BATCH_JOINS                 = 0x0001,
   HA_GS_BATCH_LEAVES                = 0x0002,
   HA_GS_BATCH_BOTH                  = 0x0003,
   HA_GS_DEACTIVATE_ON_FAILURE       = 0x0004
} ha_gs_batch_ctrl_t;
```

Figure 18. The ha_gs_batch_ctrl_t type

Each value has the following meanings:

HA_GS_NO_BATCHING
This value indicates no batching is allowed. Failure leave, cast-out, and join protocol proposals are serialized and presented to the group one at a time.

HA_GS_BATCH_JOINS
This value indicates that any number of join protocol proposals may be batched with other join protocol proposals. Failure leave and cast-out protocol proposals are not batched.

HA_GS_BATCH_FAILURES
This value indicates any number of failure leave or cast-out protocol proposals may be batched with other failure leave or cast-out protocol proposals respectively. Join protocol proposals are not batched.

HA_GS_BATCH_BOTH
This value indicates that any number of failure leave, cast-out or join protocol proposals may be batched with other failure leave, cast-out, or join protocol proposals respectively.

HA_GS_DEACTIVATE_ON_FAILURE

Enables the execution of a deactivate script when the provider is failing. This value is used by a deactivate-on-failure facility. For more information on the facility, refer to Section 3.3, “Deactivate-on-failure facility” on page 33.

3.1.7 Submitting changes with voting

The voting response to each voting phase of an n-phase protocol may contain any of the following:

- A new group state value proposal
- A provider-broadcast message
- A new default vote value for the group proposal

These choices give providers quite a bit of flexibility in managing their actions during an n-phase protocol. When one or more of these items is submitted with a voting response, Group Services sends it to all providers as part of the next notification of the protocol.

Changing the group state value during the voting phases of a protocol can be very useful. As an example, it would allow a group to update the group state value during membership change protocols, which may be very important in determining group quorum or active/inactive status.

Similarly, by submitting a provider-broadcast message with voting response, instead of or along with an updated group state value, the providers can pass data among themselves during the protocol without having to actually manipulate the group state value field.

Because each provider can submit its vote with these items, these items can be submitted with different values by providers. In this case, Group Services chooses only one value for each of the items to propagate to the providers for the next notification. Because the providers cannot control which value is chosen, they should guarantee one of the following rules:

- Only one provider submits a group state value and/or provider-broadcast message and/or new default vote value during each phase.
- All providers submit the same new group state value and/or provider-broadcast message and/or new default vote value during each phase.

If these rules are not followed, it is unpredictable which value will be chosen by the Group Services.

3.2 Responsiveness check facility

The responsiveness check facility allows the Group Services to inspect the state of the GS client periodically when there are no ongoing group activities. Group Services always monitors the GS client for an exit. A responsiveness check allows Group Services to query the actual responsiveness of the GS client. When the group is active, that is, when a protocol is running, Group Services can determine the responsiveness of the GS client by the client's response to the running protocol. Accordingly, Group Services suspends responsiveness checking during ongoing protocols.

3.2.1 Responsiveness check types

There are two responsiveness check types provided by Group Services. GS clients can specify one of the following responsiveness check types:

No responsiveness check

For this type, Group Services acts only if the GS client process exits.

Ping-like responsiveness check

For this type, Group Services periodically sends a responsiveness notification to the GS client and expects a response. The notification calls the responsiveness callback subroutine specified by the GS client. Group Services expects the responsiveness callback routine to return a code that indicates whether the GS client is operational or has detected an internal problem that prevents its correct operation.

The responsiveness check type is defined by the ha_gs_responsiveness_type_t type shown in Figure 19.

```
typedef enum {
    HA_GS_NO_RESPONSIVENESS,
    HA_GS_PING_RESPONSIVENESS,
    HA_GS_COUNTER_RESPONSIVENESS
} ha_gs_responsiveness_type_t;
```

Figure 19. The ha_gs_responsiveness_type_t type

The values have the following meanings:

HA_GS_NO_RESPONSIVENESS

This value indicates that Group Services should not perform a responsiveness check.

HA_GS_PING_RESPONSIVENESS

This value indicates that Group Services should perform a ping-like responsiveness check.

HA_GS_COUNTER_RESPONSIVENESS

This value is reserved for IBM use.

3.2.2 Utilizing a facility

To utilize a responsiveness check facility, a GS client is required to provide the responsiveness check control block when it initializes itself with Group Services. The responsiveness control block is defined as shown in Figure 20.

```
typedef struct {
    ha_gs_responsiveness_type_t     gs_responsiveness_type;
    unsigned int                    gs_responsiveness_interval;
    ha_gs_time_limit_t              gs_responsiveness_response_time_limit;
    void                            *gs_counter_location;
    unsigned int                    gs_counter_length;
} ha_gs_responsiveness_t;
```

Figure 20. The responsiveness control block

Each field contains the following information:

gs_responsiveness_type

This field contains the type of responsiveness check that is to be performed for this GS client. It may take one of the values defined by the ha_gs_responsiveness_type_t type shown in Figure 19 on page 31.

gs_responsiveness_interval

This field contains the number of seconds that Group Services should wait between executions of the specified responsiveness check.

gs_responsiveness_response_time_limit

This field contains the number of seconds that Group Services should wait for a return from the responsiveness callback subroutine. If the subroutine fails to return, Group Services assumes that the GS client has no responsiveness.

gs_counter_location

This field is reserved for IBM use.

gs_counter_length

This field is reserved for IBM use.

There is one more thing required to utilize a responsiveness check facility. If a GS client chooses a ping-like responsiveness check type, it must provide a responsiveness callback subroutine. For more information, refer to Section 4.2, “Initializing with Group Services” on page 46.

3.3 Deactivate-on-failure facility

Group Services provides a deactivate-on-failure facility. If a provider fails, this facility automatically executes a deactivate script that is provided by an application. The script could process some recovery/clean-up actions on the failed provider’s node. This facility is useful when a provider’s process fails to hold some resources and these resources must be released for the other providers. If an application provides a deactivate script that releases the resources, they are automatically released when a provider fails.

Usually, a deactivate script has the same name and directory for a group. However, it is possible to provide different deactivate scripts for each node in a group. In either case, providers on a node in a group must specify the same deactivate script.

A deactivate script is executed with the following rules:

- If Group Services needs to execute a deactivate script against multiple targeted-providers on one node in one protocol, the deactivate script will be executed once. A multiple targeted-providers list is passed to the script.
- If Group Services needs to execute a deactivate script against multiple targeted-providers on one node in separate protocols, the deactivate script is executed once per protocol.
- If there is no targeted-provider on a node, a deactivate script is not executed on the node, while a deactivate script is executed on other nodes that have targeted-providers.

The deactivate-on-failure facility is used by the following protocols:

- Failure leave protocol
- Goodbye protocol
- Expel protocol
- Cast-out protocol

For more information on using these protocols, refer to Section 5.3, “Failure leave protocol” on page 60, Section 5.7, “Goodbye protocol” on page 80, Section 5.8, “Expel protocol” on page 81, or Section 5.10, “Cast-out protocol” on page 95.

3.3.1 Utilizing a facility

To utilize a deactivate-on-failure facility, a GS client is required to set the gs_batch_control field in the group attributes block shown in Figure 1 on page 8. To enable deactivate-on-failure, this field needs to be set to a value of HA_GS_DEACTIVATE_ON_FAILURE. This value is defined by ha_gs_batch_ctrl_t type as shown in Figure 18 on page 29.

There is one more requirement to utilize a deactivate-on-failure facility: A GS client must provide a deactivate script. For more information, refer to Section 4.2, “Initializing with Group Services” on page 46.

3.3.2 Deactivate scripts

This section provides information about the execution environment, input parameters, and exit codes of deactivate scripts.

3.3.2.1 Execution environment

The script may be a shell script or any kind of executable file that conforms to the input and output rules that are specified later in this section.

Group Services does not verify that a deactivate script actually exists on a node or that it is executable until it is to be executed. If the specified deactivate script is not found or is not executable, Group Services applies the group’s default vote value for the phase in which the deactivate script should have been executed and for each subsequent voting phase if there are any.

A valid deactivate script is executed as follows: Using the following environments, the Group Services daemon on the targeted-provider’s node forks a child process that tries to execute the deactivate script:

Effective user ID (UID) and group ID (GID)

The forked process executes with the effective UID and GID of the targeted provider that the provider had when it initialized with Group Services. If the provider changed its UID or GID after initialization, the deactivate script still uses the effective UID and GID from the time it initialized. A deactivate script with a set UID bit in its file permissions executes with those values.

Working directory

The forked process begins execution in the current working directory of the targeted provider that the provider had when it initialized with Group Services. If the provider changed its current working directory after initialization, the deactivate script still uses the current working directory that existed when it initialized. A deactivate script that wants to execute in another directory must change to that directory.

Environment variables

The forked process inherits the environment variables from the Group Services daemon's environment. Therefore, the deactivate script must not make any assumptions about the environment variables (for example, the path) or access to specific directories or file systems except for those that are normally accessible to the provider's effective UID and gid.

STDIN, STDOUT, and STDERR file descriptors

On input, the STDIN, STDOUT, and STDERR file descriptors are closed (not associated with any files). To perform input or output, the deactivate script must explicitly open any input or output file that it wants to use.

3.3.2.2 Input parameters

On input, Group Services supplies the following five parameters to a deactivate script:

Process ID parameter

This parameter is always zero.

Voting time limit

This parameter contains the voting time limit in seconds as an int type (4 bytes). The deactivate script must complete and exit within this time limit.

Name of the group

This parameter contains the name of targeted-provider's group name as a null-terminated string.

Deactivate flag

This parameter is the null-terminated string specified by a provider when it proposes an expel protocol. In the case of other protocols, a failure leave, goodbye, or cast-out, this parameter is the null-terminated string *providerdied*. The deactivate script can distinguish when it is called by checking this deactivate flag.

Comma(,)-delimited list of targeted provider's instance numbers

When batching of failure leave protocols is enabled, the deactivate script can be executed once for the multiple failed-providers. This fifth parameter will indicate which providers were failing. Note that each provider instance number does not contain the node number.

3.3.2.3 Exit codes

On output, a deactivate script must supply an exit code of 0 for a successful completion. Any other exit code indicates an unsuccessful completion. It is up to the deactivate script to decide what constitutes a successful completion.

Upon receipt of an exit code indicating a successful completion within the time limit, Group Services votes *approve* for this voting phase of the protocol.

Upon receipt of an exit code indicating an unsuccessful completion before the time limit expires or if the deactivate script does not exit before the time limit expires, Group Services applies the group’s default vote for this voting phase of the protocol, and each subsequent voting phase of the protocol if there are any.

3.4 Source-target facility

It is sometimes convenient to associate several groups with a single application and to allow a process to be a member of multiple groups. Such relationships are not normally tracked by Group Services except when the source-target facility is used. To understand this facility, consider the following scenario.

If a node crashes, all of the groups with providers on that node are sent a notification simultaneously. The notification causes each group to begin reacting independently to the membership change. However, it may be better for some applications to wait until another group has completed processing this change. Such a relationship might exist, for example, between a disk recovery subsystem and a distributed database application. If the database is on a disk on the failed node, the database application must wait for the disk recovery subsystem to recover from the node failure before it can begin its recovery.

Although it is possible to deal with such relationships using subscriptions, subscriptions are loosely synchronized and may not provide the degree of timing control that is required. Instead, the source-target facility can be used.

The source-target facility allows a target group to tie itself to a source group as follows: If a failure leads to the failure of a provider in both the source and target groups, the source group completes its protocol for changing the membership list before the target group begins its protocol for changing the membership list. Thus, the providers in the target group can execute with the knowledge that the providers in the source group have already handled the failure. This knowledge is particularly useful when the recovery of the target group depends on the completion of recovery by the source group.

In the recovery scenario just described, the disk recovery subsystem is defined as the source group, and the database application is defined as the target group.

3.4.1 Configurations

When an application uses a source-target facility, it needs to pay attention to the following configuration rules:

- A group defines itself as a target-group by listing a source-group name in the group attributes block by each target-group provider. A source-group is not notified that it has been “sourced” by any groups. For the group attributes block, refer to Figure 1 on page 8.
- There may be multiple source-group and/or target-group providers on a node. A source-group may have any number of target-groups. A target-group may source only one group as illustrated in Figure 21 on page 37.

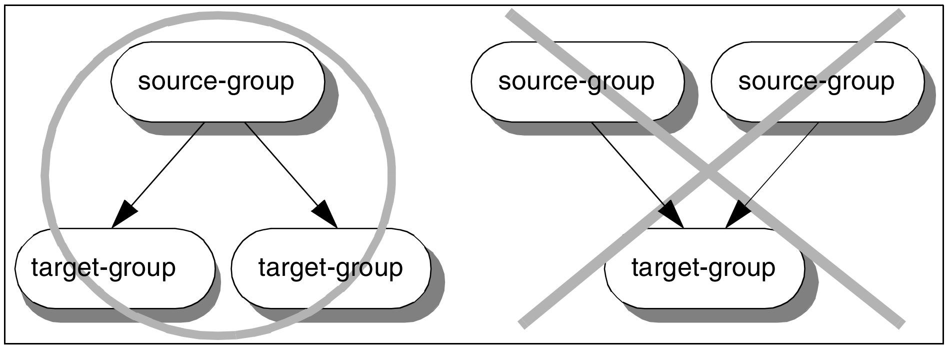


Figure 21. Multiple source-group and/or target-group providers

3.4.2 Membership list changes

With source-target groups, joins and leaves operate somewhat differently than with other groups. Here are some key differences:

- For every node on which a target-group provider wants to run, there must exist a source-group provider.

  If there is no source-group provider on a node, a potential target-group provider is not allowed to join the target group, and no membership change is proposed. The GS client attempting to join the target-group receives an asynchronous return code that indicates that there is no source-group provider active on this node.

If there is a source-group provider on a node, a potential target-group provider is allowed to join the target-group.

Figure 22 on page 38 illustrates that a target-group's provider on Node 3 cannot join the target-group. On the other hand, a target-group's provider on Node 2 can join the target-group.

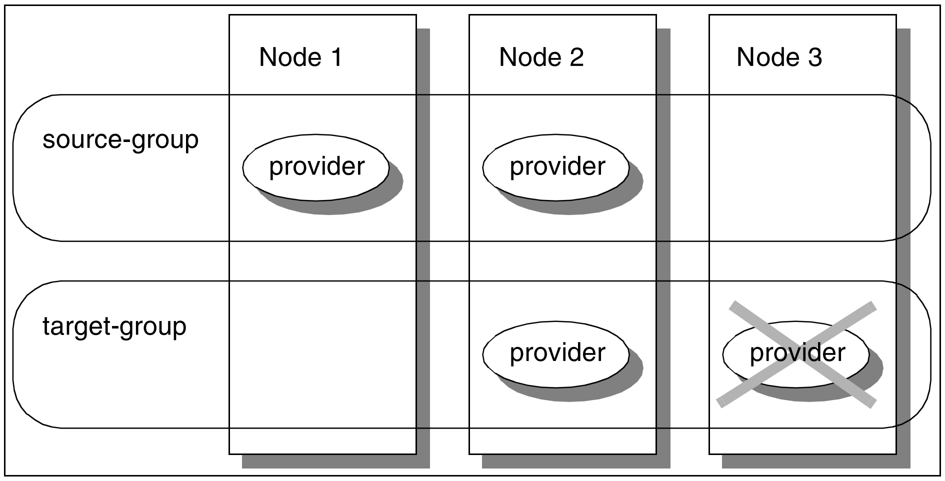


Figure 22. Joining to a target-group

- If the last remaining source-group provider on a node leaves the source-group voluntarily or involuntarily, all of the target-group providers on that node must leave the target-group.

  The source-group processes the leave(s) as a normal protocol for changing the membership list.

  Once the source-group has completed the changing membership list, a membership list change is proposed to the target-group as a cast-out of the affected providers(s) from the target-group. This proposal is called a cast-out protocol. If there is no target-group provider on that node, no cast-out protocol is proposed to the target-group providers.

  The provider(s) that are being cast out receive a notification that they have been cast out of the group. They do not otherwise participate in the cast-out protocol. For more information on a cast-out protocol, refer to Section 5.10, "Cast-out protocol" on page 95.

  Figure 23 on page 39 illustrates that when a source-group's provider on Node 2 leaves the source-group, it forces a target-group's provider on node 2 to leave the target-group. When the source-group has completed a normal protocol for changing the membership list, the target-group starts a cast-out protocol that targets the provider on node 2.

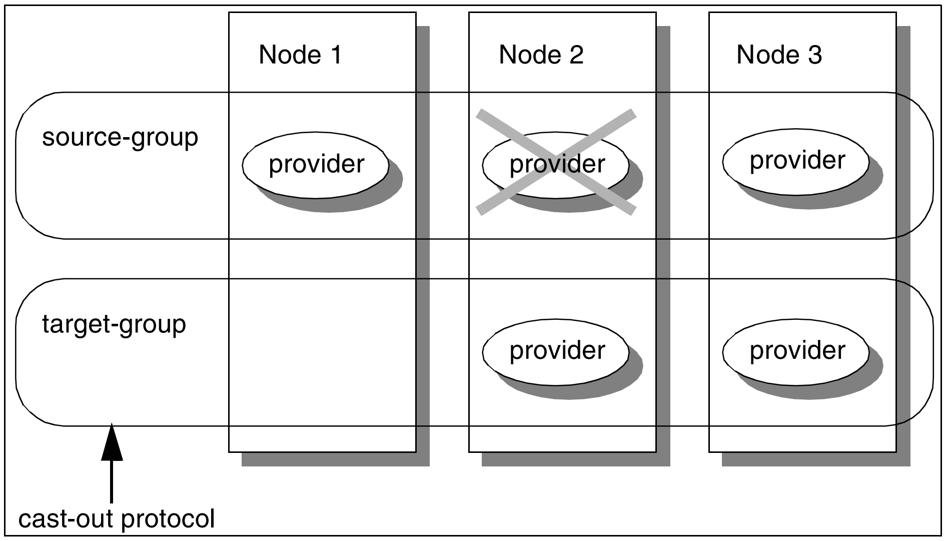


Figure 23. Leaving from a source-group

- If a node fails, the source-group starts a failure leave protocol for its leaving provider(s) on the node. When the protocol has completed, the target-group starts a cast-out protocol (instead of a failure leave protocol) for its leaving provider(s) on the node. This assumes that there are source-group and target-group providers running on a node or nodes other than the failed node.

  Figure 24 illustrates a scenario in which Node 2 has crashed. When a source-group has completed a normal protocol for changing the membership list, the target-group starts a cast-out protocol that targets the provider on Node 2 even it does not exist.

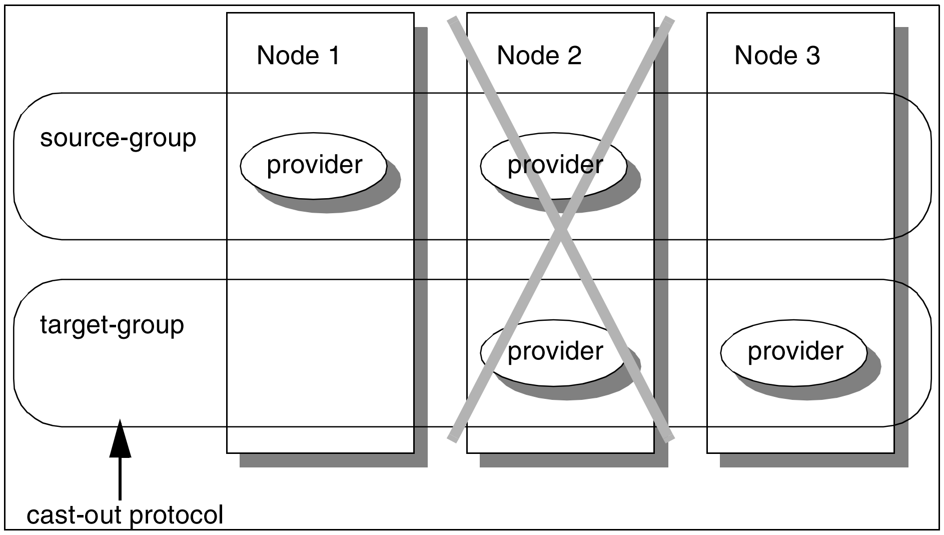


Figure 24. Node failure

- If a target-group is running a protocol and a source-group provider process fails on a node that also contains a target-group provider, the source-group runs a failure leave protocol.

  In this case, only the process of the source-group provider has failed, not the node on which it is running. Because the target-group provider process still exists, the target-group protocol can continue. However, once the source-group completes its leave protocol, the target-group provider may no longer validly belong to the target-group.

  Therefore, the Group Services subsystem considers the target-group provider(s) that will be cast-out as having failed during the protocol and treats them accordingly:

  - If the target-group's default vote is reject, the protocol is rejected, and the Group Services proposes a cast-out protocol.
  - If the default vote is approve, the protocol is approved or, if a provider votes continue, the protocol continues.
  - If the protocol continues, the failed target-group provider(s) are no longer allowed to participate. Instead, the default vote (*approve* in this case) is registered for them for each voting phase.

  Whatever the outcome of the target-group's running protocol, once it ends, Group Services immediately proposes a cast-out protocol for the target-group.

When a source-group leave prevents the last target-group provider(s) from executing protocols, those providers are given a cast-out final notification and the target-group is, in effect, dissolved.

- As part of any cast-out protocol in a target group, it will receive the source-group's current state value in the notification.

3.4.3 Group state value changes

Other than a change in the membership list, a change of the group state value is also handled by a source-target facility.

- If a source-group changes its group state value during protocols that do not result in a cast-out protocol, its associated target-group(s) receive(s) a notification.

  The notification appears to the target-group as a source-state reflection protocol. For more information about a source-state reflection protocol, refer to Section 5.11, "Source-state reflection protocol" on page 103.
- If the target-group is running a protocol when a source-group's group state value change is ready to be reflected, the running protocol continues normally, and the source-state reflection protocol is queued to be proposed later when the running protocol completes.
- If a subsequent source-group state value change appears, only the most recent one is reflected to the target-group, and the earlier change is simply dropped. In addition, if a cast-out is necessary and a source-state reflection protocol is queued, the queued protocol is dropped because the cast-out protocol reflects the most recent source-group state value.

Because a source-state reflection protocol is proposed by Group Services, it is always proposed before any pending provider-proposed protocols for the group. In addition, there is no interface for a provider to request this protocol. It is automatically proposed as a consequence of a source-group's group state value change.

3.5 Sundered namespaces

The Group Services provides a single group namespace within each domain. Given the right set of multiple network failures, a domain with multiple networks can become split. In the case of a sundered namespace, the nodes become split in such a way that they can no longer communicate with any nodes on the other side of the split. However, it is possible for each sundered portion to maintain enough information to reconstruct the groups that were in existence previously - at least those groups that still have members within any particular portion.

When a namespace is sundered, it is possible to get two instances of what should be one group. For example, in a sundered network, two nodes that own the two tails of a twin-tailed disk could end up on separate sides of the split. Because the processes of the subsystem coordinating the disk on each node would believe that the other process had disappeared, the process might want to activate its tail, which could lead to data corruption. As this example shows, it is important that each group determine whether it needs a form of quorum and use it to guide when a group is ready to perform its services.

Although the Group Services does not provide a quorum mechanism, it does provide some assistance to groups when a network is sundered. When a domain is sundered, the providers receive membership protocol proposals from Group Services that all of the providers on the “other side” of the split have failed. The providers can then execute those protocols as they normally would taking into account such factors as quorum to protect resources as necessary.

If a sundered network becomes healed and Group Services discovers separate domains, it dissolves the smaller domain, which is defined as the domain with the smaller number of nodes. Group Services sends an announcement notification that it has “died horribly” to the clients on the smaller domain. Upon receipt of the notification, the clients on the smaller domain can join the larger domain or perform any other appropriate recovery action.

Part 2. Group Services design

Chapter 4. Initializing with Group Services

To start using the services provided by Group Services, a GS client requires some preparation.

This chapter describes how a GS client chooses a domain, how it initializes with the Group Services, and how it quit using the services provided by the Group Services.

4.1 Choosing a domain

A GS client communicates with a Group Services daemon that is running on the same node as the GS client. A GS client communicates with the Group Services daemon through the Group Services Application Program Interface (GSAPI), using a Unix domain socket. Before a GS client initializes itself with the Group Services daemon, the GS client must choose a domain to which the Group Services daemon belongs.

4.1.1 Group Services PSSP domains

To choose a domain from Group Services PSSP domains, a GS client must set the following environment variable:

HA_DOMAIN_NAME
This environment variable must be set to the name of the SP system partition in which the GS client is executing.

HA_GS_SUBSYS
This environment variable must be set to hags, or it must not be defined.

Note

Before PSSP 3.1, the GS clients needed to set the environment variable HA_SYSPAR_NAME. Since PSSP 3.1, you can set either environment variable to support compatibility of older clients. However, all new GS clients should use the environment variable, HA_DOMAIN_NAME, because the environment variable, HA_SYSPAR_NAME, may eventually be unsupported.

4.1.2 Group Services HACMP/ES domains

To choose a domain from Group Services HACMP/ES domains, a GS client must set the following environment variables:

HA_DOMAIN_NAME

This environment variable must be set to the name of the HACMP/ES cluster in which the GS client is executing.

HA_GS_SUBSYS

This environment variable must be set to grpsvcs.

4.2 Initializing with Group Services

If a GS client has chosen a domain, it can initialize itself with the Group Services daemon, that is, the Group Services subsystem.

4.2.1 Subroutine call

To initialize with the Group Services subsystem, a GS client must call the ha_gs_init subroutine. After initialization with the Group Services subsystem, a GS client is allowed to join or subscribe to a group.

The syntax of the ha_gs_init subroutine is shown in Figure 25.

```
ha_gs_rc_t ha_gs_init(
   ha_gs_descriptor_t            *ha_gs_descriptor,
   const ha_gs_socket_ctrl_t     socket_options,
   const ha_gs_responsiveness_t  *responsiveness_control,
   const char                    *deactivate_script,
   ha_gs_responsiveness_cb_t     *responsiveness_callback,
   ha_gs_delayed_error_cb_t      *delayed_error_callback,
   ha_gs_query_cb_t              *query_callback)
```

Figure 25. The syntax of ha_gs_init subroutine

Each parameter requires the following information:

ha_gs_descriptor

This parameter requires a pointer to a buffer in which the Group Services subsystem will return the file descriptor that the process will use to communicate with the Group Services. The process itself must not read or write directly on this file descriptor.

socket_options

This parameter requires the value of HA_GS_SOCKET_NO_SIGNAL.

responsiveness_control

This parameter requires a pointer to a responsiveness control block. The block specifies the type, if any, that will be used to perform responsiveness checks for the process. The responsiveness control block is defined as shown in Figure 20 on page 32.

deactivate_script

This parameter requires a pointer to the path name to a deactivate script if a deactivate-on-failure is used. If not, a NULL pointer must be specified.

responsiveness_callback

This parameter requires a pointer to a responsiveness callback subroutine if a responsiveness check is required. If not, a NULL pointer must be specified. For a responsiveness notification, refer to Section 7.3, “Responsiveness notification” on page 139.

delayed_error_callback

This parameter requires a pointer to a delayed error callback subroutine. For a delayed error notification, refer to Section 7.4, “Delayed error notification” on page 141.

query_callback

This parameter should contain a NULL pointer.

4.2.2 Programming hints

This section provides programming hints.

4.2.2.1 A responsiveness check facility

When a GS client has initialized with Group Services, Group Services starts a responsiveness check as specified by the responsiveness control block. It does not matter if a GS client has been a provider and/or a subscriber or neither.

If a GS client does not use a responsiveness check facility, set it up as follows:

- The gs_responsiveness_type field in the responsiveness control block must be a value of HA_GS_NO_RESPONSIVENESS, and the responsiveness_callback parameter in the ha_gs_init subroutine can be a NULL pointer.

The required fields for each responsiveness check type are summarized in Table 1.

Table 1. Required fields for each responsiveness check type

gs_responsiveness_type	**HA_GS_NO_RESPONSIVENESS**	**HA_GS_PING_RESPONSIVENESS**
gs_responsiveness_interval	ignored	required
gs_responsiveness_response_time_limit	ignored	required

4.2.2.2 A deactivate-on-failure facility

If a GS client does not use a deactivate-on-failure facility, set it up as follows:

The gs_batch_control field in the group attributes block must not have a value of HA_GS_DEACTIVATE_ON_FAILURE, and the deactivate_script parameter in the ha_gs_init subroutine can be a NULL pointer.

4.3 Quit using Group Services

When a GS client no longer needs to use the Group Services subsystem, it should call the ha_gs_quit subroutine to quit using the Group Services subsystem. This allows Group Services to release the resources associated with the GS client.

4.3.1 Subroutine call

No parameter is required to call the ha_gs_quit subroutine. The syntax of the ha_gs_quit subroutine is shown in Figure 26 on page 48.

```
void ha_gs_quit(void)
```

Figure 26. The syntax of ha_gs_quit subroutine

4.3.2 Programming hints

If a GS client calls the ha_gs_quit subroutine while still joined as a provider to any groups, the Group Services subsystem will notify the groups that the provider has failed, and the groups will execute a failure leave protocol. If the GS client wants to leave a group without a failure leave protocol, it should call the ha_gs_leave subroutine before calling the ha_gs_quit subroutine.

Chapter 5. Proposing protocols

Once a GS client is initialized with the Group Services, it is allowed to use the subroutines provided by the Group Services shared library, that is, the Group Services Application Programming Interfaces (GSAPIs). To propose a protocol to a group, the GS client must become a provider of the group. Group Services itself also proposes a protocol to a group.

This chapter describes how a protocol is proposed to a group and how it is handled in the group.

5.1 Protocol proposal

This section provides a brief explanation of all the protocols and the information on subroutines commonly used for all the protocol proposals.

5.1.1 Protocols

The Group Services provides ten protocols. The protocols are proposed by either a GS client that has been initialized with the Group Services or the Group Services subsystem itself.

The following protocol is proposed by a GS client that is not a provider for a group that is proposed a protocol:

Join protocol
A GS client proposes this protocol when it wants to join a group as a provider. It calls the ha_gs_join subroutine to propose the protocol.

The following six protocols are proposed by a provider in a group:

State value change protocol
A provider proposes this protocol when it wants to change a group state value. It calls the ha_gs_change_state_value subroutine to propose the protocol.

Provider-broadcast message protocol
A provider proposes this protocol when it wants to send a provider-broadcast message to all the providers in the group. It calls the ha_gs_send_message subroutine to propose the protocol.

Voluntary leave protocol
A provider proposes this protocol when it wants to leave the group

voluntarily. It calls the ha_gs_leave subroutine to propose the protocol.

Goodbye protocol
A provider proposes this protocol when it wants to leave the group immediately. It calls the ha_gs_goodbye subroutine to propose the protocol.

Expel protocol
A provider proposes this protocol when it wants to expel one or more providers from the group. It calls the ha_gs_expel subroutine to propose the protocol.

Change-attributes protocol
A provider proposes this protocol when it wants to change the group attributes. It calls the ha_gs_change_attributes subroutine to propose the protocol.

The following three protocols are proposed by the Group Services subsystem itself:

Failure leave protocol
The Group Services subsystem proposes this protocol when it wants to change the membership list due to one or more provider failures.

Cast-out protocol
The Group Services subsystem proposes this protocol when it wants to change the membership list due to one or more providers being cast out by the source-target facility.

Source-state reflection protocol
The Group Services subsystem proposes this protocol when it wants to reflect to a target-group that its source-group has changed its group state value without changing the membership list.

5.1.2 Subroutines

When a GS client proposes a protocol, it is simply a matter of calling the proper subroutine. The Group Services subsystem takes care of notifying the other providers in the group that a protocol has been proposed and proceeds with it based on the number of phases and the nature of the protocol.

Most of the subroutines require *a token* and a pointer to *a proposal information block*. The token identifies the caller as a provider of the group. A proposal information block describes the information on the proposed protocol.

The ha_gs_join subroutine provides a pointer to a token instead of a value of the token. Because the subroutine needs to get a token from the Group Services subsystem. The rest of the subroutines require this token when they are called. The ha_gs_goodbye subroutine provides only a token. It does not require a pointer to a proposal information block.

The prototypes for the protocol proposal subroutines are shown in Figure 27.

```
ha_gs_rc_t ha_gs_join(
    ha_gs_token_t *,
    const ha_gs_proposal_info_t *);

ha_gs_rc_t ha_gs_change_state_value(
    ha_gs_token_t,
    const ha_gs_proposal_info_t *);

ha_gs_rc_t ha_gs_send_message(
    ha_gs_token_t,
    const ha_gs_proposal_info_t *);

ha_gs_rc_t ha_gs_leave(
    ha_gs_token_t,
    const ha_gs_proposal_info_t *);

ha_gs_rc_t ha_gs_goodbye(
    ha_gs_token_t);

ha_gs_rc_t ha_gs_expel(
    ha_gs_token_t,
    const ha_gs_proposal_info_t *);

ha_gs_rc_t ha_gs_change_attributes(
    ha_gs_token_t,
    const ha_gs_proposal_info_t *);
```

Figure 27. The protocol proposal subroutine prototypes

The token is also referred to as the *provider token* and is defined as the ha_gs_token_t type shown in Figure 7 on page 14.

The content of a proposal information block depends on each protocol. The proposal information block is defined as a ha_gs_proposal_info_t type, and it is redefined as each proposal request block using union as shown in Figure 28 on page 54.

```
typedef struct {
   union {
      ha_gs_join_request_t              _gs_join_request;
      ha_gs_state_change_request_t      _gs_state_change_request;
      ha_gs_message_request_t           _gs_message_request;
      ha_gs_leave_request_t             _gs_leave_request;
      ha_gs_expel_request_t             _gs_expel_request;
      ha_gs_subscribe_request_t         _gs_subscribe_request;
      ha_gs_attribute_change_request_t  _gs_attribute_change_request;
   } _gs_protocol_info;
} ha_gs_proposal_info_t;
```

Figure 28. The proposal information block

Note

The following definitions are available for programming convenience:

```
#definegs_join_request             _gs_protocol_info._gs_join_request
#definegs_state_change_request     _gs_protocol_info._gs_state_change_request
#definegs_message_request          _gs_protocol_info._gs_message_request
#definegs_leave_request            _gs_protocol_info._gs_leave_request
#definegs_expel_request            _gs_protocol_info._gs_expel_request
#definegs_attribute_change_request _gs_protocol_info._gs_attribute_change_request
```

Proposal request blocks and their fields are summarized in Table 2. An asterisk (*) indicates that the field is a pointer.

Table 2. Protocol request blocks and their fields

fields	**ha_gs_join_request_t**	**ha_gs_state_change_request_t**	**ha_gs_message_request_t**	**ha_gs_leave_request_t**	**ha_gs_expel_request_t**	**ha_gs_attribute_change_request_t**
*gs_group_attributes	O					O
gs_provider_instance	O					
*gs_provider_local_name	O					

fields	**ha_gs_join_request_t**	**ha_gs_state_change_request_t**	**ha_gs_message_request_t**	**ha_gs_leave_request_t**	**ha_gs_expel_request_t**	**ha_gs_attribute_change_request_t**
*gs_n_phase_protocol_callback	O					
*gs_protocol_approved_callback	O					
*gs_protocol_rejected_callback	O					
*gs_announcement_callback	O					
gs_num_phases		O	O	O	O	O
gs_time_limit		O	O	O	O	O
gs_new_state		O				
gs_message			O			
gs_leave_code				O		
gs_expel_list					O	
gs_deactivate_phase					O	
*gs_deactivate_flag					O	
*gs_backlevel_providers						O

5.2 Join protocol

Once a GS client has initialized with the Group Services, it can join a group. To join a group, it needs to propose a join protocol. A GS client is not yet a member of a group; however, it can propose a join protocol, participate in the protocol voting phases (if the protocol is n-phase), and receive the protocol proposal result.

5.2.1 Subroutine call

A GS client calls the ha_gs_join subroutine to propose a join protocol. A subroutine prototype is shown in Figure 27 on page 53. On input, the GS client provides the join request block shown in Figure 29.

```
typedef struct {
    ha_gs_group_attributes_t     *gs_group_attributes;
    short                        gs_provider_instance;
    char                         *gs_provider_local_name;
    ha_gs_n_phase_cb_t           *gs_n_phase_protocol_callback;
    ha_gs_approved_cb_t          *gs_protocol_approved_callback;
    ha_gs_rejected_cb_t          *gs_protocol_rejected_callback;
    ha_gs_announcement_cb_t      *gs_announcement_callback;
    ha_gs_merge_cb_t             *gs_merge_callback;
} ha_gs_join_request_t;
```

Figure 29. The join request block

Each field requires the following information:

gs_group_attributes
This field requires a pointer to the group attributes block shown in Figure 1 on page 8.

gs_provider_instance
This field requires an instance number to be used by this provider. This value must be unique on this node for this group.

gs_provider_local_name
This field requires a pointer to an optional byte string that contains a local name for the provider.

gs_n_phase_protocol_callback
This field requires a pointer to the callback subroutine that is to be called by an n-phase notification. For an n-phase notification, refer to Section 7.5, “N-phase notification” on page 142.

gs_protocol_approved_callback
This field requires a pointer to the callback subroutine that is to be called by a protocol approved notification. For information about the protocol approved notification, refer to Section 7.6, “Protocol approved notification” on page 143.

gs_protocol_rejected_callback
This field requires a pointer to the callback subroutine that is to be called by a protocol-rejected notification. For a protocol-rejected notification, refer to Section 7.7, “Protocol rejected notification” on page 145.

gs_announcement_callback

This field requires a pointer to the callback subroutine that is to be called by an announcement notification. For an announcement notification, refer to Section 7.8, “Announcement notification” on page 146.

gs_merge_callback

This field is reserved for IBM use. It must be a NULL pointer.

5.2.2 Protocol flow

The number of phases and the voting time limit depend on a GS client that proposes a join protocol. If it is the first GS client that proposes a join protocol to a group (in other words, if the group does not exist currently), Group Services uses the group attributes block specified in a join request block that is provided by the GS client. If not, the Group Services uses the group attributes block that is already registered for the group. In either case, the gs_num_phases field in the group attributes block is used for the number of phases, and the gs_time_limit field is used for the voting time limit.

Batching join protocols

If batching join protocols is enabled, one join protocol handles one or more joining GS clients. If not, one join protocol handles only one joining GS client. This section assumes that batching join protocols is enabled. Therefore, joining GS clients are treated as plural. If this is not the case, read it as singular.

5.2.2.1 One-phase protocol

If the protocol is a one-phase, the protocol is approved automatically.

The GS clients are added to the membership list. All the providers including the GS clients that proposed the protocol receive a protocol approved notification (ha_gs_protocol_approved_callback) with the updated membership list. The subscribers also receive the notification if they subscribed for it.

The protocol flow of the one-phase join protocol is illustrated in Figure 30 on page 58.

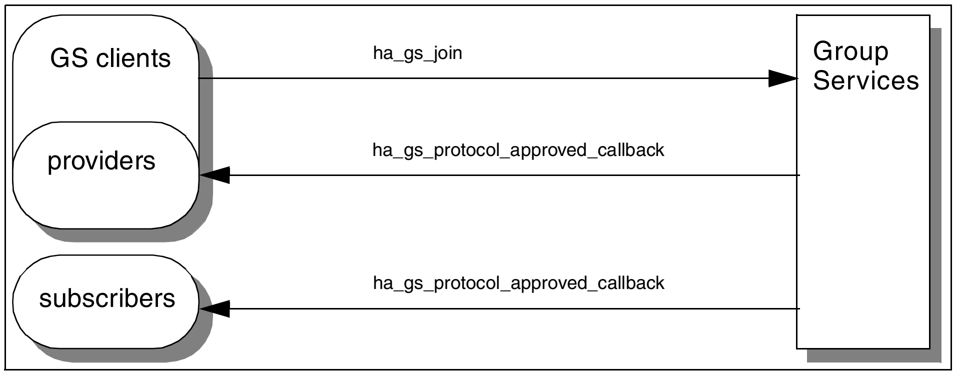

GS clients
providers
subscribers
ha_gs_join
ha_gs_protocol_approved_callback
ha_gs_protocol_approved_callback
Group Services

Figure 30. The one-phase join protocol

5.2.2.2 N-phase protocol

If the protocol is an n-phase, the protocol has voting phases. All the providers including the GS clients that proposed the protocol participate in voting phases.

If the protocol is approved, the GS clients are added to the membership list. All the providers including the GS clients that proposed the protocol receive a protocol approved notification (ha_gs_protocol_approved_callback) with the updated membership list. If a new group state value was proposed during the voting phases of the protocol, a group state value is updated, and the protocol approved notification includes the updated group state value. The subscribers also receive the notification if they subscribed for it.

The protocol flow of the approved n-phase join protocol is illustrated in Figure 31 on page 59.

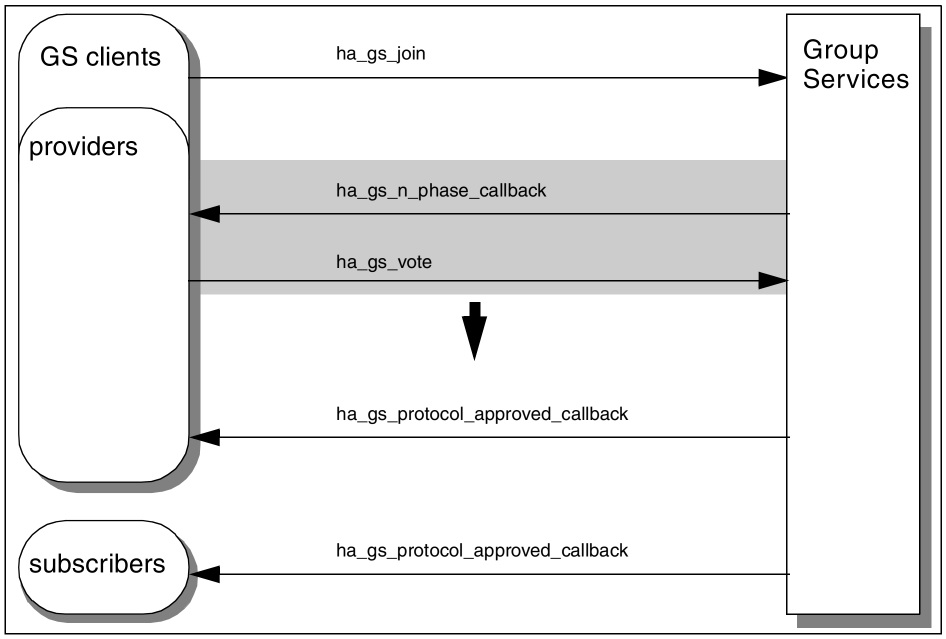


Figure 31. The approved n-phase join protocol

If the protocol is rejected, the membership list and the group state value remain unchanged. All the providers including the GS clients that proposed the protocol receive a protocol-rejected notification (ha_gs_protocol_rejected_callback). The subscribers do not receive any notification.

The protocol flow of the rejected n-phase join protocol is illustrated in Figure 32 on page 60.

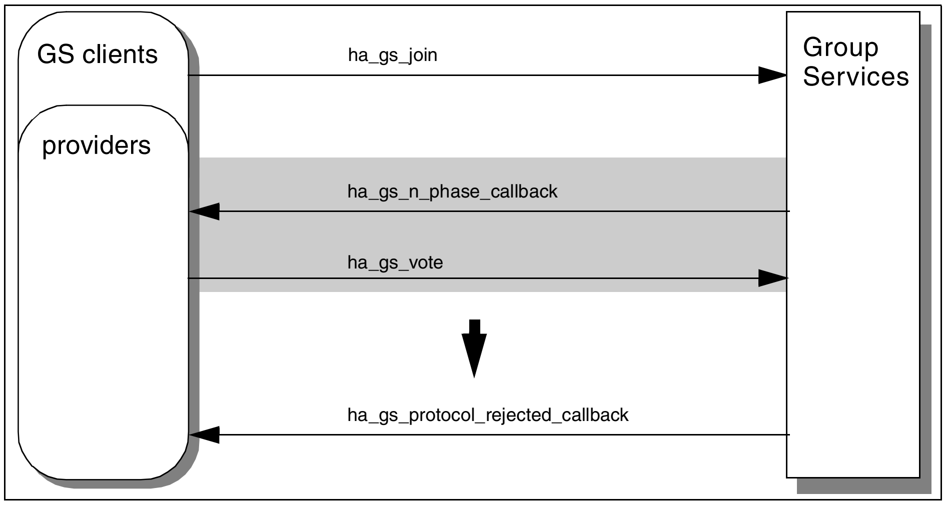


Figure 32. The rejected n-phase join protocol

5.2.3 Programming hints

Unlike the other protocols, the gs_proposed_by field in the proposal block contains the provider information block for the GS client that is executing the callback subroutine rather than the provider that initiated the join protocol.

For more information, refer to Section 7.2.2.6, “gs_proposal field” on page 132.

5.3 Failure leave protocol

Group Services proposes a failure leave protocol when it wants to remove one or more providers from the membership list when the providers’ sockets associated with the Group Services are broken.

5.3.1 Protocol proposal

The Group Services subsystem uses the following fields of the group attributes block to propose and execute a protocol:

- The gs_num_phases field is used for the number of phases.
- The gs_time_limit field is used for the voting time limit.
- The gs_batch_control field is used to enable/disable batching failure leave protocols.

- The gs_batch_control field is also used to enable/disable a deactivate-on-failure facility. A deactivate script has already been registered by the ha_gs_init subroutine.

5.3.2 Protocol flow

A protocol uses the number of phases and the voting time limit specified in the group attributes block already registered.

Because the failed providers have lost their connection to Group Services, they cannot participate in any activities of the protocol.

Note

If batching failure leave protocols is enabled, one failure leave protocol handles one or more failed providers. If not, one failure leave protocol handles only one failed provider. This section assumes that batching failure leave protocols is enabled. Therefore, failed providers are treated as plural. If this is not the case, read it as singular.

5.3.2.1 One-phase protocol without deactivate-on-failure

If the protocol is one-phase and deactivate-on-failure is disabled, the protocol is approved automatically, and a deactivate script is not executed.

The failed providers are removed from the membership list. All the remaining providers receive a protocol approved notification (ha_gs_protocol_approved_callback) with the updated membership list. The subscribers also receive the notification if they subscribed for it.

The protocol flow of the one-phase failure leave protocol without deactivate-on-failure is illustrated in Figure 33.

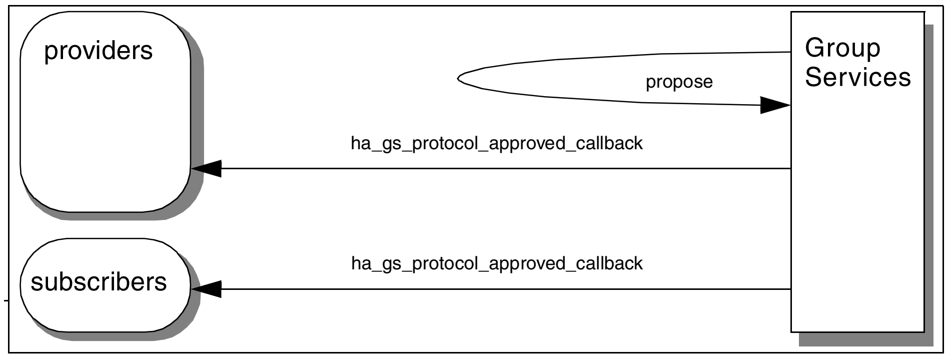

providers
Group Services
propose
ha_gs_protocol_approved_callback
subscribers
ha_gs_protocol_approved_callback

Figure 33. The one-phase failure leave protocol without deactivate-on-failure

5.3.2.2 One-phase protocol with deactivate-on-failure

If deactivate-on-failure is enabled, the protocol is approved automatically, and a deactivate script is executed against the failed providers immediately after the protocol begins execution. The Group Services subsystem does not wait for the script to complete execution. The exit code of the script is not inspected.

The failed providers are removed from the membership list. All the remaining providers receive a protocol approved notification (ha_gs_protocol_approved_callback) with the updated membership list. The subscribers also receive the notification if they subscribed for it.

The protocol flow of the one-phase failure leave protocol with deactivate-on-failure is illustrated in Figure 34.

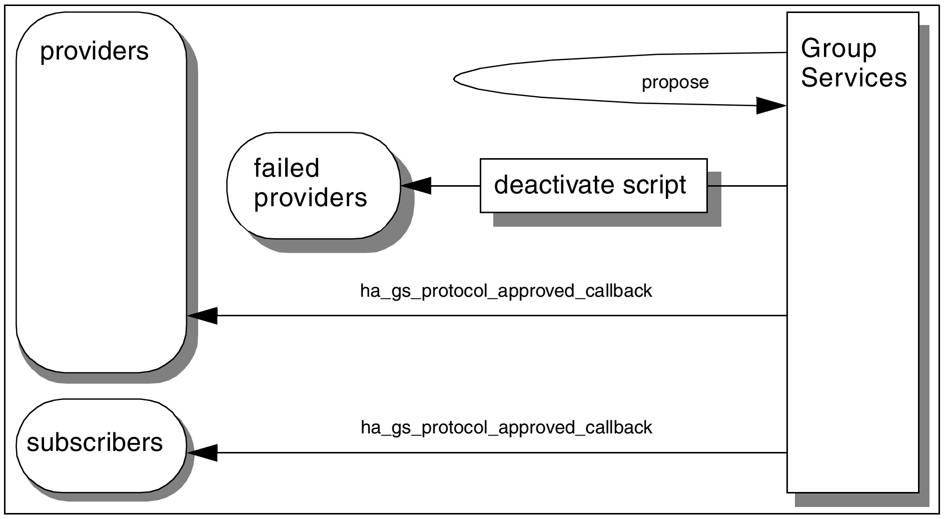


Figure 34. The one-phase failure leave protocol with deactivate-on-failure

5.3.2.3 N-phase protocol without deactivate-on-failure

If a failure leave protocol is n-phase and deactivate-on-failure is disabled, the protocol has voting phases, and a deactivate script is not executed.

The protocol flow of the beginning of an n-phase failure leave protocol without deactivate-on-failure is illustrated in Figure 35 on page 63.

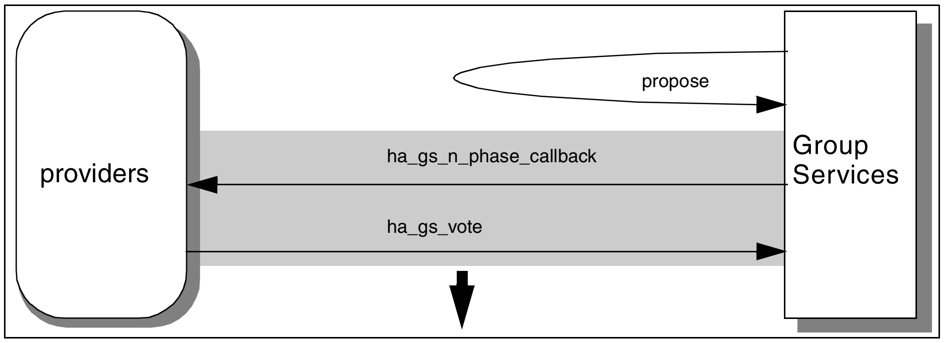


Figure 35. The n-phase failure leave protocol without deactivate-on-failure

5.3.2.4 N-phase protocol with deactivate-on-failure

If deactivate-on-failure is enabled, a deactivate script is executed against the failed providers immediately after the protocol begins execution. The Group

Services waits for the script to complete execution within the voting time limit. The exit code of the script is inspected.

According to the exit code, Group Services votes as the failed providers' vote for the phase. If the protocol requires more voting phases, Group Services continues to vote as the failed providers' vote for each subsequent voting phase. The voting value is determined by the following rule:

- The Group Services votes to approve if the exit code is 0.
- The Group Services votes current default vote value in the following cases:
  - If the exit code is not 0.
  - If the protocol has specified a voting time limit, and the script does not complete its execution within voting time limit.
  - If the script is not specified or it is not executable.

The protocol flow of the beginning of an n-phase failure leave protocol with deactivate-on-failure is illustrated in Figure 36 on page 65.

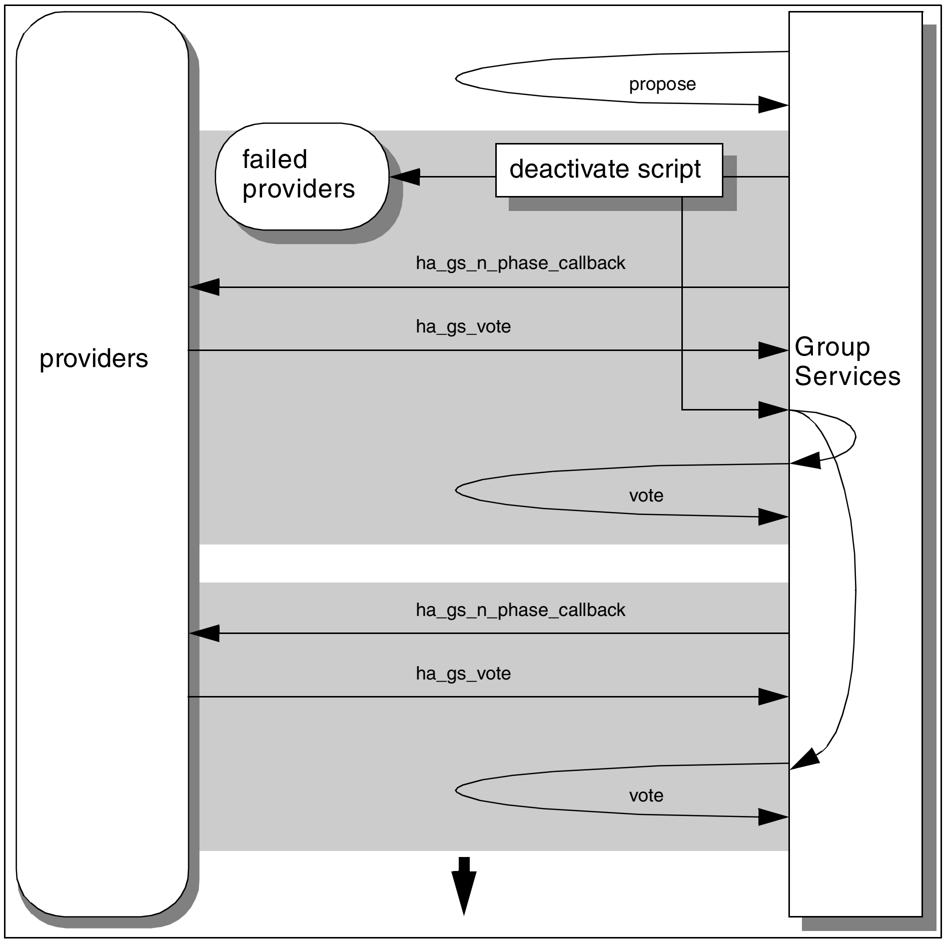


Figure 36. The n-phase failure leave protocol with deactivate-on-failure

5.3.2.5 The ending of n-phase protocol

If the protocol is approved, the failed providers are removed from the membership list. All the remaining providers receive a protocol approved notification (ha_gs_protocol_approved_callback) with the updated membership list. If a new group state value was proposed during the voting phases of the protocol, a group state value is updated, and the protocol approved notification includes the updated group state value. The subscribers also receive the notification if they subscribed for it.

The protocol flow of the ending of an approved n-phase failure leave protocol is illustrated in Figure 37 on page 66.

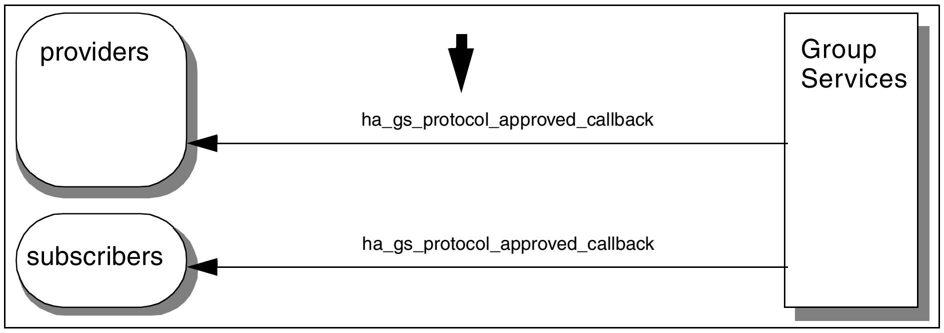

providers
Group Services
ha_gs_protocol_approved_callback
subscribers
ha_gs_protocol_approved_callback

Figure 37. The approved n-phase failure leave protocol

If the protocol is rejected, the Group Services checks the following special condition that requires special handling: If batching failure leave protocols is enabled and the rejection is caused by a default reject vote.

If this is the case, the execution of the protocol stops; however, the failed providers are *not* removed from the membership list, and the group state value remains unchanged. The group will be immediately put into a new failure leave protocol with any newly-failed providers added to the list of already-failed providers from the previous protocol.

A deactivate script will be executed only once against any single failed provider instance. Thus, during the subsequent failure protocol(s), the deactivate script will be executed only against the newly-failed providers, not against the already-failed providers.

During any subsequent failure protocols, the Group Services subsystem will vote to approve on behalf of the already-failed providers. This avoids the group being put in an infinite loop.

The protocol flow of the ending of the rejected n-phase failure leave protocol with the special condition is illustrated in Figure 38 on page 67.

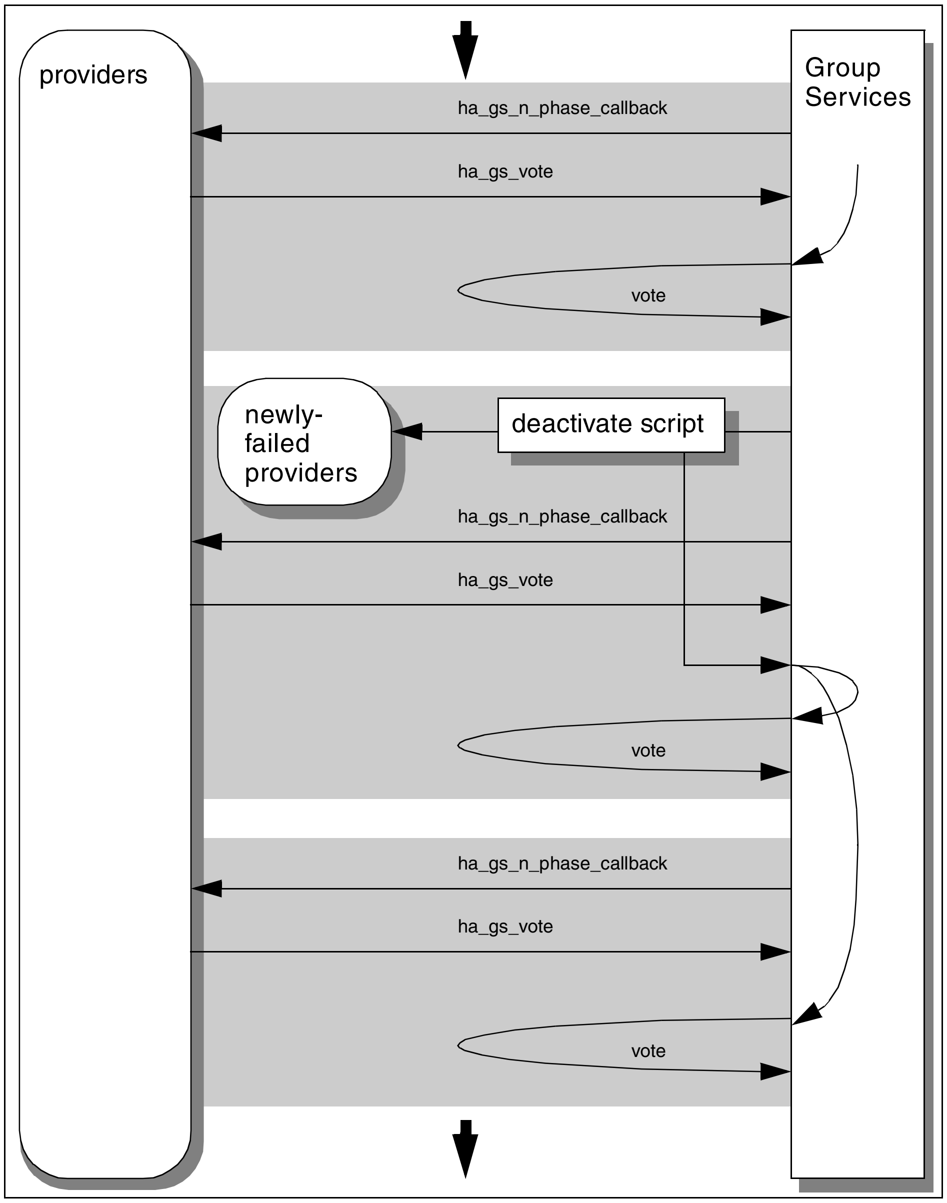


Figure 38. The rejected n-phase failure leave protocol with a special condition

If the protocol is rejected and the previous condition is not applied, the execution of the protocol stops. All the failed providers are removed from the membership list, and the group state value remains unchanged. The remaining providers receive the protocol rejected notification (ha_gs_protocol_rejected_callback) with updated membership list. The subscribers also receive the notification, if they subscribed for it.

The protocol flow of the ending of a rejected n-phase failure leave protocol without a special condition is illustrated in Figure 39.

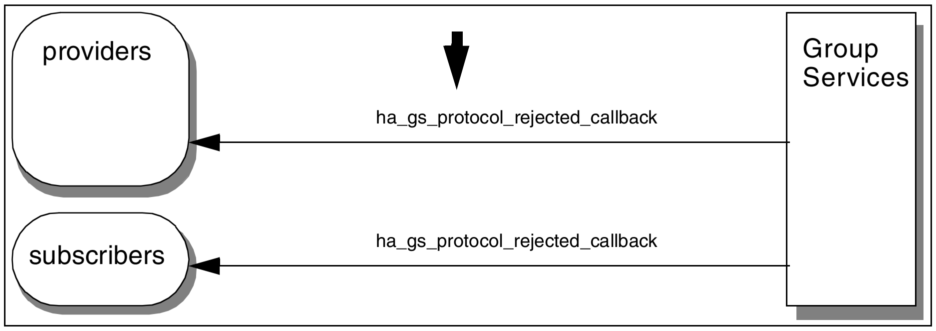

providers
Group Services
ha_gs_protocol_rejected_callback
subscribers
ha_gs_protocol_rejected_callback

Figure 39. The rejected n-phase failure leave protocol without a special condition

5.3.3 Programming hints

The following sections contain programming hints.

5.3.3.1 Targeted or not targeted

Whether a failed provider is targeted or not targeted is controlled by the fifth input parameter to a deactivate script. Therefore, it is the responsibility of a deactivate script to handle this parameter properly, if necessary.

For more information on deactivate scripts, refer to Section 3.3.2, “Deactivate scripts” on page 34.

5.3.3.2 Being multiple providers

When a failed provider has been joined as a provider to multiple groups, each group continues to execute independent failure protocols.

- If multiple groups enable deactivate-on-failure, the deactivate script will be executed during each group’s failure protocol.
- Group Services does not define the order in which the deactivate scripts will be executed by each group because the order in which the individual groups will execute the failure protocols is not defined.

5.4 State value change protocol

The group state value is unique to a group, and all the providers and subscribers can share this information. A Group Services application could utilize this value for this purpose. If the value needs to be changed, a provider can change it by proposing a state value change protocol.

5.4.1 Subroutine call

A provider calls the ha_gs_change_state_value subroutine to propose a state value change protocol. A subroutine prototype is shown in Figure 27 on page 53. On input, the provider provides the state change request block as shown in Figure 40.

```
typedef struct {
    ha_gs_num_phases_t       gs_num_phases;
    ha_gs_time_limit_t       gs_time_limit;
    ha_gs_state_value_t      gs_new_state;
} ha_gs_state_change_request_t;
```

Figure 40. The state change request block

Each field requires the following information:

gs_num_phases
This field requires the number of phases for the protocol. For the ha_gs_num_phases_t type, refer to Figure 10 on page 20.

gs_time_limit
This field requires the voting time limit for the protocol. For the ha_gs_time_limit_t type, refer to Figure 12 on page 22.

gs_new_state
This field requires the group state value information block that contains the new value for the group state value. For the group state value information block, refer to Figure 3 on page 11.

5.4.2 Protocol flow

A protocol uses the number of phases and the voting time limit specified by the provider that proposed the protocol.

5.4.2.1 One-phase protocol

If the state value change protocol is a one-phase protocol, it is approved automatically.

All the providers, including the provider that proposed the protocol, receive the protocol approved notification (ha_gs_protocol_approved_callback) with the updated group state value. The subscribers also receive the notification if they subscribed for it.

The protocol flow of the one-phase state value change protocol is illustrated in Figure 41.

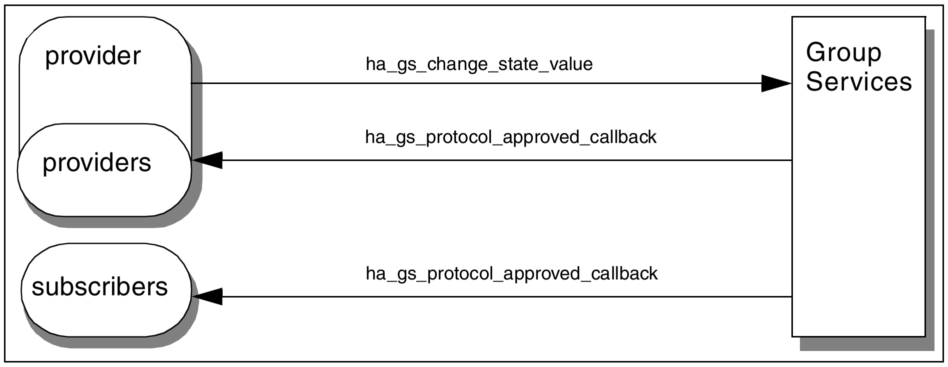


Figure 41. The one-phase state value change protocol

5.4.2.2 N-phase protocol

If the state value change protocol is an n-phase, the protocol has voting phases. All the providers, including the provider that proposed the protocol, participate in voting phases.

If the protocol is approved, the proposed value replaces the current group state value. If other values are proposed during the voting phases of the protocol, the last proposed value replaces the current group state value.

All the providers, including the provider that proposed the protocl, receive a protocol approved notification (ha_gs_protocol_approved_callback) with the updated group state value. The subscribers also receive the notification if they subscribed for it.

The protocol flow of the approved n-phase state value change protocol is illustrated in Figure 42 on page 71.

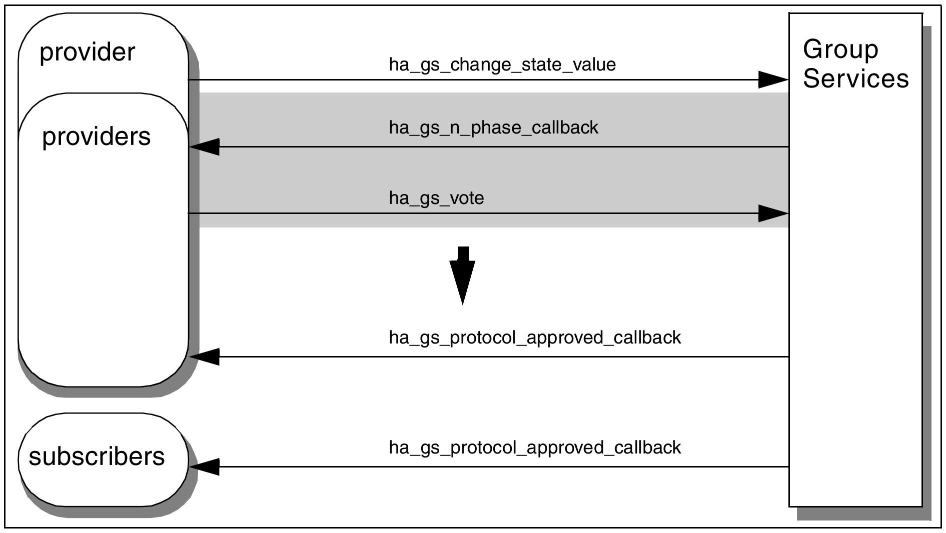


Figure 42. The approved n-phase state value change protocol

If the protocol is rejected, the group state value remains unchanged. All the providers, including the provider that proposed the protocol, receive a *protocol rejected* notification (ha_gs_protocol_rejected_callback). The subscribers do not receive any notification.

The protocol flow of the rejected n-phase state value change protocol is illustrated in Figure 43.

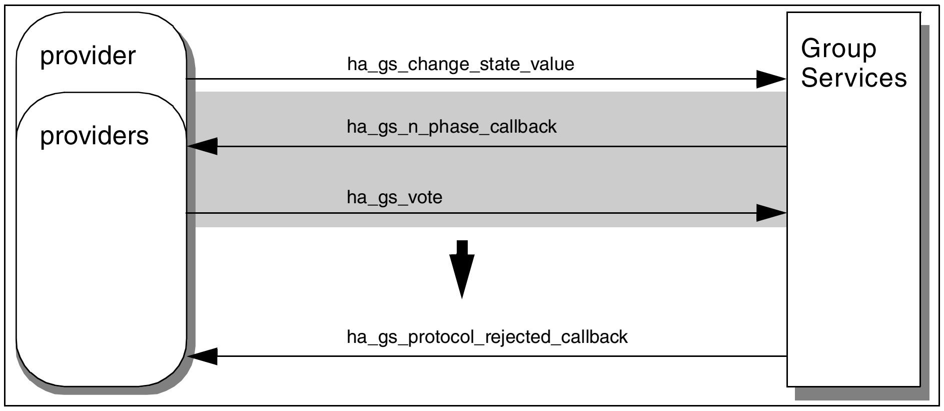


Figure 43. The rejected n-phase state value change protocol

5.4.3 Programming hints

The following sections contain programming hints.

5.4.3.1 Proposing a new group state value on voting

A new group state value can be proposed during the voting phase by a ha_gs_vote subroutine.

For more information, refer to Section 5.12.2, “Proposing a group state value” on page 108.

5.4.3.2 Source-state reflection protocol

If the protocol is approved, it changes the group sate value; therefore, if the group is assigned as a source-group, a source-state reflection protocol is proposed to its target groups.

For more information on the source-state reflection protocol, refer to Section 5.11, “Source-state reflection protocol” on page 103.

5.5 Provider-broadcast message protocol

When a provider wants to send a message to the other providers in a group, it can send a provider broadcast message. To do this, it needs to propose a provider broadcast message protocol.

5.5.1 Subroutine call

A provider calls the ha_gs_send_message subroutine to propose a provider broadcast message protocol. A subroutine prototype is shown in Figure 27 on page 53. At input, the provider provides the message request block as shown in Figure 44.

```
typedef struct {
   ha_gs_num_phases_t          gs_num_phases;
   ha_gs_time_limit_t          gs_time_limit;
   ha_gs_provider_message_t    gs_message;
} ha_gs_message_request_t;
```

Figure 44. The message request block

Each field requires the following information:

gs_num_phases

This field requires the number of phases for the protocol. For the ha_gs_num_phases_t type, refer to Figure 10 on page 20.

gs_time_limit

This field requires the voting time limit for the protocol. For the ha_gs_time_limit_t type, refer to Figure 12 on page 22.

gs_message

This field requires a provider broadcast message block that is to be broadcast to providers. For a provider-broadcast message block, refer to Figure 4 on page 11.

5.5.2 Protocol flow

A protocol uses the number of phases and the voting time limit specified by the provider that proposed the protocol.

5.5.2.1 One-phase protocol

If the provider-broadcast message protocol is a one-phase, the protocol is approved automatically.

All the providers, including the provider that proposed the protocol, receive the protocol approved notification (ha_gs_protocol_approved_callback) with the provider-broadcast message. The subscribers do not receive any notification.

The protocol flow of the one-phase provider-broadcast message protocol is illustrated in Figure 45.

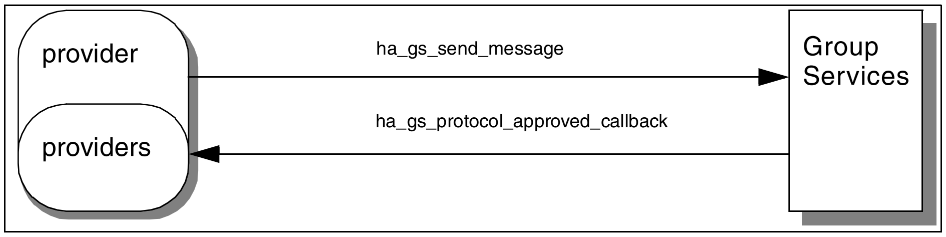


Figure 45. The one-phase provider-broadcast message protocol

5.5.2.2 N-phase protocol

If the provider-broadcast message protocol is an n-phase, the protocol has voting phases. All the providers, including the provider that proposed the protocol, participate in voting phases.

If the protocol is approved, all the providers, including the provider that proposed the protocol, receive a protocol approved notification (ha_gs_protocol_approved_callback). The subscribers do not receive any notification.

If the protocol is approved and a new group state value is proposed during the voting phases of the protocol, a group state value is updated. All the providers, including the provider that proposed the protocol, receive a protocol approved notification with the updated group state value. The subscribers also receive the notification if they subscribed for it.

The protocol flow of the approved n-phase provider-broadcast message protocol is illustrated in Figure 46. The flow assumes that the group state value was updated. If not, the subscribers do not receive any notification.

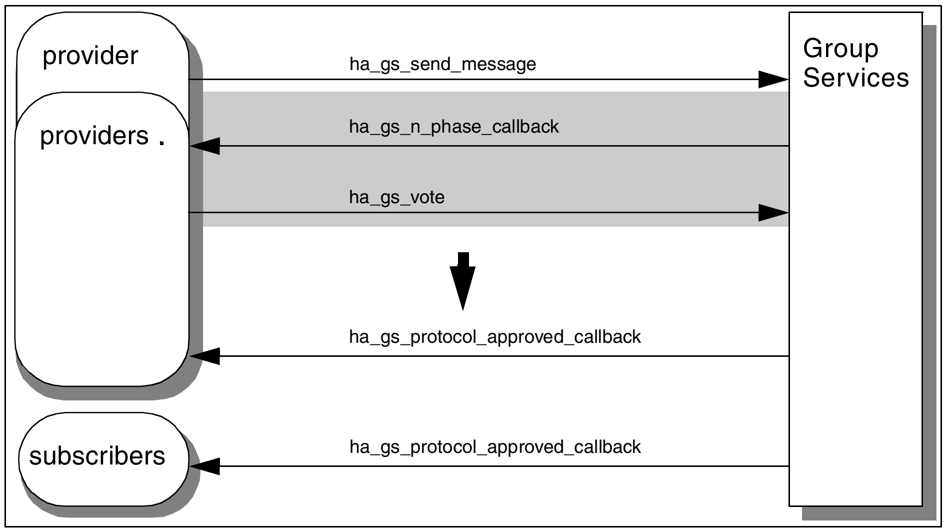


Figure 46. The approved provider-broadcast message protocol

If the protocol is rejected, the group state value remains unchanged. All the providers, including the provider that proposed the protocol, receive a protocol rejected notification (ha_gs_protocol_rejected_callback). The subscribers do not receive any notification.

The protocol flow of the rejected provider-broadcast message protocol is illustrated in Figure 47 on page 75.

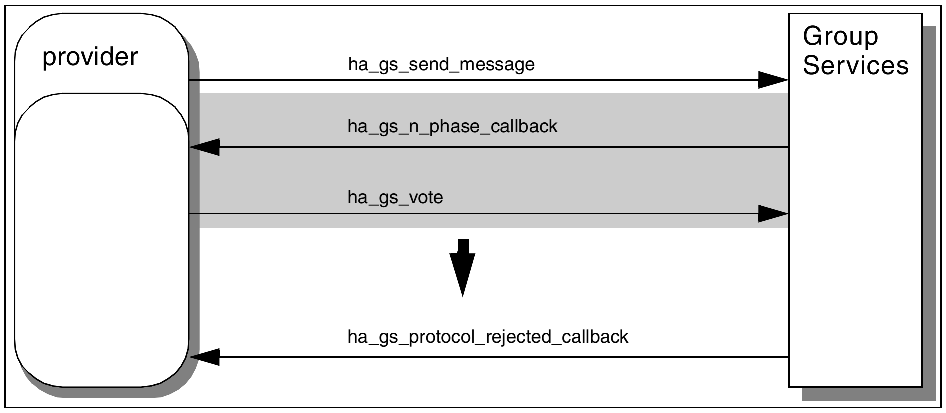


Figure 47. The rejected provider-broadcast message protocol

5.5.3 Programming hints

The following sections contain programming hints.

5.5.3.1 Only once

Unlike the group state value, a provider-broadcast message is presented only once in the proposal block provided by the next n-phase, protocol approved, or protocol rejected notification. It is the responsibility of an application to keep the message for later reference.

5.5.3.2 Sending a provider-broadcast message on voting

A provider-broadcast message can be sent during the voting phase by a ha_gs_vote subroutine.

For more information, refer to Section 5.12.3, “Sending a provider-broadcast message” on page 110.

5.6 Voluntary leave protocol

A provider uses this protocol to leave the group.

5.6.1 Subroutine call

A provider calls the ha_gs_leave subroutine to propose a voluntary leave protocol. A subroutine prototype is shown in Figure 27 on page 53. At input, the provider provides the leave request block as shown in Figure 48 on page 76.

```
typedef struct {
    ha_gs_num_phases_t       gs_num_phases;
    ha_gs_time_limit_t       gs_time_limit;
    unsigned int             gs_leave_code;
} ha_gs_leave_request_t;
```

Figure 48. The leave request block

Each field requires the following information:

gs_num_phases
This field requires the number of phases for the protocol. For the ha_gs_num_phases_t type, refer to Figure 10 on page 20.

gs_time_limit
This field requires the voting time limit for the protocol. For the ha_gs_time_limit_t type, refer to Figure 12 on page 22.

gs_leave_code
This field requires a four-byte value that is defined by the Group Services application and is controlled by the providers in a way that is meaningful to the application. When a provider leaves a group, the leave code is passed to the other providers with the n-phase, protocol approved, or protocol rejected notification. Leave codes are not interpreted by the Group Services subsystem.
The leave code will be stored in the gs_voluntary_leave_code field defined by the ha_gs_leave_info_t type shown in Figure 94 on page 136. For more information, refer to Section 7.2.2, "Notification blocks and their fields" on page 125.

5.6.2 Protocol flow

A protocol uses the number of phases and the voting time limit specified by the provider that proposed the protocol.

5.6.2.1 One-phase protocol

If the voluntary leave protocol is a one-phase protocol, the protocol is approved automatically.

All the providers, including the provider that proposed the protocol, receive the protocol approved notification (ha_gs_protocol_approved_callback) with the updated membership list. The subscribers also receive the notification if they subscribed for it.

The protocol flow of the one-phase voluntary leave protocol is illustrated in Figure 49 on page 77.

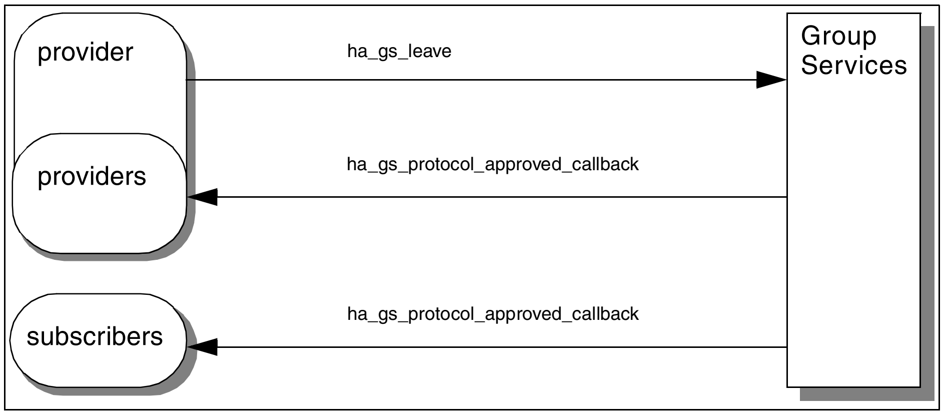

provider
providers
subscribers
ha_gs_leave
ha_gs_protocol_approved_callback
ha_gs_protocol_approved_callback
Group Services

Figure 49. The one-phase voluntary leave protocol

5.6.2.2 N-phase protocol

If the voluntary leave protocol is an n-phase, the protocol has voting phases. All the providers, except the provider that proposed the protocol, participate in voting phases.

The leaving provider receives an n-phase notification (ha_gs_n_phase_callback) as an initial notification. At this point, the leaving provider has been removed from the membership list. The other providers proceed voting phases.

If the protocol is approved, all the remaining providers receive a protocol approved notification (ha_gs_protocol_approved_callback) with the updated membership list. If a new group state value was proposed during the voting phases, the notification includes the updated group state value as well. The subscribers also receive the notification if they subscribed for it.

The protocol flow of the approved voluntary leave protocol is illustrated in Figure 50 on page 78.

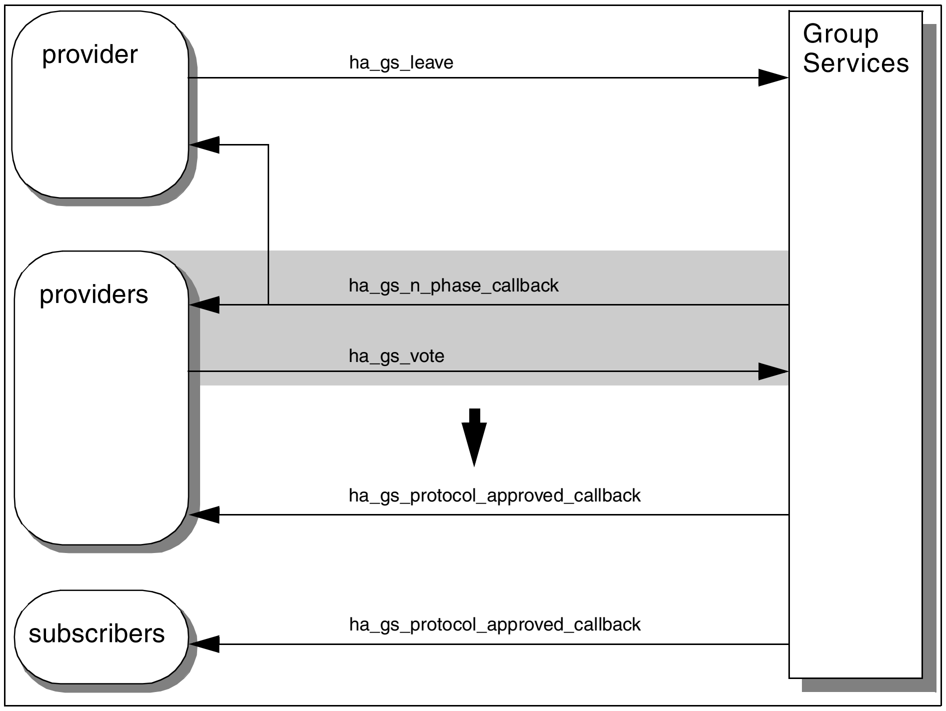


Figure 50. The approved n-phase voluntary leave protocol

If the protocol is rejected, the group state value remains unchanged; however, the provider that proposed the protocol has been removed from the membership list. All the remaining providers receive a protocol rejected notification (ha_gs_protocol_rejected_callback) with the updated membership list. The subscribers also receive the notification if they subscribed for it.

The protocol flow of the rejected voluntary leave protocol is illustrated in Figure 51 on page 79.

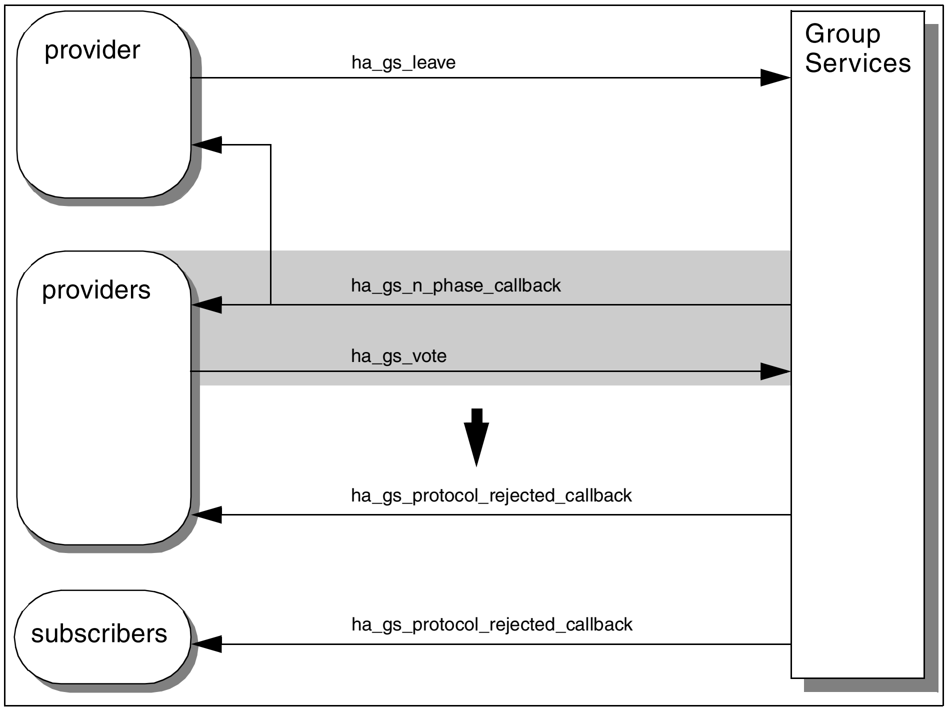


Figure 51. The rejected n-phase voluntary leave protocol

5.6.3 Programming hints

The following sections contain programming hints.

5.6.3.1 A leaving provider

After calling the ha_gs_leave subroutine, the leaving provider will receive one of the following notifications:

- Delayed error notification

  There are some reasons that a leaving provider receives this notification:

  - The given parameters or fields are not valid.
  - The connection to the Group Services is lost.
  - The provider's group is already executing another protocol.

- N-phase notification

  Regardless of whether the protocol is approved or rejected, the leaving provider has been removed from the membership list when the provider receives this notification; therefore, the provider (actually, it is not a provider at this point) should not vote for this notification. If it does, it receives a delayed error notification.

5.7 Goodbye protocol

In normal situations, a provider uses a voluntary leave protocol to leave a group. However, a provider must receive a notification to leave a group. Also, a voluntary leave protocol does not involve a deactivate-on-failure facility. If a provider wants to leave a group very quickly and wants the Group Services to do the cleanup procedure, in other words, if the provider executes a deactivate script, they have a choice to propose a goodbye protocol.

5.7.1 Subroutine call

A provider calls the ha_gs_goodbye subroutine to propose a goodbye protocol. A subroutine prototype is shown in Figure 27 on page 53. At input, the provider specifies only the provider token. This is different from the other protocols that are proposed by providers.

5.7.2 Protocol flow

The protocol flow of a goodbye protocol is exactly the same as the failure leave protocol described in Section 5.3.2, “Protocol flow” on page 61. The only difference between them is that a goodbye protocol is proposed by a provider while a failure leave protocol is proposed by the Group Services subsystem.

If the ha_gs_goodbye subroutine returns with an HA_GS_OK return code, the calling provider has been removed from the membership list. The provider will not receive any asynchronous errors.

5.7.3 Programming hints

The following sections contain programming hints.

5.7.3.1 Voluntary leave or goodbye

When a leaving provider proposes a goodbye protocol, the remaining providers receive an n-phase, protocol approved, or protocol rejected notification. The gs_protocol_type filed in these notifications’ notification block has the value of HA_GS_FAILURE_LEAVE shown in Figure 88 on page 128. In other words, a goodbye protocol is treated as a part of the failure leave protocol.

To determine whether it is a failure leave protocol or a goodbye protocol, check the gs_leave_info field in the proposal block. If the gs_voluntary_or_failure has a value of HA_GS_PROVIDER_SAID_GOODBYE as shown in Figure 95 on page 137, it is a goodbye protocol. If it has a value of HA_GS_PROVIDER_FAILURE, it is a failure leave protocol.

For more information, refer to Section 7.2.2, “Notification blocks and their fields” on page 125.

5.8 Expel protocol

The expel protocol allows providers to propose the removal of one or more providers from the group. This protocol could be useful in the following situations:

- A provider has received an announcement notification that another provider is not responsive or has detected an internal error.
- A provider has received an announcement notification that another provider failed to submit a vote within the specified time limit during a previously completed n-phase protocol.
- A provider has detected, through some other means, that another provider is not behaving as expected in the context of the application the group is running.

5.8.1 Subroutine call

A provider calls the ha_gs_expel subroutine to propose an expel protocol. A subroutine prototype is shown in Figure 27 on page 53. At input, the provider provides the expel request block as shown in Figure 52.

```
typedef struct {
    ha_gs_num_phases_t      gs_num_phases;
    ha_gs_time_limit_t      gs_time_limit;
    ha_gs_membership_t      gs_expel_list;
    int                     gs_deactivate_phase;
    char                    *gs_deactivate_flag;
} ha_gs_expel_request_t;
```

Figure 52. The expel request block

Each field requires the following information:

gs_num_phases

This field requires the number of phases for the protocol. For the ha_gs_num_phases_t type, refer to Figure 10 on page 20.

gs_time_limit

This field requires the voting time limit for the protocol. For the ha_gs_time_limit_t type, refer to Figure 12 on page 22.

gs_expel_list

This field requires a list of providers that are targeted to be expelled. The list uses the same type as the membership information block shown in Figure 2 on page 10.

gs_deactivate_phase

This field requires the phase number in which a deactivate script should be executed against the providers that are being expelled. If this field contains 0, no deactivate script will be executed.

gs_deactivate_flag

This field requires a pointer to a flag that is to be passed to the deactivate script. A flag is a null-terminated string with a maximum length of 256 bytes. If you specify a string that is longer than 256 bytes, it will be truncated. If the pointer is NULL, no flag will be passed to the deactivate script. For a deactivate script, refer to Section 3.3, “Deactivate-on-failure facility” on page 33.

5.8.2 Protocol flow

A protocol uses the number of phases and the voting time limit specified by the provider that proposed the protocol.

5.8.2.1 One-phase protocol without deactivate-on-failure

If the expel protocol is a one-phase protocol, and the value of the gs_deactivate_phase field in the expel request block is 0. The protocol is approved automatically, and no deactivate script is executed during the protocol.

All the providers, including the providers that are targeted for expulsion, and subscribers receive the protocol approved notification (ha_gs_protocol_approved_callback) with the updated membership list.

The providers that are targeted for expulsion are removed from the membership list. However, their processes are still up and running. It is the responsibility of an application to deal with these processes.

The flow of the one-phase expel protocol without deactivate-on-failure is illustrated in Figure 53 on page 83.

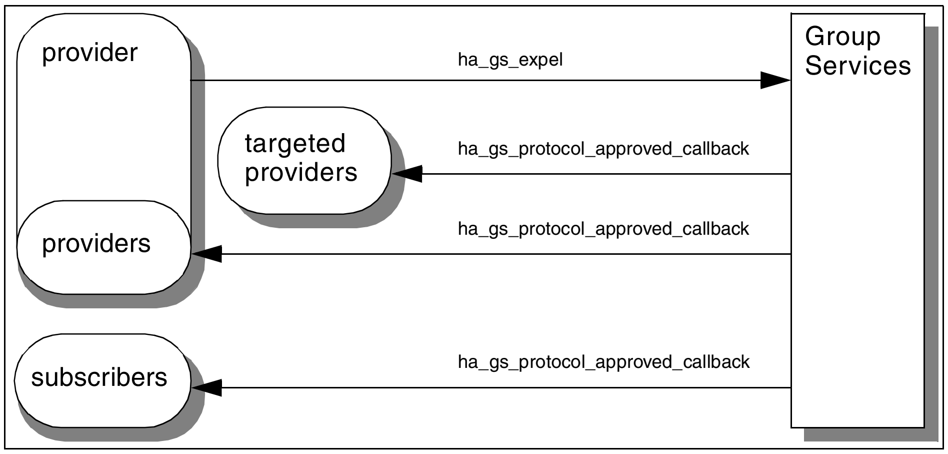

provider
ha_gs_expel
Group Services
targeted providers
ha_gs_protocol_approved_callback
providers
ha_gs_protocol_approved_callback
subscribers
ha_gs_protocol_approved_callback

Figure 53. The one-phase expel protocol without deactivate-on-failure

5.8.2.2 One-phase protocol with deactivate-on-failure

If the expel protocol is a one-phase protocol and the value of the gs_deactivate_phase field in the expel request block is 1 (notice that a one-phase protocol has only one phase), the protocol is approved automatically, and the deactivate script is executed immediately after the protocol begins execution.

The Group Services has forked a child process to execute a deactivate script; then, it sends the protocol approval notification (ha_gs_protocol_approved_callback) to the providers that are targeted for expulsion. However, it is unpredictable whether the provider will receive the notification before or after the script executes. Group Services does not wait for the script to complete execution before it sends the notification. The exit code of the script is not inspected.

The providers that are targeted for expulsion are removed from the membership list. However, their process may or may not be terminated successfully by the deactivate script. It is the responsibility of an application to deal with these processes.

The remaining providers receive the protocol approved notification (ha_gs_protocol_approved_callback) with the updated membership list. The subscribers also receive the notification if they subscribed for it.

The protocol flow of the one-phase expel protocol with deactivate-on-failure is illustrated in Figure 54.

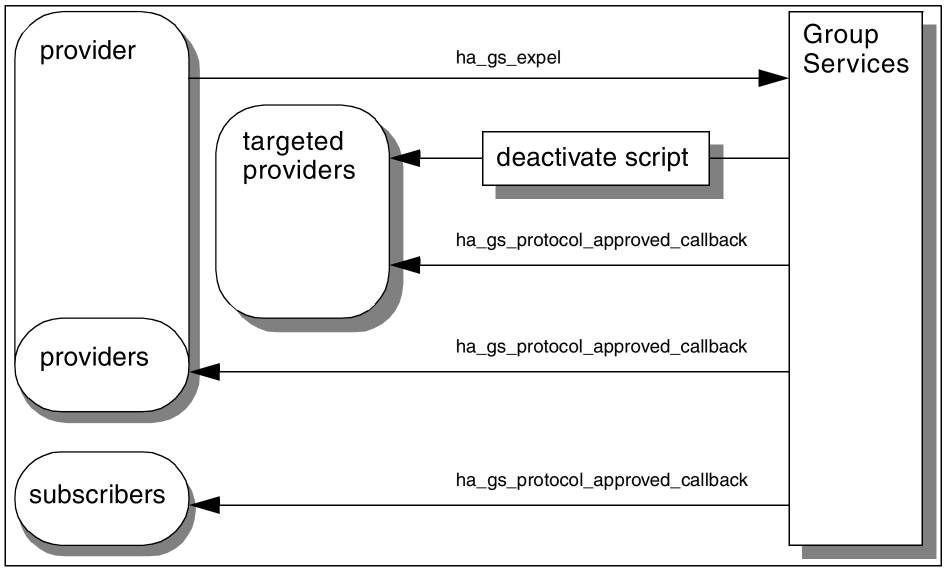

provider
ha_gs_expel
Group Services
targeted providers
deactivate script
ha_gs_protocol_approved_callback
providers
ha_gs_protocol_approved_callback
subscribers
ha_gs_protocol_approved_callback

Figure 54. The one-phase expel protocol with deactivate-on-failure

5.8.2.3 N-phase protocol without deactivate-on-failure

If the expel protocol is an n-phase and the value of the gs_deactivate_phase field in the expel request block is 0, the protocol has voting phases, and no deactivate script is executed during the protocol.

The providers that are targeted for expulsion do not participate in voting phases.

If the protocol is approved, the providers that are targeted for expulsion are removed from the membership list. All the providers, including the providers that are targeted for expulsion, receive the protocol approved notification (ha_gs_protocol_approved_callback) with the updated membership list. If a new group state value was proposed during the voting phases, the notification contains the updated group state value as well. The subscribers also receive the notification if they subscribed for it.

The providers that are targeted for expulsion are removed from the membership list. However, their processes are still up and running. It is the responsibility of an application to deal with these processes.

The protocol flow of the approved n-phase expel protocol without deactivate-on-failure is illustrated in Figure 55.

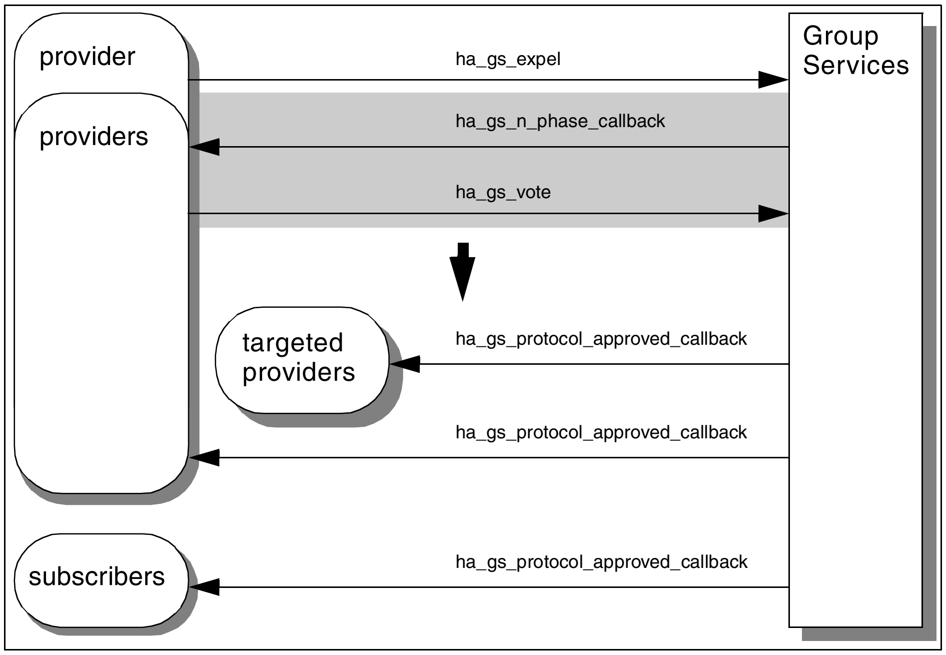


Figure 55. The approved n-phase expel protocol without deactivate-on-failure

If the protocol is rejected, the providers that are targeted for expulsion are not removed from the membership list, and the group state value remains unchanged. All the providers, except the providers that are targeted for expulsion, receive a protocol rejected notification (ha_gs_protocol_rejected_callback). The subscribers do not receive any notification.

The protocol flow of the rejected n-phase expel protocol without deactivate-on-failure is illustrated in Figure 56 on page 86.

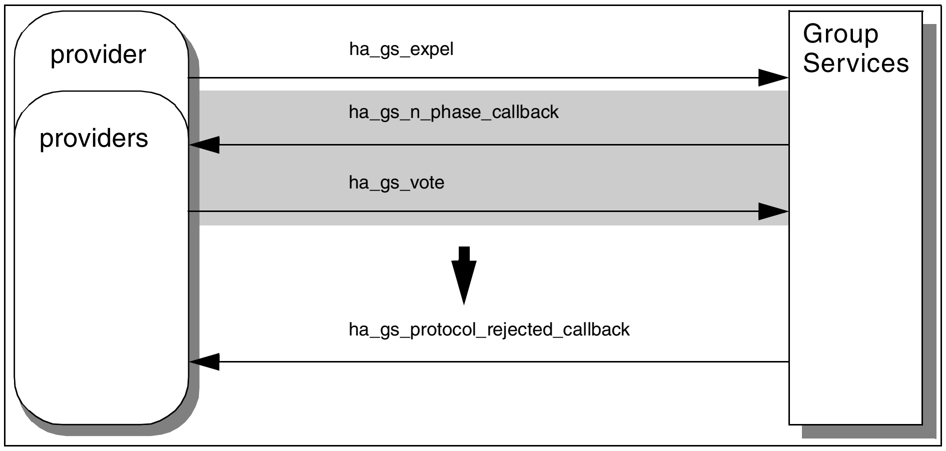


Figure 56. The rejected n-phase expel protocol without deactivate-on-failure

5.8.2.4 N-phase protocol with deactivate-on-failure

If an expel protocol is an n-phase protocol and the value of the gs_deactivate_phase field in the expel request block is not 0, the protocol has voting phases, and a deactivate script is executed during the protocol.

If the deactivate script is to be executed in a future voting phase, Group Services votes to continue as the targeted provider's vote for each interim voting phase. Therefore, even though all the providers vote to approve, the protocol still continues the voting phase.

The beginning of the protocol flow of the n-phase expel protocol with deactivate-on-failure is illustrated in Figure 57 on page 87.

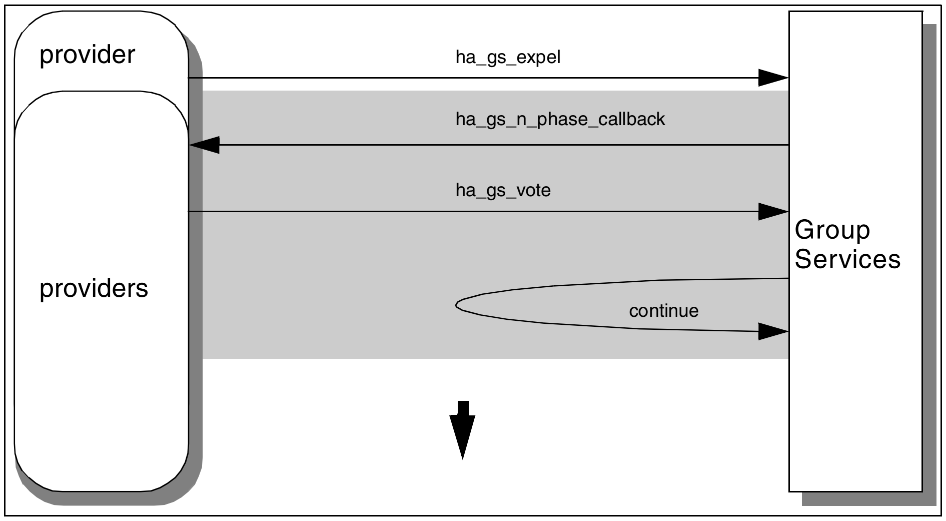


Figure 57. The n-phase expel protocol with deactivate-on-failure

The voting phase is repeated at least until the phase that has one of the following conditions:

- One or more of the providers vote to reject.
- The phase comes to the time a deactivate script is to be executed.

If one or more of the providers vote to reject before the phase in which the deactivate script is to be executed, the expel protocol is rejected, and the deactivate script will not be executed.

If the phase comes to the time that a deactivate script is to be executed and Group Services has forked a child process to execute the deactivate script, Group Services sends the protocol approval notification (ha_gs_protocol_approved_callback) to the providers that are targeted for expulsion. However, whether the provider will receive the notification before or after the script executes is unpredictable. Group Services waits for the script to complete execution within the voting time limit. The exit code of the script is inspected.

According to the exit code, Group Services votes in place of the targeted providers vote for the phase. If the protocol requires more voting phases, Group Services continues to vote as the targeted providers vote for each subsequent voting phase. The voting value is determined by the following rule:

- The Group Services votes to approve if the exit code is 0.

- The Group Services votes the current default vote value in the following cases:
  - The exit code is not 0.
  - If the protocol has specified a voting time limit and the script does not complete its execution within the voting time limit.
  - The script is not specified or it is not executable.

The protocol flow of the deactivate script execution phase is illustrated in Figure 58 on page 89.

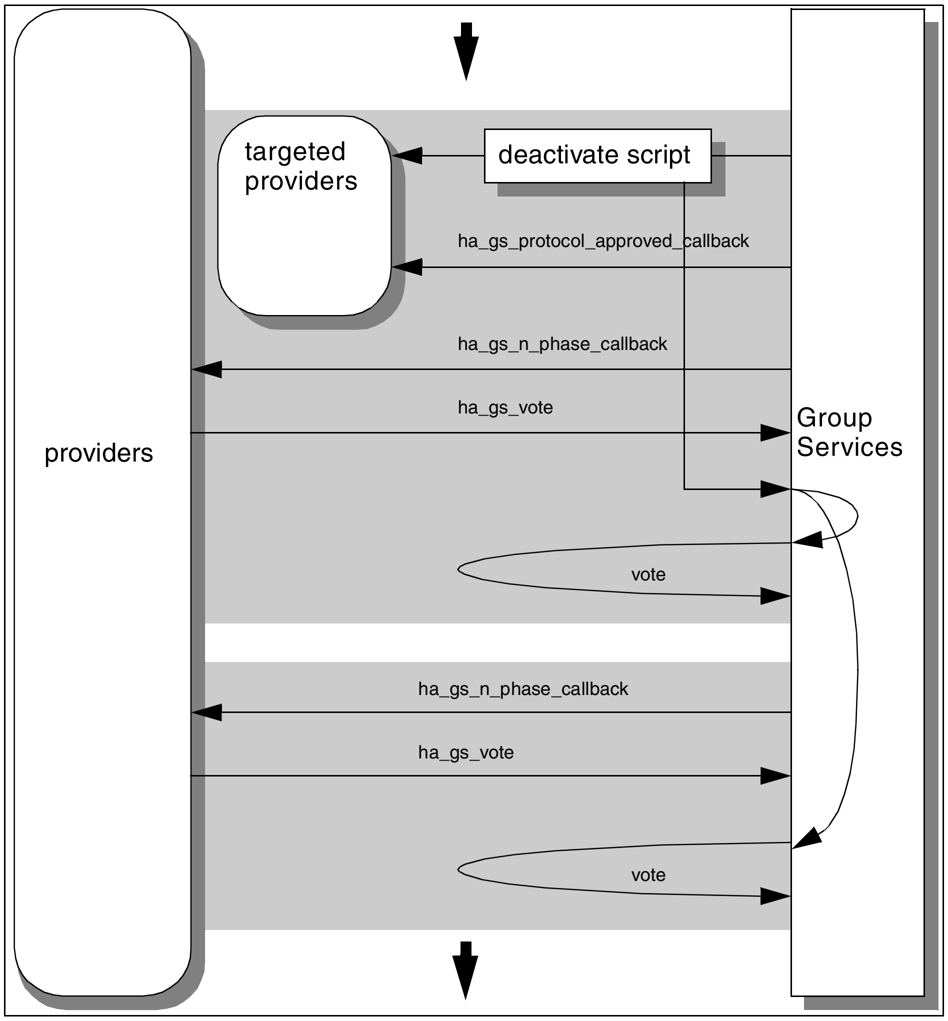


Figure 58. The deactivate script execution phase

If the protocol is approved, the providers that are targeted for expulsion are removed from the membership list. The remaining providers receive the protocol approved notification (ha_gs_protocol_approved_callback) with the updated membership list. If a new group state value was proposed during the voting phases, the notification contains the updated group state value inas well. The subscribers also receive the notification if they subscribed for it.

The protocol flow of the approved n-phase expel protocol with deactivate-on-failure is illustrated in Figure 59.

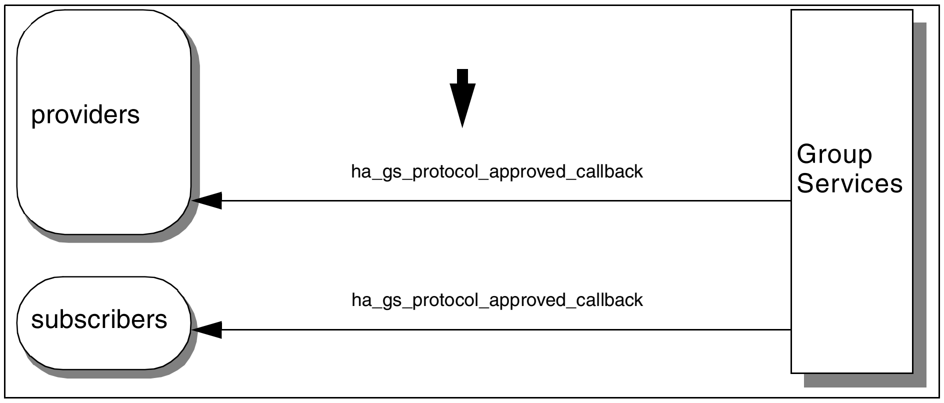

providers
Group Services
ha_gs_protocol_approved_callback
subscribers
ha_gs_protocol_approved_callback

Figure 59. The approved n-phase expel protocol with deactivate-on-failure

If the protocol is rejected, the providers that are targeted for expulsion are not removed from the membership list, and the group state value remains unchanged. The providers that are not targeted for expulsion receive a protocol rejected notification (ha_gs_protocol_rejected_callback). The subscribers do not receive any notification.

The providers that are targeted for expulsion are not removed from the membership list. However, if the deactivate script causes a provider to exit, the Group Services proposes a failure leave protocol for that provider.

The protocol flow of the rejected n-phase expel protocol with deactivate-on-failure is illustrated in Figure 60.

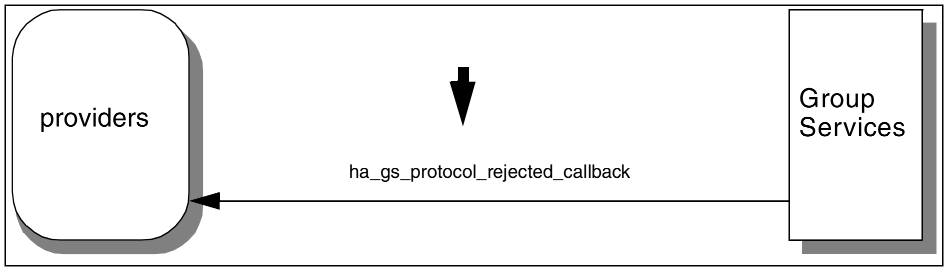


Figure 60. The rejected n-phase expel protocol with deactivate-on-failure

5.8.3 Programming hints

The following sections contain programming hints.

5.8.3.1 Joining to multiple groups (case 1)

If a single process has joined as a provider to multiple groups and one of those provider instances has been expelled from a group, the effect on the other instances is as follows:

- If the process no longer exists (it is killed or has failed) as a result of the expel protocol, the other provider instances of the process are handled through failure leave protocols in their groups.
- If the process still exists, the other provider instances of the process are not affected and continue as full participants in their groups.

5.8.3.2 Joining to multiple groups (case 2)

If a single process has joined as a provider to multiple groups and more than one of the groups are simultaneously executing expel protocols that target those providers (for example, because the process is unresponsive), the order in which deactivate scripts are executed against the process is undefined by Group Services.

Because each group’s expel protocol proceeds independently, Group Services does not coordinate the execution of the deactivate script for each group’s protocol. If all groups approve their expel protocols and the process is killed, no failure leave protocols are executed. If one or more groups reject their expel protocols, but the process is killed in the course of executing the deactivate script, those groups initiate failure leave protocols to remove the failed provider.

5.9 Change-attributes protocol

A group has its own attributes that specify characteristics of the group. These attributes are defined when the first provider in the group creates the group. Usually, the attributes will not change; however, if it is necessary, providers are allowed to change some of theses attributes.

5.9.1 Subroutine call

A provider calls the ha_gs_change_attributes subroutine to propose a change-attributes protocol. A subroutine prototype is shown in Figure 27 on page 53. At input, the provider specifies the attribute change request block as shown in Figure 61 on page 92.

```
typedef struct {
    ha_gs_num_phases_t          gs_num_phases;
    ha_gs_time_limit_t          gs_time_limit;
    ha_gs_group_attributes_t    *gs_group_attributes;
    ha_gs_membership_t          *gs_backlevel_providers;
} ha_gs_attribute_change_request_t;
```

Figure 61. The attribute change request block

Each of the fields requires the following information:

gs_num_phases
: This field requires the number of phases for the protocol. For the ha_gs_num_phases_t type, refer to Figure 10 on page 20.

gs_time_limit
: This field requires the voting time limit for the protocol. For the ha_gs_time_limit_t type, refer to Figure 12 on page 22.

gs_group_attributes
: This field requires a pointer to the group attributes block that is to replace the current group attributes. The group attributes block is shown in Figure 1 on page 8.

gs_backlevel_providers
: This field requires a NULL pointer when the subroutine is called. If the provider receives a delayed error notification with the value of HA_GS_BACKLEVEL_PROVIDERS, this field contains a pointer to a list of providers that are in the group and that were compiled and linked against an older version of Group Services shared libraries. The membership information block shown in Figure 2 on page 10 is used for this list.

5.9.2 Protocol flow

A protocol uses the number of phases and the voting time limit specified by the provider that proposed the protocol.

5.9.2.1 One-phase protocol

If the change-attributes protocol is a one-phase, the protocol is approved automatically.

All the providers, including the provider that proposed the protocol, receive the protocol approved notification (ha_gs_protocol_approved_callback) with the updated group attributes. The subscribers do not receive any notification.

The protocol flow of the one-phase state value change protocol is illustrated in Figure 62.

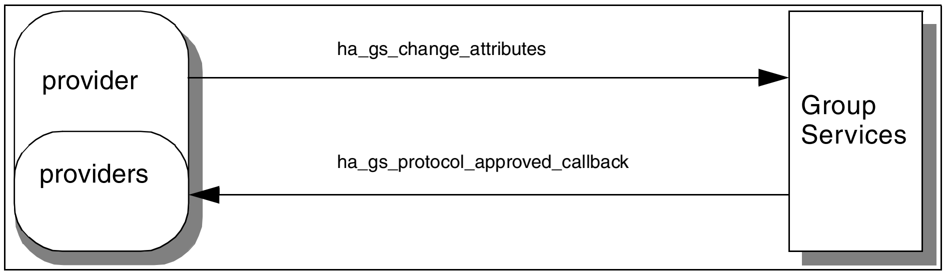


Figure 62. The one-phase change-attributes protocol

5.9.2.2 N-phase protocol

If the change-attributes protocol is an n-phase, the protocol has voting phases. All the providers, including the provider that proposed the protocol, participate in voting phases.

If the protocol is approved, the proposed group attributes replaces the current group attributes. All the providers, including the provider that proposed the protocol, receive a protocol approved notification (ha_gs_protocol_approved_callback) with the updated group attributes. The subscribers do not receive any notification.

If a new group state value was proposed during the voting phases, the notification contains the updated group state value as well. The subscribers also receive the notification, if they subscribed for it.

The protocol flow of the approved n-phase change-attributes protocol is illustrated in Figure 63 on page 94.

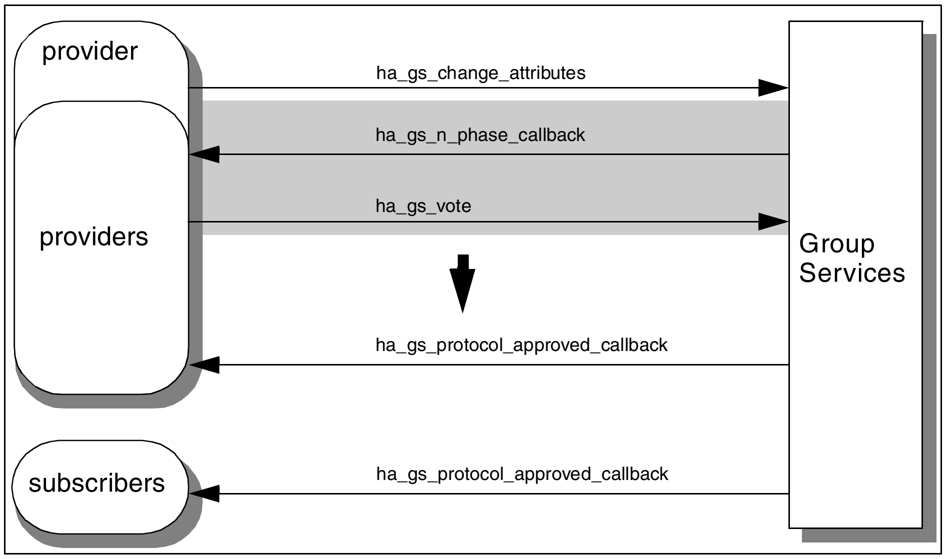


Figure 63. The approved n-phase change-attributes protocol

If the protocol is rejected, the group state value remains unchanged. All the providers, including the provider that proposed the protocol, receive a protocol rejected notification (ha_gs_protocol_rejected_callback). The subscribers do not receive any notification.

The protocol flow of the rejected n-phase state value change protocol is illustrated in Figure 64.

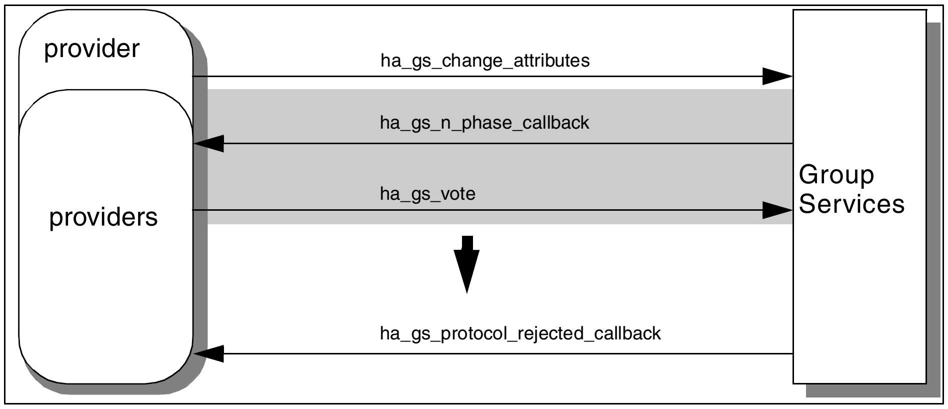


Figure 64. The rejected n-phase change-attributes protocol

5.9.3 Programming hints

The following section contains programming hints.

5.9.3.1 Changeable attributes

The following attributes can be changed through an ha_gs_change_attributes subroutine call:

- gs_client_version
- gs_batch_control
- gs_num_phases
- gs_source_reflection_num_phases
- gs_group_default_vote
- gs_merge_control
- gs_time_limit
- gs_source_reflection_time_limit

For more information on the group attributes, refer to Section 2.2.2.1, “The group attributes” on page 7.

5.10 Cast-out protocol

Group Services proposes a cast-out protocol to a group that uses a source-target facility. When a source-group has changed its membership list, Group Services may need to cast out some providers in the target-group. If this is the case, Group Services proposes a cast-out protocol to the target-group. For more details about a source-target facility, refer to Section 3.4, “Source-target facility” on page 36.

Note

This book uses the following semantics for a provider that is to be cast out from a target-group:

- If a provider's socket connection to the Group Services is broken at the time the cast-out protocol begins execution, this provider is called *a failed cast-out provider*.
- If a provider's socket connection to the Group Services is *not* broken at the time the cast-out protocol begins execution, this provider is called *an active cast-out provider*.

You do not see these semantics in the official Group Services documentations.

The behaviors of a cast-out protocol and a failure leave protocol are quite similar; the reasons why they are proposed are different, though. There are two differences between them:

1. In the case of cast-out protocols, Group Services needs to decide if it should execute a deactivate script or not when deactivate-on-failure is activated. On the other hand, the Group Services always executes a deactivate script in the case of failure leave protocols when deactivate-on-failure is activated.

   The decision is made as follows: If a provider that is to be cast out from a target-group is a failed cast-out provider, the script will be executed. If it is an active cast-out provider, the script will *not* be executed against it.

   If batching cast-out protocols is enabled (this is equal to batching failure leave protocols enabled), and if both failed cast-out providers and active cast-out providers are targeted by a cast-out protocol, the script will be executed only against the failed cast-out providers.

2. In the case of cast-out protocols, active cast-out provides receive a final notification (a protocol approved notification or a protocol rejected notification) and failed cast-out providers do not receive any notification while, in the case of failure leave protocols, failed providers do not receive any notification.

5.10.1 Protocol proposal

The Group Services subsystem uses the following fields of the group attributes block to propose and execute a protocol:

- The gs_num_phases field is used for the number of phases.

- The gs_time_limit field is used for the voting time limit.
- The gs_batch_control field is used to enable/disable batching cast-out protocols.
- The gs_batch_control field is also used to enable/disable a deactivate-on-failure facility. A deactivate script has already been registered by the ha_gs_init subroutine.

5.10.2 Protocol flow

A protocol uses the number of phases and the voting time limit specified in the group attributes block already registered.

The providers that are to be cast out from the group do not participate in any voting phases of the protocol.

Batching cast-out protocols

If batching cast-out protocols is enabled, one cast-out protocol handles one or more providers that are to be cast out. If not, one cast-out protocol handles only one provider that is to be cast out. This section assumes that batching cast-out protocols is enabled. Therefore providers that are to be cast out are treated as plural. If this is not the case, read it as singular.

5.10.2.1 One-phase protocol without deactivate-on-failure

If a cast-out protocol is one-phase and deactivate-on-failure is disabled, the protocol is approved automatically, and a deactivate script is not executed.

The providers that are to be cast out are removed from the membership list. All the remaining providers receive a protocol approved notification (ha_gs_protocol_approved_callback) with the updated membership list. The subscribers also receive the notification if they subscribed for it.

All the active cast-out providers receive a protocol approved notification. All the failed cast-out providers do not receive any notification.

The protocol flow of the one-phase cast-out protocol without deactivate-on-failure is illustrated in Figure 65 on page 98.

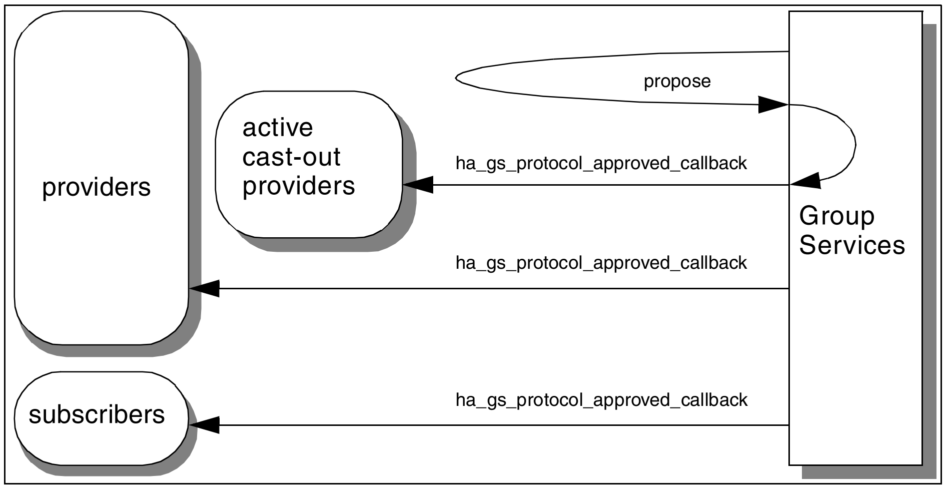

providers
active cast-out providers
subscribers
propose
ha_gs_protocol_approved_callback
ha_gs_protocol_approved_callback
ha_gs_protocol_approved_callback
Group Services

Figure 65. The one-phase cast-out protocol without deactivate-on-failure

5.10.2.2 One-phase protocol with deactivate-on-failure

If deactivate-on-failure is enabled, and if one or more failed cast-out providers are included in the cast-out targets, the protocol is approved automatically, and a deactivate script will be executed against failed cast-out providers immediately after the protocol begins execution. The Group Services does not wait for the script to complete execution. The exit code of the deactivate script is not inspected.

The providers that are to be cast out are removed from the membership list. All the remaining providers receive a protocol approved notification (ha_gs_protocol_approved_callback) with the updated membership list. The subscribers also receive the notification if they subscribed for it.

All the active cast-out providers receive a protocol approved notification. All the failed cast-out providers do not receive any notification.

The protocol flow of the one-phase cast-out protocol with deactivate-on-failure is illustrated in Figure 66 on page 99.

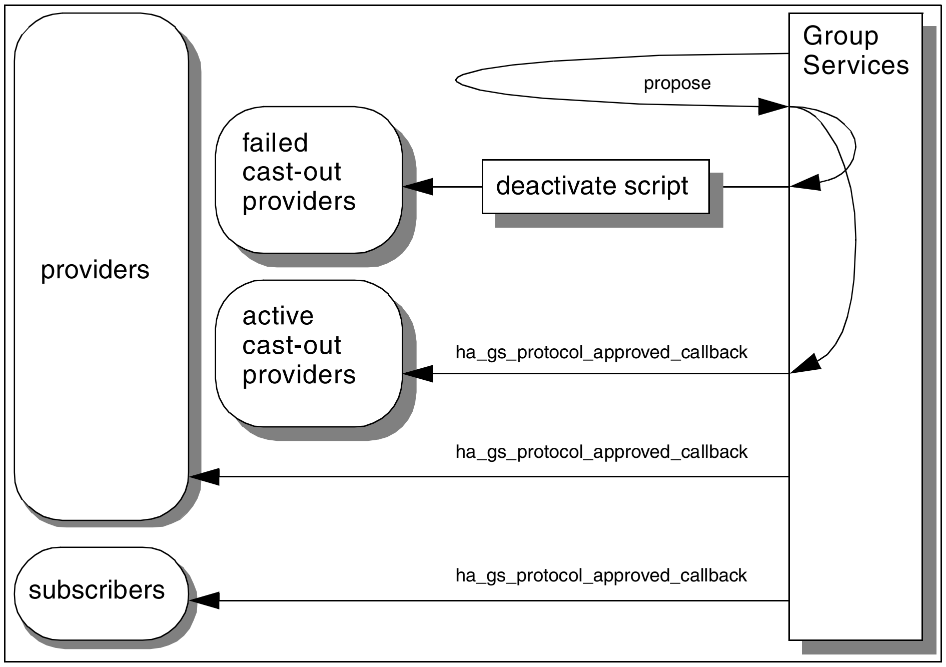

Group Services
propose
failed cast-out providers
deactivate script
providers
active cast-out providers
ha_gs_protocol_approved_callback
ha_gs_protocol_approved_callback
subscribers
ha_gs_protocol_approved_callback

Figure 66. The one-phase cast-out protocol with deactivate-on-failure

5.10.2.3 N-phase protocol beginning without deactivate-on-failure

If a cast-out protocol is n-phase and deactivate-on-failure is disabled, the protocol has voting phases and a deactivate script is not executed.

All the providers except the providers that are to be cast out from the target-group participate in the voting phases.

The flow of the beginning of an n-phase cast-out protocol without deactivate-on-failure is illustrated in Figure 67 on page 100.

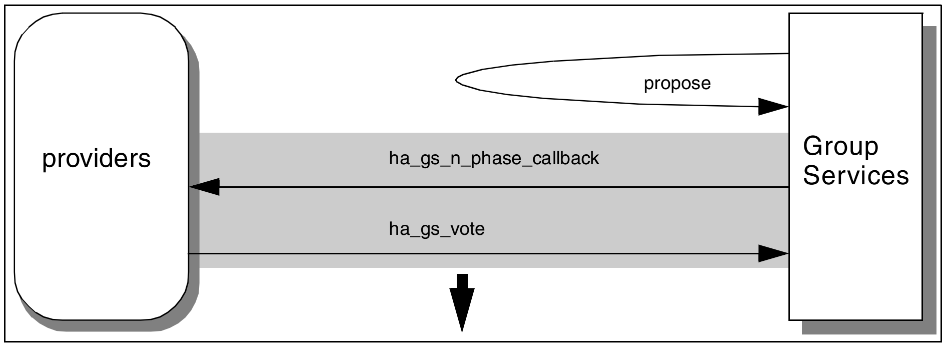


Figure 67. The beginning of protocol without deactivate-on-failure

5.10.2.4 N-phase protocol beginning with deactivate-on-failure

If deactivate-on-failure is enabled and if one or more failed cast-out providers are included in the cast-out targets, a deactivate script will be executed against failed cast-out providers immediately after the protocol begins execution. Group Services waits for the script to complete execution within the voting time limit. The exit code of the deactivate script is inspected.

According to the exit code, the Group Services votes as the failed cast-out providers' vote for the phase. If the protocol requires more voting phases, Group Services continues to vote as the failed cast-out providers' vote for each subsequent voting phase. The voting value is determined by the following rule:

- The Group Services votes to approve, if the exit code is 0.
- The Group Services votes current default vote value, in the following cases:
  - The exit code is not 0.
  - If the protocol has specified a voting time limit and the script does not complete its execution within voting time limit.
  - The script is not specified or it is not executable.

The flow of the beginning of an n-phase cast-out protocol with deactivate-on-failure is illustrated in Figure 68 on page 101.

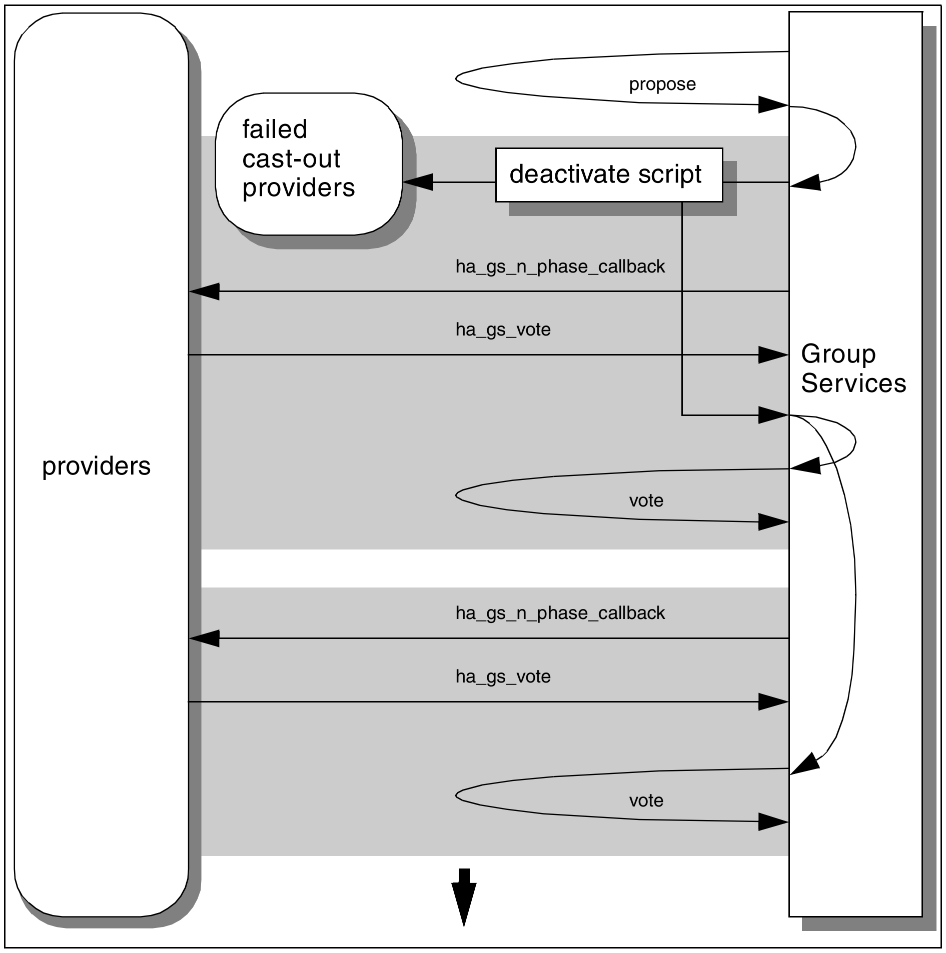


Figure 68. The beginning of protocol with deactivate-on-failure

5.10.2.5 N-phase protocol ending

If the protocol is approved, the providers that are to be cast out are removed from the membership list. All the remaining providers receive a protocol approved notification (ha_gs_protocol_approved_callback) with the updated membership list. If a new group state value was proposed during the voting phases of the protocol, a group state value is updated, and the protocol approved notification includes the updated group state value as well. The subscribers also receive the notification if they subscribed for it.

All the active cast-out providers receive the protocol approved notification. All the failed cast-out providers do not receive any notification.

The flow of the ending of an approved n-phase cast-out protocol is illustrated in Figure 69.

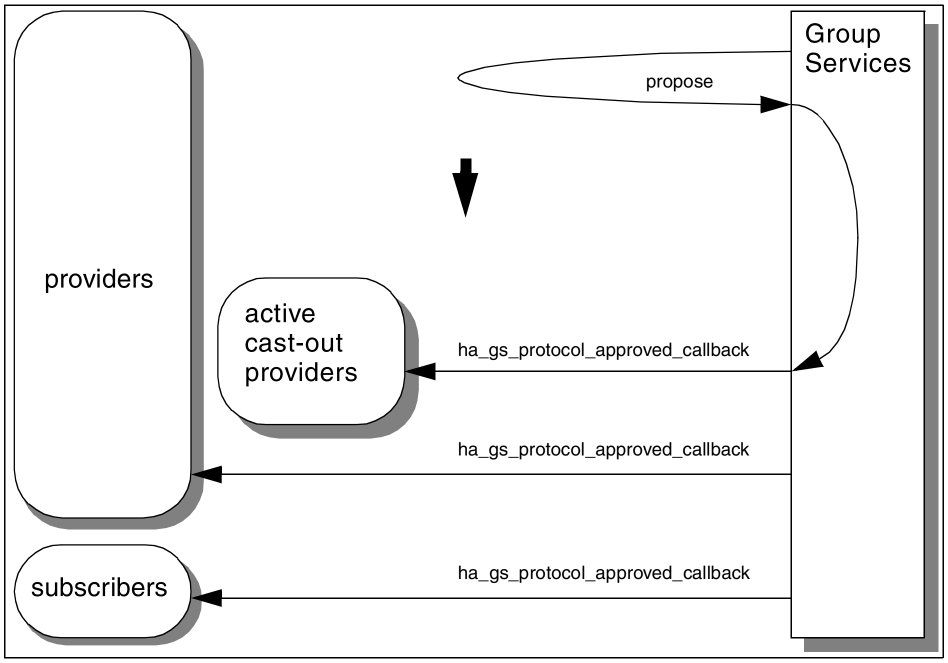

Group Services
propose
providers
active cast-out providers
ha_gs_protocol_approved_callback
ha_gs_protocol_approved_callback
subscribers
ha_gs_protocol_approved_callback

Figure 69. The ending of approved n-phase cast-out protocol

If the protocol is rejected, the providers that are to be cast out *are still removed* from the membership list, and the group state value remains unchanged. The remaining providers receive the protocol rejected notification (ha_gs_protocol_rejected_callback) with an updated membership list. The subscribers also receive the notification if they subscribed for it.

The active cast-out providers receive the protocol rejected notification. The failed cast-out providers do not receive any notification.

The flow of the ending of rejected n-phase cast-out protocol is illustrated in Figure 70 on page 103.

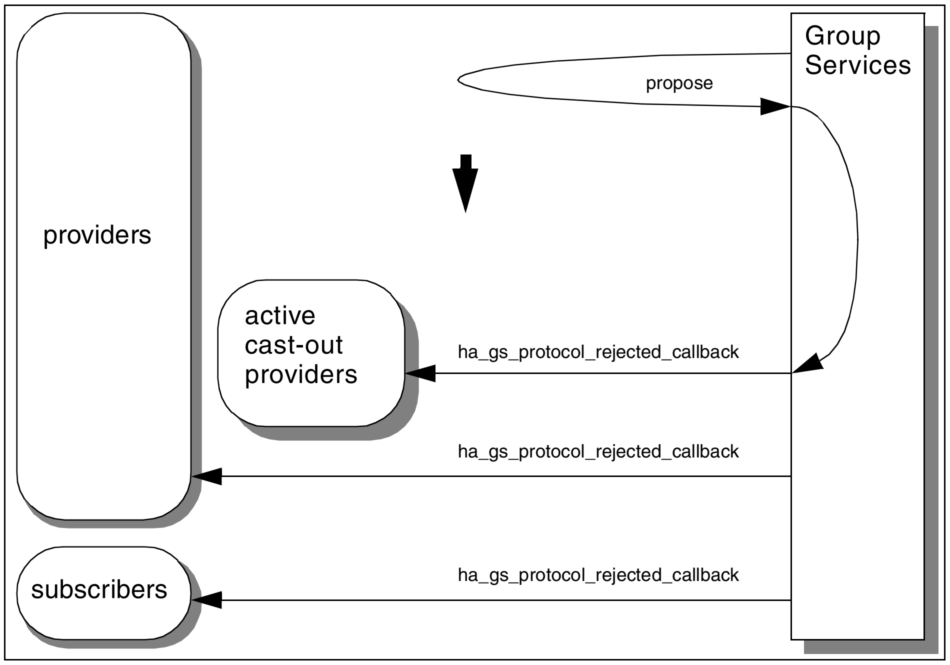

Group Services
propose
providers
active cast-out providers
ha_gs_protocol_rejected_callback
ha_gs_protocol_rejected_callback
subscribers
ha_gs_protocol_rejected_callback

Figure 70. The ending of rejected n-phase cast-out protocol

5.10.3 Programming hints

The following section contains programming hints.

5.10.3.1 Batching cast-out protocols

Batching cast-out protocols is controlled with batching failure leave protocols. There is no separate control bits for each batching control.

5.11 Source-state reflection protocol

The Group Services subsystem proposes a source-state reflection protocol to a target-group when a source-group has changed its group state value through a non-membership change protocol.

5.11.1 Protocol proposal

The Group Services subsystem uses the following fields of the group attributes block to propose and execute a protocol:

- The gs_source_reflection_num_phases field is used for the number of phases.

- The gs_source_reflection_time_limit field is used for the voting time limit.

5.11.2 Protocol flow

A protocol uses the number of phases and the voting time limit specified in the group attributes block already registered.

5.11.2.1 One-phase protocol

If a source-state reflection protocol is one-phase, the protocol is approved automatically.

The providers in the target-group receive a protocol approved notification (ha_gs_protocol_approved_callback) with the updated source-group's group state value. The subscribers do not receive any notification.

The protocol flow of the one-phase source-state reflection protocol is illustrated in Figure 71.

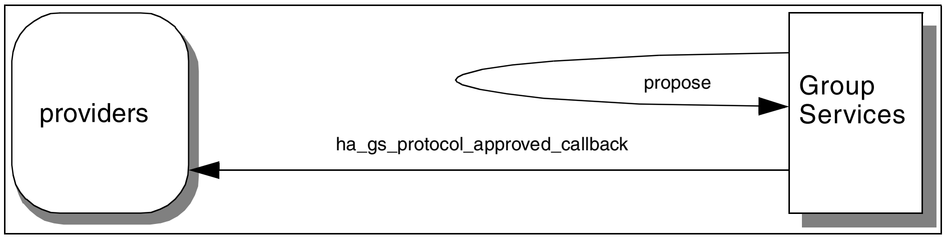


Figure 71. The one-phase source-state reflection protocol

5.11.2.2 N-phase protocol

If a source-state reflection protocol is n-phase, the protocol has voting phases. All the providers in the target-group participate in voting phases.

If the protocol is approved, all the providers in the target-group receive a protocol approved notification (ha_gs_protocol_approved_callback). The subscribers do not receive any notification. If a new group state value is proposed during the voting phases of the protocol for the target-group, the notification contains the updated group state value. The subscribers also receive the notification, if they subscribed for it.

The protocol flow of the approved n-phase source-state reflection protocol is illustrated in Figure 72 on page 105. The flow assumes that the group state value for the target-group is updated. If not, the subscribers do not receive any notification.

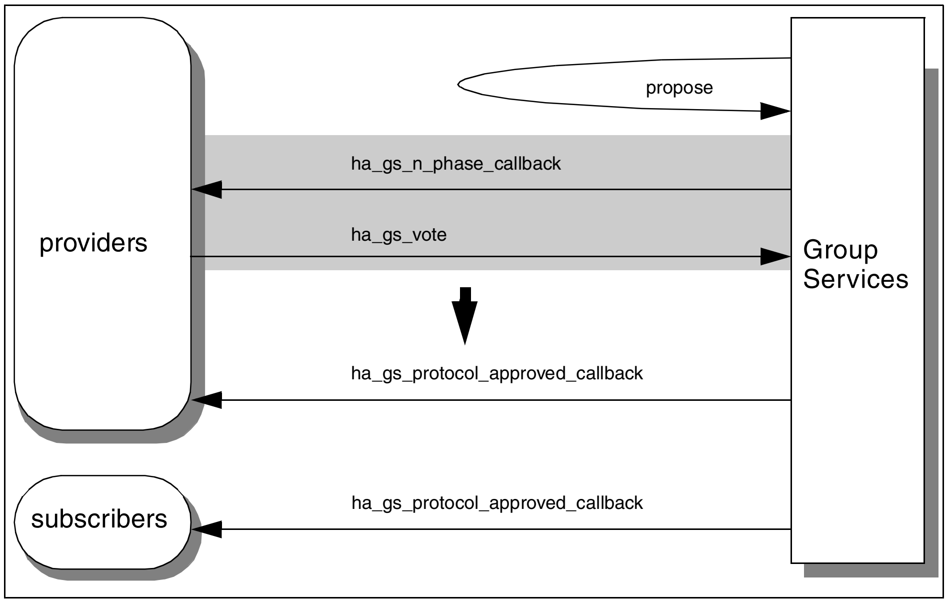


Figure 72. The approved n-phase source-state reflection protocol

If the protocol is rejected, the group state value for the target-group remains unchanged. All the providers receive a protocol rejected notification (ha_gs_protocol_rejected_callback). The subscribers do not receive any notification.

The protocol flow of the rejected n-phase source-state reflection protocol is illustrated in Figure 73 on page 106.

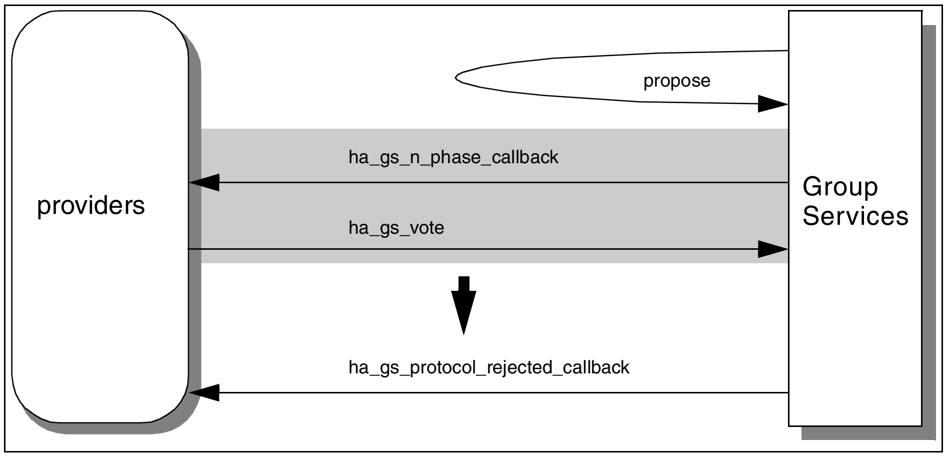


Figure 73. The rejected n-phase source-state reflection protocol

5.11.3 Programming hints

The following section contain programming hints.

5.11.3.1 Only with the first notification

The source-group's group state value is presented only with the first notification that is given to the target-group(s). It is the responsibility of the target-group providers to remember it if it is necessary for their correct operation. The first notification would be either an n-phase, protocol approved, or protocol rejected notification.

5.11.3.2 If the target-group is running another protocol

If the target-group is running another protocol when a source-group state value change is ready to be reflected, the running protocol continues normally, and the source-state reflection protocol is queued to be proposed later when the running protocol completes.

5.12 Voting on proposed protocol

If a provider receives an n-phase notification, the provider is expected to vote on a proposed protocol to the group within the voting phase time limit. The provider can vote to approve, continue, or reject the protocol proposal.

Optionally, a provider can submit the following proposals with its voting:

- A proposal to change a group state value.

- A proposal to send a provider-broadcast message.
- A proposal to change the default vote value to be used by the group until the end of a currently executing protocol. It does not change the default vote value in the group attributes block.

For more information, refer to Section 3.1.7, “Submitting changes with voting” on page 30.

5.12.1 Subroutine call

To vote on a proposed protocol, a provider calls the ha_gs_vote subroutine. The syntax of the ha_gs_vote subroutine is shown in Figure 74:

```
ha_gs_rc_t ha_gs_vote (
   ha_gs_token_t                    provider_token,
   ha_gs_vote_value_t               vote_value,
   const ha_gs_state_value_t        *proposed_state_value,
   const ha_gs_provider_message_t   *provider_message,
   ha_gs_vote_value_t               default_vote_value)
```

Figure 74. The syntax of ha_gs_vote subroutine

Each parameter requires the following information:

provider_token
This parameter requires a token that identifies the caller as a provider of the group. This token was previously initialized when the provider joined the group using the ha_gs_join subroutine.

vote_value
This parameter requires the value of the vote. It can take one of the following values:

-HA_GS_VOTE_APPROVE

-HA_GS_VOTE_CONTINUE

-HA_GS_VOTE_REJECT

They are defined by the ha_gs_vote_value_t type shown in Figure 11 on page 21.

proposed_state_value
This parameter requires an optional updated state value for the group. If the provider does not wish to propose an updated state value, specify a null pointer. For more information, refer to Section 5.12.2, “Proposing a group state value” on page 108.

provider_message

This parameter requires an optional provider-broadcast message to be sent to the providers as part of the next notification for this protocol. If the provider does not wish to send a message, specify a null pointer. For more information, refer to Section 5.12.3, “Sending a provider-broadcast message” on page 110.

default_vote_value

This parameter requires the default vote value to be used by the group until the end of the currently executing protocol. It does not change the default vote value in the group attributes block. It can take one of the following values:

-HA_GS_NULL_VOTE

-HA_GS_VOTE_APPROVE

-HA_GS_VOTE_REJECT

If the provider does not wish to change the default vote value, specify HA_GS_NULL_VOTE. They are defined by the ha_gs_vote_value_t type shown in Figure 11 on page 21.

5.12.2 Proposing a group state value

A group state value can be proposed during the voting phases by the ha_gs_vote subroutine. The group state value is presented in the proposal block provided by the next notification through the last notification for the protocol. Notifications could be an n-phase, protocol approved, and/or protocol rejected notification. If another group state value is proposed, it replaces the previously proposed group state value.

If the proposed protocol is approved, the providers receive the protocol approved notification with the updated group state value. The subscribers also receive the notification if they subscribed for it.

This flow is illustrated in Figure 75 on page 109.

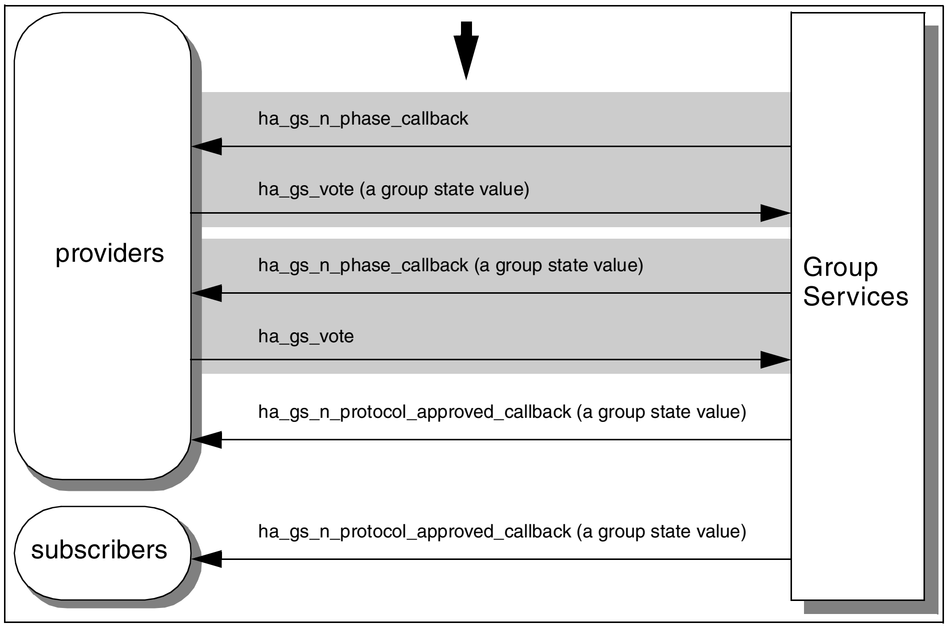

ha_gs_n_phase_callback
ha_gs_vote (a group state value)
providers
ha_gs_n_phase_callback (a group state value)
Group Services
ha_gs_vote
ha_gs_n_protocol_approved_callback (a group state value)
subscribers
ha_gs_n_protocol_approved_callback (a group state value)

Figure 75. Proposing a group state value during voting phases and approved

If the proposed protocol is rejected, the group state value remains unchanged, and the providers receive the protocol rejected notification. The subscribers do not receive any notification.

This flow is illustrated in Figure 76 on page 110.

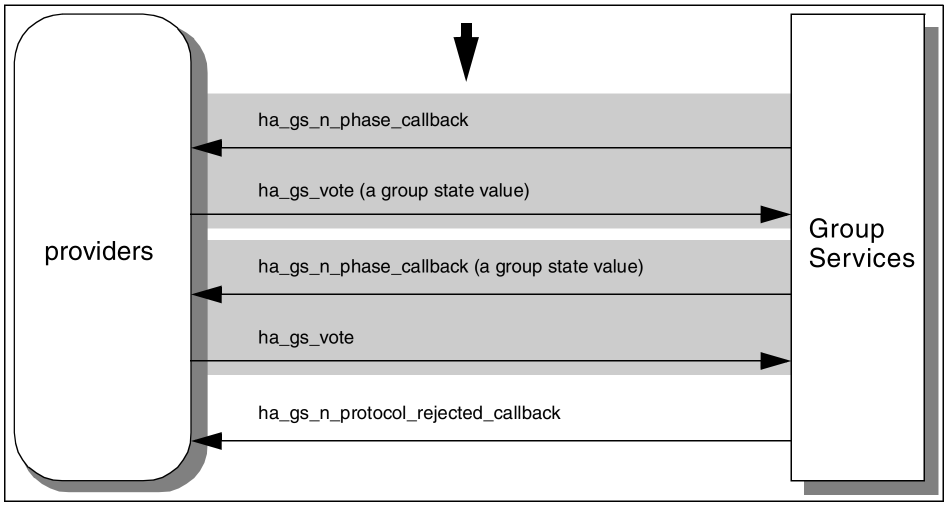

ha_gs_n_phase_callback
ha_gs_vote (a group state value)
providers
Group Services
ha_gs_n_phase_callback (a group state value)
ha_gs_vote
ha_gs_n_protocol_rejected_callback

Figure 76. Proposing a group state value during voting phases and rejected

5.12.3 Sending a provider-broadcast message

A provider-broadcast message can be sent during the voting phases by the ha_gs_vote subroutine. The provider-broadcast message is presented only once in the proposal block provided by the next notification. The notification could be an n-phase, protocol approved, or protocol rejected notification.

If the provider-broadcast message is sent in the middle of voting phases, it will be delivered by the next n-phase notification (ha_gs_n_phase_callback). This flow is illustrated in Figure 77 on page 111.

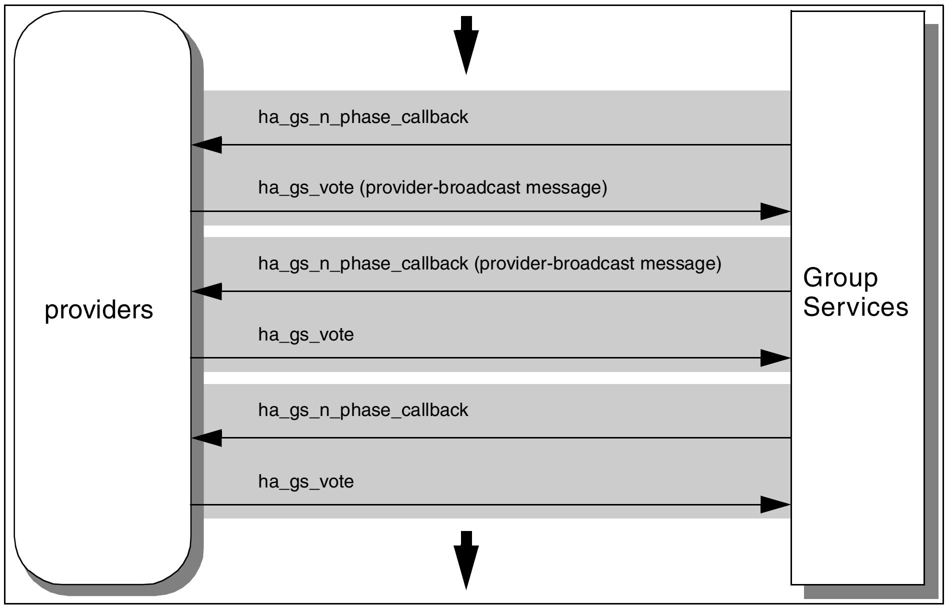

providers
ha_gs_n_phase_callback
ha_gs_vote (provider-broadcast message)
ha_gs_n_phase_callback (provider-broadcast message)
ha_gs_vote
ha_gs_n_phase_callback
ha_gs_vote
Group Services

Figure 77. Sending a message in the middle of voting phases

If the proposed protocol is approved and a provider-broadcast message is proposed in the last voting phase, providers receive a protocol approved notification (ha_gs_approved_notification_callback) with the provider-broadcast message. The subscribers do not receive any notification.

If a new group state value was proposed during the voting phases, the notification contains the updated group state value as well. The subscribers also receive the notification if they subscribed for it.

This flows are illustrated in Figure 78 on page 112.

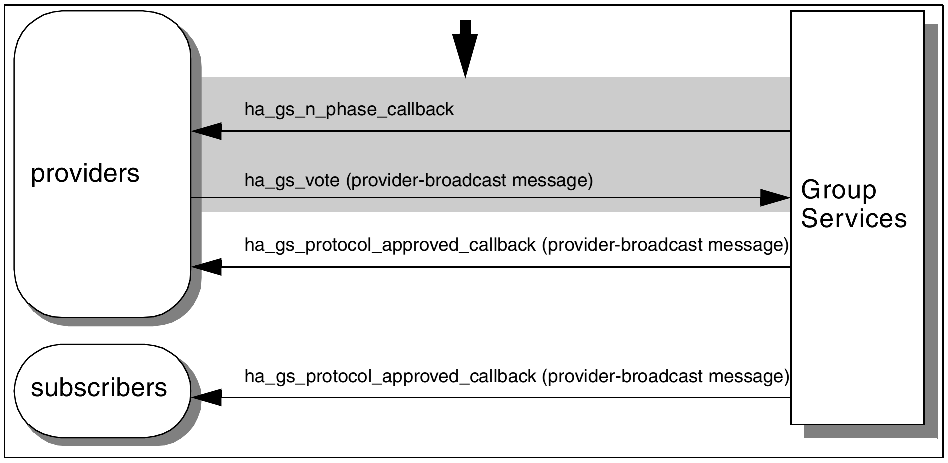

providers
subscribers
Group Services
ha_gs_n_phase_callback
ha_gs_vote (provider-broadcast message)
ha_gs_protocol_approved_callback (provider-broadcast message)
ha_gs_protocol_approved_callback (provider-broadcast message)

Figure 78. Sending an approved message in the last voting phase

If the proposed protocol is rejected and a provider-broadcast message is proposed in the last voting phase, it will be delivered by a protocol rejected notification (ha_gs_rejected_notification_callback). A group state value remains unchanged. The subscribers do not receive any notification.

These flows are illustrated in Figure 79.

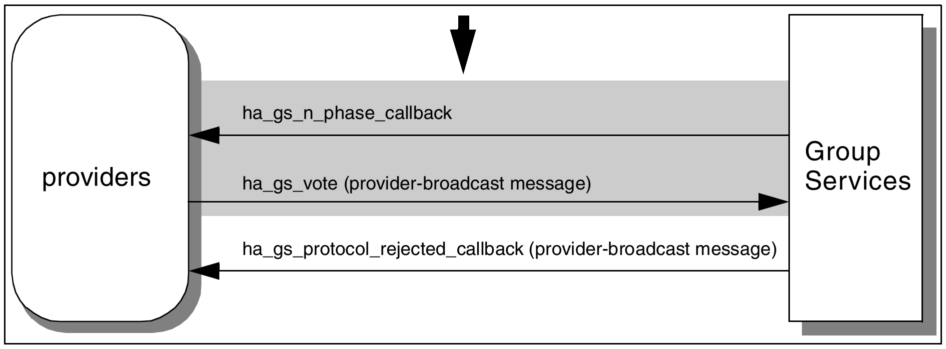

providers
Group Services
ha_gs_n_phase_callback
ha_gs_vote (provider-broadcast message)
ha_gs_protocol_rejected_callback (provider-broadcast message)

Figure 79. Sending a rejected message in the last voting phase

5.12.4 Programming hints

If multiple providers in the same voting phase propose group state values, the Group Services chooses only one of them. Therefore, if different providers propose different group state values, it is unpredictable which group state value the Group Services will choose.

If multiple providers in the same voting phase send provider-broadcast messages, Group Services chooses only one of them. Therefore, if different providers send different provider-broadcast messages, it is unpredictable which messages the Group Services will choose.

Chapter 6. Subscribing to a group

If a GS client only needs information about a membership list and a group state value change on a group, it is not necessary to become a provider of the group. Group Services provides a subscriber with information about these changes.

This chapter describes how a GS client subscribes and unsubscribes to and from a group.

6.1 Subscribe to a group

When it is appropriate for a process to monitor a group without taking part in the control of the group's activities, the Group Services allows the process to subscribe to the group.

6.1.1 Subroutine call

If a GS client has initialized itself with the Group Services, it can call the ha_gs_subscribe subroutine to subscribe a group. The subroutine requires a pointer to a token and a pointer to a proposal information block. A token will be returned by the Group Services that is used to identify a GS client in the group as a subscriber. A proposal information block describes the information on subscription.

The prototype for the ha_gs_subscribe subroutine is shown in Figure 80.

```
ha_gs_rc_t ha_gs_subscribe(
    ha_gs_token_t *,
    const ha_gs_proposal_info_t *);
```

Figure 80. The ha_gs_subscribe subroutine prototype

The token is also referred as *the subscriber token* and defined as ha_gs_token_t type as shown in Figure 7 on page 14. The definition is the same as the provider token.

The proposal information block is redefined as *the subscribe request block* as shown in Figure 28 on page 54.

Note

The following definition is available for convenience of programming:

```
#define  gs_subscribe_request _gs_protocol_info._gs_subscribe_request
```

The subscribe request block is defined as shown in Figure 81.

```
typedef struct {
   ha_gs_subscription_ctrl_t    gs_subscription_control;
   ha_gs_group_name_t           gs_subscription_group;
   ha_gs_subscription_cb_t      *gs_subscription_callback;
} ha_gs_subscribe_request_t;
```

Figure 81. The subscribe request block

Each field requires the following information:

gs_subscription_control

This field requires one or more flags that indicate the types of information that the subscriber wishes to receive about changes to the subscribed-to group. A GS client may subscribe to changes in the group state value, the membership list, or both.
The flags are not exclusive and may be specified in any combination by ORing the individual flags together. They are defined by the ha_gs_subscription_ctrl_t type shown in Figure 82.

```
typedef enum {
   HA_GS_SUBSCRIBE_STATE                    = 0x01,
   HA_GS_SUBSCRIBE_DELTA_JOINS              = 0x02,
   HA_GS_SUBSCRIBE_DELTA_LEAVES             = 0x04,
   HA_GS_SUBSCRIBE_DELTAS_ONLY              = 0x06,
   HA_GS_SUBSCRIBE_MEMBERSHIP               = 0x08,
   HA_GS_SUBSCRIBE_ALL_MEMBERSHIP           = 0x0e,
   HA_GS_SUBSCRIBE_STATE_AND_MEMBERSHIP     = 0x0f
} ha_gs_subscription_ctrl_t;
```

Figure 82. The ha_gs_subscription_ctrl_t type

Each value indicates the following information:

HA_GS_SUBSCRIBE_STATE

This value indicates that the subscriber wants to receive the group's state value whenever the state value is updated.

HA_GS_SUBSCRIBE_DELTA_JOINS

This value indicates that the subscriber wants to receive the set of providers that are joining the group, whenever a join occurs.

HA_GS_SUBSCRIBE_DELTA_LEAVES
This value indicates that the subscriber wants to receive the set of providers that are leaving the group, whenever a voluntary leave or an involuntary leave (failure leave) occurs.

HA_GS_SUBSCRIBE_DELTAS_ONLY
This value indicates that, whenever a join or leave occurs, the subscriber wants to receive both the set of providers that are joining the group and the set of providers that are leaving the group.

HA_GS_SUBSCRIBE_MEMBERSHIP
This value indicates that the subscriber wants to receive a full list of providers in the group whenever a membership change (a join or leave) occurs. If this flag is specified along with either of the delta flags, the delta list and the full membership list are given as two separate lists during membership changes. The delta flags free the subscriber from having to determine the changing members by comparing full membership lists after getting notifications.
If HA_GS_SUBSCRIBE_MEMBERSHIP is not specified but at least one of the delta flags is specified, the subscriber still receives the full list of providers in the group on the first subscription notification that contains membership data for the group. Subsequent notifications contain only the delta list of joining or leaving providers.

HA_GS_SUBSCRIBE_ALL_MEMBERSHIP
This value indicates that, on all subscription notifications that contain membership information, the subscriber wants to receive both the full set of providers in the group and the delta list of joining or leaving providers.

HA_GS_SUBSCRIBE_STATE_AND_MEMBERSHIP
This value indicates that the subscriber wants to receive all of the information described by the other flags.

gs_subscription_group
This field requires a pointer to the name of the group to which the caller wishes to subscribe. The name must be equal to the gs_group_name field in the group attributes block shown in Figure 1 on page 8.

gs_subscription_callback
This field requires a pointer to the callback subroutine that is to be called by a subscription notification. For a subscription notification, refer to Section 7.9, “Subscription notification” on page 147.

6.1.2 Programming hints

If the ha_gs_subscribe subroutine call is successful, it returns HA_GS_OK, and the subscriber_token field is set to the token that identifies this subscriber's connection to the group. However, it does not guarantee that the subscription itself is successful. If subscription is successful, the subscriber will receive a subscription notification that contains the current information. If not, it will receive a delayed error notification.

For a subscription notification, refer to Section 7.9, "Subscription notification" on page 147. For a delayed error notification, refer to Section 7.4, "Delayed error notification" on page 141.

6.2 Unsubscribe from a group

When it is no longer necessary to monitor a group, a subscriber can unsubscribe the group.

6.2.1 Subroutine call

If a GS client is a subscriber, it can call the ha_gs_unsubscribe subroutine to unsubscribe from a group. The subroutine only requires a subscriber token that was returned when it called the ha_gs_subscribe subroutine to subscribe to the group.

The prototype for the ha_gs_unsubscribe subroutine is shown in Figure 83.

```
ha_gs_rc_t ha_gs_unsubscribe(ha_gs_token_t);
```

Figure 83. The prototype for the ha_gs_subscribe subroutine

The subscriber token is defined as the ha_gs_token_t type shown in Figure 7 on page 14. The definition is the same as the provider token.

6.2.2 Programming hints

If a group that a subscriber has subscribed is dissolved, the subscriber receives a final subscription notification with a value of HS_GS_SUBSCRIPTION_DISSOLVED. At this point, the group does not exist; therefore, the subscriber (actually, it is not a subscriber anymore) should not call the ha_gs_unsubscribe subroutine to that group.

Chapter 7. Getting notifications

During activities in a group, providers and subscribers receive notifications. Upon receiving notifications, providers and subscribes are required to take appropriate actions.

This chapter describes how a notification is created and how it is handled in a group.

7.1 Overview

When Group Services has information that should be delivered to its clients, it uses notifications. A notification is sent to GS clients, and it executes a callback subroutine that is provided by GS clients. The callback subroutine checks the information that is sent by the Group Services and takes appropriate actions, if necessary.

There are seven notifications used by Group Services:

- Responsiveness notification
- Delayed error notification
- N-phase notification
- Protocol approved notification
- Protocol rejected notification
- Announcement notification
- Subscription notification

7.1.1 Notifications and callback subroutines

Each notification needs its corresponding callback subroutine. A GS client is required to provide them. The required callback subroutines depend on the type of GS client: Provider or subscriber. A provider needs to provide six callback subroutines. A subscriber needs to provide three callback subroutines. Table 3 summarizes notification, its callback subroutine, and GS clients that need to provide it.

Table 3. Notification and GS clients

Notification	Callback subroutine name	Provider	Subscriber
responsiveness	ha_gs_responsiveness_callback	yes	yes
delayed error	ha_gs_delayed_error_callback	yes	yes

Notification	Callback subroutine name	Provider	Subscriber
n-phase	ha_gs_n_phase_callback	yes	no
protocol approved	ha_gs_protocol_approved_callback	yes	no
protocol rejected	ha_gs_protocol_rejected_callback	yes	no
announcement	ha_gs_announcement_callback	yes	no
subscription	hs_gs_subscriber_callback	no	yes

Note

A GS client can use any name for callback subroutines. Group Services requires the address of callback subroutines, not their name. However, this book uses the names shown in Table 3 for readability.

These callback subroutines must be registered when a GS client registers itself to the Group Services. The subroutines that register callback subroutines are ha_gs_init, ha_gs_join, or ha_gs_subscribe. Table 4 summarizes callback subroutines and the subroutines that register them.

Table 4. Callback subroutine and the subroutine that registers it

Callback subroutine	Subroutine registers callback subroutine
ha_gs_responsiveness_callback	ha_gs_init
ha_gs_delayed_error_callback	ha_gs_init
ha_gs_n_phase_callback	ha_gs_join
ha_gs_protocol_approved_callback	ha_gs_join
ha_gs_protocol_rejected_callback	ha_gs_join
ha_gs_announcement_callback	ha_gs_join
hs_gs_subscriber_callback	ha_gs_subscribe

Figure 84 on page 121 illustrates the relationship between notifications and callback subroutines.

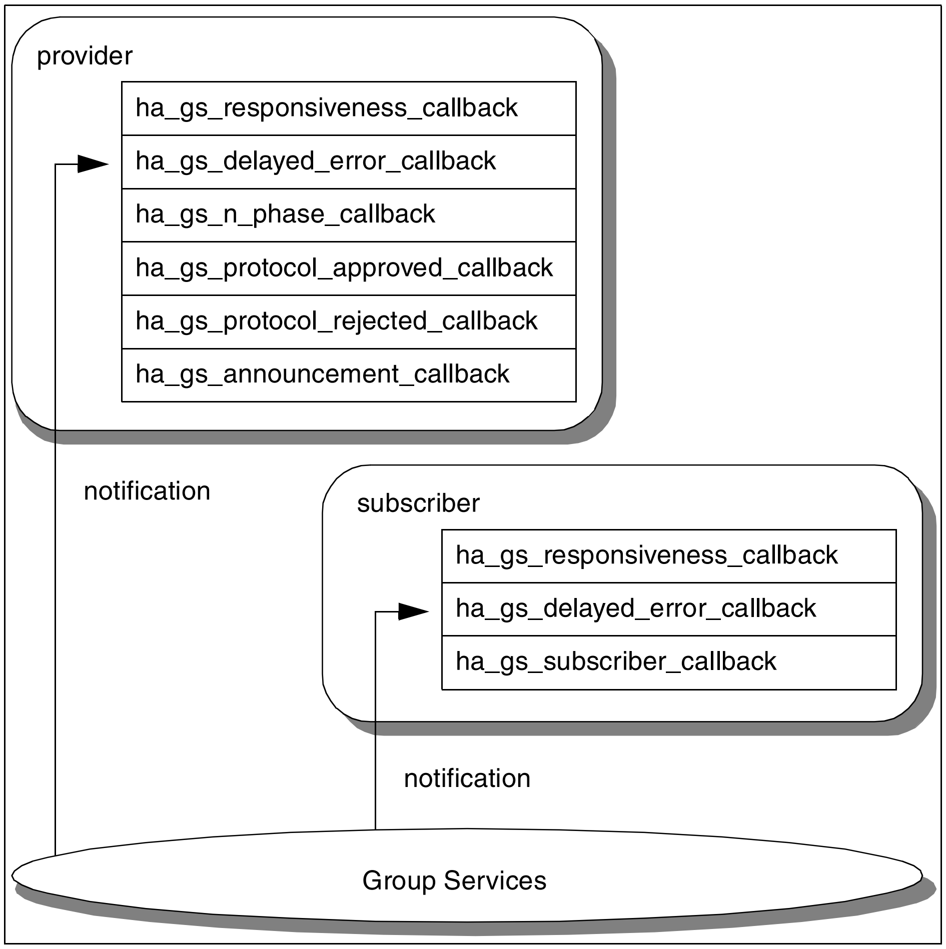

provider
ha_gs_responsiveness_callback
ha_gs_delayed_error_callback
ha_gs_n_phase_callback
ha_gs_protocol_approved_callback
ha_gs_protocol_rejected_callback
ha_gs_announcement_callback
notification
subscriber
ha_gs_responsiveness_callback
ha_gs_delayed_error_callback
ha_gs_subscriber_callback
notification
Group Services

Figure 84. Notification and callback subroutine

7.1.2 Executing callback subroutines

The previous section explained that with using a notification the Group Services executes a callback subroutine that is provided by a GS client. From the programming point of view, a GS client and the Group Services are different processes. Therefore, it is not possible for the Group Services process to call the subroutine that resides in the GS client's process directly.

To solve this problem, Group Services provides the ha_gs_dispatch subroutine. This subroutine resides in the Group Services shared library just like other subroutines.

When the ha_gs_init, ha_gs_join, or ha_gs_subscribe subroutine is called, it registers the addresses of appropriate callback subroutines to a table that resides in the data segment. The ha_gs_init subroutine also creates a socket to communicate with the Group Services daemon.

The GS client program must call the ha_gs_dispatch subroutine regularly. The ha_gs_dispatch subroutine checks the socket to see whether the Group Services daemon has written notifications to it. If it has, the subroutine reads the notifications and executes callback subroutines according to the notifications. The subroutine uses the table that contains the address of callback subroutines.

The GS client process itself must not read or write directly on the socket.

Figure 85 on page 123 illustrates the mechanism of the ha_gs_dispatch subroutine and the callback subroutine.

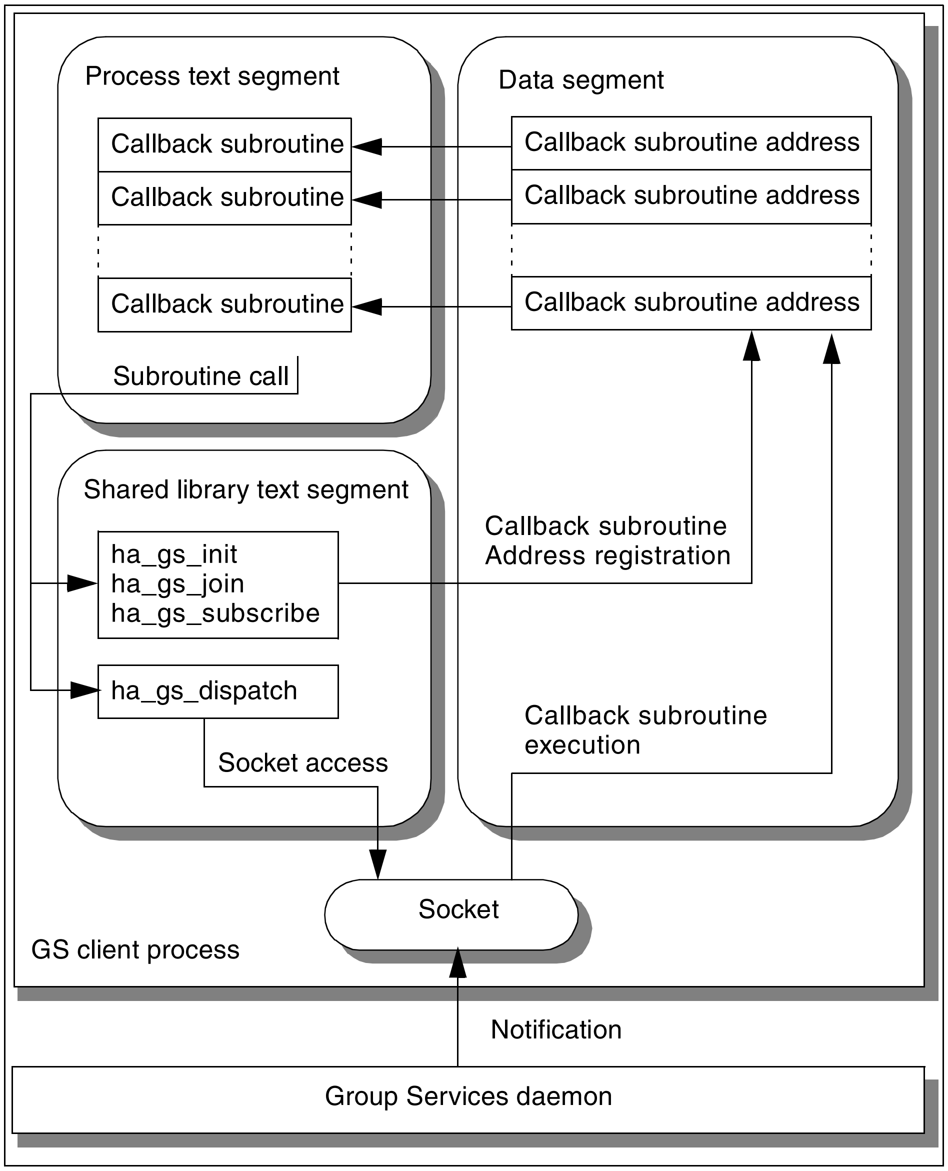


Figure 85. ha_gs_dispatch subroutine and callback subroutines

7.1.3 Programming hints

When a GS client registers its callback subroutines, Group Services requires their addresses.

Group Services provides a number of separate callback routines, each of which expects to receive a different type of notification. However, each notification block also specifies its type. This design allows you to code callback routines using either of the following two strategies or a combination of the two:

- Code a number of specialized callback routines.

  This reduces the amount of checking each callback routine must perform when it receives a notification. You might want to use this approach if performance and path length are considerations when your application handles a notification.

- Code a general callback routine that parses the notifications it receives.

  This reduces the number of callback routines you need to code, but it increases the amount of work each routine must do to determine the type of notification it has received.

7.2 Common design

This section provides information on notification designs that are commonly used for all the notifications.

7.2.1 Callback subroutine prototypes

All callback subroutines use only one parameter. Figure 86 on page 125 shows prototypes of callback subroutines. Again, the program can use any name for callback subroutines; however, this book uses the names shown in Table 3 on page 119 for readability.

```
ha_gs_callback_rc_t ha_gs_responsiveness_callback(
   const ha_gs_responsiveness_notification_t *);

void ha_gs_delayed_error_callback(
   const ha_gs_delayed_error_notification_t *);

void ha_gs_n_phase_callback(
   const ha_gs_n_phase_notification_t *);

void ha_gs_protocol_approved_callback(
   const ha_gs_approved_notification_t *);

void ha_gs_protocol_rejected_callback(
   const ha_gs_rejected_notification_t *);

void ha_gs_announcement_callback(
   const ha_gs_announcement_notification_t *);

void ha_gs_subscriber_callback(
   const ha_gs_subscription_notification_t *);
```

Figure 86. The callback subroutine prototypes

7.2.2 Notification blocks and their fields

The parameter of a callback subroutine points to a notification block. The notification blocks are defined differently from each callback subroutine:

- A responsiveness notification block is defined by the ha_gs_responsiveness_notification_t type.
- A delayed error notification block is defined by the ha_gs_delayed_error_notification_t type.
- An n-phase notification block is defined by the ha_gs_n_phase_notification_t type.
- A protocol approved notification block is defined by the ha_gs_approved_notification_t type.
- A protocol rejected notification block is defined by the ha_gs_rejected_notification_t type.
- An announcement notification block is defined by the ha_gs_announcement_notification_t type.
- A subscriber notification block is defined by the ha_gs_subscriber_notification_t type.

Each notification block contains two to seven fields as summarized in Table 5. An asterisks (*) indicates that a field is a pointer.

Table 5. Notification blocks and their fields

Field in notification block	**Responsiveness notification block**	**Delayed error notification block**	**n-phase notification block**	**Protocol approved notification block**	**Protocol rejected notification block**	**Announcement notification block**	**Subscriber notification block**
gs_notification_type	O	O	O	O	O	O	O
gs_provider_token			O	O	O	O	
gs_subscriber_token							O
gs_protocol_type		O	O	O	O		
gs_summary_code			O	O	O	O	
*gs_proposal			O	O	O		
gs_responsiveness_information	O						
gs_request_token		O					
gs_delayed_return_code		O					
*gs_failing_request		O					
gs_time_limit			O				
*gs_announcement						O	
gs_subscription_type							O
*gs_state_value							O
*gs_full_membership							O
*gs_changing_membership							O
*gs_subscription_special_data							O

The following sections describe the commonly-used fields in a notification block.

7.2.2.1 gs_notification_type field

The gs_notification_type field contains the type of notification. It is defined by the ha_gs_notification_type_t type shown in Figure 87.

```
typedef enum {
    HA_GS_RESPONSIVENESS_NOTIFICATION,
    HA_GS_QUERY_NOTIFICATION,
    HA_GS_DELAYED_ERROR_NOTIFICATION,
    HA_GS_N_PHASE_NOTIFICATION,
    HA_GS_APPROVED_NOTIFICATION,
    HA_GS_REJECTED_NOTIFICATION,
    HA_GS_ANNOUNCEMENT_NOTIFICATION,
    HA_GS_SUBSCRIPTION_NOTIFICATION,
    HA_GS_MERGE_NOTIFICATION
} ha_gs_notification_type_t;
```

Figure 87. The ha_gs_notification_type_t type

Each value indicates the following information:

HA_GS_RESPONSIVENESS_NOTIFICATION
This value indicates that a notification block is for a responsiveness notification.

HA_GS_QUERY_NOTIFICATION
This value is reserved for IBM use.

HA_GS_DELAYED_ERROR_NOTIFICATION
This value indicates that a notification block is for a delayed error notification.

HA_GS_N_PHASE_NOTIFICATION
This value indicates that a notification block is for an n-phase notification.

HA_GS_APPROVED_NOTIFICATION
This value indicates that a notification block is for a protocol approved notification.

HA_GS_REJECTED_NOTIFICATION
This value indicates that a notification block is for a protocol rejected notification.

HA_GS_ANNOUNCEMENT_NOTIFICATION
This value indicates that a notification block is for an announcement notification.

HA_GS_SUBSCRIPTION_NOTIFICATION
This value indicates that a notification block is for a subscriber notification.

HA_GS_MERGE_NOTIFICATION
This value is reserved for IBM use.

7.2.2.2 gs_provider_token field

The gs_provider_token field contains a token that identifies the caller as a provider of the group. The token indicates to which provider the notification is delivered. An application needs to use this field in case it is a provider of multiple groups. The token is defined by the ha_gs_token_t type as shown in Figure 7 on page 14.

7.2.2.3 gs_subscriber_token field

The gs_subscribe_token field contains a token that identifies the caller as a subscriber of the group. The token indicates to which subscriber the notification is delivered. An application needs to use this field in case it is a subscriber of multiple groups. The token is defined by the ha_gs_token_t type as shown in Figure 7 on page 14.

7.2.2.4 gs_protocol_type field

The gs_protocol_type field contains the protocol type for which the notification is being delivered. It is defined by the ha_gs_request_t type shown in Figure 88.

```
typedef enum {
    HA_GS_RESPONSIVENESS,
    HA_GS_JOIN,
    HA_GS_FAILURE_LEAVE,
    HA_GS_LEAVE,
    HA_GS_EXPEL,
    HA_GS_STATE_VALUE_CHANGE,
    HA_GS_PROVIDER_MESSAGE,
    HA_GS_CAST_OUT,
    HA_GS_SOURCE_STATE_REFLECTION,
    HA_GS_MERGE,
    HA_GS_SUBSCRIPTION,
    HA_GS_GROUP_ATTRIBUTE_CHANGE,
    MAX_REQUEST = HA_GS_GROUP_ATTRIBUTE_CHANGE
} ha_gs_request_t;
```

Figure 88. The ha_gs_request_t type

Each value indicates the following information:

HA_GS_JOIN
This value indicates that a notification is delivered for a join protocol.

HA_GS_FAILURE_LEAVE
This value indicates that a notification is delivered for a failure leave protocol.

HA_GS_LEAVE
This value indicates that a notification is delivered for a voluntary leave protocol.

HA_GS_EXPEL
This value indicates that a notification is delivered for an expel protocol.

HA_GS_STATE_VALUE_CHANGE
This value indicates that a notification is delivered for a state value change protocol.

HA_GS_PROVIDER_MESSAGE
This value indicates that a notification is delivered for a provider-broadcast message protocol.

HA_GS_CAST_OUT
This value indicates that a notification is delivered for a cast-out protocol.

HA_GS_SOURCE_STATE_REFLECTION
This value indicates that a notification is delivered for a source-state reflection protocol.

HA_GS_MERGE
This value is reserved for IBM use.

HA_GS_GROUP_ATTRIBUTE_CHANGE
This value indicates that a notification is delivered for a change-attributes protocol.

7.2.2.5 gs_summary_code field

The gs_summary_code field contains one or more flags that indicate a summary of notification. Information includes: Voting activities, responsiveness checks, and/or deactivate-on-failure activities. The field can contain one or more of the flags defined by the ha_gs_summary_code_t type shown in Figure 89 on page 130.

```
typedef enum {
    HA_GS_MIN_SUMMARY_CODE                      = 0x0001,
    HA_GS_EXPLICIT_APPROVE                      = 0x0001,
    HA_GS_EXPLICIT_REJECT                       = 0x0002,
    HA_GS_DEFAULT_APPROVE                       = 0x0004,
    HA_GS_DEFAULT_REJECT                        = 0x0008,
    HA_GS_TIME_LIMIT_EXCEEDED                   = 0x0010,
    HA_GS_PROVIDER_FAILED                       = 0x0020,
    HA_GS_RESPONSIVENESS_NO_RESPONSE            = 0x0040,
    HA_GS_RESPONSIVENESS_RESPONSE               = 0x0080,
    HA_GS_GROUP_DISSOLVED                       = 0x0100,
    HA_GS_GROUP_SERVICES_HAS_DIED_HORRIBLY      = 0x0200,
    HA_GS_DEACTIVATE_UNSUCCESSFUL               = 0x0400,
    HA_GS_DEACTIVATE_TIME_LIMIT_EXCEEDED        = 0x0800,
    HA_GS_GROUP_ATTRIBUTES_CHANGED              = 0x1000,
    HA_GS_MAX_SUMMARY_CODE                      = 0x1000
} ha_gs_summary_code_t;
```

Figure 89. The ha_gs_summary_code_t type

Each value indicates the following information:

HA_GS_EXPLICIT_APPROVE

This flag is set for a protocol approved notification if all approval votes in the tally were explicitly submitted by the providers. No other flags are set with this flag.

HA_GS_EXPLICIT_REJECT

This flag is set for a protocol rejected notification if one or more rejection votes in the tally were explicitly submitted by the providers.

HA_GS_DEFAULT_APPROVE

This flag is set if one or more approval votes in the tally were recorded by default. If this flag is set, the HA_GS_TIME_LIMIT_EXCEEDED flag, the HA_GS_PROVIDER_FAILED flag, or both are also set.

HA_GS_DEFAULT_REJECT

This flag is set if one or more rejection votes in the tally were recorded by default. If this flag is set, the HA_GS_TIME_LIMIT_EXCEEDED flag, the HA_GS_PROVIDER_FAILED flag, or both are also set.

HA_GS_TIME_LIMIT_EXCEEDED

This flag is set when a default approval vote or a default rejection vote was recorded because one or more providers failed to vote in time.

HA_GS_PROVIDER_FAILED

This flag is set when a default approval vote or a default rejection vote was recorded because one or more providers failed (because the node or process failed). The reason for the failure will be provided during the subsequent failure leave protocol.

HA_GS_RESPONSIVENESS_NO_RESPONSE

This flag is set for an announcement notification when one or more providers failed a responsiveness check. The gs_announcement field of the announcement notification block points to the list of providers that failed the responsiveness check.

HA_GS_RESPONSIVENESS_RESPONSE

This flag is set for an announcement notification when one or more providers that previously failed responsiveness checks are now responding successfully. The gs_announcement field of the announcement notification block points to the list of providers that are now responding successfully.

HA_GS_GROUP_DISSOLVED

This flag is reserved for IBM use.

HA_GS_GROUP_SERVICES_HAS_DIED_HORRIBLY

This flag is set for an announcement notification when the Group Services daemon has died.

HA_GS_DEACTIVATE_UNSUCCESSFUL

This flag is set when a deactivate script exited with an unsuccessful return value.

HA_GS_DEACTIVATE_TIME_LIMIT_EXCEEDED

This flag is set when a deactivate script did not exit within the specified time limit.

HA_GS_GROUP_ATTRIBUTES_CHANGED

This flag is reserved for IBM use.

Notification and its possible summary codes are summarized in Table 6.

Table 6. Notification and its possible summary codes

Summary code	n-phase notification	Protocol approved notification	Protocol rejected notification	Announcement notification
HA_GS_EXPLICIT_APPROVE		O		
HA_GS_EXPLICIT_REJECT			O	
HA_GS_DEFAULT_APPROVE	O	O		
HA_GS_DEFAULT_REJECT	O		O	
HA_GS_TIME_LIMIT_EXCEEDED	O	O	O	O
HA_GS_PROVIDER_FAILED	O	O	O	
HA_GS_RESPONSIVENESS_NO_RESPONSE				O
HA_GS_RESPONSIVENESS_RESPONSE				O
HA_GS_GROUP_SERVICES_HAS_DIED_HORRIBLY				O
HA_GS_DEACTIVATE_UNSUCCESSFUL		O	O	
HA_GS_DEACTIVATE_TIME_LIMIT_EXCEEDED		O	O	

7.2.2.6 gs_proposal field

The gs_proposal field contains a pointer to the proposal block for the proposal on which the vote is requested. The proposal block is defined by the ha_gs_proposal_t type shown in Figure 90 on page 133.

```
typedef struct {
   ha_gs_phase_info_t          gs_phase_info;
   ha_gs_provider_t            gs_proposed_by;
   ha_gs_updates_t             gs_whats_changed;
   ha_gs_membership_t          *gs_current_providers;
   ha_gs_membership_t          *gs_changing_providers;
   ha_gs_leave_array_t         *gs_leave_info;
   ha_gs_expel_info_t          *gs_expel_info;
   ha_gs_state_value_t         *gs_current_state_value;
   ha_gs_state_value_t         *gs_proposed_state_value;
   ha_gs_state_value_t         *gs_source_state_value;
   ha_gs_provider_message_t    *gs_provider_message;
   ha_gs_group_attributes_t    *gs_new_group_attributes;
} ha_gs_proposal_t;
```

Figure 90. The proposal block

Each field contains the following information:

gs_phase_info

This field contains information about the type of protocol that is executing and the phase number to which this notification applies. The field is defined by the ha_gs_phase_info_t type shown in Figure 91.

```
typedef struct {
   unsigned short          gs_num_phases;
   unsigned short          gs_phase_number;
} ha_gs_phase_info_t;
```

Figure 91. The ha_gs_phase_info_t type

Each field contains the following information:

gs_num_phases

This field contains HA_GS_1_PHASE if the executing protocol is a one-phase or HA_GS_N_PHASE if the executing protocol is an n-phase.

gs_phase_number

This field contains the phase number to which the notification applies.

gs_proposed_by

This field contains the provider information block that identifies the provider (or the Group Services subsystem itself) that proposed the executing protocol. The provider information block is defined as shown in Figure 8 on page 14.

On all join protocols, this field always contains the provider information block for the GS client that is executing the callback subroutine rather than the provider that proposed the join protocol. This allows each provider to capture its own provider information block.

gs_whats_changed

This field contains one or more flags that indicate whether the membership list and/or the group state value is changed and/or if the notification contains a provider-broadcast message. The flags are defined by the ha_gs_updates_t type shown in Figure 92.

```
typedef enum {
    HA_GS_NO_CHANGE                         = 0x0000,
    HA_GS_PROPOSED_MEMBERSHIP               = 0x0001,
    HA_GS_ONGOING_MEMBERSHIP                = 0x0002,
    HA_GS_PROPOSED_STATE_VALUE              = 0x0004,
    HA_GS_ONGOING_STATE_VALUE               = 0x0008,
    HA_GS_UPDATED_PROVIDER_MESSAGE          = 0x0010,
    HA_GS_UPDATED_MEMBERSHIP                = 0x0020,
    HA_GS_REJECTED_MEMBERSHIP               = 0x0040,
    HA_GS_UPDATED_STATE_VALUE               = 0x0080,
    HA_GS_REFLECTED_SOURCE_STATE_VALUE      = 0x0100,
    HA_GS_EXPEL_INFORMATION                 = 0x0200,
    HA_GS_PROPOSED_GROUP_ATTRIBUTES         = 0x0400,
    HA_GS_ONGOING_GROUP_ATTRIBUTES          = 0x0800,
    HA_GS_UPDATED_GROUP_ATTRIBUTES          = 0x1000,
    HA_GS_REJECTED_GROUP_ATTRIBUTES         = 0x2000
} ha_gs_updates_t;
```

Figure 92. The ha_gs_updates_t type

Each value indicates the following information:

HA_GS_NO_CHANGE

No fields have been updated from a previous notification.

HA_GS_PROPOSED_MEMBERSHIP

Membership changes are proposed. The gs_changing_providers field points to a list of joining or leaving providers. For joining providers, the gs_current_providers field points to a list of the current members of the group.

HA_GS_ONGOING_MEMBERSHIP

An ongoing membership change protocol is executing. The gs_changing_providers field points to a list of joining or leaving providers and this field will not change during the protocol.

HA_GS_PROPOSED_STATE_VALUE

A change to the group state value is proposed. The gs_proposed_state_value field points to a proposed new group state value. If providers submit group state value changes with their voting responses, this field may be updated during the protocol. The gs_current_state_value field contains the group's current (last approved) state value.

HA_GS_ONGOING_STATE_VALUE
The gs_proposed_state_value field points to a proposed new group state value, but the value is unchanged from a previous notification. The gs_current_state_value field contains the group's current (last approved) state value.

HA_GS_UPDATED_PROVIDER_MESSAGE
The gs_provider_message field points to a provider-broadcast message. This flag may be set on both n-phase notification and final notifications (protocol approved or protocol rejected). A message is presented only once.

HA_GS_REFLECTED_SOURCE_STATE_VALUE
The source-group updated its group state value during either a membership change protocol or a group state value change protocol. The source-group's state value is presented only with the first notification that is given to the target-group(s). It is the responsibility of the target-group providers to remember it if it is necessary for their correct operation.

HA_GS_PROPOSED_GROUP_ATTRIBUTES
The gs_new_group_attributes field contains the new group attributes that were proposed by a change-attributes protocol.

HA_GS_ONGOING_GROUP_ATTRIBUTES
The gs_new_group_attributes field contains the new group attributes that were proposed by a change-attributes protocol, and these are unchanged from a previous notification.

HA_GS_UPDATED_GROUP_ATTRIBUTES
This flag is set on the protocol approved notification for a change-attributes protocol. The gs_new_group_attributes field contains the new group attributes.

HA_GS_REJECTED_GROUP_ATTRIBUTES
This flag is set on the protocol rejected notification for a change-attributes protocol. The gs_new_group_attributes field contains the rejected group attributes.

gs_current_providers
This field contains a pointer to a membership information block shown in Figure 2 on page 10. The block contains a list of providers that currently belong to the group.

gs_changing_providers

This field contains a pointer to a membership information block shown in Figure 2 on page 10. The block contains a list of providers that are joining or leaving the group through the protocol.

gs_leave_info

This field contains a pointer to an array that contains the reason codes for each provider specified in the gs_changing_providers field that is leaving the group. The field is defined by the ha_gs_leave_array_t type shown in Figure 93.

```
typedef struct {
   unsigned int           gs_count;
   ha_gs_leave_info_t     *gs_leave_codes;
} ha_gs_leave_array_t;
```

Figure 93. The ha_gs_leave_array_t type

Each field contains the following information:

gs_count

This field contains the number of providers that are leaving.

gs_leave_codes

This field contains a pointer to an entry for each provider that is leaving the group that specifies the protocol and the reason for the leave. The field is defined by the ha_gs_leave_info_t type shown in Figure 94 on page 136. The leave reason entries are in the same order in which the providers are listed in the gs_changing_providers field.

```
typedef struct {
   unsigned int           gs_voluntary_or_failure;
   unsigned int           gs_voluntary_leave_code;
} ha_gs_leave_info_t;
```

Figure 94. The ha_gs_leave_info_t type

Each field contains the following information:

gs_voluntary_or_failure

This field contains the protocol that caused providers to leave. It can contain one or more of the flags defined by the ha_gs_leave_reasons_t type shown in Figure 95.

```
typedef enum {
    HA_GS_VOLUNTARY_LEAVE            = 0x0001,
    HA_GS_PROVIDER_FAILURE           = 0x0002,
    HA_GS_HOST_FAILURE               = 0x0004,
    HA_GS_PROVIDER_EXPELLED          = 0x0008,
    HA_GS_SOURCE_PROVIDER_LEAVE      = 0x0010,
    HA_GS_PROVIDER_SAID_GOODBYE      = 0x0020
} ha_gs_leave_reasons_t;
```

Figure 95. The ha_gs_leave_reasons_t type

Each value indicates the following information:

HA_GS_VOLUNTARY_LEAVE

The provider has requested to leave voluntarily. If this flag is set, it is the only flag in the gs_voluntary_or_failure field.
The gs_voluntary_leave_code field contains the application-defined leave code that was specified by a voluntary leave protocol.
If this flag is not set, the gs_voluntary_leave_code field is not used and is undefined.

HA_GS_PROVIDER_FAILURE

The provider is leaving the group because its process has failed. This flag could be set with the HA_GS_HOST_FAILURE flag.

HA_GS_HOST_FAILURE

The provider is leaving the group because its node has failed. This flag could be set with the HA_GS_PROVIDER_FAILURE flag.

HA_GS_PROVIDER_EXPELLED

The provider is leaving the group because of an expel protocol.

HA_GS_SOURCE_PROVIDER_LEAVE

The provider is being cast out of the group because of a cast-out protocol. If a node failure causes both a source-group and a target-group to lose providers, this flag could be set with the HA_GS_HOST_FAILURE flag.

HA_GS_PROVIDER_SAID_GOODBYE

The provider proposed a goodbye protocol and has left the group.

gs_voluntary_leave_code

This field contains the application-defined leave code that was specified on input to the ha_gs_leave subroutine. Refer to the gs_leave_code field in Figure 48 on page 76.

gs_expel_info

This field contains a pointer to an expel information block. The block is defined by the ha_gs_expel_info_t type shown in Figure 96.

```
typedef struct {
    int                       gs_deactivate_phase;
    int                       gs_expel_flag_length;
    char                      *gs_expel_flag;
} ha_gs_expel_info_t;
```

Figure 96. The expel information block

Each field contains the following information:

gs_deactivate_phase
This field contains the phase number in which the deactivate script should be executed against any providers that are being expelled. If this field contains 0, no deactivate script is executed.

gs_expel_flag_length
This field contains the length of the expel flag.

gs_expel_flag
This field contains a flag that is to be passed to the deactivate script. It is a pointer to a null-terminated string with a maximum length of 256 bytes. If the pointer is null, no flag is passed to the deactivate script.

gs_current_state_value
This field contains a pointer to a group state value information block (shown in Figure 3 on page 11) that contains the current group state value. This is the latest approved group state value, which is the same group state value as at the beginning of the protocol.

gs_proposed_state_value
This field contains a pointer to a group state value information block (shown in Figure 3 on page 11) that contains the proposed value for the group state value. The gs_whats_changed field contains a value of either HA_GS_PROPOSED_STATE_VALUE or HA_GS_ONGOING_STATE_VALUE. If there is no new state value for the protocol, this field is null.

gs_source_state_value
This field contains a pointer to a group state value information block (shown in Figure 3 on page 11) that contains the updated group state value of this group's source-group, if the proposal is the result of a change in the source-group. The

gs_whats_changed field contains a value of HA_GS_REFLECTED_SOURCE_VALUE. Otherwise, this field is null.

gs_provider_message
This field contains a pointer to a provider-broadcast message block (shown in Figure 4 on page 11) that contains the provider-broadcast message, if any. The gs_whats_changed field contains a value of HA_GS_UPDATED_PROVIDER_MESSAGE. Otherwise, the field is null.

gs_new_group_attributes
This field contains a pointer to a group attributes block (shown in Figure 1 on page 8) that contains the proposed value for the group attributes. The gs_whats_changed field contains a value of either HA_GS_PROPOSED_GROUP_ATTRIBUTES, HA_GS_ONGOING_GROUP_ATTRIBUTES, HA_GS_UPDATED_GROUP_ATTRIBUTES, or HA_GS_REJECTED_GROUP_ATTRIBUTES. If there are no new group attributes for the protocol, this field is null.

7.3 Responsiveness notification

The responsiveness callback routine is intended to provide Group Services with a means of removing a provider that fails a responsiveness check. The callback subroutine should perform any cleanup actions that are required by the GS client. It also allows the GS client to perform any periodic validity checks on its own operation or its environment that may be needed.

Group Services performs responsiveness checks once the GS client has initialized. If a responsiveness check fails and the GS client is a provider, Group Services places it on a list of nonresponsive providers. Then, Group Services sends an announcement notification containing the list to all of the group's providers. Group Services takes no other direct action. For more information on an announcement notification, refer to Section 7.8, "Announcement notification" on page 146.

Upon receipt of the announcement notification, a provider could propose an expel protocol to remove the nonresponsive providers from the group, if appropriate. For more information on an expel protocol, refer to Section 5.8, "Expel protocol" on page 81. Group Services tries to contact nonresponsive providers. If a previously nonresponsive provider responds, Group Services sends an announcement notification containing the list to all of the group's providers.

Note that because Group Services continues to perform responsiveness checks for nonresponsive providers, the group can determine how quickly it should respond to announcement notifications. A group can expel a nonresponsive provider after receiving the first announcement notification, or it can wait to see if the provider becomes responsive again.

If a GS client is a subscriber or no part of a provider or a subscriber, it is just ignored.

For more information on a responsiveness check facility, refer to Section 3.2, “Responsiveness check facility” on page 31.

7.3.1 Subroutine call

To deliver a responsiveness notification to GS clients, Group Services uses their ha_gs_responsiveness_callback subroutine. At input, Group Services provides a pointer to the responsiveness notification block shown in Figure 97.

```
typedef struct {
    ha_gs_notification_type_t    gs_notification_type;
    ha_gs_responsiveness_t       gs_responsiveness_information;
} ha_gs_responsiveness_notification_t;
```

Figure 97. The responsiveness notification block

Each field contains the following information:

gs_notification_type
This field contains a value of HA_GS_RESPONSIVENESS_NOTIFICATION defined by the ha_gs_notification_type_t type shown in Figure 87 on page 127.

gs_responsiveness_information
This field contains the pointer to the responsiveness control block shown in Figure 20 on page 32. It was specified on input to the ha_gs_init subroutine when this process initialized itself with the Group Services.

7.3.2 Programming hints

The ha_gs_responsiveness_callback subroutine is the only callback subroutine that requires a return code. If the GS client is operational, it should return a value of HA_GS_OK. If the GS client has detected an internal problem that prevents its correct operation, it should return a value of

HA_GS_CALLBACK_NOT_OK. These values are defined by the ha_gs_callback_rc_t type shown in Figure 98.

```
typedef enum {
    HA_GS_CALLBACK_NOT_OK,
    HA_GS_CALLBACK_OK
} ha_gs_callback_rc_t;
```

Figure 98. The ha_gs_callback_rc_t type

7.4 Delayed error notification

An application must prepare for two kind of errors in the Group Services environment. One is called a synchronous error and is returned immediately. This type of error is commonly used for other subroutines. The other type of error is unique to Group Services and is called an asynchronous error. No error was detected when an application called a subroutine; however, Group Services found the reason that it could not execute that subroutine later. In this case, Group Services delivers an asynchronous error code with a notification. This notification is called *a delayed error notification*.

For more information about synchronous/asynchronous errors, refer to Section 8.1, “Synchronous/asynchronous errors” on page 153.

7.4.1 Subroutine call

To deliver a delayed error notification to GS clients, Group Services uses their ha_gs_delayed_error_callback subroutine. On input, Group Services provides a pointer to the delayed error notification block shown in Figure 99.

```
typedef struct {
    ha_gs_notification_type_t    gs_notification_type;
    ha_gs_token_t                gs_request_token;
    ha_gs_request_t              gs_protocol_type;
    ha_gs_rc_t                   gs_delayed_return_code;
    ha_gs_proposal_info_t        *gs_failing_request;
} ha_gs_delayed_error_notification_t;
```

Figure 99. The delayed error notification block

Each field contains the following information:

gs_notification_type

This field contains the

HA_GS_DELAYED_ERROR_NOTIFICATION value described in Section 7.2.2.1, “gs_notification_type field” on page 127.

gs_request_token
This field contains a provider token or subscriber token described in Section 7.2.2.2, “gs_provider_token field” on page 128, or in Section 7.2.2.3, “gs_subscriber_token field” on page 128.

gs_protocol_type
This field contains the protocol type described in Section 7.2.2.4, “gs_protocol_type field” on page 128.

gs_delayed_return_code
This field contains the error number of the delayed error. For error codes, refer to Section 8.2, “Error code” on page 154.

gs_proposal
This field contains the pointer to the proposal information block described in Section 7.2.2.6, “gs_proposal field” on page 132.

7.5 N-phase notification

If a proposed protocol is an n-phase protocol, providers receive an n-phase notification as the first notification. The callback subroutine should check its environment or perform its operation according to the proposed protocol.

The providers are expected to vote to approve, reject, or continue for the proposed protocol within the voting time limit. To do this, they call the ha_gs_vote subroutine. For more information on the ha_gs_vote subroutine, refer to Section 5.12, “Voting on proposed protocol” on page 106.

If a provider fails to vote within the voting time limit, the Group Services applies the group’s default vote value for this provider for the rest of the phases of the ongoing protocol. For more information on voting, refer to Section 3.1.2, “Voting phase” on page 20.

7.5.1 Subroutine call

To deliver an n-phase notification to GS clients, Group Services uses their ha_gs_n_phase_callback subroutine. On input, Group Services provides a pointer to the n-phase notification block shown in Figure 100 on page 143.

```
typedef struct {
    ha_gs_notification_type_t    gs_notification_type;
    ha_gs_token_t                gs_provider_token;
    ha_gs_request_t              gs_protocol_type;
    ha_gs_summary_code_t         gs_summary_code;
    ha_gs_time_limit_t           gs_time_limit;
    ha_gs_proposal_t             *gs_proposal;
} ha_gs_n_phase_notification_t;
```

Figure 100. The n-phase notification block

Each field contains the following information:

gs_notification_type
: This field contains a value of HA_GS_N_PHASE_NOTIFICATION described in Section 7.2.2.1, “gs_notification_type field” on page 127.

gs_provider_token
: This field contains a provider token described in Section 7.2.2.2, “gs_provider_token field” on page 128.

gs_protocol_type
: This field contains the protocol type described in Section 7.2.2.4, “gs_protocol_type field” on page 128.

gs_summary_code
: This field contains summary codes described in Section 7.2.2.5, “gs_summary_code field” on page 129.

gs_time_limit
: This field contains the time limit, in seconds, within which the GS client must submit its vote for this notification.

gs_proposal
: This field contains the pointer to the proposal information block described in Section 7.2.2.6, “gs_proposal field” on page 132.

7.6 Protocol approved notification

When a proposed protocol has been approved, Group Services delivers a protocol approved notification to all the providers. The subscribers also receive this notification if they have a subscriber for it.

For an n-phase protocol, this notification is delivered after the protocol has been approved by voting. A one-phase protocol is automatically approved and receives this notification.

A notification contains the information if one or more approval votes in the tally were recorded by default. It also contains its reason: Votes were not submitted within the voting time limit, or providers failed due to the node or process failure. However, the notification does not contain the list of providers that failed to vote. Therefore, the Group Services also delivers an announcement notification containing this list. For more information on an announcement notification, refer to Section 7.8, “Announcement notification” on page 146.

7.6.1 Subroutine call

To deliver a protocol approved notification to GS clients, Group Services uses their ha_gs_protocol_approved_callback subroutine. On input, Group Services provides a pointer to the protocol approved notification block shown in Figure 101.

```
typedef struct {
   ha_gs_notification_type_t   gs_notification_type;
   ha_gs_token_t               gs_provider_token;
   ha_gs_request_t             gs_protocol_type;
   ha_gs_summary_code_t        gs_summary_code;
   ha_gs_proposal_t            *gs_proposal;
} ha_gs_approved_notification_t;
```

Figure 101. The protocol approved notification block

Each field contains the following information:

gs_notification_type
This field contains a value of HA_GS_APPROVED_NOTIFICATION described in Section 7.2.2.1, “gs_notification_type field” on page 127.

gs_provider_token
This field contains a provider token described in Section 7.2.2.2, “gs_provider_token field” on page 128.

gs_protocol_type
This field contains the protocol type described in Section 7.2.2.4, “gs_protocol_type field” on page 128.

gs_summary_code
This field contains summary codes described in Section 7.2.2.5, “gs_summary_code field” on page 129.

gs_proposal
This field contains the pointer to the proposal information block described in Section 7.2.2.6, “gs_proposal field” on page 132.

7.7 Protocol rejected notification

When a proposed protocol has been rejected, Group Services delivers a protocol rejected notification to all the providers. The subscribers do not receive this notification.

For an n-phase protocol, this notification is delivered after the protocol has been rejected by voting. For a one-phase protocol, this notification is not delivered because it cannot be rejected.

A notification contains the information if one or more rejection votes in the tally were recorded by default. It also contains its reason: Votes were not submitted within the voting time limit, or providers failed due to the node or process failure. However, the notification does not contain the list of providers that failed to vote. Therefore, Group Services also delivers an announcement notification containing this list. For more information on an announcement notification, refer to Section 7.8, “Announcement notification” on page 146.

7.7.1 Subroutine call

To deliver a protocol rejected notification to GS clients, Group Services uses their ha_gs_protocol_rejected_callback subroutine. On input, Group Services provides a pointer to the protocol rejected notification block shown in Figure 102.

```
typedef struct {
    ha_gs_notification_type_t    gs_notification_type;
    ha_gs_token_t                gs_provider_token;
    ha_gs_request_t              gs_protocol_type;
    ha_gs_summary_code_t         gs_summary_code;
    ha_gs_proposal_t             *gs_proposal;
} ha_gs_rejected_notification_t;
```

Figure 102. The protocol rejected notification block

Each field contains the following information:

gs_notification_type
: This field contains a value of HA_GS_REJECTED_NOTIFICATION described in Section 7.2.2.1, “gs_notification_type field” on page 127.

gs_provider_token
: This field contains a provider token described in Section 7.2.2.2, “gs_provider_token field” on page 128.

gs_protocol_type
This field contains the protocol type described in Section 7.2.2.4, “gs_protocol_type field” on page 128.

gs_summary_code
This field contains summary codes described in Section 7.2.2.5, “gs_summary_code field” on page 129.

gs_proposal
This field contains the pointer to the proposal information block described in Section 7.2.2.6, “gs_proposal field” on page 132.

7.8 Announcement notification

When abnormal conditions (other than a complete failure) that affect one or more providers in the group occur, Group Services delivers an announcement notification to providers.

How to deal with announcement notifications is up to the application itself. A possible reaction may be proposing an expel protocol against the faulty provider by one of the other group members. The approval of an expel protocol results in the removal of the provider from the group.

A provider receives announcement notifications for the following reasons:

- One or more providers failed to vote within the voting time limit.
- One or more providers failed a responsiveness check.
- One or more providers that previously failed responsiveness checks are now responding successfully.
- The GS daemon has died.

7.8.1 Subroutine call

To deliver an announcement notification to GS clients, Group Services uses their ha_gs_announcement_callback subroutine. On input, Group Services provides a pointer to the announcement notification block shown in Figure 103 on page 147.

```
typedef struct {
    ha_gs_notification_type_t    gs_notification_type;
    ha_gs_token_t                gs_provider_token;
    ha_gs_summary_code_t         gs_summary_code;
    ha_gs_membership_t           *gs_announcement;
} ha_gs_announcement_notification_t;
```

Figure 103. The announcement notification block

Each field contains the following information:

gs_notification_type
: This field contains a value of HA_GS_ANNOUNCEMENT_NOTIFICATION described in Section 7.2.2.1, “gs_notification_type field” on page 127.

gs_provider_token
: This field contains a provider token that is described in Section 7.2.2.2, “gs_provider_token field” on page 128.

gs_summary_code
: This field contains summary codes described in Section 7.2.2.5, “gs_summary_code field” on page 129.

gs_announcement
: This field contains the pointer to a membership information block (shown in Figure 2 on page 10) of providers that are affected by the condition that is being reported by this announcement.

7.9 Subscription notification

When a proposed protocol changes the membership list and/or the group state value, Group Services delivers a subscription notification to the subscribers. If subscribers want to receive theses notifications, they must subscribe to them. For information about subscribing to a group, refer to Section 6.1, “Subscribe to a group” on page 115.

A subscriber receives subscription notifications for the following reasons:

- The membership list is updated.
- The group state value is updated.
- The group that was subscribed to has dissolved because all providers have left the group.
  The subscription is deactivated. To start receiving notifications again, the subscriber must resubscribe to the group. If the group does not exist

because providers have not rejoined it, each subscription request receives an asynchronous error code of HA_GS_UNKNOWN_GROUP.

- The group that was subscribed to has dissolved because the Group Services daemon has died.
  The subscription is deactivated, and the subscriber's connection to the Group Services daemon is terminated. Before calling any Group Services subroutines, the (former) subscriber must wait until control returns from the ha_gs_dispatch subroutine. Failure to do so may result in an application hang.
  After the ha_gs_dispatch subroutine returns, the former subscriber must re-initialize the connection to Group Services by calling the ha_gs_init subroutine and then taking any other necessary actions to resubscribe to the group.

7.9.1 Subroutine call

To deliver a subscription notification to subscribers, Group Services uses their ha_gs_subscriber_callback subroutine. On input, the Group Services provides a pointer to the subscription notification block shown in Figure 104.

```
typedef struct {
   ha_gs_notification_type_t   gs_notification_type;
   ha_gs_token_t               gs_subscriber_token;
   ha_gs_subscription_type_t   gs_subscription_type;
   ha_gs_state_value_t         *gs_state_value;
   ha_gs_membership_t          *gs_full_membership;
   ha_gs_membership_t          *gs_changing_membership;
   ha_gs_special_data_t        *gs_subscription_special_data;
} ha_gs_subscription_notification_t;
```

Figure 104. The subscription notification block

Each field contains the following information:

gs_notification_type
This field contains a value of HA_GS_SUBSCRIPTION_NOTIFICATION described in Section 7.2.2.1, “gs_notification_type field” on page 127.

gs_subscriber_token
This field contains a subscriber token described in Section 7.2.2.3, “gs_subscriber_token field” on page 128.

gs_subscription_type
This field contains the type of change for which this subscription notification is being delivered. It can contain one or more of the

flags defined by the ha_gs_subscription_type_t type shown in Figure 105.

```
typedef enum {
   HA_GS_SUBSCRIPTION_STATE            = 0x01,
   HA_GS_SUBSCRIPTION_DELTA_JOIN       = 0x02,
   HA_GS_SUBSCRIPTION_DELTA_LEAVE      = 0x04,
   HA_GS_SUBSCRIPTION_MEMBERSHIP       = 0x08,
   HA_GS_SUBSCRIPTION_SPECIAL_DATA     = 0x40,
   HA_GS_SUBSCRIPTION_DISSOLVED        = 0x80,
   HA_GS_SUBSCRIPTION_GS_HAS_DIED      = 0x100
} ha_gs_subscription_type_t;
```

Figure 105. The ha_gs_subscription_type_t type

Each value indicates the following information:

HA_GS_SUBSCRIPTION_STATE
This value indicates that the notification contains the updated group state value. This flag may appear with any of the other flags.

HA_GS_SUBSCRIPTION_DELTA_JOIN
This value indicates that the notification contains the set of joining providers. Joining and leaving providers are not listed together in a single notification. Therefore, no notification will contain both the HA_GS_SUBSCRIPTION_DELTA_JOIN and HA_GS_SUBSCRIPTION_DELTA_LEAVE flags.

HA_GS_SUBSCRIPTION_DELTA_LEAVE
This value indicates that the notification contains the set of leaving providers. Joining and leaving providers are not listed together in a single notification. Therefore, no notification will contain both the HA_GS_SUBSCRIPTION_DELTA_JOIN and HA_GS_SUBSCRIPTION_DELTA_LEAVE flags.

HA_GS_SUBSCRIPTION_MEMBERSHIP
This value indicates that the notification contains the complete updated membership list. This flag may appear with either the HA_GS_SUBSCRIPTION_DELTA_JOIN or HA_GS_SUBSCRIPTION_DELTA_LEAVE flag.

HA_GS_SUBSCRIPTION_DISSOLVED
This value indicates that the group that was subscribed to has dissolved; all providers have left the group. This flag may appear with any of the other flags.

HA_GS_SUBSCRIPTION_GS_HAS_DIED
This value indicates that the group that was subscribed to has

dissolved because the Group Services daemon has died. This flag appears with the HA_GS_SUBSCRIPTION_DISSOLVED flag.

gs_state_value
This field contains a pointer to the group state value information block (shown in Figure 3 on page 11) that contains the new group state value, if the HA_GS_SUBSCRIPTION_STATE flag is set in the gs_subscription_type field..

gs_full_membership
This field contains a pointer to the membership information block (shown in Figure 2 on page 10) that contains providers currently belong to the group, if the HA_GS_SUBSCRIBE_MEMBERSHIP flag is set in the gs_subscription_type field.

gs_changing_membership
This field contains a pointer to the membership information block (shown in Figure 2 on page 10) that contains changing (either joining or leaving) providers, if the HA_GS_SUBSCRIBE_DELTA_JOIN or HA_GS_SUBSCRIBE_DELTA_LEAVE flag is set in the gs_subscription_type field.

gs_subscription_special_data
This field contains the special group-specific subscription data, if the HA_GS_SUBSCRIPTION_SPECIAL_DATA flag is set in the gs_subscription_type field.
For more information on subscription special data, refer to Appendix A, "Subscription Special Data" in *Group Services Programming Guide and Reference*, SA22-7355.

7.10 Dispatching notifications

The ha_gs_dispatch subroutine is used by a process to receive notifications from the Group Services shared library. It is vital that the GS client call this subroutine on a regular basis to be able to receive notifications from the GS shared library by executing the appropriate callback routines. This allows the GS client to respond to any protocols that may be executing in the group. The parameter to this subroutine controls the behavior of ha_gs_dispatch once all outstanding notifications have been delivered.

Although the ha_gs_dispatch subroutine needs to be called regularly, exactly how often will differ for each GS client. The most important factor is that the GS client should be ready to respond to arriving notifications as quickly as possible to allow it to respond to changes in its group or the system as quickly as possible.

Once the ha_gs_dispatch subroutine is called, it will process all notifications that have arrived, which may result in multiple GS client callback routines being executed.

7.10.1 Subroutine call

To receive notifications from Group Services and to execute corresponding callback subroutines, GS client must execute the ha_gs_dispatch subroutine regularly. On input, the GS client provides the flag that indicates how notifications are to be processed.

The syntax of the ha_gs_dispatch subroutine is shown in Figure 106.

```
ha_gs_rc_t ha_gs_dispatch(
   const ha_gs_dispatch_flag_t dispatch_flags)
```

Figure 106. The syntax of ha_gs_dispatch subroutine

The dispatch_flags parameter can be one of the following values:

HA_GS_NON_BLOCKING
The GSAPI should check for notifications that have arrived on the GSAPI socket. If any notifications have arrived, the GSAPI should call the appropriate callback subroutines. If no notifications have arrived, the GSAPI should return control immediately.
This model is appropriate to single-threaded (or non-threaded) GS clients. It is expected that the GS client will remain responsive to arriving notifications using the select subroutine or similar mechanisms.

HA_GS_BLOCKING
The GSAPI should check for notifications that have arrived on the GSAPI socket. As notifications arrive, the GSAPI will call the appropriate callback subroutines, and it will continue to do so until an error occurs or the connection is broken.
This model is appropriate to multi-threaded GS clients, although it may be used by single-threaded (or non-threaded) GS clients.

Chapter 8. Error handling

As with general subroutines, the subroutines provided by Group Services return an error code. However, Group Services uses a unique mechanism to return an error code. Even a subroutine does not return an error immediately. The subroutine may return an error code later; therefore, you must be careful to handle error codes.

This chapter covers an error handling mechanism of the Group Services.

8.1 Synchronous/asynchronous errors

When an application calls subroutines provided by the Group Services shared library, Group Services checks the subroutine call for errors. If the subroutine call is syntactically invalid, the application receives a syntax error code. If the group currently has an executing protocol and an application proposes another protocol, the application receives an error code that indicates a collision between competing protocols. This type of error code is returned synchronously, and, therefore, it is called a *synchronous error*.

If there is no synchronous error, the Group Services tentatively accepts the subroutine call, and the application receives a successful return code synchronously. However, if collision errors are detected asynchronously (because other providers or Group Services itself submit a proposal at the same time), Group Services returns an error code. This type of error code is returned asynchronously, and, therefore, it is called an *asynchronous error*.

An asynchronous error will be delivered by the ha_gs_delayed_error_callback subroutine. This subroutine is provided by a GS client when it registers itself to the Group Services by using the ha_gs_init subroutine.

Not all subroutines have both synchronous and asynchronous errors. Table 7 summarizes the subroutines and their error types.

Table 7. Subroutines and synchronous/asynchronous error

Subroutines	Synchronous	Asynchronous
For GS clients		
ha_gs_init	Yes	No
ha_gs_quit	No	No
For providers		

Subroutines	Synchronous	Asynchronous
ha_gs_join	Yes	Yes
ha_gs_change_state_value	Yes	Yes
ha_gs_send_message	Yes	Yes
ha_gs_leave	Yes	Yes
ha_gs_goodbye	Yes	No
ha_gs_expel	Yes	Yes
ha_gs_change_attributes	Yes	Yes
ha_gs_vote	Yes	No
For subscribers		
ha_gs_subscribe	Yes	Yes
ha_gs_unsubscribe	Yes	No
For callback subroutines		
ha_gs_responsiveness_callback	Yes	No
ha_gs_delayed_error_callback	No	No
ha_gs_n_phase_callback	No	No
ha_gs_protocol_approved_callback	No	No
ha_gs_protocol_rejected_callback	No	No
ha_gs_announcement_callback	No	No
ha_gs_subscriber_callback	No	No
For dispatch subroutines		
ha_gs_dispatch	Yes	No

8.2 Error code

Error codes used for synchronous and asynchronous errors are defined by the ha_gs_rc_t type shown in Figure 107 on page 155. Some error codes appear with only one of these errors, and others appear with both of them.

```
typedef enum {
    HA_GS_OK,
    HA_GS_OK_SO_FAR = HA_GS_OK,
    HA_GS_NOT_OK,
    HA_GS_EXISTS,
    HA_GS_NO_INIT,
    HA_GS_NAME_TOO_LONG,
    HA_GS_NO_MEMORY,
    HA_GS_NOT_A_MEMBER,
    HA_GS_BAD_CLIENT_TOKEN,
    HA_GS_BAD_MEMBER_TOKEN,
    HA_GS_BAD_PARAMETER,
    HA_GS_UNKNOWN_GROUP,
    HA_GS_INVALID_GROUP,
    HA_GS_NO_SOURCE_GROUP_PROVIDER,
    HA_GS_BAD_GROUP_ATTRIBUTES,
    HA_GS_WRONG_OLD_STATE,
    HA_GS_DUPLICATE_INSTANCE_NUMBER,
    HA_GS_COLLIDE,
    HA_GS_SOCK_CREATE_FAILED,
    HA_GS_SOCK_INIT_FAILED,
    HA_GS_CONNECT_FAILED,
    HA_GS_VOTE_NOT_EXPECTED,
    HA_GS_NOT_SUPPORTED,
    HA_GS_INVALID_SOURCE_GROUP,
    HA_GS_UNKNOWN_PROVIDER,
    HA_GS_INVALID_DEACTIVATE_PHASE,
    HA_GS_PROVIDER_APPEARS_TWICE,
    HA_GS_BACKLEVEL_PROVIDERS
} ha_gs_rc_t;
```

Figure 107. The ha_gs_rc_t type

Each value indicates the following information:

HA_GS_OK
The subroutine was successful. This return code is returned synchronously.

HA_GS_NOT_OK
An error occurred. This error is returned synchronously.

HA_GS_EXISTS
The GSAPI has already been initialized by a previous call to the ha_gs_init subroutine. This error is returned synchronously.

HA_GS_NO_INIT
An attempt was made to use the GSAPI without initializing it by calling the ha_gs_init subroutine. This error is returned synchronously.

HA_GS_NAME_TOO_LONG
A name string was specified that was longer than that given by the

HA_GS_MAX_GROUP_NAME_LENGTH symbolic constant. This error is returned synchronously.

HA_GS_NO_MEMORY
The Group Services subsystem could not allocate the required memory. This error is returned synchronously.

HA_GS_NOT_A_MEMBER
The provider that is proposing the protocol is no longer a provider for the specified group. This error is returned asynchronously. It can be returned in response to the protocol requests resulting from calls to the following subroutines: ha_gs_change_state_value, ha_gs_send_message, and ha_gs_leave.

HA_GS_BAD_CLIENT_TOKEN
This value is reserved for IBM use.

HA_GS_BAD_MEMBER_TOKEN
The specified token does not represent a valid provider or subscriber instance for this client. This error is returned synchronously.

HA_GS_BAD_PARAMETER
The specified parameter was not valid. This error can be returned either synchronously or asynchronously, depending on when it was detected.

HA_GS_UNKNOWN_GROUP
The group that was specified on the call to the ha_gs_subscribe subroutine does not exist. This error is returned asynchronously.

HA_GS_INVALID_GROUP
The process does not have permission to join the group that was specified on the call to the ha_gs_join subroutine. For example, this error would be returned in response to an attempt to join a system-defined group, such as the host membership group or an adapter membership group. This error is returned asynchronously.

HA_GS_NO_SOURCE_GROUP_PROVIDER
A call to the ha_gs_join subroutine specified a source-group name, and there is no provider from that source-group already active on this node. This error is returned asynchronously.

HA_GS_BAD_GROUP_ATTRIBUTES
The group attributes that were specified on a call to the ha_gs_join subroutine are either invalid or do not the match the group attributes that were specified by the providers that already belong

to the group. This error can be returned either synchronously or asynchronously, depending on when it was detected.

HA_GS_WRONG_OLD_STATE
This value is reserved for IBM use.

HA_GS_DUPLICATE_INSTANCE_NUMBER
The provider instance number that was specified on a call to the ha_gs_join subroutine is already in use for this group on this node. This error is returned asynchronously.

HA_GS_COLLIDE
Another protocol is already active for this group. This error can be returned either synchronously or asynchronously, depending on when it was detected. This error is returned in response to the protocol requests resulting from calls to the following subroutines: ha_gs_change_state_value, ha_gs_send_message, and ha_gs_leave, ha_gs_change_attributes.

HA_GS_SOCK_CREATE_FAILED
The Group Services subsystem could not create a socket for communication. This error is returned synchronously.

HA_GS_SOCK_INIT_FAILED
The Group Services subsystem could not initialize the socket for communication. This error is returned synchronously.

HA_GS_CONNECT_FAILED
The Group Services subsystem could not complete the connection. Possible causes are: The Group Services daemon is not running or it is not ready to accept connections. This error is returned synchronously.

HA_GS_VOTE_NOT_EXPECTED
A vote was received but was not expected. Either no protocol was in progress or the Group Services subsystem already received a vote for this protocol. This error is returned synchronously.

HA_GS_NOT_SUPPORTED
The requested function is not currently supported. This error is returned synchronously.

HA_GS_INVALID_SOURCE_GROUP
The process specified an invalid source group on the call to the ha_gs_join subroutine. For example, this error would be returned in response to an attempt to specify as a source group a system-defined group, such as the host membership group or an adapter membership group. This error is returned synchronously.

HA_GS_UNKNOWN_PROVIDER

At least one of the providers that was specified in an expel protocol is not a member of the specified group. This error can be returned either synchronously or asynchronously, depending on when it was detected. This error is returned in response to the protocol requests resulting from calls to the ha_gs_expel subroutine.

HA_GS_INVALID_DEACTIVATE_PHASE

The process specified a phase other than 0 or 1 on the call to the ha_gs_expel subroutine for a one-phase expel protocol. This error is returned synchronously.

HA_GS_PROVIDER_APPEARS_TWICE

A provider to be expelled is listed twice in the given gs_expel_list provided by the expel request block shown in Figure 52 on page 81.

HA_GS_BACKLEVEL_PROVIDERS

A protocol request was made, and the group contains active providers that were compiled against an older level of the Group Services shared library that does not support the new protocol request. This error is returned asynchronously.

Part 3. Group Services programming

Chapter 9. Recoverable Network File System

Network File System (NFS) could be the most popular solution for sharing data throughout the distributed application environment. One node exports its local file system and all the nodes mount this file system as their network file system. As long as all the applications use their network file system, they are accessing an identical file system and sharing the data.

However, the NFS server node may not be up and running all the time. It might need to be shut down for maintenance, or it may fail by accident. In either case, a system operator needs to do some work. Assigning a new server node from the available nodes, creating a local file system on it, copying the data from the old server node to the new server node, and, finally, remounting all the node's network file systems.

To automate this recovery procedure, this chapter introduces a unique environment called the Recoverable Network File System (RNFS) environment. There are two programs used to provide an RNFS environment: RNFS and the Recoverable Network File System Monitor (RNFSM). The RNFS program executes NFS recovery procedures automatically. The RNFSM program monitors the state of the RNFS environment. These programs utilize many services provided by Group Services. If you compare the complexity of the recovery procedure with the size of programs that are short enough to read through, you will realize how Group Services is useful and powerful.

Note

The purpose of this chapter is helping you understand the Group Services programming not providing you with a complete solution. Therefore the programs are incomplete to use in the real environment.

In this chapter, in some cases, the term, *node n*, is used for *the rnfs program running on node n*. For example, *node n proposes a protocol* actually means *the rnfs program running on node n proposes a protocol*.

9.1 Mechanism

The *rnfs* program runs on multiple nodes and creates the RNFS group (the program uses *rnfs_group* for the group name). In the group, there is only one NFS server node. All the nodes, including the server node, mount the server node's local file system (the program uses */local_nfs* for the local file system name) as network file system (the program uses */shared_nfs* for the network file system name). When the server node takeover occurs, one available node in the group is selected as a new server node. All the nodes, including the new server node, remount the new server node's /local_nfs as /shared_nfs. The new server node's /local_nfs must be updated as soon as possible.

To satisfy this requirement, the program uses the following tactics:

- All the nodes in the group, rnfs_group, have their own local file system, /local_nfs, and they export it; so, any other node in the group can mount it at any time.
- To keep the shared data as new as possible for a case of server node takeover, all the nodes, except a server node, replicate /shared_nfs to their /local_nfs once in a while.
- If server node takeover occurs as planned, the remaining nodes are required to replicate /shared_nfs to their /local_nfs before remounting the /shared_nfs.
- If server node takeover occurs by accident, the remaining nodes just remount the /shared_nfs.
- An application that wants to use an RFNS environment must access /shared_nfs instead of /local_nfs. They are on the node on which the application is running.

Figure 108 on page 163 illustrates the mechanism of the rnfs program. Currently, node 0 is the server node, and all the nodes (nodes 0, 1, and 2) mount node 0's /local_nfs as /shared_nfs. To keep a client node's /local_nfs up to date, the client nodes (nodes 1 and 2) must, occasionally, replicate /shared_nfs to their /local_nfs. When server node takeover occurs, node 1 is selected as a new server node in this example. All the nodes (nodes 1 and 2) umount /shared_nfs and remount node 1's /local_nfs as /shared_nfs.

The rnfs program does not have the rnfs server program or the rnfs client program. This means that you do not need to think about which node will be the server node. You can also add nodes to the group or delete nodes from the group dynamically.

The *rnfsm* program monitors the state of nodes in the group. If a server node is changed, it reports the new server node number. If nodes are added or deleted, it reports all the nodes currently in the group.

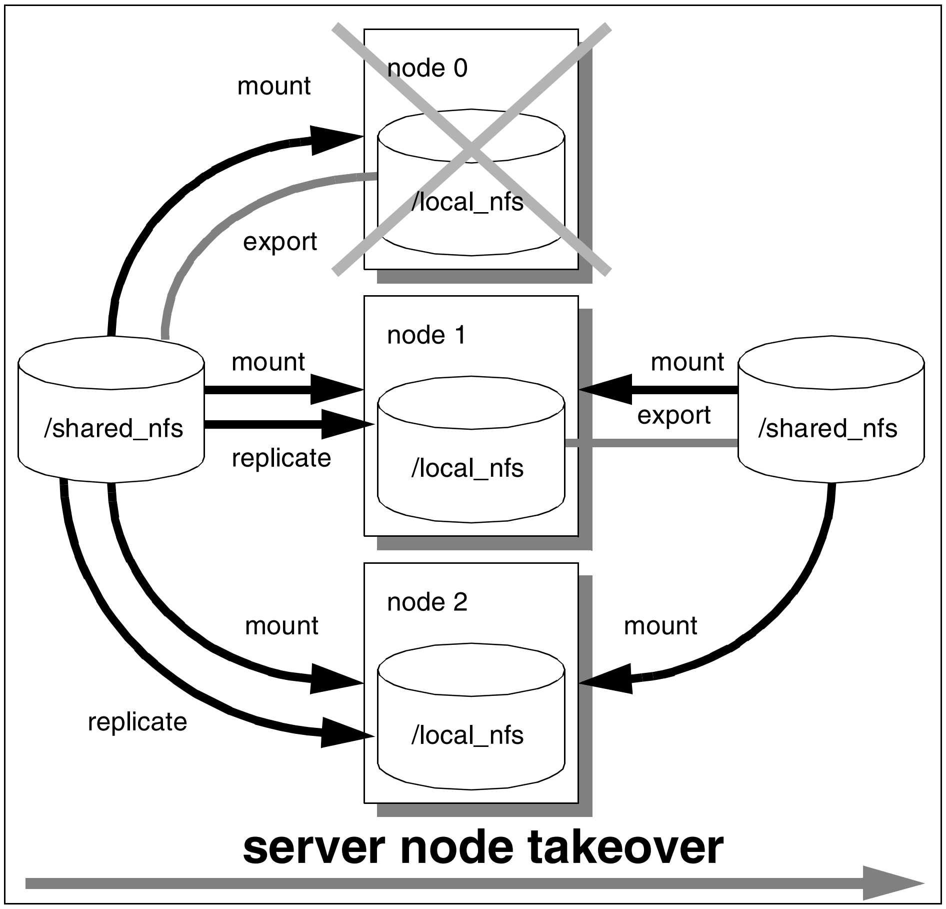


Figure 108. Program mechanism

9.2 rnfs program overview

The rnfs program is required to predict all the possible situations that could occur during normal operation, and it must prepare for them. The following nine situations are managed by the program:

1. Checking responsiveness
2. Creating the group
3. Adding a node
4. Replicating a file system
5. Server node shutdown

6. Client node shutdown
7. Server node failure
8. Client node failure
9. Receiving announcement
10. Receiving delayed error

How the program manages these situations is described in Section 9.3, “rnfs program in details” on page 170.

9.2.1 Program state

The program has its own state variables to manage its state. The following two global variables are used for this purpose:

- **ima** - This variable specifies the role of the program in the group. The variable takes one of the following values:
  - **RNFS_SERVER** - The program roles of an NFS server node.
  - **RNFS_CLIENT** - The program roles of an NFS client node.
- **imdoing** - This variable specifies the program’s condition in the group. The variable takes one of the following values:
  - **RNFS_JOINING** - The program is joining the RNFS group.
  - **RNFS_STABLE** - The program has joined the RNFS group and /shared_nfs is mounted. All the other programs have responsiveness.
  - **RNFS_UNSTABLE** - The program has joined the RNFS group; however, /shared_nfs is not mounted and/or some other programs do not have responsiveness.
  - **RNFS_LEAVING** - The program is leaving the RNFS group.

The program defines these values and variables as follows:

```
typedef enum {
   RNFS_CLIENT,
   RNFS_SERVER
} rnfs_ima_t;
typedef enum {
   RNFS_JOINING,
   RNFS_STABLE,
   RNFS_UNSTABLE,
   RNFS_LEAVING
} rnfs_imdoing_t;

rnfs_ima_t              ima;
rnfs_imdoing_t          imdoing;
```

The state diagram of the program is illustrated in Figure 109 on page 167.

When the program has initialized with the Group Services by calling the ha_gs_init subroutine, it is ready to propose a join protocol (①). If the program is the first provider, it creates the group, becomes a server node, and mounts its own /local_nfs to /shared_nfs (②). If the program is not the first provider, it joins the group, becomes a client node, and mounts the server node's /local_nfs to /shared_nfs (③).

When a server node shutdown occurs as planned, the server node umounts /shared_nfs and leaves the group (⑥). At the same time, the client nodes replicate /shared_nfs to their /local_nfs and then umount /shared_nfs. If a client node is listed at the top of the membership list after the server node leaves, this client node becomes a new server node (④). The other client nodes remain as client nodes (⑤). The new server node registers its hostname (②) so that all the client nodes can mount the new server node's /local_nfs to /shared_nfs (③). The new server node also needs to mount its /local_nfs to /shared_nfs. When a client node shutdown occurs as planned, the client node umounts /shared_nfs and leaves the group (⑦).

When a server node failure occurs by accident, the client nodes do the same procedure as when a server node shutdown occurs, except for file system replication, because /shared_nfs (that is, /local_nfs of the server node) has been lost. When a client node failure occurs by accident, no action is taken for this.

The file system replication is required when the program is a client node and has mounted /shared_nfs (③).

The responsiveness check is applied for all the states. If one or more programs lose their responsiveness, all the programs get notified and put themselves in the RNFS_UNSTABLE state (④ or ⑤). If a program is in the

RNFS_UNSTABLE state, it rejects a join protocol to prevent the addition of a node to the group. When the programs that previously lost their responsiveness now start responding, all the programs get notified and put themselves in the RNFS_STABLE state (② or ③).

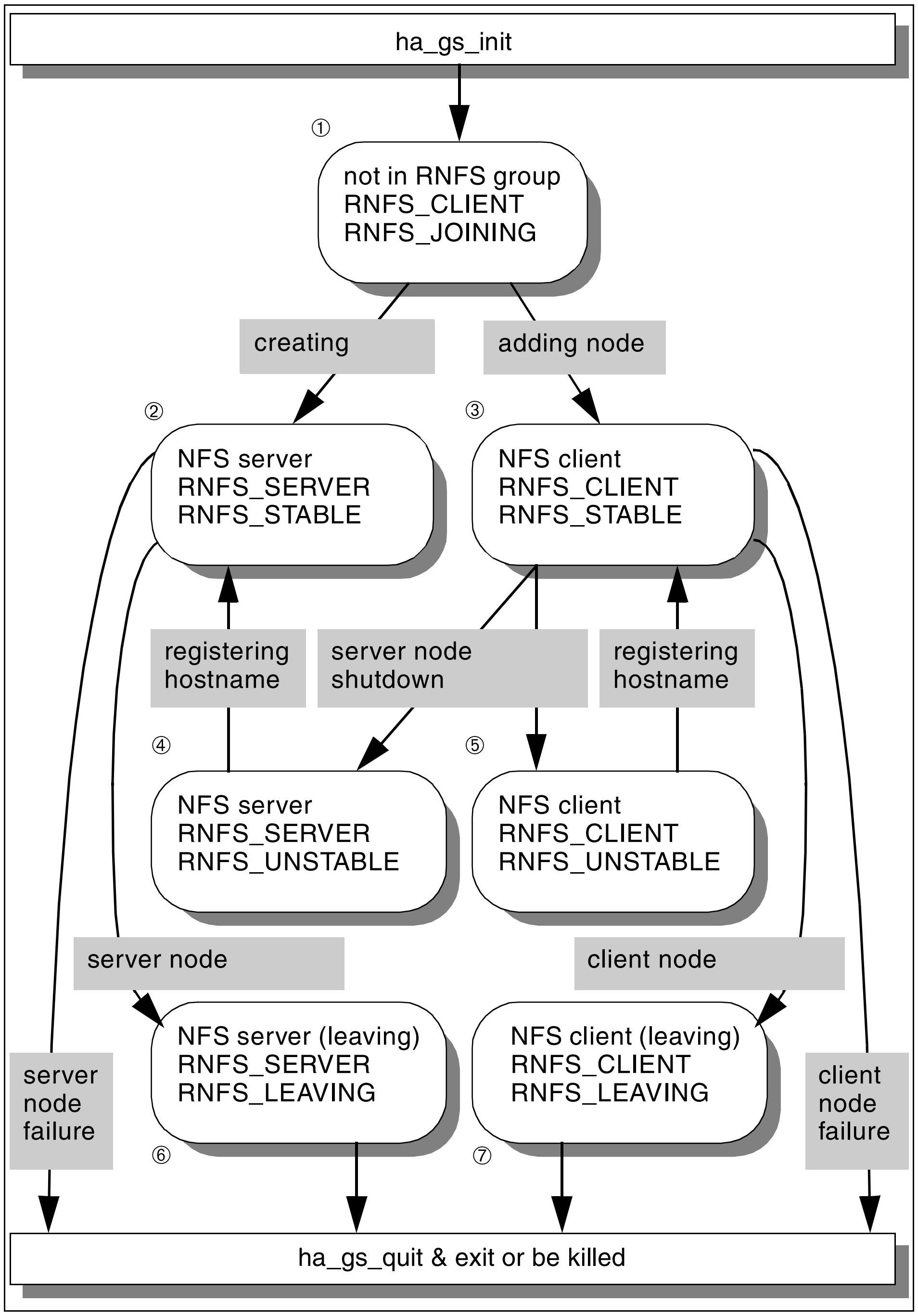


Figure 109. Program state diagram

9.2.2 Utilizing Group Services

The program utilizes many services provided by the Group Services. The following sections describes these services and how the program uses them.

9.2.2.1 Services

The following services are used by the program:

The membership list
The list contains all the available nodes in the group. The node listed at the top of the list plays the role a server node.

The group state value
The value contains the hostname of a server node. All the nodes use this value when they mount a server node's /local_nfs to /shared_nfs.

The provider-broadcast message
All the client nodes are required to keep their /local_nfs up to date in case it becomes a new server node. The server node occasionally sends a provider-broadcast message to ask them to replicate the file system.

Responsiveness check
All the nodes are periodically checked for responsiveness. If a node loses its responsiveness, all the nodes receive this information. If one or more nodes lose their responsiveness, the group considers itself unstable and rejects a join protocol.

Deactivate-on-failure
When node failure occurs, a deactivate script is executed to umount /shared_nfs for the failure node.

N-phase protocol
All the nodes are required to mount, umount, or replicate a file system within a voting time limit. If they have not completed, Group Services uses a default vote value; this default vote value is *approve*, and the protocol completes. However, all the nodes asynchronously receive the information about which nodes have not completed their job. Voting to continue is not used by the program. Therefore, the program has, at most, two phases.

9.2.2.2 Protocols

The following protocols are used by the program:

Join protocol
This protocol is used to create the group or to add a node to the group.

Voluntary leave protocol
This protocol is used for a node shutdown. A server node shutdown allows the remaining nodes to replicate the file system.

Failure leave protocol
This protocol is used for a node failure.

State value change protocol
This protocol is used for a server node's hostname registration. If a client node becomes a server node, it proposes this protocol.

Provider-broadcast message protocol
This protocol is used for file system replication. A server node broadcasts a file system replication message once in a while. Upon receiving this message, client nodes replicate the file system.

9.2.2.3 Notifications

The following notifications are used by the program:

Responsiveness notification
After initializing with the Group Services, all the nodes are expected to return the value of HA_GS_CALLBACK_OK to this notification.

N-phase notification
A required action by this notification depends on each protocol. In most cases, a node is required to vote when the required action is completed.

Protocol approved notification
A required action by this notification depends on each protocol.

Protocol rejected notification
A join protocol could be rejected if the group is unstable.

Announcement notification
When nodes lose their responsiveness or they have not completed a required action within the voting time, all the nodes receive this notification.

Delayed error notification
If the program is already running on a node, or the hostname registration request or file system replication request are canceled, a program receives this notification.

9.3 rnfs program in details

This section describes the rnfs program in detail. The entire program is provided in Section C.1, “rnfs.c” on page 261.

9.3.1 main routine

The first half of main routine is shown in Figure 110 on page 171. In this part, the program does the following operations:

1. Set domain name, group name, and instance number. The domain name is given as a parameter of the program. The group name is defined as:

   ```
   #define RNFS_GROUP_NAME   "rnfs_group"
   ```

   The instance number is defined as:

   ```
   #define RNFS_INSTANCE_NUM 5523
   ```

   The number is fixed; therefore, it prevents multiple instances from running on the same node.

2. Set some global variables.
3. It calls the init_program subroutine to initialize the program with the Group Services.
4. It calls propose_join subroutine to join the group. This will be the case described in Section 9.3.3, “Creating the group” on page 175, or Section 9.3.4, “Adding a node” on page 180.

```
/*************************************
* main
*************************************/
int main(int argc, char **argv) {
   char           key;
   fd_set         my_fd;
   struct timeval timeout;
   int            replicate;

   if(argc != 2) {
      printf("Usage: %s domain_name\n", argv[0]);
      exit(argc);
   }
   strcpy(domain_name, "HA_DOMAIN_NAME=");
   strcat(domain_name, argv[1]);
   putenv(domain_name);
   printf("domain name: %s, ", getenv("e"));
   printf("group name: %s, ", RNFS_GROUP_NAME);
   printf("instance number: %d\n", RNFS_INSTANCE_NUM);

   replicate       = 0;
   ima             = RNFS_CLIENT;
   descriptor      = 0;
   timeout.tv_sec  = 1;
   timeout.tv_usec = 0;

   init_program();
   propose_join();
```

Figure 110. main routine (the first half)

The initialization subroutine is shown in Figure 111 on page 172. In this subroutine, the program performs the following operations:

1. Set the responsiveness control block as follows:
   - The program uses ping type responsiveness checks.
   - The Group Services checks responsiveness every two seconds, and the program must reply within one second. These are defined as:

     ```
     #define RNFS_RESPONSE_RATE           2
     #define RNFS_RESPONSE_TIME_LIMIT     1
     ```
2. Set the deactivate script defined as:

   ```
   #define RNFS_DEACTIVATE    "./rnfs_deact.ksh"
   ```
3. Set the address for the following callback subroutines:

   ```
   ha_gs_responsiveness_callback
   ha_gs_delayed_error_callback
   ```

```
/**************************************
* init_program (ha_gs_init)
**************************************/
void init_program() {

   responsiveness.gs_responsiveness_type     = HA_GS_PING_RESPONSIVENESS;
   responsiveness.gs_responsiveness_interval = RNFS_RESPONSE_RATE;
   responsiveness.gs_responsiveness_response_time_limit = RNFS_RESPONSE_TIME_LIMIT;
   responsiveness.gs_counter_location        = NULL;
   responsiveness.gs_counter_length          = NULL;

   gs_rc = ha_gs_init(
      &descriptor,
      HA_GS_SOCKET_NO_SIGNAL,
      &responsiveness,
      RNFS_DEACTIVATE,
      ha_gs_responsiveness_callback,
      ha_gs_delayed_error_callback,
      NULL);
   if(gs_rc != HA_GS_OK) {
      printf("*** ha_gs_init failed rc=%d ***\n", gs_rc);
      exit(-1);
   }
   return;
}
```

Figure 111. Initialization subroutine

The last half of the main routine is shown in Figure 112 on page 173. In this part, the program enters an infinite loop and performs the following operations:

1. Set two file descriptors for the select subroutine. One for stdin and the other for the Group Services. Then, the program calls the select subroutine. It waits one second, at most, and then returns.
2. If there is data from stdin, the program calls the suspend_program subroutine to suspend the program and get your command.
3. If there is data from Group Services, the program calls the ha_gs_dispatch subroutine to receive notification.
4. If the program runs on a server node, if it is not leaving the group, and if almost 10 seconds have passed since the last file system replication, the program calls the propose_message subroutine to propose a provider-broadcast message protocol. This case is described in Section 9.3.5, “Replicating a file system” on page 183.

```
   printf("hit <Enter> key to suspend\n");
   for(;;) {
      FD_ZERO(&my_fd);
      FD_SET(0, &my_fd);
      FD_SET(descriptor, &my_fd);
      rc = select(descriptor + 1, &my_fd, NULL, NULL, &timeout);
      if(rc < 0) {
         printf("*** select failed rc=%d ***\n", rc);
         exit(rc);
      }
      if(FD_ISSET(0, &my_fd)) {
         suspend_program();
      }
      if(descriptor && FD_ISSET(descriptor, &my_fd)) {
         gs_rc = ha_gs_dispatch(HA_GS_NON_BLOCKING);
         if(gs_rc != HA_GS_OK) {
            printf("*** ha_gs_dispatch failed rc=%d ***\n", gs_rc);
         }
      }
      if(ima == RNFS_SERVER) {
         if((imdoing != RNFS_LEAVING) && (replicate > RNFS_REPLICATE_RATE)) {
            propose_message();
            replicate = 0;
         } else {
            replicate++;
         }
      }
   }
}
```

Figure 112. main routine (the last half)

The suspension subroutine is shown in Figure 113 on page 174. If there is data from stdin (in other words, if you press the Enter key), the program suspends and awaits your command. You can choose one of the following operations:

- Press the **r** or **R** key to resume the program.
- Press the **l** or **L** key to leave the group.

The resume operation simply returns to the main routine. The leave operation calls the propose_leave subroutine to propose a voluntary leave protocol. This case is described in Section 9.3.6, “Server node shutdown” on page 186, and Section 9.3.7, “Client node shutdown” on page 194.

As you may have noticed, you can press any keys, other than l or L, to resume operation.

```
/*************************************
* suspend_program
*************************************/
void suspend_program() {
   char proposal[32];

   gets(proposal); /* remove previously input strings */
   printf("[ program suspended ] l(eave) or r(esume)?: ");
   scanf("%s", proposal);
   switch((int)proposal[0]) {
   case 'l': case 'L':
      propose_leave();
      break;
   default:
      break;
   }
   gets(proposal); /* remove extra strings */
   return;
}
```

Figure 113. Suspension subroutine

9.3.2 Checking responsiveness

To make the RNFS environment reliable, the program utilizes the responsiveness check facility. Group Services checks the responsiveness of nodes occasionally, and, if the node loses its responsiveness, Group Services notifies all the nodes in the group.

Figure 114 on page 175 illustrates the control flow for checking responsiveness. When nodes 0, 1, and 2 have initialized with the Group Services successfully (①), they start receiving a responsiveness notification from the Group Services (②). The Group Services sends this notification every two seconds. This interval is defined as follows:

```
#define RNFS_RESPONSE_RATE 2
```

Every time the node receives the notification, it is required to return the value of HA_GS_CALLBACK_OK (③). This must be done within one second. This time limit is defined as follows:

```
#define RNFS_RESPONSE_TIME_LIMIT 1
```

A node receives a responsiveness notification regardless of whether it has already joined the group or not.

If a node fails to return the value, Group Services sends an announcement notification to all the nodes. For information about this situation, refer to Section 9.3.10, “Receiving an announcement” on page 202.

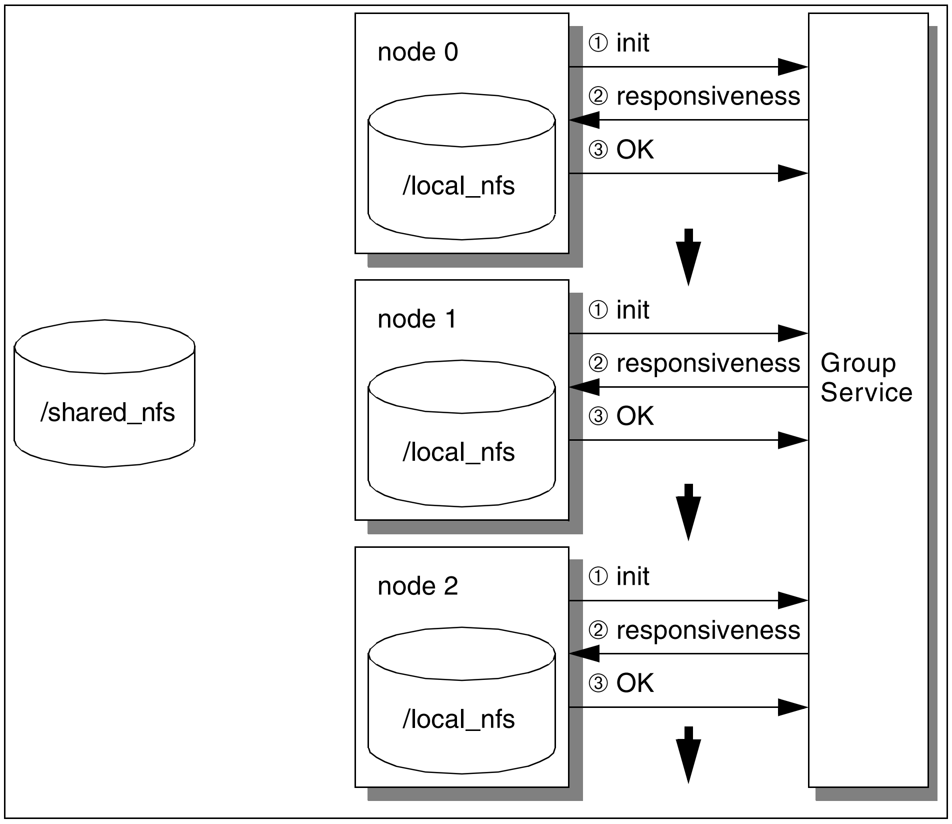


Figure 114. Checking responsiveness

The responsiveness notification is implemented as shown in Figure 115.

```
/****************************************
* responsiveness notification
****************************************/
ha_gs_callback_rc_t ha_gs_responsiveness_callback(
   const ha_gs_responsiveness_notification_t *block) {

   return(HA_GS_CALLBACK_OK);
}
```

Figure 115. Responsiveness notification

9.3.3 Creating the group

When it is time to create the RNFS environment, more than one node is going to join the group at the same time. At this point, there is no server-client relationship between the nodes, and no node knows the hostname of a server node.

Figure 116 on page 177 illustrates the control flow for creating the group. Three nodes propose a join protocol at the same time (①). These join proposals are handled by Group Services one-by-one. Because the group has batching join protocols disabled. Assuming that node 0 is selected by Group Services for the first join protocol.

When node 0 proposes a join protocol, it receives an n-phase notification (②). This notification specifies how many nodes are currently in the group. This should be zero because this is the first join protocol for the group. At this point, node 0 realizes it will be a server node. Therefore, node 0 votes to approve and, thereby, proposes its hostname for the group state value (③). This must be done within five seconds. This time limit is defined as follows:

```
#define RNFS_JOIN_FAILURE_TIME_LIMIT 5
```

Then, it receives a protocol approved notification with the updated membership list (④). Using the group state value, the node executes the rnfs_mount script (refer to Section 9.4.3, “rnfs_mount shell script” on page 209) to mount node 0’s /local_nfs to /shared_nfs.

If node 0 fails to vote within the time limit, Group Services sends an announcement notification to all the nodes (in this case, node 0). For this situation, refer to Section 9.3.10, “Receiving an announcement” on page 202.

The rest of the nodes (node 1 and 2) are handled by Group Services in the manner described in Section 9.3.4, “Adding a node” on page 180.

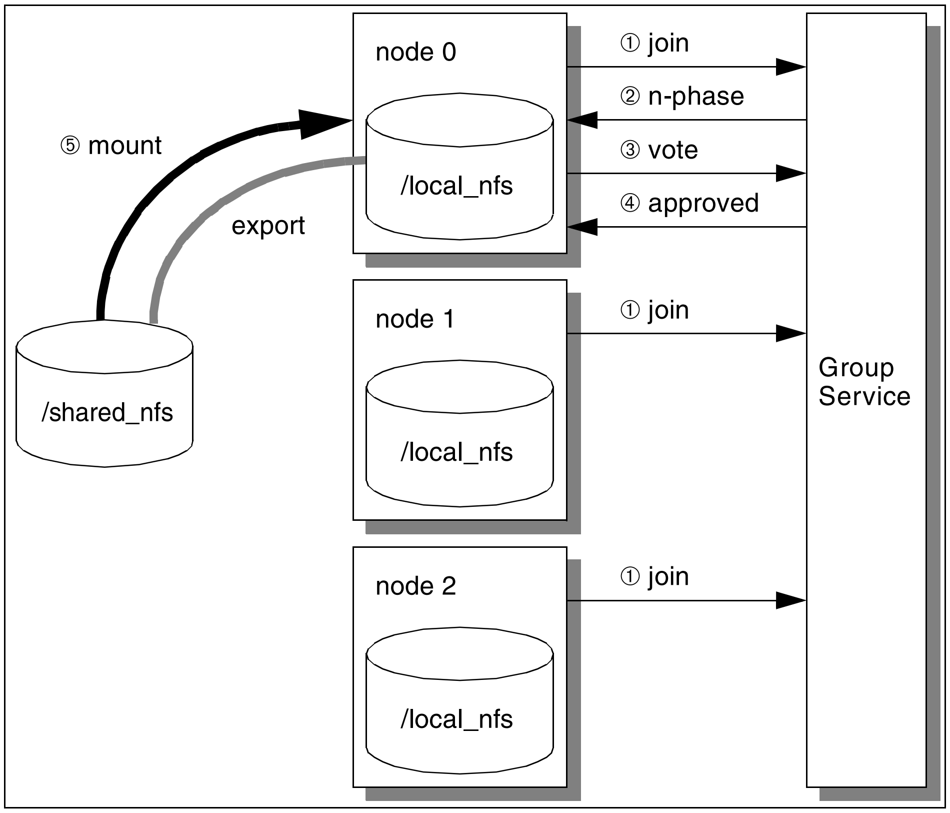


Figure 116. Creating the group

The program state changes as shown in Table 8.

Table 8. Program state change for creating the group

Node	Before protocol	After protocol
node 0	RNFS_CLIENT RNFS_JOINIG	RNFS_SERVER RNFS_STABLE
node 1	RNFS_CLIENT RNFS_JOINIG	No change
node 2	RNFS_CLIENT RNFS_JOINIG	No change

The join protocol proposal is implemented as shown in Figure 117 on page 178. Before calling the ha_gs_join subroutine, the imdoing variable is required to be set to RNFS_JOINING.

```
/***************************************
* propose_join (ha_gs_join)
***************************************/
void propose_join() {

   proposal_info.gs_join_request.gs_group_attributes    = &group_attributes;
   proposal_info.gs_join_request.gs_provider_instance   = RNFS_INSTANCE_NUM;
   proposal_info.gs_join_request.gs_provider_local_name = RNFS_LOCAL_NAME;
   proposal_info.gs_join_request.gs_n_phase_protocol_callback
      = ha_gs_n_phase_callback;
   proposal_info.gs_join_request.gs_protocol_approved_callback
      = ha_gs_protocol_approved_callback;
   proposal_info.gs_join_request.gs_protocol_rejected_callback
      = ha_gs_protocol_rejected_callback ;
   proposal_info.gs_join_request.gs_announcement_callback
      = ha_gs_announcement_callback;
   proposal_info.gs_join_request.gs_merge_callback      = NULL;

   group_attributes.gs_version                       = 1;
   group_attributes.gs_sizeof_group_attributes
      = sizeof(ha_gs_group_attributes_t);
   group_attributes.gs_client_version                = 1;
   group_attributes.gs_batch_control
      = HA_GS_NO_BATCHING | HA_GS_DEACTIVATE_ON_FAILURE;
   group_attributes.gs_num_phases                    = HA_GS_N_PHASE;
   group_attributes.gs_source_reflection_num_phases = HA_GS_1_PHASE;
   group_attributes.gs_group_default_vote            = HA_GS_VOTE_APPROVE;
   group_attributes.gs_merge_control                 = HA_GS_DISSOLVE_MERGE;
   group_attributes.gs_time_limit                    = RNFS_JOIN_FAILURE_TIME_LIMIT;
   group_attributes.gs_source_reflection_time_limit = NULL;
   group_attributes.gs_group_name                    = RNFS_GROUP_NAME;
   group_attributes.gs_source_group_name             = NULL;

   imdoing = RNFS_JOINING;

   gs_rc = ha_gs_join(
      &provider_token,
      &proposal_info);
   if(gs_rc != HA_GS_OK) {
      printf("*** ha_gs_join failed rc=%d **\n", gs_rc);
   }
   return;
}
```

Figure 117. Join protocol proposal

The n-phase notification for a join protocol is implemented as shown in Figure 118 on page 179. In this situation, the following condition is true:

```
!block->gs_proposal->gs_current_providers->gs_count
```

Therefore, node 0 joins the group as a server node and must register its hostname by voting to approve.

```
case HA_GS_JOIN:
     if(block->gs_proposal->gs_current_providers->gs_count == 0) {
        if(gethostname(hostname, 256)) {
           printf("*** gethostname failed ***\n");
        }
        host_name.gs_length = strlen(hostname) + 1;
        host_name.gs_state = hostname;
        vote_protocol(HA_GS_VOTE_APPROVE, &host_name);
     } else if(imdoing == RNFS_UNSTABLE) {
        vote_protocol(HA_GS_VOTE_REJECT, NULL);
     } else {
        vote_protocol(HA_GS_VOTE_APPROVE, NULL);
     }
     break;
```

Figure 118. N-phase notification (HA_GS_JOIN)

The voting subroutine is implemented as shown in Figure 119. Node 0 must specify its hostname in the second parameter.

```
/****************************************
* vote_protocol
****************************************/
void vote_protocol(
   ha_gs_vote_value_t vote_value,
   const ha_gs_state_value_t *host_name) {

   gs_rc = ha_gs_vote(
      provider_token,
      vote_value,
      host_name,
      NULL,
      HA_GS_NULL_VOTE);
   if(gs_rc != HA_GS_OK) {
      printf("*** ha_gs_vote failed rc=%d ***\n", gs_rc);
   }
   return;
}
```

Figure 119. Voting subroutine

The protocol approved notification for a join protocol is implemented as shown in Figure 120 on page 180. In this situation, the following conditions are true:

```
imdoing == RFNS_JOINING
```

and

```
mynodeis == serveris
```

Therefore, node 0 joins the group as a server node and mounts node 0's /local_nfs by using the rnfs_mount script.

```
case HA_GS_JOIN:
     if(imdoing == RNFS_JOINING) {
        mynodeis = block->gs_proposal->gs_proposed_by.gs_node_number;
        serveris = block->gs_proposal->gs_current_providers->gs_providers->gs_node_num
        if(mynodeis == serveris) {
           ima = RNFS_SERVER;
           printf("[ joined as server ] ");
        } else {
           ima = RNFS_CLIENT;
           printf("[ joined as client ] ");
        }
        printf("mount network file system from %s\n",
           block->gs_proposal->gs_current_state_value->gs_state);
        strcpy(mount_command, RNFS_MOUNT);
        strcat(mount_command,
           block->gs_proposal->gs_current_state_value->gs_state);
        if(rc = system(mount_command)) {
           printf("\n*** system failed rc=%d ***\n", rc);
        }
        imdoing = RNFS_STABLE;
     }
     break;
```

Figure 120. Protocol approved notification (HA_GS_JOIN)

9.3.4 Adding a node

One of the great abilities of the Group Services application is its scalability. You can dynamically add any number of nodes to the group without modifying the program. Also, you do not need to worry about whether the node is going to be a server or client node.

Figure 121 on page 181 illustrates the control flow for adding a node to the group. Assuming that nodes 0 and 1 have been in the group already, and node 0 is the server node. Then, node 2 is being added to the group.

When node 2 proposes a join protocol (①), all the nodes receive an n-phase notification (②). They are required to vote to approve or reject immediately (③). If the group is unstable (the variable, imdoing, is equal to the value of RNFS_UNSTABLE), they vote to reject; otherwise, they vote to approve. This must be done within five seconds. This time limit is defined as follows:

```
#define RNFS_JOIN_FAILURE_TIME_LIMIT 5
```

Then, they receive a protocol approved or rejected notification depending on the condition of the imdoing variable (④). Upon receiving the protocol approved notification, node 2 executes the rnfs_mount script (refer to Section

9.4.3, “rnfs_mount shell script” on page 209) to mount node 0’s /local_nfs as /shared_nfs using the group state value (⑤). Upon receiving the protocol rejected notification, node 2 exits the program, and nodes 0 and 1 do not have any changes. It is the responsibility of node 2 to execute the program again to join the group.

If a node fails to vote within the time limit, Group Services sends an announcement notification to all the nodes. For this situation, refer to Section 9.3.10, “Receiving an announcement” on page 202.

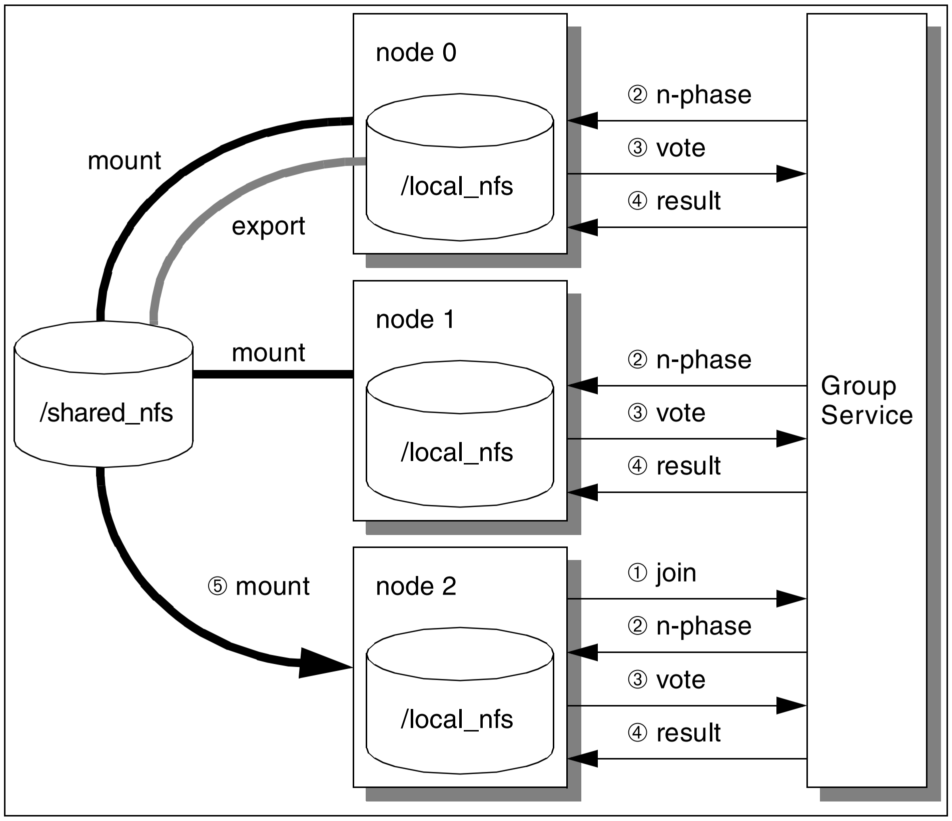


Figure 121. Adding a node

The program state changes as shown in Table 9.

Table 9. Program state change for adding a node

Node	Before protocol	After protocol
node 0	RNFS_SERVER RNFS_STABLE	no change

Node	Before protocol	After protocol
node 1	RNFS_CLIENT RNFS_STABLE	no change
node 2	RNFS_CLIENT RNFS_JOINIG	If approved, RNFS_CLIENT RNFS_STABLE If rejected, does not exist

The join protocol proposal is implemented as shown in Figure 117 on page 178. Before calling the ha_gs_join subroutine, the imdoing variable is required to be set to RNFS_JOINING.

The n-phase callback for a join protocol is implemented as shown in Figure 118 on page 179. In this situation, the following condition is false for all the nodes.

```
block->gs_proposal->gs_current_providers->gs_count == 0
```

Therefore, if the group is unstable, a node votes to reject or approve.

The protocol approved notification for a join protocol is implemented as shown in Figure 120 on page 180. In this situation, the condition

```
imdoing == RFNS_JOINIG
```

is false for node 0 and 1, and true for node 2.

For node 2, the condition

```
mynodeis == serveris
```

is false. Therefore, node 2 joins the group as a client node and mounts node 0's /local_nfs by using the rnfs_mount script.

The protocol rejected notification for a join protocol is implemented as shown in Figure 122 on page 183. In this situation, the condition

```
imdoing == RFNS_JOINIG
```

is false for node 0 and 1, and true for node 2. Therefore, nodes 0 and 1 do nothing, and node 2 exits the program.

```
/*************************************
* protocol rejected notification
*************************************/
void ha_gs_protocol_rejected_callback(
   const ha_gs_rejected_notification_t *block) {

   switch(block->gs_protocol_type) {
   case HA_GS_JOIN:
      if(imdoing == RNFS_JOINING) {
         printf("* warning * the group is unstable, join later - exit\n");
         exit(-1);
      }
      break;
   default:
      printf("*** protocol rejected notification is not expected ***\n");
      break;
   }
   return;
}
```

Figure 122. Protocol rejected notification

9.3.5 Replicating a file system

The server node takeover occurs by any chances. In this case, Group Services assigns a new server node from the available nodes in the group. The new server node is required to provide its /local_nfs as /shared_nfs. Therefore, the data in /local_nfs must be as new as possible. To achieve this goal, the current server node asks the client nodes to replicate /shared_nfs to their /local_nfs once in a while.

Figure 123 on page 184 illustrates the control flow for replicating file system. Assuming that node 0 is a server node and node 1 and 2 are client nodes.

The node 0 proposes a provider-broadcast message protocol with the message (①). This message is defined as follows:

```
#define RNFS_MEAAGE "replicate file system\0"
```

All the nodes receive an n-phase notification and confirm the contents of the message (②). Nodes 1 and 2 execute the rnfs_replicate script (refer to Section 9.4.4, “rnfs_replicate” on page 210) to replicate /shared_nfs to /local_nfs (③) and then vote to approve (④). This must be done within five seconds. This time limit is defined as follows:

```
#define RNFS_REPLICATE_TIME_LIMIT 5
```

All the nodes receive a protocol approved notification (⑤); however, no action is required for this.

If a node fails to vote within the time limit, Group Services sends an announcement notification to all the nodes. For this situation, refer to Section 9.3.10, “Receiving an announcement” on page 202.

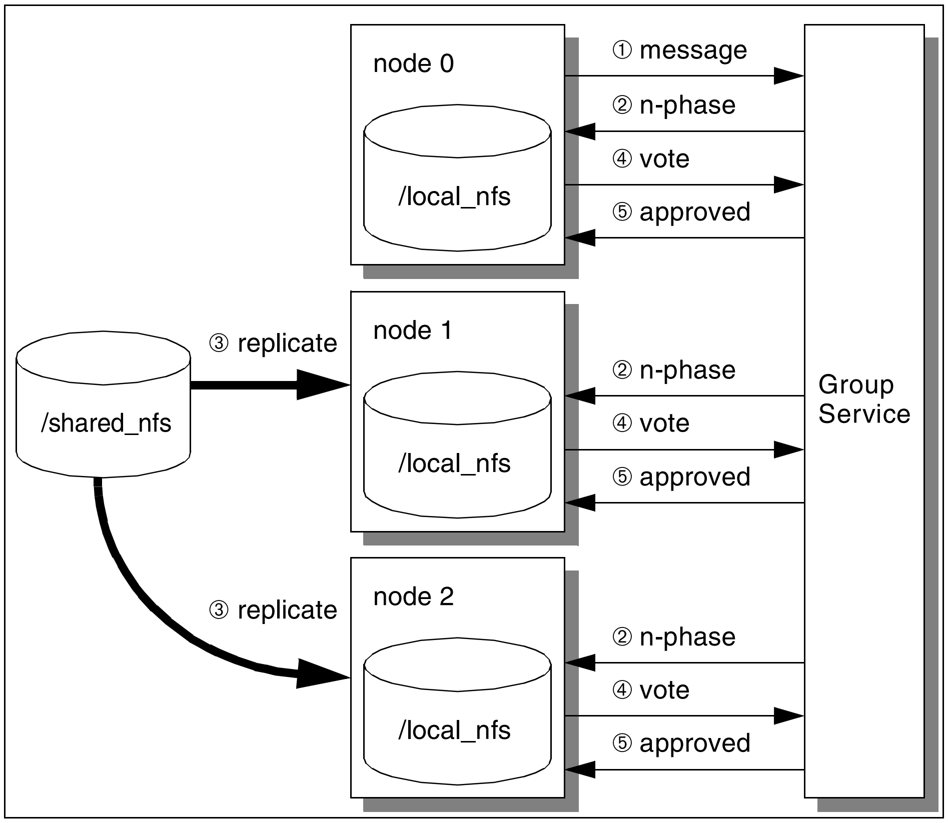


Figure 123. Replicating file system

The provider-broadcast message protocol proposal is implemented as shown in Figure 124 on page 185. If any other protocol is currently running, the proposal is canceled.

```
/*************************************
* propose_message (ha_gs_send_message)
*************************************/
void propose_message() {
   char                    message[2048];

   strcpy(message, RNFS_MESSAGE);

   proposal_info.gs_message_request.gs_num_phases         = HA_GS_N_PHASE;
   proposal_info.gs_message_request.gs_time_limit         = RNFS_REPLICATE_TIME_LIMIT;
   proposal_info.gs_message_request.gs_message.gs_length  = strlen(message) + 1;
   proposal_info.gs_message_request.gs_message.gs_message = message;

   gs_rc = ha_gs_send_message(
      provider_token,
      &proposal_info);
   if(gs_rc != HA_GS_OK) {
      if(gs_rc == HA_GS_COLLIDE) {
         printf("* warning * replication is canceled\n");
      } else {
         printf("*** ha_gs_send_message failed rc=%d ***\n", gs_rc);
      }
   }
   return;
}
```

Figure 124. Provider-broadcast message protocol proposal

The n-phase callback for a provider-broadcast message protocol is implemented as shown in Figure 125 on page 186. In this situation, the condition

```
ima == RFNS_CLIENT
```

is false for node 0 and true for node 1 and 2. All the nodes check the contents of the message, then node 1 and 2 execute rnfs_replicate script to replicate /shared_nfs to /local_nfs. Finally, all the nodes vote to approve to complete the notification.

```
case HA_GS_PROVIDER_MESSAGE:
     if(strcmp(RNFS_MESSAGE,
        block->gs_proposal->gs_provider_message->gs_message)) {
        printf("*** provider-broadcast message is not expected ***\n");
        break;
     }
     if(ima == RNFS_CLIENT) {
        printf("[ replicate ] replicate file system\n");
        if(rc = system(RNFS_REPLICATE)) {
           printf("\n*** system failed rc=%d ***\n", rc);
        }
     }
     vote_protocol(HA_GS_VOTE_APPROVE, NULL);
     break;
```

Figure 125. N-phase notification (HA_GS_PROVIDER_MESSAGE)

The protocol approved notification for a provider-broadcast message protocol is implemented as shown in Figure 126. No action is required for all the nodes. The n-phase protocol is used to detect the node that did not replicate the file system within a time limit.

```
case HA_GS_PROVIDER_MESSAGE:
     break;
```

Figure 126. Protocol approved notification (HA_GS_PROVIDER_MESSAGE)

9.3.6 Server node shutdown

The server node shutdown might be required for maintenance purposes. In this situation, all nodes in the group are required to umount /shared_nfs because a server node takeover will occur soon. Any available node can be a new server node. Therefore, the remaining nodes are required to replicate /shared_nfs to their /local_nfs before umount /shared_nfs.

Figure 136 on page 195 illustrates control flow for server node shutdown. Assuming that node 0 is a current server node and has shut down as planned, node 1 will be a new server node.

Node 0 proposes a voluntary leave protocol with the leaving code of RNFS_SERVER (①). Then, all the nodes receive an n-phase notification (②). At this moment, node 0 is removed from the membership list; therefore, it calls the ha_gs_quit subroutine and then umount /shared_nfs (③). Nodes 1 and 2 are required to check the leaving code. In this case, it is a server node (RNFS_SERVER). Therefore, they execute the rnfs_replicate script (refer to Section 9.4.4, “rnfs_replicate” on page 210) to replicate the file system (④) and then execute the rnfs_umount script (refer to Section 9.4.5, “rnfs_umount” on page 210) to umount /shared_nfs (⑤). When it is completed, they vote to approve (⑥). This must be done within five seconds. This time limit is defined as follows:

```
#define RNFS_SHUTDOWN_TIME_LIMIT 5
```

Nodes 1 and 2 receive a protocol approved notification with an updated membership list (⑦). Assuming that node 1 is listed at the top of the membership list, node 1 becomes a new server node and is required to propose a state value change protocol to register its hostname to the group state value. This must be done immediately. For information about registering a hostname, refer to the section entitled “Registering a hostname” on page 191.

If a node fails to vote within the time limit, Group Services sends an announcement notification to all the nodes. For this situation, refer to Section 9.3.10, “Receiving an announcement” on page 202.

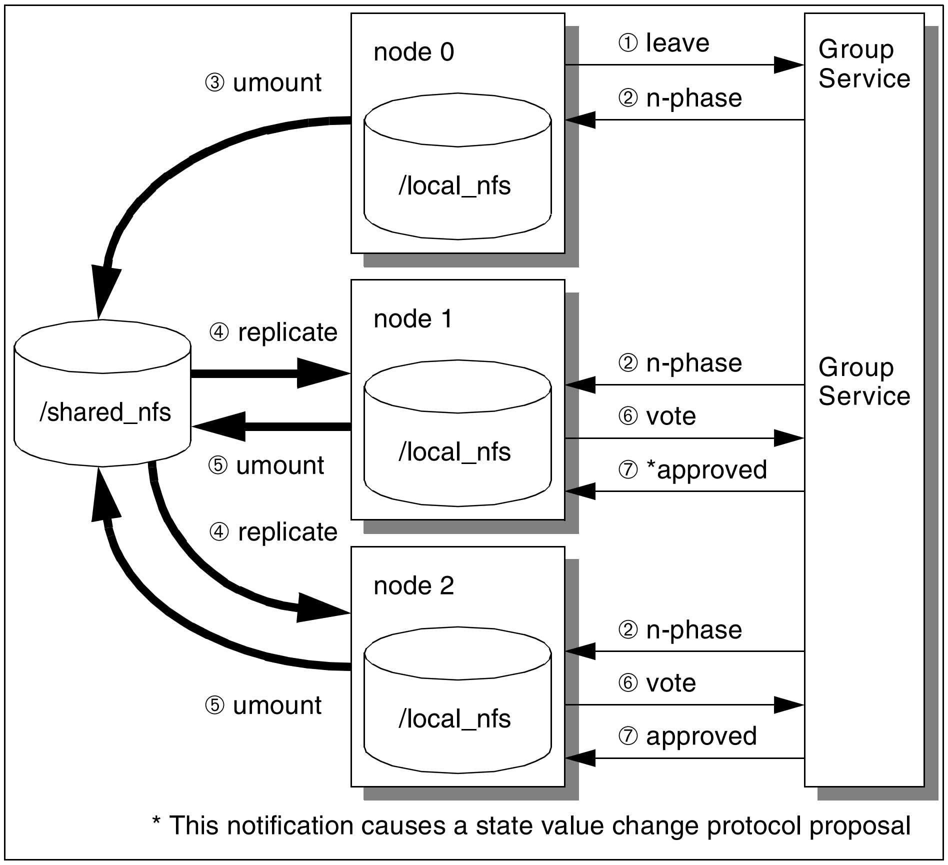


Figure 127. Server node shutdown

The program state is changed as shown in Table 10.

Table 10. Program state change for server node shutdown

Node	Before protocol	After protocol
node 0	RNFS_SERVER RNFS_LEAVING	Does not exist
node 1	RNFS_CLIENT RNFS_STABLE	RNFS_SERVER RNFS_UNSTABLE
node 2	RNFS_CLIENT RNFS_STABLE	RNFS_CLIENT RNFS_UNSTABLE

The voluntary leave protocol proposal is implemented as shown in Figure 128 on page 189. The program needs to set the leaving code to the value of the ima variable, and the imdoing variable to the value of RNFS_LEAVING. If any other protocol is currently running, the proposal is canceled. It is the responsibility of the server node to propose a voluntary leave protocol again.

```
/*************************************
* propose_leave (ha_gs_leave)
*************************************/
void propose_leave() {

   proposal_info.gs_leave_request.gs_num_phases = HA_GS_N_PHASE;
   proposal_info.gs_leave_request.gs_time_limit = RNFS_SHUTDOWN_TIME_LIMIT;
   proposal_info.gs_leave_request.gs_leave_code = ima;

   imdoing = RNFS_LEAVING;

   gs_rc = ha_gs_leave(
      provider_token,
      &proposal_info);
   if(gs_rc != HA_GS_OK) {
      if(gs_rc == HA_GS_COLLIDE) {
         printf("* warning * leaving the group is canceled\n");
      } else {
         printf("*** ha_gs_leave failed rc=%d ***\n", gs_rc);
      }
   }
   return;
}
```

Figure 128. Voluntary leave protocol proposal

The n-phase notification for a voluntary leave protocol is implemented as shown in Figure 129 on page 190. In this situation, the condition

```
imdoing != RNFS_LEAVING
```

is false for node 0 and true for nodes 1 and 2.

The following condition is true:

```
block->gs_proposal->gs_leave_info->gs_leave_codes->gs_voluntary_leave_code
== RNFS_SERVER
```

Therefore, nodes 1 and 2 call the rnfs_replicate script to replicate /shared_nfs to /local_nfs. Then, they call the rnfs_umount script to umount /shared_nfs. They must set the group to unstable to reject upcoming join protocols (if there are to be any) because a joining node could pick up node 0 as a server node instead of node 1. Then, they vote to approve.

```
case HA_GS_LEAVE:
     if(imdoing != RNFS_LEAVING) {
        if(block->gs_proposal->gs_leave_info->gs_leave_codes->gs_voluntary_leave_code
           == RNFS_SERVER) {
           printf("[ server shutdown ] replicate file system\n");
           if(rc = system(RNFS_REPLICATE)) {
              printf("\n*** system failed rc=%d ***\n", rc);
           }
           printf("[ server shutdown ] umount network file system\n");
           if(rc = system(RNFS_UMOUNT)) {
              printf("\n*** system failed rc=%d ***\n", rc);
           }
           imdoing = RNFS_UNSTABLE;
        }
        vote_protocol(HA_GS_VOTE_APPROVE, NULL);
     } else {
        quit_program();
     }
     break;
```

Figure 129. N-phase notification (HA_GS_LEAVE)

This is the last notification for node 0. Node 0 is not a provider of the group at this point and should not call the ha_gs_vote subroutine. Instead, it calls the quit_program subroutine to quit using GSAPIs as shown in Figure 130.

```
/***************************************
* quit_program (ha_gs_quit)
***************************************/
void quit_program() {

   gs_rc = ha_gs_quit();
   if (gs_rc != HA_GS_OK) {
      printf("*** ha_gs_quit failed rc=%d ***\n", gs_rc);
   } else {
      printf("[ server takeover ] umount network file system\n");
      if(rc = system(RNFS_UMOUNT)) {
         printf("\n*** system failed rc=%d ***\n", rc);
      }
      exit(0);
   }
   return;
}
```

Figure 130. Quit subroutine

The protocol approved notification for a voluntary leave protocol is implemented as shown in Figure 131 on page 191. Only nodes 1 and 2 receive this notification. In this situation, the condition

```
ima != RNFS_SERVER
```

is true. The condition

```
mynodeis ==
block->gs_proposal->gs_current_providers->gs_providers->gs_node_number
```

is true for node 1 and false for node 2. Therefore, node 1 starts registering its hostname.

```
case HA_GS_LEAVE:
     if(ima != RNFS_SERVER) {
        if(mynodeis ==
           block->gs_proposal->gs_current_providers->gs_providers->gs_node_number) {
           ima = RNFS_SERVER;
           propose_state();
        }
     }
     break;
```

Figure 131. Protocol approved notification (HA_GS_LEAVE)

Registering a hostname

Every time a server node takeover occurrs, a new server node is required to register its hostname to the group state value. This registration is notified to all the nodes in the group. Upon receiving this notification, all the nodes must mount the server node’s /local_nfs to /shared_nfs.

Figure 132 on page 192 illustrates the control flow for registering a hostname to the group state value. Assume that node 0 has left the group, node 1 has become a server node, and node 2 is still a client node.

Node 1 realizes it becomes a server node by receiving the protocol approved notification described in Section 9.3.6, “Server node shutdown” on page 186, or Section 9.3.8, “Server node failure” on page 196. Node 1 is required to register its hostname to the group state value so that nodes 1 and 2 can mount node 1’s /local_nfs as /shared_nfs. To do this, node 1 proposes a state value change protocol (①). Then, nodes 1 and 2 receive an n-phase notification with the proposed (not updated, because it is not approved) group state value (②). Upon receiving the notification, all the nodes mount node 1’s /local_nfs (③). When this is completed, they vote to approve (④). This must be done within five seconds. This time limit is defined as follows:

```
#define RNFS_TAKEOVER_TIME_LIMIT 5
```

Nodes 1 and 2 receive a protocol approved notification with an updated group state value (⑤). No action is required for this notification.

If a node fails to vote within the time limit, Group Services sends an announcement notification to all the nodes. For this situation, refer to Section 9.3.10, “Receiving an announcement” on page 202.

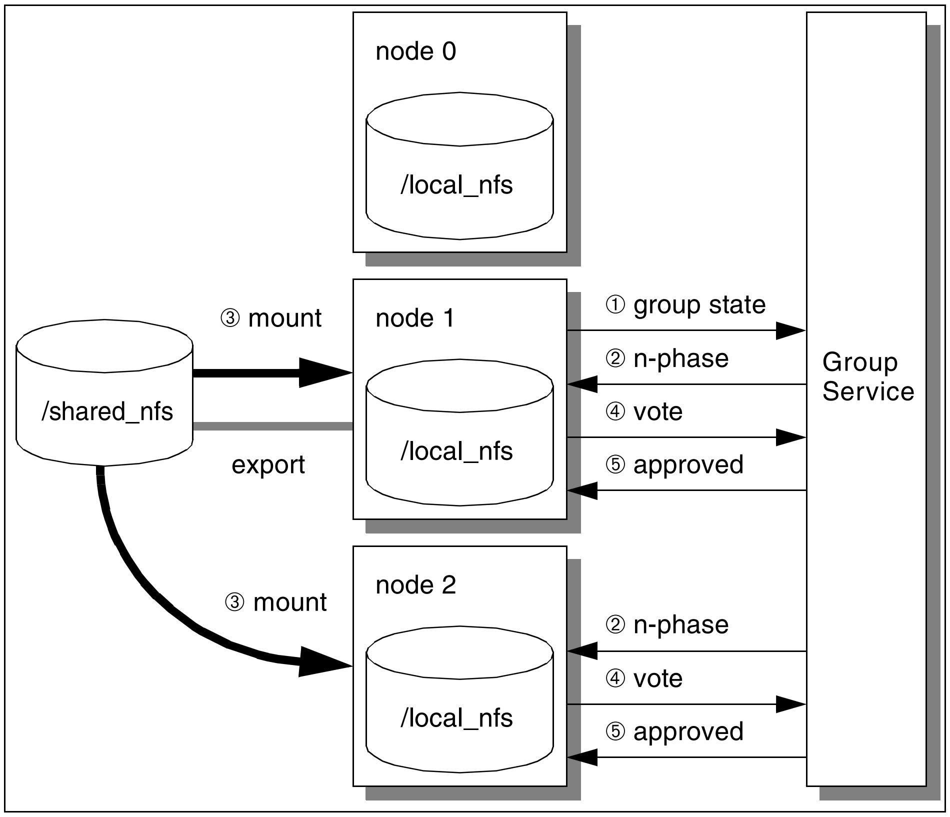


Figure 132. Registering hostname

The program state changes as shown in Table 11.

Table 11. Program state for registering hostname

Node	Before protocol	After protocol
node 1	RNFS_SERVER RNFS_UNSTABLE	RNFS_SERVER RNFS_STABLE
node 2	RNFS_CLIENT RNFS_UNSTABLE	RNFS_CLIENT RNFS_STABLE

The state value change protocol proposal is implemented as shown in Figure 133 on page 193. If any other protocol is currently running, the proposal is canceled. However, node 0’s hostname must be registered. Therefore, using the collide variable, the program retries the proposal until it is started.

```
/*************************************
* propose_state (ha_gs_chage_state_value)
*************************************/
void propose_state() {
   char hostname[256];
   int  collide;

   if(gethostname(hostname, 256)) {
      printf("*** gethostname failed ***\n");
   }

   proposal_info.gs_state_change_request.gs_num_phases         = HA_GS_N_PHASE;
   proposal_info.gs_state_change_request.gs_time_limit
      = RNFS_TAKEOVER_TIME_LIMIT;
   proposal_info.gs_state_change_request.gs_new_state.gs_length
      = strlen(hostname) + 1;
   proposal_info.gs_state_change_request.gs_new_state.gs_state = hostname;

   for(; collide;) {
      gs_rc = ha_gs_change_state_value(
         provider_token,
         &proposal_info);
      if(gs_rc != HA_GS_OK) {
         if(gs_rc == HA_GS_COLLIDE) {
            printf("* warning * hostname registration is canceled - retry\n");
            collide = 1;
         } else {
            printf("*** ha_gs_change_state_value failed rc=%d ***\n", gs_rc);
            collide = 0;
         }
      } else {
         collide = 0;
      }
   }
   return;
}
```

Figure 133. Change state value protocol proposal

The n-phase callback for a state value change protocol is implemented as shown in Figure 134 on page 194. Nodes 1 and 2 execute the rnfs_mount script using node 1’s hostname.

```
case HA_GS_STATE_VALUE_CHANGE:
     printf("[ server takeover ] mount network file system from %s\n",
        block->gs_proposal->gs_proposed_state_value->gs_state);
     strcpy(mount_command, RNFS_MOUNT);
     strcat(mount_command,
        block->gs_proposal->gs_proposed_state_value->gs_state);
     if(rc = system(mount_command)) {
        printf("\n*** system failed rc=%d ***\n", rc);
     }
     serveris = block->gs_proposal->gs_proposed_by.gs_node_number;
     vote_protocol(HA_GS_VOTE_APPROVE, NULL);
     break;
```

Figure 134. N-phase notification (HA_GS_STATE_VALUE_CHANGE)

The protocol approved notification for a state value change protocol is implemented as shown in Figure 135. Nodes 1 and 2 set the group to stable so that a new node can join the group. The n-phase protocol is used to detect the node that did not mount node 1's /local_nfs as /shared_nfs within the time limit.

```
case HA_GS_STATE_VALUE_CHANGE:
     imdoing = RNFS_STABLE;
     break;
```

Figure 135. Protocol approved notification (HA_GS_STATE_VALUE_CHANGE)

9.3.7 Client node shutdown

A client node shutdown might be required for maintenance purposes as a server node. This situation does not require involved procedures. The client node is required to umount the /shared_nfs, and that is all.

Figure 136 on page 195 illustrates control flow for a client node shutdown. Assuming that node 2 is a client node and has shut down as planned.

Node 2 proposes a voluntary leave protocol with a leaving code of RNFS_CLIENT (①). Then, nodes 0, 1, and 2 receive an n-phase notification (②). At this moment, node 2 has been removed from the membership list; therefore, it calls the ha_gs_quit subroutine and then umounts /shared_nfs (③). Nodes 0 and 1 are required to check the leaving code. In this case, it is a client node (RNFS_CLIENT). Therefore, they vote to approve immediately (④). This must be done within five seconds. This time limit is defined as follows:

```
#define RNFS_SHUTDOWN_TIME_LIMIT 5
```

Nodes 0 and 1 receive a protocol approved notification (⑤); however, no action is required for this notification.

If a node fails to vote within the time limit, Group Services sends an announcement notification to all the nodes. For this situation, refer to Section 9.3.10, “Receiving an announcement” on page 202.

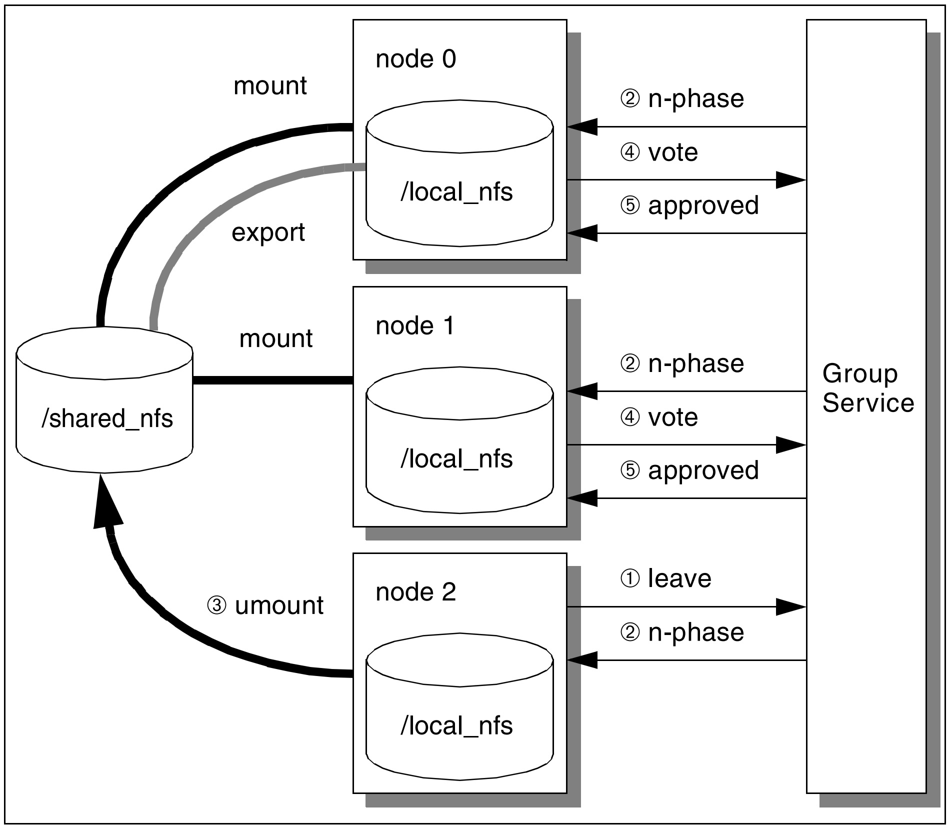


Figure 136. Client node shutdown

The program state changes are shown in Table 12.

Table 12. Program state for client node shutdown

Node	Before protocol	After protocol
node 0	RNFS_SERVER RNFS_STABLE	no change
node 1	RNFS_CLIENT RNFS_STABLE	no change

Node	Before protocol	After protocol
node 2	RNFS_CLIENT RNFS_LEAVING	does not exist

The voluntary leave protocol proposal is implemented as shown in Figure 128 on page 189. The program needs to set the leaving code to the value of the ima variable and the imdoing variable to the value of RNFS_LEAVING. If any other protocol is currently running, the proposal is canceled. It is the responsibility of the client node to propose a voluntary leave protocol again.

The n-phase notification for a voluntary leave protocol is implemented as shown in Figure 129 on page 190. In this situation, the condition

```
imdoing != RNFS_LEAVING
```

is false for node 2 and true for nodes 0 and 1.

The condition

```
block->gs_proposal->gs_leave_info->gs_leave_codes->gs_voluntary_leave_code
== RNFS_SERVER
```

is false. Therefore, nodes 0 and 1 simply vote to approve.

This is the last notification for node 2. Node 2 is not a provider of the group at this point and should not call the ha_gs_vote subroutine. Instead, it calls the quit_program subroutine to quit using GSAPIs as shown in Figure 130 on page 190.

The protocol approved notification for a voluntary leave protocol is implemented as shown in Figure 131 on page 191. Only nodes 0 and 1 receive this notification. In this situation, the condition

```
ima != RNFS_SERVER
```

is false for node 0 and true for node 1. The condition

```
mynodeis ==
block->gs_proposal->gs_current_providers->gs_providers->gs_node_number
```

is false for node 1. Therefore, no action is required for nodes 0 or 1.

9.3.8 Server node failure

The server node failure is similar to the server node shutdown described in Section 9.3.6, “Server node shutdown” on page 186. However, the node has failed and cannot umount /shared_nfs. Therefore, Group Services does this operation for a failed server node. Also, unlike the server node shutdown, it is impossible for the remaining nodes to replicate /shared_nfs to their /local_nfs.

Figure 137 on page 198 illustrates control flow for a server node failure. Assuming that node 0 is a current server node that fails by accident, node 1 will be the new server node.

When node 0 fails, Group Services notices this situation and proposes a failure leave protocol. At the same time, Group Services executes the deactivate script (refer to Section 9.4.1, “rnfs_deact.ksh shell script” on page 208) against node 0 (①) to umount /shared_nfs (②). Then, nodes 1 and 2 receive an n-phase notification with a changing providers list (③). This list contains the node number of node 0; therefore, nodes 1 and 2 know node 0 was failed. Then, they are required to umount /shared_nfs (④). When this is complete, they vote to approve (⑤). This must be done within five seconds. This time limit is defined as follows:

```
#define RNFS_JOIN_FAILURE_TIME_LIMIT 5
```

Nodes 1 and 2 receive a protocol approved notification with an updated membership list (⑥). Node 1 is listed at the top of the membership list. Therefore, node 1 becomes a new server node and is required to propose a state value change protocol to register its hostname to the group state value. This must be done immediately. For hostname registration, refer to the section entitled “Registering a hostname” on page 191.

If a node fails to vote within the time limit, Group Services sends an announcement notification to all the nodes. For this situation, refer to Section 9.3.10, “Receiving an announcement” on page 202.

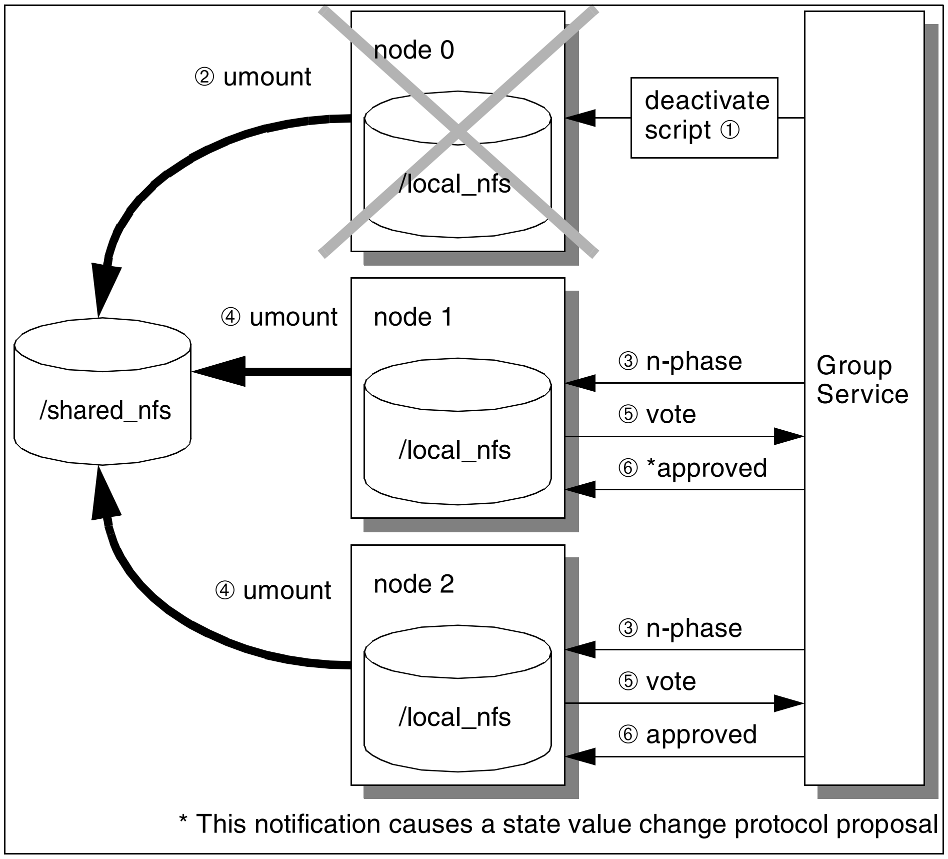


Figure 137. Server node failure

The program state changes as shown in Table 13.

Table 13. Program state for server node failure

Node	Before protocol	After protocol
node 0	RNFS_SERVER RNFS_STABLE	Does not exist
node 1	RNFS_CLIENT RNFS_STABLE	RNFS_SERVER RNFS_UNSTABLE
node 2	RNFS_CLIENT RNFS_STABLE	RNFS_CLIENT RNFS_UNSTABLE

The n-phase notification for a failure leave protocol is implemented as shown in Figure 138 on page 199. In this situation, the condition

```
ima != RNFS_SERVER
```

is true and the condition

```
serveris ==
block->gs_proposal->gs_changing_providers->gs_providers->gs_node_number
```

is also true. Therefore, nodes 1 and 2 realize that node 0 was failed and execute the rnfs_umount script to umount /shared_nfs. They must set the group to unstable to reject the upcoming join protocols (if there will be any) because a joining node could pick up node 0 as a server node instead of node 1. Then, they vote to approve.

```
case HA_GS_FAILURE_LEAVE:
     if(ima != RNFS_SERVER) {
        if(serveris ==
           block->gs_proposal->gs_changing_providers->gs_providers->gs_node_number) {
           printf("[ server failure ] umount network file system\n");
           if(rc = system(RNFS_UMOUNT)) {
              printf("\n*** system failed rc=%d ***\n", rc);
           }
           imdoing = RNFS_UNSTABLE;
        }
     }
     vote_protocol(HA_GS_VOTE_APPROVE, NULL);
     break;
```

Figure 138. N-phase notification (HA_GS_FAILURE_LEAVE)

The protocol approved notification for a failure leave protocol is implemented as shown in Figure 139 on page 200. In this situation, the condition

```
ima != RNFS_SERVER
```

is true and the condition

```
mynodeis ==
block->gs_proposal->gs_current_providers->gs_providers->gs_node_number
```

is true for node 1 and false for node 2. Therefore, node 1 starts registering its hostname.

```
case HA_GS_FAILURE_LEAVE:
     if(ima != RNFS_SERVER) {
        if(mynodeis ==
           block->gs_proposal->gs_current_providers->gs_providers->gs_node_number) {
           ima = RNFS_SERVER;
           propose_state();
        }
     }
     break;
```

Figure 139. Protocol approved notification (HA_GS_FAILURE_LEAVE)

9.3.9 Client node failure

The client node failure is similar to the client node shutdown described in Section 9.3.7, “Client node shutdown” on page 194. However, the node has failed and cannot perform umount /shared_nfs. Therefore, Group Services does this operation for a failed client node.

Figure 140 on page 201 illustrates control flow for a client node failure, assuming that node 2 is a client node and fails by accident.

When node 2 fails, Group Services notices this situation and proposes a failure leave protocol. At the same time, Group Services executes the deactivate script (refer to Section 9.4.1, “rnfs_deact.ksh shell script” on page 208) against node 2 (①) to umount /shared_nfs (②). Then, nodes 0 and 1 receive an n-phase notification with a changing provider list (③). This list contains the node number of node 2, and nodes 0 and 1 know that the failure node is not node 0. Therefore, they vote to approve immediately (④). This must be done within five seconds. This time limit is defined as follows:

```
#define RNFS_JOIN_FAILURE_TIME_LIMIT 5
```

Nodes 0 and 1 receive a protocol approved notification (⑤); however, no action is required for this.

If a node fails to vote within the time limit, Group Services sends an announcement notification to all the nodes. For this situation, refer to Section 9.3.10, “Receiving an announcement” on page 202.

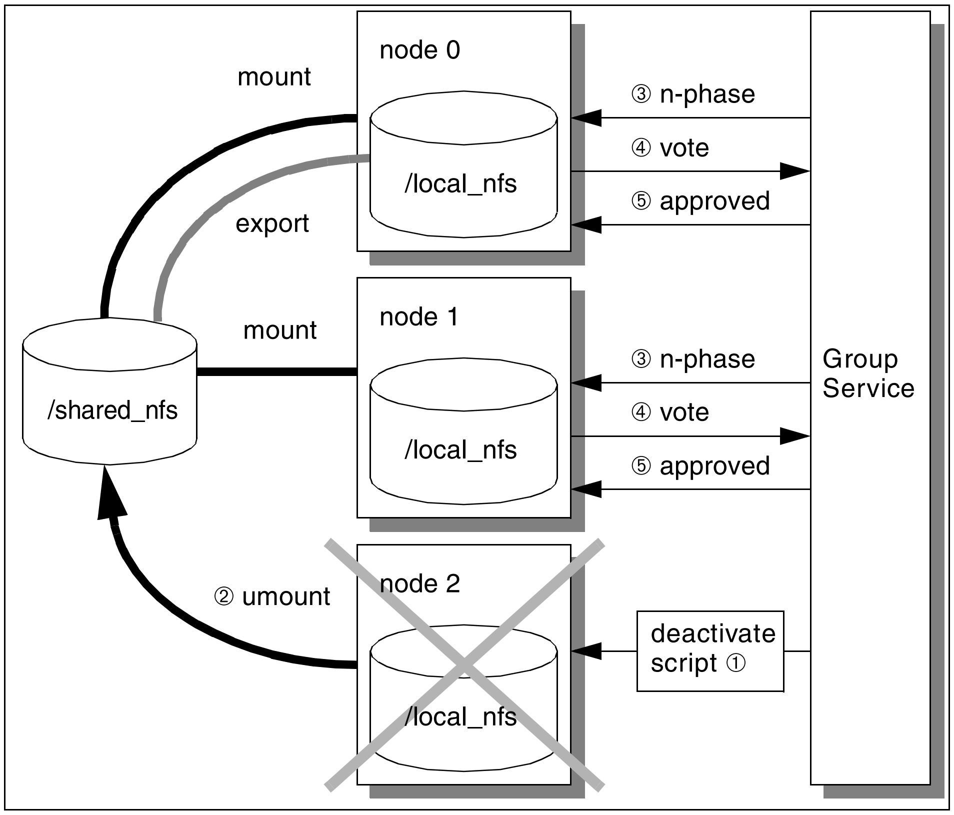


Figure 140. Client node failure

The program state changes as shown in Table 14.

Table 14. Program state for having NFS client failure

Node	Before protocol	After protocol
node 0	RNFS_SERVER RNFS_STABLE	no change
node 1	RNFS_CLIENT RNFS_STABLE	no change
node 2	RNFS_CLIENT RNFS_STABLE	does not exist

The n-phase notification for a failure leave protocol is implemented as shown in Figure 138 on page 199. In this situation, the condition

```
ima != RNFS_SERVER
```

is false for node 0 and true for node 1. The condition

```
serveris ==
block->gs_proposal->gs_changing_providers->gs_providers->gs_node_number
```

is false. Therefore, nodes 0 and 1 simply vote to approve.

The protocol approved notification for a failure leave protocol is implemented as shown in Figure 139 on page 200. In this situation, the condition

```
ima != RNFS_SERVER
```

is false for node 0 and true for node 1. The condition

```
mynodeis ==
block->gs_proposal->gs_current_providers->gs_providers->gs_node_number
```

is true for node 0 and false for node 1. Therefore, no action is required for nodes 0 and 1.

9.3.10 Receiving an announcement

A node receives an announcement notification in the following three situations:

- When a node in the group has lost its responsiveness
- When a node in the group that lost its responsiveness has recovered
- When a node in the group has failed to vote within the time limit
- When the last node leaves the group
- When a node has disconnected unexpectedly from the Group Services, or the Group Services daemon has died

In the first case, the program displays a warning message similar to the following:

```
* warning * responsiveness check has failed on node 0 node 1
```

In the second case, the program displays a warning message similar to the following:

```
* warning * responsiveness check has recovered on node 0 node 1
```

These situations are described in Section 9.3.2, “Checking responsiveness” on page 174.

In the third case, the program is required to check to which protocol this announcement notification has been sent. Unfortunately, there is no information in the announcement notification block about this. Therefore, the program needs to track the protocol it has processed. To do this, the program assigns the latest protocol to the latest_protocol variable each time it receives an n-phase notification as follows:

```
switch(latest_protocol = block->gs_protocol_type) {
```

The program translates the protocols and displays warning messages similar to the following:

Join protocol (HA_GS_JOIN)

```
* warning * it might be very busy on node 0
```

This situation is described in Section 9.3.3, “Creating the group” on page 175 and Section 9.3.4, “Adding a node” on page 180.

Failure leave protocol (HA_GS_FAILURE_LEAVE)

```
* warning * umounting file system might have failed on node 0
```

This situation is described in Section 9.3.8, “Server node failure” on page 196.

Voluntary leave protocol (HA_GS_LEAVE)

```
* warning * replicating/umounting file system might have failed
on node 0
```

This situation is described in Section 9.3.6, “Server node shutdown” on page 186.

State value change protocol (HA_GS_STATE_VALUE_CHANGE)

```
* warning * mounting file system might have failed on node 0
```

This situation is described in “Registering a hostname” on page 191.

Provider-broadcast message protocol (HA_GS_PROVIDER_MESSAGE)

```
* warning * replicating file system might have failed on node 0
```

This situation is described in Section 9.3.5, “Replicating a file system” on page 183.

Figure 141 on page 204 illustrates control flow for receiving an announcement notification.

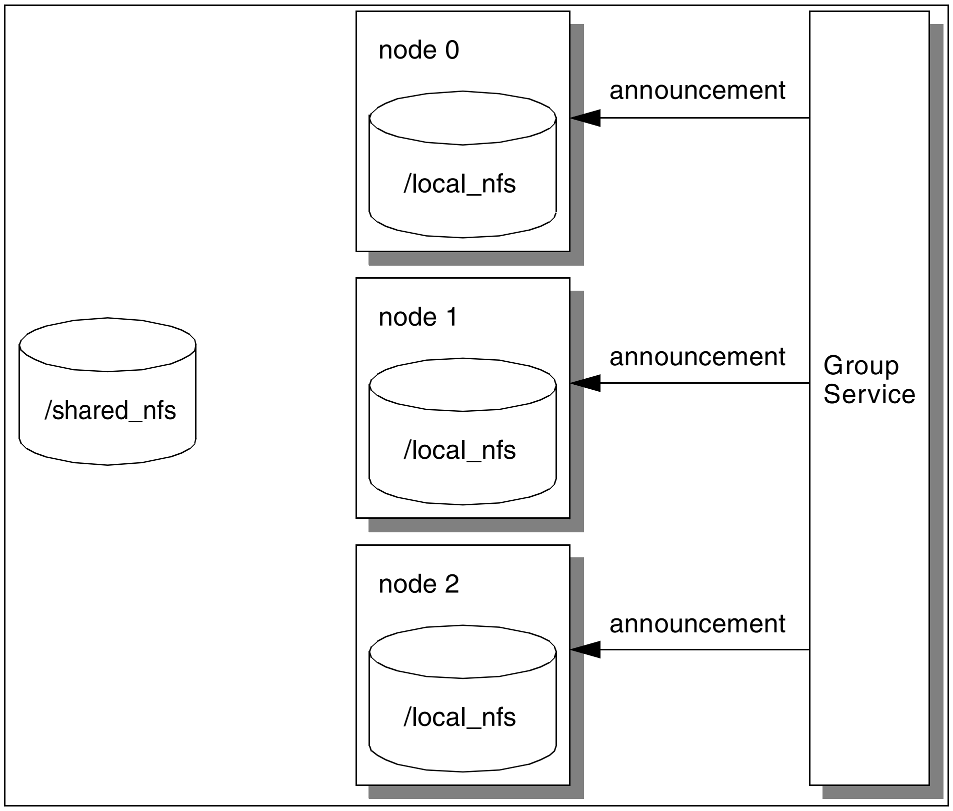


Figure 141. Receiving an announcement

The announcement notification is implemented as shown in Figure 142 on page 205.

```
/**************************************
* announcement notification
**************************************/
void ha_gs_announcement_callback(
   const ha_gs_announcement_notification_t *block) {

   switch(block->gs_summary_code) {
   case HA_GS_RESPONSIVENESS_NO_RESPONSE:
      printf("* warning * responsiveness check has failed on node ");
      if(imdoing != RNFS_LEAVING) {
         imdoing = RNFS_UNSTABLE;
      }
      break;
   case HA_GS_RESPONSIVENESS_RESPONSE:
      printf("* warning * responsiveness check has recovered on node ");
      if(imdoing != RNFS_LEAVING) {
         imdoing = RNFS_STABLE;
      }
      break;
   case HA_GS_TIME_LIMIT_EXCEEDED:
      switch(latest_protocol) {
      case HA_GS_JOIN:
         printf("* warning * it might be very busy on node ");
         break;
      case HA_GS_FAILURE_LEAVE:
         printf("* warning * umounting file system might have failed on node ");
         break;
      case HA_GS_LEAVE:
         printf("* warning * replicating/umounting file system might have failed on node ");
         break;
      case HA_GS_STATE_VALUE_CHANGE:
         printf("* warning * mounting file system might have failed on node ");
         break;
      case HA_GS_PROVIDER_MESSAGE:
         printf("* warning * replicating file system might have failed on node ");
         break;
      default:
         printf("*** announcement notification is not expected ***\n");
         return;
      }
      break;
   case HA_GS_GROUP_SERVICES_HAS_DIED_HORRIBLY:
      printf("*** Group Services has died ***\n");
      break;
   case HA_GS_GROUP_DISSOLVED:
      printf("*** rnfs_group has dissolved ***\n");
      break;
   default:
      printf("*** announcement notification is not expected ***\n");
      return;
   }
   print_nodes(block->gs_announcement);
   return;
}
```

Figure 142. Announcement notification

The print_nodes subroutine shown in Figure 143 on page 206 prints out the node numbers in the gs_announcement field in the announcement notification block.

```
/*************************************
* print_nodes
*************************************/
void print_nodes(ha_gs_membership_t *membership_list) {
   int              number_of_nodes;
   ha_gs_provider_t *member;

   member = membership_list->gs_providers;
   for(number_of_nodes = 0;
      number_of_nodes < membership_list->gs_count;
      number_of_nodes++, member++) {
      printf("%d ", member->gs_node_number);
   }
   printf("\n");
   return;
}
```

Figure 143. Print nodes subroutine

9.3.11 Receiving error

A node receives a delayed error notification in the following three situations:

- When multiple rnfs program instances are executed on a node
- When a protocol has collided

In the first case, the program displays a warning message similar to the following and exits:

```
* warning * another rnfs program is running on this node - exit
```

This situation is described in Section 9.3.3, “Creating the group” on page 175, and Section 9.3.4, “Adding a node” on page 180.

In the second case, the program is required to check to which protocol this delayed error notification has been sent. The delayed error notification block has this information.

The program translates the protocols and displays warning messages similar to the following:

Voluntary leave protocol (HA_GS_LEAVE)

```
* warning * leaving the group is canceled
```

This situation is described in Section 9.3.6, “Server node shutdown” on page 186, and Section 9.3.7, “Client node shutdown” on page 194.

State value change protocol (HA_GS_STATE_VALUE_CHANGE)

`* warning * hostname registration is canceled - retry`

This situation is described in the section entitled “Registering a hostname” on page 191.

Provider-broadcast message protocol (HA_GS_PROVIDER_MESSAGE)

`* warning * replication is canceled`

This situation is described in Section 9.3.5, “Replicating a file system” on page 183.

The delayed error notification is implemented as shown in Figure 144.

```
/**************************************
* delayed error notification
**************************************/
void ha_gs_delayed_error_callback(
   const ha_gs_delayed_error_notification_t *block) {

   switch(block->gs_protocol_type) {
   case HA_GS_JOIN:
      if(block->gs_delayed_return_code == HA_GS_DUPLICATE_INSTANCE_NUMBER) {
         printf("* warning * another rnfs is running on this node - exit\n");
         exit(-1);
      }
      break;
   case HA_GS_LEAVE:
      if(block->gs_delayed_return_code == HA_GS_COLLIDE) {
         printf("* warning * leaving the group is canceled\n");
      }
      break;
   case HA_GS_STATE_VALUE_CHANGE:
      if(block->gs_delayed_return_code == HA_GS_COLLIDE) {
         printf("* warning * hostname registration is canceled - retry\n");
         propose_state();
      }
      break;
   case HA_GS_PROVIDER_MESSAGE:
      if(block->gs_delayed_return_code == HA_GS_COLLIDE) {
         printf("* warning * replication is canceled\n");
      }
      break;
   default:
      printf("*** delayed error notification is not expected ***\n");
      break;
   }
   return;
}
```

Figure 144. Delayed error notification

9.4 Shell script, Perl script, and log file

If you do not have an environment to modify and recompile the rnfs program, the program provides you with flexibility by using shell scripts. The following operations are done by the shell scripts:

- Deactivate-on-failure facility (a deactivate script)
- mount /shared_nfs file system
- replicate /shared_nfs file system to /local_nfs file system
- umount /shared_nfs file system

Instead of executing actual commands, these shell scripts write their activities to the log file, rnfs.log. You can modify theses shell scripts to execute actual commands or to do something else.

Entire shell scripts are provided in Appendix C, “Recoverable Network File System programs” on page 261.

9.4.1 rnfs_deact.ksh shell script

The rnfs_deact.ksh shell script is a deactivate script called by Group Services when a failure leave protocol is proposed.

The actual deactivate script is written by the Perl script. Group Services only allows the use of a shell script or an executable program for a deactivate script. Since a Perl script is neither of these, the shell script is required as a wrapper to execute the Perl script. Therefore, two scripts are used for a deactivate script: One for a wrapper written as a shell script and one for a deactivate script written as a Perl script.

The name of the shell script is defined in the rnfs program as:

```
#define RNFS_DEACTIVATE    "./rnfs_deact.ksh"
```

The shell script is implemented as shown in Figure 145.

```
./rnfs_deact.perl $1 $2 $3 $4 $5

exit $?
```

Figure 145. The rnfs_deact.ksh shell script

It forwards the five parameters given by Group Services to the rnfs_deact.perl Perl script. It also forwards the return value from the Perl script to the Korn shell. This value is checked and used by Group Services for some cases.

9.4.2 rnfs_deact.perl Perl script

The rnfs_deact.perl Perl script is the real deactivate script and is called by the rnfs_deact.ksh shell script. The Perl script opens the rnfs log file (rnfs.log) and writes the date and five parameters: Process ID, voting time limit, failed group, deactivate flag, and failed provider. Then, it writes the operation (umount /shared_nfs) and closes the log file. Finally, it returns to the deact.ksh shell script. A successfully executed deactivate script must give a return value of 0.

This Perl script is implemented as shown in Figure 146.

```
$FILE_SYSTEM = "/shared_nfs";
$LOG_FILE    = "./rnfs.log";

if($#ARGV == 4) {
   open(STDOUT, ">> $LOG_FILE");
   `date >> $LOG_FILE`;
   print "   deactivate script executed by the Group Services\n";
   print "      Process ID:         $ARGV[0]\n";
   print "      Voting Time Limit:  $ARGV[1]\n";
   print "      Failed group:       $ARGV[2]\n";
   print "      Deactivate flag:    $ARGV[3]\n";
   print "      Failed provider(s): $ARGV[4]\n";
   print "   umount $FILE_SYSTEM\n";
   close STDOUT;
} else {
   print "*** the number of parameters is not expected ***\n";
   exit -1;
}

# exit with  return code 0
exit 0;
```

Figure 146. The rnfs_deact.perl Perl script

9.4.3 rnfs_mount shell script

The rnfs_mount shell script is used to mount a server node's /local_nfs local file system as /shared_nfs network file system. Instead of executing an actual command, it writes the operation (the mount command) to the log file (rnfs.log) with a time stamp.

The name of the shell script is defined in the rnfs program as:

```
#define RNFS_MOUNT        "./rnfs_mount "
```

The shell script is implemented as shown in Figure 147 on page 210.

```
date >> ./rnfs.log
print "   mount $1:/local_nfs /shared_nfs" >> ./rnfs.log
```

Figure 147. The rnfs_mount shell script

9.4.4 rnfs_replicate

The rnfs_replicate shell script is used to replicate /shared_nfs network file system to /local_nfs local file system. Instead of executing an actual command, it writes the operation (the cp command) to the log file (rnfs.log) with a time stamp.

The name of the shell script is defined in the rnfs program as:

```
#define RNFS_REPLICATE    "./rnfs_replicate"
```

The script is implemented as shown in Figure 148 on page 210.

```
date >> ./rnfs.log
print "   cp -R /shared_nfs/. /local_nfs" >> ./rnfs.log
```

Figure 148. The rnfs_mount shell script

9.4.5 rnfs_umount

The rnfs_umount shell script is used to umount the /shared_nfs network file system. In stead of executing an actual command, it writes the operation (the umount command) to the log file (rnfs.log) with time stamp.

The name of the shell script is defined in the rnfs program as:

```
#define RNFS_UMOUNT       "./rnfs_umount"
```

The shell scrip is implemented as shown in Figure 149 on page 210.

```
date >> ./rnfs.log
print "   umount /shared_nfs" >> ./rnfs.log
```

Figure 149. The rnfs_mount shell script

9.5 rnfsm program

The rnfsm program monitors the status of the RNFS environment; if a server node shutdown or failure occurs, it displays a new server node on the screen.

If client node shutdown or failure has happened, it displays all the available nodes in the group on the screen.

The rnfs program is a provider's program, and the rnfsm program is a subscriber's program. Therefore, the rnfsm program does not have any protocol-related subroutines. Instead, it has subscriber-related subroutines.

This section describes the rnfsm program in detail. The entire program is provided in Section C.7, "rnfsm.c" on page 273.

9.5.1 main routine

The main routine of the rnfsm program is shown in Figure 150 on page 212. In this routine, the program performs the following operations:

1. Set the domain name and group name. The domain name is given as a parameter of the program. The group name is defined as:

   ```
   #define RNFS_GROUP_NAME   "rnfs_group"
   ```

   Unlike the rnfs program, an instance number is not required. This means that you can execute the rnfsm program on a node as many times as you want.
2. Set some global variables.
3. It calls the init_program subroutine to initialize the program with the Group Services.
4. It calls the propose_subscribe subroutine to subscribe the group.

In an infinite loop, the program performs the following operations:

1. Set two file descriptors for the select subroutine, one for stdin and the other for the Group Services. Then, the program calls the select subroutine. It waits one second, at most, and then returns.
2. If there is data from stdin, the program calls the suspend_program subroutine to suspend the program and get your command. This is the case described in Section 9.5.3, "Unsubscribing the group" on page 217.
3. If there is data from the Group Services, the program calls the ha_gs_dispatch subroutine to receive notifications.

```
/*************************************
* main
*************************************/
int main(int argc, char **argv) {
   char           key;
   fd_set         my_fd;
   struct timeval timeout;

   if(argc != 2) {
      printf("Usage: %s domain_name\n", argv[0]);
      exit(argc);
   }
   strcpy(domain_name, "HA_DOMAIN_NAME=");
   strcat(domain_name, argv[1]);
   putenv(domain_name);
   printf("domain name: %s, ", getenv("HA_DOMAIN_NAME"));
   printf("group name: %s\n", RNFS_GROUP_NAME);

   descriptor = 0;
   timeout.tv_sec  = 1;
   timeout.tv_usec = 0;
   init_program();
   propose_subscribe();

   printf("hit <Enter> key to suspend\n");
   for(;;) {
      FD_ZERO(&my_fd);
      FD_SET(0, &my_fd);
      FD_SET(descriptor, &my_fd);
      rc = select(descriptor + 1, &my_fd, NULL, NULL, &timeout);
      if(rc < 0) {
         printf("*** select failed rc=%d ***\n", rc);
         exit(rc);
      }
      if(FD_ISSET(0, &my_fd)) {
         suspend_program();
      }
      if(descriptor && FD_ISSET(descriptor, &my_fd)) {
         gs_rc = ha_gs_dispatch(HA_GS_NON_BLOCKING);
         if(gs_rc != HA_GS_OK) {
            printf("*** ha_gs_dispatch failed rc=%d ***\n", gs_rc);
         }
      }
   }
}
```

Figure 150. Main routine

The initialization subroutine is shown in Figure 151 on page 213. In this subroutine, the program performs the following operations:

1. Set the responsiveness control block as follows:
    - The program uses no responsiveness check.

This does not mean you cannot use the responsiveness check for a subscriber. However, Group Services does not take any action when it notices that a subscriber has lost its responsiveness. Therefore, an application is required to provide its own mechanism in the responsiveness callback subroutine, if necessary.

2. Set the address for the following callback subroutine:

   `ha_gs_delayed_error_callback`

```
/**************************************
* init_program (ha_gs_init)
**************************************/
void init_program() {

   /* responsiveness control block */
   responsiveness.gs_responsiveness_type     = HA_GS_NO_RESPONSIVENESS;
   responsiveness.gs_responsiveness_interval = NULL;
   responsiveness.gs_responsiveness_response_time_limit = NULL;
   responsiveness.gs_counter_location        = NULL;
   responsiveness.gs_counter_length          = NULL;

   gs_rc = ha_gs_init(
      &descriptor,
      HA_GS_SOCKET_NO_SIGNAL,
      &responsiveness,
      NULL,
      NULL,
      ha_gs_delayed_error_callback,
      NULL);
   if(gs_rc != HA_GS_OK) {
      printf("*** ha_gs_init failed rc=%d ***\n", gs_rc);
   }
   return;
}
```

Figure 151. Initialization subroutine

9.5.2 Subscribing the group

To subscribe the group, the program calls the subscription subroutine shown in Figure 152 on page 214. In this subroutine, the program performs the following operations:

1. Set the subscribe request block as follows:
   - Subscribe both the group state value change and the membership list change.
   - Subscribe the group defined as:

     `#define RNFS_GROUP_NAME "rnfs_group"`

   - Set the address for the following callback subroutine:

```
ha_gs_subscriber_callback
```

2. It calls the ha_gs_subscribe subroutine to subscribe the group.

```
/**************************************
* propose_subscribe (ha_gs_subacribe)
**************************************/
void propose_subscribe() {

   proposal_info.gs_subscribe_request.gs_subscription_control
      = HA_GS_SUBSCRIBE_STATE_AND_MEMBERSHIP;
   proposal_info.gs_subscribe_request.gs_subscription_group
      = RNFS_GROUP_NAME;
   proposal_info.gs_subscribe_request.gs_subscription_callback
      = ha_gs_subscriber_callback;

   gs_rc = ha_gs_subscribe(
      &subscriber_token,
      &proposal_info);
   if(gs_rc != HA_GS_OK) {
      printf("*** ha_gs_subscribe failed rc=%d **\n", gs_rc);
   }
   return;
}
```

Figure 152. Subscription subroutine

If the rnfsm program is started before the rnfs program has created the RNFS environment, the rnfsm program will receive a delayed error notification as shown in Figure 153 on page 215. The notification indicates that the group being subscribed is unknown.

If this is the case, the program calls the quit_program subroutine shown in Figure 154 on page 215 to terminate the program with a message similar to the following:

```
$ rnfsm sp6en0
domain name: sp6en0, group name: rnfs_group
hit <Enter> key to suspend
*** no RNFS node is available currently ***
$
```

```
/**************************************
* delayed error notification
**************************************/
void ha_gs_delayed_error_callback(
   const ha_gs_delayed_error_notification_t *block) {

   switch(block->gs_protocol_type) {
   case HA_GS_SUBSCRIPTION:
      if(block->gs_delayed_return_code == HA_GS_UNKNOWN_GROUP) {
         printf("*** no RNFS node is available currently ***\n");
         quit_program();
      }
      break;
   default:
      printf("*** delayed error notification is not expected ***\n");
      break;
   }
   return;
}
```

Figure 153. Delayed error notification

```
/**************************************
* quit_program (ha_gs_quit)
**************************************/
void quit_program() {

   gs_rc = ha_gs_quit();
   if (gs_rc != HA_GS_OK) {
      printf("*** ha_gs_quit failed rc=%d ***\n", gs_rc);
   } else {
      exit(0);
   }
   return;
}
```

Figure 154. Quit subroutine

If the rnfsm program has successfully subscribed the group, the program receives the subscriber notification that contains the current state of the group. The subscriber notification is shown in Figure 155 on page 216.

There are three situations in which the program receives a subscriber notification:

- When the RNFS environment has been dissolved.
  In other words, when all the providers (rnfs program) have left the group, the program displays a message similar to the following:

  ```
  *** no RNFS node is available any more ***
  ```

- When the server node has been changed
  In other words, when the group state has been changed, the program displays a message similar to the following:

  ```
  server node hostname: sp6n01
  ```

- When the client node has been added or deleted
  In other words, when the membership list has been changed, the program displays a message similar to the following:

  ```
  client nodes: 1 3 5
  ```

```
/**************************************
* subscriber notification
**************************************/
void ha_gs_subscriber_callback(
   const ha_gs_subscription_notification_t *block) {

   printf("\n");
   if(block->gs_subscription_type & HA_GS_SUBSCRIPTION_DISSOLVED) {
      printf("*** no RNFS node is available any more ***\n");
      quit_program();
   } else {
      if(block->gs_subscription_type & HA_GS_SUBSCRIPTION_STATE) {
         printf("server node hostname: %s\n", block->gs_state_value->gs_state);
      }
      if(block->gs_subscription_type & HA_GS_SUBSCRIPTION_MEMBERSHIP) {
         printf("client nodes: ");
         print_members(block->gs_full_membership);
      }
   }
   return;
}
```

Figure 155. Subscriber notification

The print_members subroutine is shown in Figure 156 on page 217.

```
/***************************************
* print_members
***************************************/
void print_members(ha_gs_membership_t *membership_list) {
   int               i;
   ha_gs_provider_t *member;

   if(membership_list->gs_count) {
      member = membership_list->gs_providers;
      for(i = 0; i < membership_list->gs_count; i++, member++){
         printf("%d ", member->gs_node_number);
      }
   }
   printf("\n");
   return;
}
```

Figure 156. Print members subroutine

9.5.3 Unsubscribing the group

If you want to stop monitoring the RNFS environment, that is, if the rnfsm program wants to unsubscribe the group, you need to press the **Enter** key. If you press the **Enter** key, the program calls the suspension subroutine shown in Figure 157 on page 218 to suspend and await your command. You can choose one of the following operations:

- Press the **r** or **R** key to resume the program.
- Press the **u** or **U** key to unsubscribe the group.

The resume operation simply returns to the main routine. The unsubscribe operation calls the propose_unscribe subroutine shown in Figure 158 on page 218 to unsubscribe the group.

As you may have noticed, you can press any keys other than u or U to resume operation.

```
/**************************************
* suspend_program
**************************************/
void suspend_program() {
   char proposal[32];

   gets(proposal); /* remove previously input strings */
   printf("[ program suspended ] u(nsubscribe) or r(esume)?: ");
   scanf("%s", proposal);
   switch((int)proposal[0]) {
   case 'u': case 'U':
      propose_unsubscribe();
      break;
   default:
      break;
   }
   gets(proposal); /* remove extra strings */
   return;
}
```

Figure 157. Suspension subroutine

```
/**************************************
* propose_unsubscribe (ha_gs_unsubscribe)
**************************************/
void propose_unsubscribe() {

   gs_rc = ha_gs_unsubscribe(
      subscriber_token);
   if(gs_rc != HA_GS_OK) {
      printf("*** ha_gs_unsubscribe failed rc=%d ***\n", gs_rc);
   } else {
      quit_program();
   }
   return;
}
```

Figure 158. Unsubscription subroutine

9.6 Operation example

This section provides you with an operation example of the rnfs and rnfsm programs. This example uses the following environment:

- The domain name is sp6en0.
- There are three nodes in the domain. Their node numbers are 1, 3, and 5. Their hostnames are sp6n01, sp6n03, and sp6n05.

The following is the operation scenario. The sequential numbers, 1 through 15, are referred as event numbers in Table 15 on page 222 and Figure 159 on page 224 through Figure 165 on page 228:

1. Node 1 (sp6n01) joins the group.

   To do this, execute rnfs sp6en0 on node 1. Node 1 joins the group as a server node and displays the following message:

   ```
   [ joined as server ] mount network file system from sp6n01
   ```

2. Start RNFS monitor.

   To do this, execute rnfsm sp6en0 on any node in the domain. The rnfsm program displays the following message and starts monitoring:

   ```
   server node hostname: sp6n01
   client nodes: 1
   ```

3. Node 3 (sp6n03) joins the group.

   To do this, execute “rnfs sp6en0” on node 3. Node 3 joins the group as a client node and displays the following message:

   ```
   [ joined as client ] mount network file system from sp6n01
   ```

   Because node 3 is a client node, it replicates the file system every 10 seconds and displays the following message:

   ```
   [ replicate ] replicate file system
   ```

4. Node 5 (sp6n05) joins the group.

   To do this, execute “rnfs sp6en0” on node 5. Node 5 joins the group as a client node and displays the following message:

   ```
   [ joined as client ] mount network file system from sp6n01
   ```

   Because node 5 is a client node, it replicates the file system every 10 seconds and displays the following message:

   ```
   [ replicate ] replicate file system
   ```

5. Node 5 becomes busy and stops processing any notification.

   To do this, press **Enter** and wait with the following message:

   ```
   [ program suspended ] l(eave) or r(esume)?:
   ```

   Because node 5 does not replicate the file system, nodes 1 and 3 are notified of this situation and display the following message:

   ```
   * warning * replicating file system might have failed on node 5
   ```

   Because node 5 does not reply to a responsiveness notification either, nodes 1 and 3 are notified of this situation and display the following message:

```
* warning * responsiveness check has failed on node 5
```

6. Node 5 becomes normal and starts processing notifications.

   To do this, type `r` or `R`, and then press the **Enter** key as follows:

   ```
   [ program suspended ] l(eave) or r(esume)?: r
   ```

   Because node 5 starts replying to a responsiveness notification, nodes 1 and 3 are notified of this situation and display the following message:

   ```
   * warning * responsiveness check has recovered on node 5
   ```

   Node 5 processes all the suspended notifications and displays the following messages:

   ```
   [ replicate ] replicate file system
   * warning * replicating file system might have failed on node 5
   * warning * responsiveness check has failed on node 5
   [ replicate ] replicate file system
   * warning * responsiveness check has recovered on node 5
   ```

7. Node 1 leaves the group.

   To do this, press the **Enter** key, type `l` or `L`, and then press the **Enter** key again as follows:

   ```
   [ program suspended ] l(eave) or r(esume)?: l
   ```

   Node 1 umounts the file system before the rnfs program exits and displays the following message:

   ```
   [ server takeover ] umount network file system
   ```

   Because node 1 is a server node, a server node takeover occurs. Node 3 is the node that joined the group next to node 1; therefore, it becomes a new server node.

   Nodes 3 and 5 replicate the file system and display the following message:

   ```
   [ server shutdown ] replicate file system
   ```

   Then, they umount the file system and display the following message:

   ```
   [ server shutdown ] umount network file system
   ```

   Finally, they mount the network file system from node 3 and display the following message:

   ```
   [ server takeover ] mount network file system from sp6n03
   ```

8. Node 3 becomes busy and stops processing any notification.

   To do this, press the **Enter** key, and wait with the following message:

   ```
   [ program suspended ] l(eave) or r(esume)?:
   ```

Because node 3 does not reply to a responsiveness notification, node 5 is notified of this situation and displays the following message:

```
* warning * responsiveness check has failed on node 3
```

9. Node 1 tries to join the group.

   To do this, execute rnfs sp6en0. This execution fails because node 3 is currently not responding, and it has made the group unstable. Node 1 exits and displays the following message:

   ```
   * warning * the group is unstable, join later - exit
   ```

   Node 5 explicitly voted to reject on the join protocol proposal. However, Group Services voted to approve (the default vote value) as node 3's vote. Even though the join protocol has been completed, node 3 did not vote within the voting time limit. Node 5 is notified of this situation and displays the following message:

   ```
   * warning * it might be very busy on node 3
   ```

10. Node 3 finally fails.

    To do this, press **Ctrl-C** to terminate the program as follows:

    ```
    [ program suspended ] l(eave) or r(esume)?:^C$
    ```

    This is a node failure; therefore, node 3 cannot umount the file system by itself. Group Services proposes a failure leave protocol and executes the deactivate script on node 3 to umount the file system. This is recorded in the rnfs.log file.

    Node 5 umounts the file system without the file system replication and displays the following message:

    ```
    [ server failure ] umount network file system
    ```

    Node 5 becomes a new server node. It mounts its own file system and displays the following messages:

    ```
    [ server takeover ] mount network file system from sp6n05
    ```

11. Node 1 joins the group.

    To do this, execute `rnfs sp6en0`.

    Node 1 joins the group and displays the following message:

    ```
    [ joined as client ] mount network file system from sp6n05
    ```

12. Node 5 suddenly fails.

    To do this, press **Ctrl-C** and terminate the program on node 5.

    This is a node failure; therefore, node 5 cannot umount the file system by itself. Group Services proposes a failure leave protocol and executes the deactivate script on node 5 to umount the file system. This is recorded in the rnfs.log file.

Node 1 umounts the file system and displays the following message:

```
[ server failure ] umount network file system
```

13. Node 1 tries to leave the group.

    To do this, press the **Enter** key, type `l` or `L`, and press the **Enter** key again as follows:

    ```
    [ program suspended ] l(eave) or r(esume)?: l
    ```

    A server node takeover is currently occurring, that is, a state value change protocol is running. Therefore, a voluntary leave protocol collides with this protocol. The voluntary leave protocol is canceled and displays the following message:

    ```
    * warning * leaving the group is canceled
    ```

14. Node 1 leaves the group.

    To do this, press the **Enter** key, type `l` or `L` , and press the **Enter** key again.

    The group becomes stable when a server node takeover has completed with the following message:

    ```
    [ server takeover ] mount network file system from sp6n01
    ```

    Node 1 umounts the file system and exits with the following message:

    ```
    [ server takeover ] umount network file system
    ```

15. Because the group has been dissolved, the rnfsm program is notified of this situation and exits with the following message:

    ```
    *** no RNFS node is available any more ***
    ```

9.6.1 Events summary

Table 15 summarizes the event number and the event on each node. (S) indicates a server node and (C) indicates a client node. If nothing is indicated, a node is not in the group when an event occurs.

Table 15. Event number and the event on nodes

Event number	rnfs on node 1	rnfs on node 3	rnfs on node 5	rnfsm on any node
1	(S) join			
2	(S)			start
3	(S)	(C) join		

Event number	rnfs on node 1	rnfs on node 3	rnfs on node 5	rnfsm on any node
4	(S)	(C)	(C) join	
5	(S)	(C)	(C) suspend	
6	(S)	(C)	(C) resume	
7	(S) leave	(C)	(C)	
8		(S) suspend	(C)	
9	join (reject)	(S)	(C)	
10		(S) node failure	(C)	
11	(C) join		(S)	
12	(C)		(S) node failure	
13	(S) leave (reject)			
14	(S)leave			
15				exit

9.6.2 rnfs execution output

Figure 159 on page 224 through Figure 161 on page 225 show executions of the rnfs program on nodes 1, 3, and 5. Arrows with a number indicate the point at which the numbered event has occurred.

```
$ rnfs sp6en0 <----------------------------------------------------(1)
domain name: sp6en0, group name: rnfs_group, instance number: 5523
hit <Enter> key to suspend
[ joined as server ] mount network file system from sp6n01
* warning * replicating file system might have failed on node 5 <--(5)
* warning * responsiveness check has failed on node 5
* warning * responsiveness check has recovered on node 5 <---------(6)

[ program suspended ] l(eave) or r(esume)?: l <--------------------(7)
* warning * responsiveness check has failed on node 1
[ server takeover ] umount network file system
$ rnfs sp6en0 <----------------------------------------------------(9)
domain name: sp6en0, group name: rnfs_group, instance number: 5523
hit <Enter> key to suspend
* warning * the group is unstable, join later - exit
$ rnfs sp6en0 <----------------------------------------------------(11)
domain name: sp6en0, group name: rnfs_group, instance number: 5523
hit <Enter> key to suspend
[ joined as client ] mount network file system from sp6n05
[ replicate ] replicate file system
[ server failure ] umount network file system <--------------------(12)

[ program suspended ] l(eave) or r(esume)?: l <--------------------(13)
* warning * leaving the group is canceled
[ server takeover ] mount network file system from sp6n01

[ program suspended ] l(eave) or r(esume)?: l <--------------------(14)
[ server takeover ] umount network file system
$
```

Figure 159. rnfs execution on node1

```
$ rnfs sp6en0 <----------------------------------------------------(3)
domain name: sp6en0, group name: rnfs_group, instance number: 5523
hit <Enter> key to suspend
[ joined as client ] mount network file system from sp6n01
[ replicate ] replicate file system
[ replicate ] replicate file system
[ replicate ] replicate file system
* warning * replicating file system might have failed on node 5 <--(5)
* warning * responsiveness check has failed on node 5
[ replicate ] replicate file system
* warning * responsiveness check has recovered on node 5 <---------(6)
[ replicate ] replicate file system
* warning * responsiveness check has failed on node 1
[ server shutdown ] replicate file system <------------------------(7)
[ server shutdown ] umount network file system
[ server takeover ] mount network file system from sp6n03

[ program suspended ] l(eave) or r(esume)?:^C$ <-------------------(8)(10)
```

Figure 160. rnfs execution on node3

```
$ rnfs sp6en0 <-----------------------------------------------------(4)
domain name: sp6en0, group name: rnfs_group, instance number: 5523
hit <Enter> key to suspend
[ joined as client ] mount network file system from sp6n01
[ replicate ] replicate file system

[ program suspended ] l(eave) or r(esume)?: r <--------------------(5,6)
[ replicate ] replicate file system
* warning * replicating file system might have failed on node 5
* warning * responsiveness check has failed on node 5
[ replicate ] replicate file system
* warning * responsiveness check has recovered on node 5
[ replicate ] replicate file system
* warning * responsiveness check has failed on node 1
[ server shutdown ] replicate file system <------------------------(7)
[ server shutdown ] umount network file system
[ server takeover ] mount network file system from sp6n03
[ replicate ] replicate file system
* warning * responsiveness check has failed on node 3 <------------(8)
* warning * it might be very busy on node 3 <----------------------(9)
[ server failure ] umount network file system <--------------------(10)
[ server takeover ] mount network file system from sp6n05
^C$ <--------------------------------------------------------------(12)
```

Figure 161. rnfs execution on node5

9.6.3 rnfs log file

Figure 162 on page 226 through Figure 164 on page 227 show the rnfs.log log file on nodes 1, 3, and 5. Arrows with a number indicate the point at which the numbered event has occurred. The log file records the following information:

- The `mount` command execution
- The `umount` command execution
- The `cp` command execution
- The deactivate script execution.

```
$ cat rnfs.log
Fri Dec 10 15:56:18 EST 1999 <--------------------------------------(1)
   mount sp6n01:/local_nfs /shared_nfs
Fri Dec 10 15:57:02 EST 1999 <--------------------------------------(7)
   umount /shared_nfs
Fri Dec 10 15:57:33 EST 1999 <--------------------------------------(11)
   mount sp6n05:/local_nfs /shared_nfs
Fri Dec 10 15:57:37 EST 1999
   cp -R /shared_nfs/. /local_nfs
Fri Dec 10 15:57:41 EST 1999 <--------------------------------------(12)
   umount /shared_nfs
Fri Dec 10 15:57:46 EST 1999
   mount sp6n01:/local_nfs /shared_nfs
Fri Dec 10 15:57:50 EST 1999 <--------------------------------------(14)
   umount /shared_nfs
$
```

Figure 162. rnfs.log on node1

```
$ cat rnfs.log
Fri Dec 10 15:56:22 EST 1999 <--------------------------------------(3)
   mount sp6n01:/local_nfs /shared_nfs
Fri Dec 10 15:56:26 EST 1999
   cp -R /shared_nfs/. /local_nfs
Fri Dec 10 15:56:32 EST 1999
   cp -R /shared_nfs/. /local_nfs
Fri Dec 10 15:56:39 EST 1999
   cp -R /shared_nfs/. /local_nfs
Fri Dec 10 15:56:48 EST 1999
   cp -R /shared_nfs/. /local_nfs
Fri Dec 10 15:56:57 EST 1999
   cp -R /shared_nfs/. /local_nfs
Fri Dec 10 15:57:02 EST 1999 <--------------------------------------(7)
   cp -R /shared_nfs/. /local_nfs
Fri Dec 10 15:57:02 EST 1999
   umount /shared_nfs
Fri Dec 10 15:57:02 EST 1999
   mount sp6n03:/local_nfs /shared_nfs
Fri Dec 10 15:57:27 EST 1999 <--------------------------------------(10)
   deactivate script executed by the Group Services
      Process ID:         0
      Voting Time Limit:  5
      Failed group:       rnfs_group
      Deactivate flag:    providerdied
      Failed provider(s): 5523
   umount /shared_nfs
$
```

Figure 163. rnfs.log on node3

```
$ cat rnfs.log
Fri Dec 10 15:56:27 EST 1999 <--------------------------------------(4)
   mount sp6n01:/local_nfs /shared_nfs
Fri Dec 10 15:56:32 EST 1999
   cp -R /shared_nfs/. /local_nfs
Fri Dec 10 15:56:52 EST 1999
   cp -R /shared_nfs/. /local_nfs
Fri Dec 10 15:56:53 EST 1999
   cp -R /shared_nfs/. /local_nfs
Fri Dec 10 15:56:57 EST 1999
   cp -R /shared_nfs/. /local_nfs
Fri Dec 10 15:57:02 EST 1999 <--------------------------------------(7)
   cp -R /shared_nfs/. /local_nfs
Fri Dec 10 15:57:02 EST 1999
   umount /shared_nfs
Fri Dec 10 15:57:02 EST 1999
   mount sp6n03:/local_nfs /shared_nfs
Fri Dec 10 15:57:09 EST 1999
   cp -R /shared_nfs/. /local_nfs
Fri Dec 10 15:57:27 EST 1999 <--------------------------------------(10)
   umount /shared_nfs
Fri Dec 10 15:57:32 EST 1999
   mount sp6n05:/local_nfs /shared_nfs
Fri Dec 10 15:57:41 EST 1999 <--------------------------------------(12)
   deactivate script executed by the Group Services
      Process ID:         0
      Voting Time Limit:  5
      Failed group:       rnfs_group
      Deactivate flag:    providerdied
      Failed provider(s): 5523
   umount /shared_nfs
$
```

Figure 164. rnfs.log on node5

9.6.4 rnfsm execution output

Finally, Figure 165 shows an execution of the rnfsm program. Arrows with a number indicate the point at which the numbered event has occurred. This program can be executed on any node in the domain. You can also execute as many instances as you want. The program must be started after the first rnfs program joins the group. When all the nodes leave the group, the program terminates automatically.

```
$ rnfsm sp6en0
domain name: sp6en0, group name: rnfs_group
hit <Enter> key to suspend

server node hostname: sp6n01 <-------------------------------------(2)
client nodes: 1

server node hostname: sp6n01 <-------------------------------------(3)
client nodes: 1 3

server node hostname: sp6n01 <-------------------------------------(4)
client nodes: 1 3 5

server node hostname: sp6n01 <-------------------------------------(7)
client nodes: 3 5

server node hostname: sp6n03

server node hostname: sp6n03 <-------------------------------------(10)
client nodes: 5

server node hostname: sp6n05

server node hostname: sp6n05 <-------------------------------------(11)
client nodes: 5 1

server node hostname: sp6n05 <-------------------------------------(12)
client nodes: 1

server node hostname: sp6n01

*** no RNFS node is available any more *** <-----------------------(14,15)
$
```

Figure 165. rnfsm execution

Chapter 10. Checking your program

During the GS application programing, you may want to check that your program is working as you expected. For example, you think the program has joined the group; however, Group Services may not think so.

Group Services provides you with several commands to look inside the Group Services subsystem. This chapter takes a close look at the following commands:

- hagsgr
- hagscl
- hagsvote

These commands are provided for development team use; therefore, they are undocumented, and it is difficult to understand their output completely. This chapter focuses only on the part of their output that is useful for checking your program.

Note

The `hagsgr`, `hagscl`, and `hagsvote` commands are owned by the user, *bin*, and belong to the group, *bin*. Prior to using them, make sure that you have the proper access permissions.

10.1 Command usage

The commands mentioned in this chapter use the following command usage:

command_name [-h host] [-l] -g group_name
Using the `-g` flag, you can specify a group name and display only its information. The group name is a name used by the System Resource Controller (SRC).

command_name [-h host] [-l] -s subsystem_name
Using the `-s` flag, you can specify a subsystem name and display only its information. The subsystem name is a name used by SRC.

command_name [-h host] [-l] -p subsystem_pid
Using the `-p` flag, you can specify the process ID of the Group Services daemon and display only its information.

The `hagsgr` and `hagsvote` commands use an additional flag:

[-a group_name]

Using the `-a` flag, you can specify a group name and display only its information. The group name is a name used by Group Services.

10.2 Command examples

Using the commands mentioned in this chapter, you can check the following information:

- The group, providers, and subscribers
- The group attributes
- The group state value
- The providers and subscribers in detail
- A deactivate script
- A responsiveness check
- The protocol currently executing

The following sections explain how to get this information. All the sections use the operation example described in Section 9.6, “Operation example” on page 218. This chapter assumes that the rnfsm program is to run on node 1.

10.2.1 Checking the group, providers, and subscribers

To check whether the group has been created and whether the program has become a provider or subscriber, you can use the `hagsgr` command.

When nodes 1, 3, and 5 have joined the group and the rnfsm program has started on node 1 (event number 4), execute the `hagsgr` command on node 1. Figure 166 shows the command output:

```
# hagsgr -a rnfs_group -s hags
Number of groups: 5
Group name [rnfs_group]  group state[Inserted |Idle |]
Providers[[5523/1][5523/3][5523/5]]
Local subscribers[[10/1]]

#
```

Figure 166. Checking the group, providers, and subscribers on node 1

The output provides you with the following information:

- The group name is rnfs_group (Group name [rnfs_group]).

- There is a provider running on nodes 1, 3, and 5. All of them use the instance number 5523 (Providers[[5523/1][5523/3][5523/5]]).
- There is a subscriber running on node 1 (Local subscribers[[10/1]]). The instance number of a subscriber is assigned by Group Services automatically.

If you execute the command on another node, such as node 3 or 5, the output does not provide subscribers with information because subscribers are managed by a local Group Services daemon. Figure 167 shows the command output when the command is executed on node 3:

```
# hagsgr -a rnfs_group -s hags
Number of groups: 5
Group name [rnfs_group]  group state[Inserted |Idle |]
Providers[[5523/1][5523/3][5523/5]]
Local subscribers[]

#
```

Figure 167. Checking the group, providers, and subscribers on node 3

Subscriber information is not provided (Local subscribers[]).

An alternative to the `hagsgr` command is the `lssrc` command. The command is provided by the SRC as a standard AIX command. To check similar information, execute the `lssrc` command on node 1. Figure 168 shows the output:

```
# lssrc -ls hags
Subsystem         Group            PID     Status
 hags             hags             19092   active
4 locally-connected clients.  Their PIDs:
18788(rnfs) 6018(haemd) 18354(hagsglsmd) 6418(rnfsm)
HA Group Services domain information:
Domain established by node 11
Number of groups known locally: 3
                   Number of   Number of local
Group name         providers   providers/subscribers
cssMembership          10          1           0
rnfs_group              3          1           1
ha_em_peers            11          1           0
#
```

Figure 168. Checking the group, providers, & subscribers with the lssrc command

The output provides you with the following information:

- The group name is rnfs_group (Group name - rnfs_group).

- The number of providers in the group is 3 (Number of providers - 3).
- The number of providers on node 1 is 1 (Number of local providers - 1).
- The number of subscribers on node 1 is 1 (Number of local subscribers -1).

10.2.2 Checking the group attributes

To check whether the group uses the group attributes that you expected, you can use the `hagsgr` command.

When nodes 1, 3, and 5 have joined the group and the rnfsm program has started on node 1 (event number 4), execute the `hagsgr` command on any node. Figure 169 on page 233 through Figure 171 on page 235 show the output executed on node 1. Output is lengthy; therefore, numbers are added in the figures as a reference mark.

```
# hagsgr -l -a rnfs_group -s hags
Number of groups: 5
Information for SGroup: Group name [rnfs_group]
I am *not* Group Leader!  Created by subscription request.
group state[Inserted |Idle |]
ProtocolToken[3219/6355]
 counts: (prov/localprov/subs) [3/1/1]
 delayed join count [0]
 protocol counts:  received [0]
  dropped(total/DaemonMsg/ProtMgr) [4/0/1]
 message counts:  current queued [0] future queued/cumulative [0/0]
Group attributes[{group name:  value: <------------------------------------ (1)
rnfs_group
}
 batching[No batching| DeactivateOnFailure] membership phases[N phases] reflecti
on phases[?? unknown number of phases ??]
default vote[Approve] merge control[Dissolve]
membership time limit[5] reflection time limit[0]
client version[1] version[1] size[40]

Group state value: [min/max lengths (1/256)] actual length[7]  value: <----- (2)
0x7370366e 0x303100
sp6n01.

--------------------
 Provider list:
SProvider(ProviderId[5523/1]conditionalListPosition[-1]
SVSuppMember: [owned by:Client: socketFd[4] pid[18788] progname[rnfs]] token[0]
status[MemberIn ] name[SMemberName: (min/max)length: (1/16)4
value:
rNFS
]
 supp ptr: 0x301b5658 group ptr: 0x300bbb48 groupListPosition: 0 nodeListPositio
n: 0 Need Vote/Voted Yet[0/1]
0x301d1528 [votingParticipant])[end SProvider]

SProvider(ProviderId[5523/3]conditionalListPosition[-1]
SVSuppMember:  token[0] status[MemberIn ]
 supp ptr: 0x0 group ptr: 0x300bbb48 groupListPosition: 1 nodeListPosition: 0 Ne
ed Vote/Voted Yet[0/0]
 [votingParticipant])[end SProvider]

SProvider(ProviderId[5523/5]conditionalListPosition[-1]
SVSuppMember:  token[0] status[MemberIn ]
 supp ptr: 0x0 group ptr: 0x300bbb48 groupListPosition: 2 nodeListPosition: 0 Ne
ed Vote/Voted Yet[0/0]
 [votingParticipant])[end SProvider]
```

Figure 169. Checking the group attributes (1 of 3)

```
--------------------

Local provider list: <----------------------------------------------------- (3)
SProvider(ProviderId[5523/1]conditionalListPosition[-1]
SVSuppMember: [owned by:Client: socketFd[4] pid[18788] progname[rnfs]] token[0]
status[MemberIn ] name[SMemberName: (min/max)length: (1/16)4
value:
rNFS
]
 supp ptr: 0x301b5658 group ptr: 0x300bbb48 groupListPosition: 0 nodeListPositio
n: 0 Need Vote/Voted Yet[0/1]
0x301d1528 [votingParticipant])[end SProvider]

--------------------

Local subscriber list: <--------------------------------------------------- (4)
SSubscriber(SVSuppMember: [owned by:Client: socketFd[10] pid[6418] progname[rnfs
m]] token[0] status[MemberIn ] name[SMemberName: (min/max)len
gth: (1/16)11 value:
noNameGiven
]
 supp ptr: 0x301cd608 group ptr: 0x300bbb48 groupListPosition: 0 nodeListPositio
n: 1 Need Vote/Voted Yet[0/0]
 Last Request:0x301cd8f8
 subscriptions[HA_GS_SUBSCRIBE_STATE HA_GS_SUBSCRIBE_DELTA_JOINS HA_GS_SUBSCRIBE
_DELTA_LEAVES HA_GS_SUBSCRIBE_MEMBERSHIP]
number notifications sent[46]
[end SSubscriber]

--------------------

Protocol Manager summary information:
Current count: 0
total count: executed/approved/rejected[678/667/11]
failure count: executed/approved/rejected(explicit/implicit)[54/54/0(0/0)]
join count: executed/approved/rejected[93/83/10]
expel count: executed/approved/rejected[0/0/0]
attribute change count: executed/approved/rejected[0/0/0]
leave count: executed/approved/rejected[26/26/0]
state change count: executed/approved/rejected[29/29/0]
PBM count: executed/approved/rejected[380/380/0]
source reflection count: executed/approved/rejected[0/0/0]
subscription count: executed/approved/rejected[2/1/1]
announcement count: executed/approved/rejected[94/94/0]
```

Figure 170. Checking the group attributes (2 of 3)

```
--------------------
No transient protocol
--------------------
No currently executing protocol
--------------------
Unsent queue:[No entries]
--------------------
Sent queue:[No entries]
--------------------
Failure queue:[No entries]
--------------------
Join queue:[No entries]
--------------------
Subscribe queue:[No entries]
--------------------
Announcement queue:[No entries]

#
```

Figure 171. Checking the group attributes (3 of 3)

You can find the group attributes information at mark number 1 in Figure 169 on page 233:

```
Group attributes[{group name:  value:
rnfs_group
}
 batching[No batching| DeactivateOnFailure] membership phases[N phases] reflecti
on phases[?? unknown number of phases ??]
default vote[Approve] merge control[Dissolve]
membership time limit[5] reflection time limit[0]
client version[1] version[1] size[40]
```

This output provides you with the following information:

- The group name is rnfs_group ({group name: value: rnfs_group}).
- Batching protocols is not used and deactivate on failure is used (batching[No batching| DeactivateOnFailure]).
- The number of protocol phases is N-phase (membership phases[N phases]).
- The number of protocol phases for a source-state reflection protocol is not defined (reflection phases[?? unknown number of phases ??]).
- The default vote value is approve (default vote[Approve]).
- The merge control has a value of HA_GS_DISSOLVE_MERGE (merge control[Dissolve]).
- The voting time limit is five seconds (membership time limit[5]).

- The voting time limit for a source-state reflection protocol is not defined (reflection time limit[0]).
- The user-defined client version is 1 (client version[1]).
- The version of he Group Services library is 1 (version[1]).
- The group attributes block has a 40 byte length (size[40]).

10.2.3 Checking the group state value

To check whether the group uses the group state value that you expected, you can use the `hagsgr` command.

When nodes 1, 3, and 5 have joined the group and the rnfsm program has started on node 1 (event number 4), execute the `hagsgr` command on any node. Figure 169 on page 233 through Figure 171 on page 235 show the output executed on node 1. Output is lengthy; therefore, numbers are added in the figures as a reference mark.

You can find the group attributes information at mark number 2 in Figure 169 on page 233:

```
Group state value: [min/max lengths (1/256)] actual length[7]  value:
0x7370366e 0x303100
sp6n01.
```

This output provides you with the following information:

- The group state value has a seven byte length (actual length[7]).
- The group state value has a hexadecimal value of 0x7370366e 0x303100 and an ASCII value of sp6n01 (value: 0x7370366e 0x303100 sp6n01.).

At this moment, node 1 is a server node; therefore, the group state value is equal to the hostname of node 1.

Execute the command when node 1 has left the group (event number 7). The output follows:

```
Group state value: [min/max lengths (1/256)] actual length[7]  value:
0x7370366e 0x303300
sp6n03.
```

At this moment, node 3 is a server node; therefore, the group state value is equal to the hostname of node 3 (value: 0x7370366e 0x303300 sp6n03.).

If a group does not make use of the group state value, the group state value is initialized with a length of four bytes of 0. Then, the output should look like the following:

```
Group state value: [min/max lengths (1/256)] actual length[4]  value:
0x00000000
....
```

10.2.4 Checking providers and subscribers in detail

To find more detailed information on providers and subscribers, you can use the `hagsgr` command.

When nodes 1, 3, and 5 have joined the group and the rnfsm program has started on node 1 (event number 4), execute the `hagsgr` command on the node the providers and subscribers are running. The command only provides detailed information for local providers and subscribers. Figure 169 on page 233 through Figure 171 on page 235 show the output executed on node 1. Output is lengthy; therefore, numbers are added in the figures as a reference mark.

10.2.4.1 Detailed providers information

You can find the detailed providers information at mark number 3 in Figure 170 on page 234:

```
Local provider list:
SProvider(ProviderId[5523/1]conditionalListPosition[-1]
SVSuppMember: [owned by:Client: socketFd[4] pid[18788] progname[rnfs]] token[0]
status[MemberIn ] name[SMemberName: (min/max)length: (1/16)4
value:
rNFS
]
 supp ptr: 0x301b5658 group ptr: 0x300bbb48 groupListPosition: 0 nodeListPositio
n: 0 Need Vote/Voted Yet[0/1]
0x301d1528 [votingParticipant])[end SProvider]
```

The output provides you with the following information:

- The process ID is 18788 (pid[18788]).
- The program name is rnfs (progname[rnfs]).
- The local name is rNFS using four bytes (name[SMemberName: (min/max)length: (1/16)4 value: rNFS]).
- The position in the membership list is top (groupListPosition: 0).

10.2.4.2 Detailed subscribers information

You can find the following detailed subscribers information at mark number 4 in Figure 170 on page 234:

```
Local subscriber list:
SSubscriber(SVSuppMember: [owned by:Client: socketFd[10] pid[6418] progname[rnfs
m]] token[0] status[MemberIn ] name[SMemberName: (min/max)len
gth: (1/16)11 value:
noNameGiven
]
 supp ptr: 0x301cd608 group ptr: 0x300bbb48 groupListPosition: 0 nodeListPositio
n: 1 Need Vote/Voted Yet[0/0]
 Last Request:0x301cd8f8
 subscriptions[HA_GS_SUBSCRIBE_STATE HA_GS_SUBSCRIBE_DELTA_JOINS HA_GS_SUBSCRIBE
_DELTA_LEAVES HA_GS_SUBSCRIBE_MEMBERSHIP]
number notifications sent[46]
[end SSubscriber]
```

The output provides you with the following information:

- The process ID is 6418 (pid[6418]).
- The program name is rnfsm (progname[rnfsm]).
- The subscriber subscribes to
  - HA_GS_SUBSCRIBE_STATE
  - HA_GS_SUBSCRIBE_DELTA_JOINS
  - HA_GS_SUBSCRIBE_DELTA_LEAVES
  - HA_GS_SUBSCRIBE_MEMBERSHIP

  (subscriptions[HA_GS_SUBSCRIBE_STATE HA_GS_SUBSCRIBE_DELTA_JOINS HA_GS_SUBSCRIBE _DELTA_LEAVES HA_GS_SUBSCRIBE_MEMBERSHIP]).

  Actually the rnfsm program uses the value of HA_GS_SUBSCRIBE_STATE_AND_MEMBERSHIP; however, this value includes all the listed values.

10.2.5 Checking a deactivate script

When you use a deactivate script, you have to be careful with:

Effective user ID (UID) and group ID (GID)
A deactivate script will be executed with the effective UID and GID of the targeted provider that the provider had when it initialized with Group Services.

Working directory
A deactivate script will be executed in the current working

directory of the targeted provider that the provider had when it initialized with Group Services.

If Group Services uses effective uid, gid, and/or working directories that are different than what you expected, a deactivate script may fail; therefore, checking this information is important. To check deactivate script information, you can use the `hagscl` command. This command only provides information to local providers and subscribers. It does not allow you to use the -a flag; therefore, output contains all the local client information.

When nodes 1, 3, and 5 have joined the group and the rnfsm program has started on node 1 (event number 4), execute the `hagscl` command on the node on which you want to check a deactivate script. Figure 172 on page 240 shows the output executed on node 1. Because the output is lengthy, a part of it is removed.

```
# hagscl -l -s hags
Client Control layer summary:
  Number of clients connected: 4
  Cumulative number of clients connected: 41
  Total number of client requests: 78
  Number of client hash table conflicts: 0

--------------------------------------------------
Client: socketFd[4] pid[18788] progname[rnfs]Total number of Clients: 4
 Client initialized: pid: 18788 progname: rnfs
  uid/gid/version: [201/203/5]
  client directory: [SuppName: length: 10 value:
/tmp/yoshi

Number of local providers/subscribers: 1/0
Responsiveness information for Client: socketFd[4] pid[18788] progname[rnfs] <--- (1)
Type[ type[HA_GS_PING_RESPONSIVENESS]] interval[2] response time limit[1]
Checks done/bypassed[12/0] lastResponse[OK]]
Results(good/bad/late)[12/0/0]

Deactivate script information:SuppName: length: 17 value:
./rnfs_deact.ksh
Membership list:
slot    info
0       [{provider}Member token[0] Client: socketFd[4] pid[18788] progname[rnfs]
ProviderId[5523/1]]
--------------------------------------------------

<<< output is partially removed >>>

--------------------------------------------------
Client: socketFd[10] pid[6418] progname[rnfsm]Total number of Clients: 4
 Client initialized: pid: 6418 progname: rnfsm
  uid/gid/version: [201/203/5]
  client directory: [SuppName: length: 10 value:
/tmp/yoshi

Number of local providers/subscribers: 0/1
Membership list:
slot    info
0       [{subscriber}Member token[0] Client: socketFd[10] pid[6418] progname[rnf
sm]]
#
```

Figure 172. Checking a deactivate script and responsiveness check

The output provides you with the following information:

- Effective UID and GID are initialized as 201 and 203 (uid/gid/version: [201/203/5]).
- Working directory is initialized as /tmp/yoshi and has a 10 byte length (client directory: [SuppName: length: 10 value: /tmp/yoshi).
- A deactivate script is ./rnfs_deact.ksh and has 17 byte length (Deactivate script information:SuppName: length: 17 value: ./rnfs_deact.ksh).

In our environment, UID 201 is a general user named *gs* and only belongs to the group named *hagsuser* that uses GID 203. The rnfs program resides in the /tmp/yoshi directory.

10.2.6 Checking responsiveness check

To check whether the responsiveness check has been done as you expected, you can use the `hagscl` command.

When nodes 1, 3, and 5 have joined the group and the rnfsm program has started on node 1 (event number 4), execute the `hagscl` command on the node on which you want to check responsiveness. Figure 172 on page 240 shows the output executed on node 1. Because the output is lengthy, a part of it is removed.

You can find the responsiveness check information at mark number 1 in Figure 172 on page 240:

```
Responsiveness information for Client: socketFd[4] pid[18788] progname[rnfs]
Type[ type[HA_GS_PING_RESPONSIVENESS]] interval[2] response time limit[1]
Checks done/bypassed[12/0] lastResponse[OK]]
Results(good/bad/late)[12/0/0]
```

The output provides you with the following information:

- The group uses the ping type responsiveness check (type[HA_GS_PING_RESPONSIVENESS]).
- Responsiveness is checked every two seconds (interval[2]).
- A provider must respond within one second (response time limit[1]).
- A responsiveness check has been done 12 times and canceled 0 times (Checks done/bypassed[12/0]).
- The last responsiveness check was OK (lastResponse[OK]]).
- So far, responsiveness has been OK 12 times and not OK 0 times (Results(good/bad/late)[12/0/0]).

If you execute the command when node 5 is suspended (event number 5), you will have the following output. Make sure that you execute the command on node 5 because the `hagscl` command only provides local provider information:

```
Responsiveness information for Client: socketFd[4] pid[18124] progname[rnfs]
Type[ type[HA_GS_PING_RESPONSIVENESS]] interval[2] response time limit[1]
Checks done/bypassed[14/2] lastResponse[Not OK]]
Results(good/bad/late)[10/4/4]
```

This time, the output is different than the previous time:

- The responsiveness check has been done 14 times and canceled two times (Checks done/bypassed[14/2]). A provider-broadcast message protocol is proposed almost every 10 seconds to replicate the file system. If a provider responds to this proposal properly, Group Services decides that the provider has responsiveness and cancels the responsiveness check. Responsiveness check is scheduled every two seconds; therefore, it will be canceled only once in a while.
- The last response is not OK (lastResponse[Not OK]).
- So far, responsiveness has been OK 10 times and not OK four times (Results(good/bad/late)[10/4/4]).

10.2.7 Checking the protocol currently executing

To check the protocol currently executing, you can use the `hagsvote` command.

The rnfs group executes a provider-broadcast message protocol every 10 seconds. However, checking this protocol is not so easy. The rnfs program uses only two phases for a protocol, and they do not last long enough to catch. When you execute the `hagsvote` command, you usually get output similar to that shown in Figure 173:

```
# hagsvote -l -a rnfs_group -s hags
Number of groups: 5
Group name [rnfs_group] GL node [1] voting data:
No protocol is currently executing in the group.

#
```

Figure 173. Checking the protocol currently executing

There are two chances that you can check the protocol currently executing:

- When node 5 is suspended (event number 5), you can check a provider-broadcast message protocol.
- When node 1 is joining while node 3 is suspended (event number 9), you can check a join protocol.

In the first case, node 1 proposes a provider-broadcast message protocol to replicate the file system. Node 3 votes to approve, but node 5 cannot because it is suspended. Group Services waits five seconds before it votes to approve for node 5's vote using the default vote value.

Figure 174 shows the command output on node 3.

```
# hagsvote -l -a rnfs_group -s hags
Number of groups: 5
Group name [rnfs_group] GL node [1] voting data:
Not GL in phase [1] of n-phase protocol of type [ProviderMessage].
Local voting data:
Number of providers: 1
Number of providers not yet voted: 0 (vote submitted).
Given vote:[Approve vote] Default vote:[No vote value]
ProviderId      Voted?  Failed? Conditional?
[5523/3]        Yes     No      No


#
```

Figure 174. Checking a provider-broadcast message protocol on node 3

The output provides the following information:

- This is the first phase of the protocol (phase [1]).
- The current protocol is a provider-broadcast message protocol (type [ProviderMessage]).
- The provider has voted (Voted? - Yes) to approve (Given vote:[Approve vote]).

Figure 175 shows the command output on node 5.

```
# hagsvote -l -a rnfs_group -s hags
Number of groups: 5
Group name [rnfs_group] GL node [1] voting data:
Not GL in phase [1] of n-phase protocol of type [ProviderMessage].
Local voting data:
Number of providers: 1
Number of providers not yet voted: 1 (vote not submitted).
Given vote:[No vote value] Default vote:[No vote value]
ProviderId      Voted?  Failed? Conditional?
[5523/5]        No      No      No


#
```

Figure 175. Checking a provider-broadcast message protocol on node 5

The difference from the command output of node 3 is that node 5 has not voted (Voted? - No) yet (Given vote:[No vote value]).

In the second case, node 1 proposes a join protocol to join the group. Node 5 votes to reject because the group is unstable. Node 3 cannot vote because it is suspended. Group Services waits five seconds before it votes to approve for node 3's vote using the default vote value.

Figure 176 shows the command output on node 5.

```
# hagsvote -l -a rnfs_group -s hags
Number of groups: 5
Group name [rnfs_group] GL node [1] voting data:
Not GL in phase [1] of n-phase protocol of type [Join].
Local voting data:
Number of providers: 1
Number of providers not yet voted: 0 (vote submitted).
Given vote:[Reject vote] Default vote:[No vote value]
ProviderId      Voted?  Failed? Conditional?
[5523/5]        Yes     No      No


#
```

Figure 176. Checking a join protocol on node 5

The output provides the following information:

- This is the first phase of the protocol (phase [1]).
- The current protocol is a join protocol (type [Join]).
- The provider has voted (Yes) to reject (Given vote:[Reject vote]).

Figure 177 shows the command output on node 3.

```
# hagsvote -l -a rnfs_group -s hags
Number of groups: 5
Group name [rnfs_group] GL node [1] voting data:
Not GL in phase [1] of n-phase protocol of type [Join].
Local voting data:
Number of providers: 1
Number of providers not yet voted: 1 (vote not submitted).
Given vote:[No vote value] Default vote:[No vote value]
ProviderId      Voted?  Failed? Conditional?
[5523/3]        No      No      No


#
```

Figure 177. Checking a join protocol on node 3

The difference from the command output of node 5 is that node 3 has not voted (Voted? - No) yet (Given vote:[No vote value]).

Part 4. Appendices

Appendix A. Programming environment

This appendix provides information on the Group Services programming environment.

A.1 The Group Services shared libraries

The Group Services Application Programming Interface (GSAPI) is a shared library that a GS client uses to obtain the services of the Group Services subsystem. This shared library is supplied in two versions: One for non-thread safe programs and one for thread-safe programs. These libraries are referenced by the following path names:

- /usr/lib/libha_gs.a (non-thread safe version)
- /usr/lib/libha_gs_r.a (thread-safe version)

These path names are actually symbolic links to /usr/sbin/rsct/lib/libha_gs.a and /usr/sbin/rsctl/lib/libha_gs_r.a, respectively. The symbolic links are placed in /usr/lib for ease of use. For serviceability, the actual libraries are placed in the /usr/sbin/rsct/lib directory. These libraries are supplied as shared libraries, also for serviceability.

To allow non-root users to use the Group Services shared library, perform the following steps:

1. Create a group named hagsuser.
2. Add the desired user IDs to the hagsuser group.
3. Stop and restart hags (if it was running before you created the hagsuser group).

Users in the created hagsuser group can use the Group Services shared library.

A.2 Link and compile options

This redbook uses the following link and compile options for the rnfs and rnfsm program:

-g	Include debugging information.
-qnofold	Suppress compile-time evaluation of constant floating-point expressions.
-DBSD	Set the BSD option to 1 (same as #define BSD).

-bloadmap:<file_name> Specifies the name of the map-file.

-lha_gs Uses non-thread safe version Group Services library.

-lbsd Uses BSD shared library.

For information on makefile, refer to Section C.8, “makefile” on page 277.

A.3 The man pages

All the subroutines provided by the Group Services shared library are documented as man pages. To access the man pages, use the following steps:

1. Install the PSSP file set ssp.docs on your system.
2. Make sure that the MANPATH environment variable includes the path, /usr/lpp/ssp/man.

To display the man pages, issue the `man` command with the subroutine name:

```
# man <subroutine name>
```

For example, to display the man pages of the ha_gs_init suroutine, issue the `man` command as follows:

```
# man ha_gs_init
```

Appendix B. ha_gs.h

This appendix provides you with the ha_gs.h file, the header file provided by Group Services.

B.1 ha_gs.h

```
/* IBM_PROLOG_BEGIN_TAG                                                   */
/* This is an automatically generated prolog.                             */
/*                                                                        */
/*                                                                        */
/*                                                                        */
/* Licensed Materials - Property of IBM                                   */
/*                                                                        */
/* (C) COPYRIGHT International Business Machines Corp. 1996,1999          */
/* All Rights Reserved                                                    */
/*                                                                        */
/* US Government Users Restricted Rights - Use, duplication or            */
/* disclosure restricted by GSA ADP Schedule Contract with IBM Corp.      */
/*                                                                        */
/* IBM_PROLOG_END_TAG                                                     */
#ifndef _HA_GS_H_
#define _HA_GS_H_
/*********************************************************************/
/*                                                                   */
/* CPRY PGM                                                          */
/*                                                                   */
/*  Licensed Materials - Property of IBM                             */
/*                                                                   */
/*  5765-529 PSSP                                                    */
/*                                                                   */
/*  (C) Copyright IBM Corp. 1996 All Rights Reserved.                */
/*                                                                   */
/*  US Government Users Restricted Rights - Use, duplication or disclosure */
/*  restricted by GSA ADP Schedule Contract with IBM Corp.           */
/*                                                                   */
/* CPRY                                                              */
/*********************************************************************/

static char *ha_gs_h_sccsid = "@(#)86   1.28   src/rsct/pgs/pgslib/ha_gs.h, gsapi,
rsct_rmoh, rmoht5d9 4/6/99 21:33:46";

#ifdef __cplusplus
extern "C" {
#endif

#include <sys/types.h>
#include <rsct/ct_ffdc.h>

#define HA_GS_RELEASE 5

typedef enum
{
    HA_GS_OK,                           /* 0 */
    HA_GS_OK_SO_FAR = HA_GS_OK,         /* 0 */
    HA_GS_NOT_OK,                       /* 1 */
    HA_GS_EXISTS,                       /* 2 */
    HA_GS_NO_INIT,                      /* 3 */
    HA_GS_NAME_TOO_LONG,                /* 4 */
    HA_GS_NO_MEMORY,                    /* 5 */
    HA_GS_NOT_A_MEMBER,                 /* 6 */
    HA_GS_BAD_CLIENT_TOKEN,             /* 7 */
    HA_GS_BAD_MEMBER_TOKEN,             /* 8 */
    HA_GS_BAD_PARAMETER,                /* 9 */
    HA_GS_UNKNOWN_GROUP,                /* 10 */
    HA_GS_INVALID_GROUP,                /* 11 */
    HA_GS_NO_SOURCE_GROUP_PROVIDER,     /* 12 */
    HA_GS_BAD_GROUP_ATTRIBUTES,         /* 13 */
    HA_GS_WRONG_OLD_STATE,              /* 14 */
    HA_GS_DUPLICATE_INSTANCE_NUMBER,    /* 15 */
    HA_GS_COLLIDE,                      /* 16 */
    HA_GS_SOCK_CREATE_FAILED,           /* 17 */
    HA_GS_SOCK_INIT_FAILED,             /* 18 */
    HA_GS_CONNECT_FAILED,               /* 19 */
    HA_GS_VOTE_NOT_EXPECTED,            /* 20 */
    HA_GS_NOT_SUPPORTED,                /* 21 */
    HA_GS_INVALID_SOURCE_GROUP,         /* 22 */
    HA_GS_UNKNOWN_PROVIDER,             /* 23 */
    HA_GS_INVALID_DEACTIVATE_PHASE,     /* 24 */
    HA_GS_PROVIDER_APPEARS_TWICE,       /* 25 */
    HA_GS_BACKLEVEL_PROVIDERS           /* 26 */
}ha_gs_rc_t;                    /* Return Codes */

typedef enum
{
    HA_GS_NO_BATCHING   = 0x0000,
    HA_GS_BATCH_JOINS   = 0x0001,
    HA_GS_BATCH_LEAVES  = 0x0002,
    HA_GS_BATCH_BOTH    = 0x0003,
    HA_GS_DEACTIVATE_ON_FAILURE = 0x0004
} ha_gs_batch_ctrl_t;           /* Controls Batching of Requests */

typedef enum
{
    HA_GS_1_PHASE               = 0x0001,
    HA_GS_N_PHASE               = 0x0002
} ha_gs_num_phases_t;           /* Protocol number of Phases selection */

typedef enum
{
    HA_GS_FIRST_MERGE_TYPE,             /* 0 */
    HA_GS_DISSOLVE_MERGE = HA_GS_FIRST_MERGE_TYPE, /* 0 */
    HA_GS_LARGER_MERGE,                            /* 1 */
    HA_GS_SMALLER_MERGE,                           /* 2 */
    HA_GS_DONTCARE_MERGE,                          /* 3 */
    HA_GS_LAST_MERGE_TYPE = HA_GS_DONTCARE_MERGE   /* 3 */
} ha_gs_merge_ctrl_t;           /* Controlling Merges */

typedef enum
{
    HA_GS_NULL_VOTE,
    HA_GS_VOTE_APPROVE,
    HA_GS_VOTE_CONTINUE,
    HA_GS_VOTE_REJECT
} ha_gs_vote_value_t;           /* Allowable Vote Responses */

typedef enum
{
    HA_GS_SOCKET_NO_SIGNAL,
    HA_GS_SOCKET_SIGNAL
} ha_gs_socket_ctrl_t;          /* Socket Control */
```

```
typedef enum
{
    HA_GS_NON_BLOCKING,
    HA_GS_BLOCKING
} ha_gs_dispatch_flag_t;        /* Modify behavior of ha_gs_dispatch */

typedef enum
{
    HA_GS_RESPONSIVENESS_NOTIFICATION,  /* 0 */
    HA_GS_QUERY_NOTIFICATION,           /* 1 */
    HA_GS_DELAYED_ERROR_NOTIFICATION,   /* 2 */
    HA_GS_N_PHASE_NOTIFICATION,         /* 3 */
    HA_GS_APPROVED_NOTIFICATION,        /* 4 */
    HA_GS_REJECTED_NOTIFICATION,        /* 5 */
    HA_GS_ANNOUNCEMENT_NOTIFICATION,    /* 6 */
    HA_GS_SUBSCRIPTION_NOTIFICATION,    /* 7 */
    HA_GS_MERGE_NOTIFICATION,           /* 8 */
    HA_GS_NOTIFICATION_RESERVED_1 = 99  /* 99 */
} ha_gs_notification_type_t;            /* Identify types of notifications */

typedef enum
{
    HA_GS_RESPONSIVENESS,                       /* 0 */
    HA_GS_JOIN,                                 /* 1 */
    HA_GS_FAILURE_LEAVE,                        /* 2 */
    HA_GS_LEAVE,                                /* 3 */
    HA_GS_EXPEL,                                /* 4 */
    HA_GS_STATE_VALUE_CHANGE,                   /* 5 */
    HA_GS_PROVIDER_MESSAGE,                     /* 6 */
    HA_GS_CAST_OUT,                             /* 7 */
    HA_GS_SOURCE_STATE_REFLECTION,              /* 8 */
    HA_GS_MERGE,                                /* 9 */
    HA_GS_SUBSCRIPTION,                         /* 10 */
    HA_GS_GROUP_ATTRIBUTE_CHANGE,               /* 11 */
    MAX_REQUEST = HA_GS_GROUP_ATTRIBUTE_CHANGE, /* 11 */
    HA_GS_REQ_RESERVED_1 = 99                   /* 99 */
} ha_gs_request_t;              /* Type of request a notification was for */

typedef enum
{
    HA_GS_NO_RESPONSIVENESS,
    HA_GS_PING_RESPONSIVENESS,
    HA_GS_COUNTER_RESPONSIVENESS
} ha_gs_responsiveness_type_t;          /* Type of responsiveness checking */

typedef enum
{
    HA_GS_NO_CHANGE                     = 0x0000, /* 0 */
    HA_GS_PROPOSED_MEMBERSHIP           = 0x0001, /* 1 */
    HA_GS_ONGOING_MEMBERSHIP            = 0x0002, /* 2 */
    HA_GS_PROPOSED_STATE_VALUE          = 0x0004, /* 4 */
    HA_GS_ONGOING_STATE_VALUE           = 0x0008, /* 8 */
    HA_GS_UPDATED_PROVIDER_MESSAGE      = 0x0010, /* 16 */
    HA_GS_UPDATED_MEMBERSHIP            = 0x0020, /* 32 */
    HA_GS_REJECTED_MEMBERSHIP           = 0x0040, /* 64 */
    HA_GS_UPDATED_STATE_VALUE           = 0x0080, /* 128 */
    HA_GS_REFLECTED_SOURCE_STATE_VALUE  = 0x0100, /* 256 */
    HA_GS_EXPEL_INFORMATION             = 0x0200, /* 512 */
    HA_GS_PROPOSED_GROUP_ATTRIBUTES     = 0x0400, /* 1024 */
    HA_GS_ONGOING_GROUP_ATTRIBUTES      = 0x0800, /* 2048 */
    HA_GS_UPDATED_GROUP_ATTRIBUTES      = 0x1000, /* 4096 */
    HA_GS_REJECTED_GROUP_ATTRIBUTES     = 0x2000  /* 8192 */
} ha_gs_updates_t;                                /* Whats Changed */
```

```
typedef enum
{
    HA_GS_MIN_SUMMARY_CODE                      = 0x0001, /* 1 */
    HA_GS_EXPLICIT_APPROVE                      = 0x0001, /* 1 */
    HA_GS_EXPLICIT_REJECT                       = 0x0002, /* 2 */
    HA_GS_DEFAULT_APPROVE                      = 0x0004, /* 4 */
    HA_GS_DEFAULT_REJECT                        = 0x0008, /* 8 */
    HA_GS_TIME_LIMIT_EXCEEDED                   = 0x0010, /* 16 */
    HA_GS_PROVIDER_FAILED                       = 0x0020, /* 32 */
    HA_GS_RESPONSIVENESS_NO_RESPONSE            = 0x0040, /* 64 */
    HA_GS_RESPONSIVENESS_RESPONSE               = 0x0080, /* 128 */
    HA_GS_GROUP_DISSOLVED                       = 0x0100, /* 256 */
    HA_GS_GROUP_SERVICES_HAS_DIED_HORRIBLY      = 0x0200, /* 512 */
    HA_GS_DEACTIVATE_UNSUCCESSFUL               = 0x0400, /* 1024 */
    HA_GS_DEACTIVATE_TIME_LIMIT_EXCEEDED        = 0x0800, /* 2048 */
    HA_GS_GROUP_ATTRIBUTES_CHANGED              = 0x1000, /* 4096 */
    HA_GS_MAX_SUMMARY_CODE                      = 0x1000  /* 4096 */
} ha_gs_summary_code_t;         /* Notification summary */

typedef enum
{
    HA_GS_CALLBACK_NOT_OK,
    HA_GS_CALLBACK_OK
} ha_gs_callback_rc_t;          /* Callback Return Codes */

typedef enum
{
    HA_GS_VOLUNTARY_LEAVE       = 0x0001, /* 1 */
    HA_GS_PROVIDER_FAILURE      = 0x0002, /* 2 */
    HA_GS_HOST_FAILURE          = 0x0004, /* 4 */
    HA_GS_PROVIDER_EXPELLED     = 0x0008, /* 8 */
    HA_GS_SOURCE_PROVIDER_LEAVE = 0x0010, /* 16 */
    HA_GS_PROVIDER_SAID_GOODBYE = 0x0020  /* 32 */
} ha_gs_leave_reasons_t;

typedef enum
{
    HA_GS_QUERY_ALL,
    HA_GS_QUERY_GROUP
} ha_gs_query_type_t;

typedef enum
{
    HA_GS_SUBSCRIBE_STATE               = 0x01,
    HA_GS_SUBSCRIBE_DELTA_JOINS         = 0x02,
    HA_GS_SUBSCRIBE_DELTA_LEAVES        = 0x04,
    HA_GS_SUBSCRIBE_DELTAS_ONLY         = 0x06,
    HA_GS_SUBSCRIBE_MEMBERSHIP          = 0x08,
    HA_GS_SUBSCRIBE_ALL_MEMBERSHIP      = 0x0e,
    HA_GS_SUBSCRIBE_STATE_AND_MEMBERSHIP= 0x0f
    } ha_gs_subscription_ctrl_t;
typedef enum
{
    HA_GS_SUBSCRIPTION_STATE            = 0x01, /* 1 */
    HA_GS_SUBSCRIPTION_DELTA_JOIN       = 0x02, /* 2 */
    HA_GS_SUBSCRIPTION_DELTA_LEAVE      = 0x04, /* 4 */
    HA_GS_SUBSCRIPTION_MEMBERSHIP       = 0x08, /* 8 */
    HA_GS_SUBSCRIPTION_SPECIAL_DATA     = 0x40, /* 64 */
    HA_GS_SUBSCRIPTION_DISSOLVED        = 0x80, /* 128 */
    HA_GS_SUBSCRIPTION_GS_HAS_DIED      = 0x100 /* 256 */
} ha_gs_subscription_type_t;
```

```
typedef int ha_gs_token_t;
typedef int ha_gs_descriptor_t;
typedef unsigned short ha_gs_time_limit_t;

#define HA_GS_MAX_GROUP_NAME_LENGTH     32
typedef char *ha_gs_group_name_t;

/* Use this name to subscribe to processor membership. */
#define HA_GS_HOST_MEMBERSHIP_GROUP "HostMembership"

#define HA_GS_ENET_MEMBERSHIP_GROUP "enMembership"
#define HA_GS_CSS_MEMBERSHIP_GROUP "cssMembership"
#define HA_GS_CSSRAW_MEMBERSHIP_GROUP "cssRawMembership"
#define HA_GS_TOKENRING_MEMBERSHIP_GROUP "trMembership"
#define HA_GS_FDDI_MEMBERSHIP_GROUP "fddiMembership"
#define HA_GS_RS232_MEMBERSHIP_GROUP "rs232Membership"
#define HA_GS_TMSCSI_MEMBERSHIP_GROUP "tmscsiMembership"
#define HA_GS_TMSSA_MEMBERSHIP_GROUP "tmssaMembership"
#define HA_GS_SLIP_MEMBERSHIP_GROUP "slipMembership"
#define HA_GS_ATM_MEMBERSHIP_GROUP "atmMembership"

typedef struct
{
    short               gs_version;
    short               gs_sizeof_group_attributes;
    short               gs_client_version;
    ha_gs_batch_ctrl_t  gs_batch_control;
    ha_gs_num_phases_t  gs_num_phases;
    ha_gs_num_phases_t  gs_source_reflection_num_phases;
    ha_gs_vote_value_t  gs_group_default_vote;
    ha_gs_merge_ctrl_t  gs_merge_control;
    ha_gs_time_limit_t  gs_time_limit;
    ha_gs_time_limit_t  gs_source_reflection_time_limit;
    ha_gs_group_name_t  gs_group_name;
    ha_gs_group_name_t  gs_source_group_name;
} ha_gs_group_attributes_t;             /* Identify Group Attributes */

const   short   HA_GS_node_number = -1;
const   short   HA_GS_instance_number = -1;

#define gs_node_number _gs_provider_info._gs_node_number
#define gs_instance_number _gs_provider_info._gs_instance_number

typedef union
{
    struct
    {
        short   _gs_instance_number;
        short   _gs_node_number;
    } _gs_provider_info;
    int gs_provider_id;
} ha_gs_provider_t;                     /* Provider ID */


typedef struct
{
    int         gs_length;
    char        *gs_state;
} ha_gs_state_value_t;          /* State Vector */

typedef struct
{
    short               gs_version;
    ha_gs_state_value_t gs_group_state_value;
} ha_gs_group_state_t;          /* encapsulation of state vector */

typedef struct
{
    int         gs_length;
    char        *gs_message;
} ha_gs_provider_message_t;     /* provider message */

typedef struct
{
    ha_gs_responsiveness_type_t gs_responsiveness_type;
    unsigned int                gs_responsiveness_interval;
    ha_gs_time_limit_t          gs_responsiveness_response_time_limit;
    void                        *gs_counter_location;
    unsigned int                gs_counter_length;
} ha_gs_responsiveness_t;               /* responsiveness attributes */

typedef union
{
    struct {
        ha_gs_state_value_t     *_gs_info_state;
        ha_gs_provider_t        *_gs_info_providers;
    } _gs_group_info;
    ha_gs_group_name_t  gs_groups;
} ha_gs_group_info_t;

#define gs_group_info_state     _gs_group_info._gs_info_state
#define gs_group_info_providers _gs_group_info._gs_info_providers

typedef struct
{
    ha_gs_query_type_t  gs_query_type;
    ha_gs_rc_t          gs_query_return_code;
    int                 gs_number_of_groups;
    ha_gs_group_info_t  *gs_group_info;
} ha_gs_query_info_t;

typedef struct
{
    unsigned int        gs_count;
    ha_gs_provider_t    *gs_providers;
} ha_gs_membership_t;           /* Membership List */

typedef struct {
    int                 gs_deactivate_phase;
    int                 gs_expel_flag_length;
    char                *gs_expel_flag;
} ha_gs_expel_info_t;

typedef struct
{
    unsigned int        gs_voluntary_or_failure;
    unsigned int        gs_voluntary_leave_code;
} ha_gs_leave_info_t;

typedef struct
{
    unsigned int        gs_count;
    ha_gs_leave_info_t  *gs_leave_codes;
} ha_gs_leave_array_t;

typedef struct {
    unsigned short      gs_num_phases;
    unsigned short      gs_phase_number;
} ha_gs_phase_info_t;

typedef enum {
    HA_GS_ADAPTER_DEATH_ARRAY   = 0x01,
    HA_GS_CURRENT_ADAPTER_ALIAS_ARRAY   = 0x02,
    HA_GS_CHANGING_ADAPTER_ALIAS_ARRAY  = 0x04
} ha_gs_subscription_special_type_t;

typedef enum
{
    HA_GS_ADAPTER_DEAD          = 0x0001,
    HA_GS_ADAPTER_REMOVED       = 0x0002
} ha_gs_adapter_death_t;

typedef struct  {
    int                 gs_length;
    unsigned int        gs_flag;
    void                *gs_special_data;
} ha_gs_special_data_t;

typedef struct ha_gs_special_block_t {
    unsigned int        gs_special_flag;
    struct ha_gs_special_block_t        *gs_next_special_block;
    int                 gs_special_num_entries;
    int                 gs_special_length;
    void                *gs_special;
} ha_gs_special_block_t;

typedef struct
{
    ha_gs_phase_info_t          gs_phase_info;
    ha_gs_provider_t            gs_proposed_by;
    ha_gs_updates_t             gs_whats_changed;
    ha_gs_membership_t          *gs_current_providers;
    ha_gs_membership_t          *gs_changing_providers;
    ha_gs_leave_array_t         *gs_leave_info;
    ha_gs_expel_info_t          *gs_expel_info;
    ha_gs_state_value_t         *gs_current_state_value;
    ha_gs_state_value_t         *gs_proposed_state_value;
    ha_gs_state_value_t         *gs_source_state_value;
    ha_gs_provider_message_t    *gs_provider_message;
    ha_gs_group_attributes_t    *gs_new_group_attributes;
} ha_gs_proposal_t;

typedef struct
{
    ha_gs_notification_type_t   gs_notification_type;
    ha_gs_responsiveness_t      gs_responsiveness_information;
} ha_gs_responsiveness_notification_t;

typedef struct
{
    ha_gs_notification_type_t   gs_notification_type;
    unsigned int                gs_number_of_queries;
    ha_gs_query_info_t          *gs_query_info;
} ha_gs_query_notification_t;

typedef struct
{
    ha_gs_notification_type_t   gs_notification_type;
    ha_gs_token_t               gs_provider_token;
    ha_gs_request_t             gs_protocol_type;
    ha_gs_summary_code_t        gs_summary_code;
    ha_gs_time_limit_t          gs_time_limit;
    ha_gs_proposal_t            *gs_proposal;
} ha_gs_n_phase_notification_t;

typedef struct
{
    ha_gs_notification_type_t   gs_notification_type;
    ha_gs_token_t               gs_provider_token;
    ha_gs_request_t             gs_protocol_type;
    ha_gs_summary_code_t        gs_summary_code;
    ha_gs_proposal_t            *gs_proposal;
} ha_gs_approved_notification_t;

typedef struct
{
    ha_gs_notification_type_t   gs_notification_type;
    ha_gs_token_t               gs_provider_token;
    ha_gs_request_t             gs_protocol_type;
    ha_gs_summary_code_t        gs_summary_code;
    ha_gs_proposal_t            *gs_proposal;
} ha_gs_rejected_notification_t;

typedef struct
{
    ha_gs_notification_type_t   gs_notification_type;
    ha_gs_token_t               gs_provider_token;
    ha_gs_summary_code_t        gs_summary_code;
    ha_gs_membership_t          *gs_announcement;
} ha_gs_announcement_notification_t;

typedef struct
{
    ha_gs_notification_type_t   gs_notification_type;
    ha_gs_token_t               gs_provider_token;
    ha_gs_request_t             gs_protocol_type;
    ha_gs_proposal_t            *gs_proposal;
    ha_gs_merge_ctrl_t          gs_merge_control;
    ha_gs_group_state_t         gs_alpha_group_state;
    ha_gs_group_state_t         gs_omega_group_state;
} ha_gs_merge_notification_t;

typedef struct
{
    ha_gs_notification_type_t   gs_notification_type;
    ha_gs_token_t               gs_subscriber_token;
    ha_gs_subscription_type_t   gs_subscription_type;
    ha_gs_state_value_t         *gs_state_value;
    ha_gs_membership_t          *gs_full_membership;
    ha_gs_membership_t          *gs_changing_membership;
    ha_gs_special_data_t        *gs_subscription_special_data;
} ha_gs_subscription_notification_t;

typedef void (ha_gs_subscription_cb_t)(const ha_gs_subscription_notification_t*);

typedef void (ha_gs_query_cb_t)(const ha_gs_query_notification_t*);

typedef ha_gs_callback_rc_t (ha_gs_responsiveness_cb_t)(const ha_gs_responsivene
ss_notification_t*);

typedef void (ha_gs_n_phase_cb_t)(const ha_gs_n_phase_notification_t*);
typedef void (ha_gs_approved_cb_t)(const ha_gs_approved_notification_t*);

typedef void (ha_gs_rejected_cb_t)(const ha_gs_rejected_notification_t*);

typedef void (ha_gs_announcement_cb_t)(const ha_gs_announcement_notification_t*);

typedef void (ha_gs_merge_cb_t)(const ha_gs_merge_notification_t*);

typedef struct {
    ha_gs_group_attributes_t    *gs_group_attributes;
    short                       gs_provider_instance;
    char                        *gs_provider_local_name;
    ha_gs_n_phase_cb_t          *gs_n_phase_protocol_callback;
    ha_gs_approved_cb_t         *gs_protocol_approved_callback;
    ha_gs_rejected_cb_t         *gs_protocol_rejected_callback;
    ha_gs_announcement_cb_t     *gs_announcement_callback;
    ha_gs_merge_cb_t            *gs_merge_callback;
} ha_gs_join_request_t;

typedef struct {
    ha_gs_num_phases_t          gs_num_phases;
    ha_gs_time_limit_t          gs_time_limit;
    ha_gs_state_value_t         gs_new_state;
} ha_gs_state_change_request_t;

typedef struct {
    ha_gs_num_phases_t          gs_num_phases;
    ha_gs_time_limit_t          gs_time_limit;
    ha_gs_provider_message_t    gs_message;
} ha_gs_message_request_t;

typedef struct {
    ha_gs_num_phases_t          gs_num_phases;
    ha_gs_time_limit_t          gs_time_limit;
    unsigned int                gs_leave_code;
} ha_gs_leave_request_t;

typedef struct {
    ha_gs_num_phases_t          gs_num_phases;
    ha_gs_time_limit_t          gs_time_limit;
    ha_gs_membership_t          gs_expel_list;
    int                         gs_deactivate_phase;
    char                        *gs_deactivate_flag;
} ha_gs_expel_request_t;

typedef struct  {
    ha_gs_subscription_ctrl_t   gs_subscription_control;
    ha_gs_group_name_t          gs_subscription_group;
    ha_gs_subscription_cb_t     *gs_subscription_callback;
} ha_gs_subscribe_request_t;

typedef struct {
    ha_gs_num_phases_t          gs_num_phases;
    ha_gs_time_limit_t          gs_time_limit;
    ha_gs_group_attributes_t    *gs_group_attributes;
    ha_gs_membership_t          *gs_backlevel_providers;
} ha_gs_attribute_change_request_t;

#define gs_join_request         _gs_protocol_info._gs_join_request
#define gs_state_change_request _gs_protocol_info._gs_state_change_request
#define gs_message_request      _gs_protocol_info._gs_message_request
#define gs_leave_request        _gs_protocol_info._gs_leave_request
#define gs_expel_request        _gs_protocol_info._gs_expel_request
```

```
#define gs_subscribe_request    _gs_protocol_info._gs_subscribe_request
#define gs_attribute_change_request     _gs_protocol_info._gs_attribute_change_request

typedef struct {
    union {
    ha_gs_join_request_t                _gs_join_request;
    ha_gs_state_change_request_t        _gs_state_change_request;
    ha_gs_message_request_t             _gs_message_request;
    ha_gs_leave_request_t               _gs_leave_request;
    ha_gs_expel_request_t               _gs_expel_request;
    ha_gs_subscribe_request_t           _gs_subscribe_request;
    ha_gs_attribute_change_request_t    _gs_attribute_change_request;
    }   _gs_protocol_info;
} ha_gs_proposal_info_t;

typedef struct
{
    ha_gs_notification_type_t   gs_notification_type;
    ha_gs_token_t               gs_request_token;
    ha_gs_request_t             gs_protocol_type;
    ha_gs_rc_t                  gs_delayed_return_code;
    ha_gs_proposal_info_t       *gs_failing_request;
} ha_gs_delayed_error_notification_t;

typedef void (ha_gs_delayed_error_cb_t)(const ha_gs_delayed_error_notification_t*);

ha_gs_rc_t ha_gs_init(ha_gs_descriptor_t *,
                      const ha_gs_socket_ctrl_t,
                      const ha_gs_responsiveness_t *,
                      const char *,
                      ha_gs_responsiveness_cb_t*,
                      ha_gs_delayed_error_cb_t*,
                      ha_gs_query_cb_t*);

ha_gs_rc_t ha_gs_dispatch(const ha_gs_dispatch_flag_t);
ha_gs_rc_t ha_gs_join(ha_gs_token_t *,
                      const ha_gs_proposal_info_t *);

ha_gs_rc_t ha_gs_change_state_value(ha_gs_token_t,
                                    const ha_gs_proposal_info_t *);

ha_gs_rc_t ha_gs_send_message(ha_gs_token_t,
                                       const ha_gs_proposal_info_t *);

ha_gs_rc_t ha_gs_leave(ha_gs_token_t,
                       const ha_gs_proposal_info_t *);

ha_gs_rc_t ha_gs_expel(ha_gs_token_t,
                       const ha_gs_proposal_info_t *);

ha_gs_rc_t ha_gs_change_attributes(ha_gs_token_t,
                                   const ha_gs_proposal_info_t *);

ha_gs_rc_t ha_gs_goodbye(ha_gs_token_t);

ha_gs_rc_t ha_gs_vote(ha_gs_token_t,
                      ha_gs_vote_value_t,
                      const ha_gs_state_value_t *,
                      const ha_gs_provider_message_t *,
                      ha_gs_vote_value_t);

ha_gs_rc_t ha_gs_quit(void);
ha_gs_rc_t ha_gs_query_group_list(void);

ha_gs_rc_t ha_gs_query_group_info(const ha_gs_group_name_t);

ha_gs_rc_t ha_gs_subscribe(ha_gs_token_t *,
                           const ha_gs_proposal_info_t *);

ha_gs_rc_t ha_gs_unsubscribe(ha_gs_token_t);

void    ha_gs_copy_group_attributes(ha_gs_group_attributes_t *gAttrsTarg,
                                    ha_gs_group_attributes_t *gAttrsSrc);

ha_gs_rc_t ha_gs_get_ffdc_id(fc_eid_t fcid);

#ifdef __cplusplus
}                                       /* end extern "C" */
#endif

#endif                                  /* _HA_GS_H_ */
```

Appendix C. Recoverable Network File System programs

This appendix provides you with all the programs described in Chapter 9, “Recoverable Network File System” on page 161.

You can download the source codes and executable modules of these programs from the IBM Redbooks Web server. For more information, refer to Appendix D, “Using the additional material” on page 279.

C.1 rnfs.c

```
/**************************************
*
* RSCT Group Services:
* Programming Cluster Applications
*
* SG24-5523-00
*
* Recoverable Network File System
*
* rnfs.c
*
**************************************/

/**************************************
* header files
**************************************/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/select.h>
#include <ha_gs.h>

/**************************************
* definitions
**************************************/
void init_program(void);
void propose_join(void);
void propose_state(void);
void propose_message(void);
void propose_leave(void);
void quit_program(void);
void suspend_program(void);
ha_gs_callback_rc_t ha_gs_responsiveness_callback(
   const ha_gs_responsiveness_notification_t *);
void ha_gs_n_phase_callback(
   const ha_gs_n_phase_notification_t *);
void ha_gs_protocol_approved_callback(
   const ha_gs_approved_notification_t *);
void ha_gs_protocol_rejected_callback(
   const ha_gs_rejected_notification_t *);
void ha_gs_announcement_callback(
   const ha_gs_announcement_notification_t *);
void ha_gs_delayed_error_callback(
   const ha_gs_delayed_error_notification_t *);
void vote_protocol(ha_gs_vote_value_t, const ha_gs_state_value_t *);
void print_nodes(ha_gs_membership_t *);

typedef enum {
   RNFS_CLIENT,
   RNFS_SERVER
} rnfs_ima_t;
typedef enum {
   RNFS_JOINING,
   RNFS_STABLE,
   RNFS_UNSTABLE,
   RNFS_LEAVING
} rnfs_imdoing_t;

#define RNFS_RESPONSE_RATE            2
#define RNFS_RESPONSE_TIME_LIMIT      1
#define RNFS_REPLICATE_RATE          10
#define RNFS_REPLICATE_TIME_LIMIT     5
#define RNFS_SHUTDOWN_TIME_LIMIT      5
#define RNFS_JOIN_FAILURE_TIME_LIMIT  5
#define RNFS_TAKEOVER_TIME_LIMIT      5

#define RNFS_GROUP_NAME   "rnfs_group"
#define RNFS_INSTANCE_NUM 5523
#define RNFS_LOCAL_NAME   "rNFS"

#define RNFS_MESSAGE      "replicate file system\0"
#define RNFS_DEACTIVATE   "./rnfs_deact.ksh"
#define RNFS_REPLICATE    "./rnfs_replicate"
#define RNFS_UMOUNT       "./rnfs_umount"
#define RNFS_MOUNT        "./rnfs_mount "

/**************************************
* global variables
**************************************/
int                      rc;
ha_gs_rc_t               gs_rc;

rnfs_ima_t               ima;
rnfs_imdoing_t           imdoing;
ha_gs_request_t          latest_protocol;

short                    mynodeis;
short                    serveris;

char                     domain_name[256];

ha_gs_descriptor_t       descriptor;
ha_gs_responsiveness_t   responsiveness;
ha_gs_token_t            provider_token;
ha_gs_proposal_info_t    proposal_info;
ha_gs_group_attributes_t group_attributes;

/**************************************
* main
**************************************/
int main(int argc, char **argv) {
   char           key;
   fd_set         my_fd;
   struct timeval timeout;
   int            replicate;

   if(argc != 2) {
      printf("Usage: %s domain_name\n", argv[0]);
      exit(argc);
   }
   strcpy(domain_name, "HA_DOMAIN_NAME=");
   strcat(domain_name, argv[1]);
   putenv(domain_name);
   printf("domain name: %s, ", getenv("HA_DOMAIN_NAME"));
   printf("group name: %s, ", RNFS_GROUP_NAME);
   printf("instance number: %d\n", RNFS_INSTANCE_NUM);

   replicate       = 0;
   ima             = RNFS_CLIENT;
   descriptor      = 0;
   timeout.tv_sec  = 1;
   timeout.tv_usec = 0;

   init_program();
   propose_join();

   printf("hit <Enter> key to suspend\n");
   for(;;) {
      FD_ZERO(&my_fd);
      FD_SET(0, &my_fd);
      FD_SET(descriptor, &my_fd);
      rc = select(descriptor + 1, &my_fd, NULL, NULL, &timeout);
      if(rc < 0) {
         printf("*** select failed rc=%d ***\n", rc);
         exit(rc);
      }
      if(FD_ISSET(0, &my_fd)) {
         suspend_program();
      }
      if(descriptor && FD_ISSET(descriptor, &my_fd)) {
         gs_rc = ha_gs_dispatch(HA_GS_NON_BLOCKING);
         if(gs_rc != HA_GS_OK) {
            printf("*** ha_gs_dispatch failed rc=%d ***\n", gs_rc);
         }
      }
      if(ima == RNFS_SERVER) {
         if((imdoing != RNFS_LEAVING) && (replicate > RNFS_REPLICATE_RATE)) {
            propose_message();
            replicate = 0;
         } else {
            replicate++;
         }
      }
   }
}

/***************************************
* suspend_program
***************************************/
void suspend_program() {
   char proposal[32];

   gets(proposal); /* remove previously input strings */
   printf("[ program suspended ] l(eave) or r(esume)?: ");
   scanf("%s", proposal);
   switch((int)proposal[0]) {
   case 'l': case 'L':
      propose_leave();
      break;
   default:
      break;
   }
   gets(proposal); /* remove extra strings */
   return;
}

/*************************************
* init_program (ha_gs_init)
*************************************/
void init_program() {

   responsiveness.gs_responsiveness_type     = HA_GS_PING_RESPONSIVENESS;
   responsiveness.gs_responsiveness_interval = RNFS_RESPONSE_RATE;
   responsiveness.gs_responsiveness_response_time_limit = RNFS_RESPONSE_TIME_LIMIT;
   responsiveness.gs_counter_location        = NULL;
   responsiveness.gs_counter_length          = NULL;

   gs_rc = ha_gs_init(
      &descriptor,
      HA_GS_SOCKET_NO_SIGNAL,
      &responsiveness,
      RNFS_DEACTIVATE,
      ha_gs_responsiveness_callback,
      ha_gs_delayed_error_callback,
      NULL);
   if(gs_rc != HA_GS_OK) {
      printf("*** ha_gs_init failed rc=%d ***\n", gs_rc);
      exit(-1);
   }
   return;
}

/*************************************
* propose_join (ha_gs_join)
*************************************/
void propose_join() {

   proposal_info.gs_join_request.gs_group_attributes    = &group_attributes;
   proposal_info.gs_join_request.gs_provider_instance   = RNFS_INSTANCE_NUM;
   proposal_info.gs_join_request.gs_provider_local_name = RNFS_LOCAL_NAME;
   proposal_info.gs_join_request.gs_n_phase_protocol_callback
      = ha_gs_n_phase_callback;
   proposal_info.gs_join_request.gs_protocol_approved_callback
      = ha_gs_protocol_approved_callback;
   proposal_info.gs_join_request.gs_protocol_rejected_callback
      = ha_gs_protocol_rejected_callback ;
   proposal_info.gs_join_request.gs_announcement_callback
      = ha_gs_announcement_callback;
   proposal_info.gs_join_request.gs_merge_callback      = NULL;

   group_attributes.gs_version                        = 1;
   group_attributes.gs_sizeof_group_attributes
      = sizeof(ha_gs_group_attributes_t);
   group_attributes.gs_client_version                 = 1;
   group_attributes.gs_batch_control
      = HA_GS_NO_BATCHING | HA_GS_DEACTIVATE_ON_FAILURE;
   group_attributes.gs_num_phases                     = HA_GS_N_PHASE;
   group_attributes.gs_source_reflection_num_phases = HA_GS_1_PHASE;
   group_attributes.gs_group_default_vote           = HA_GS_VOTE_APPROVE;
   group_attributes.gs_merge_control                = HA_GS_DISSOLVE_MERGE;
   group_attributes.gs_time_limit                   = RNFS_JOIN_FAILURE_TIME_LIMIT;
   group_attributes.gs_source_reflection_time_limit = NULL;
   group_attributes.gs_group_name                   = RNFS_GROUP_NAME;
   group_attributes.gs_source_group_name            = NULL;
   imdoing = RNFS_JOINING;

   gs_rc = ha_gs_join(
      &provider_token,
      &proposal_info);
   if(gs_rc != HA_GS_OK) {
      printf("*** ha_gs_join failed rc=%d **\n", gs_rc);
   }
   return;
}

/**************************************
* propose_state (ha_gs_chage_state_value)
**************************************/
void propose_state() {
   char hostname[256];
   int  collide;

   if(gethostname(hostname, 256)) {
      printf("*** gethostname failed ***\n");
   }

   proposal_info.gs_state_change_request.gs_num_phases         = HA_GS_N_PHASE;
   proposal_info.gs_state_change_request.gs_time_limit
      = RNFS_TAKEOVER_TIME_LIMIT;
   proposal_info.gs_state_change_request.gs_new_state.gs_length
      = strlen(hostname) + 1;
   proposal_info.gs_state_change_request.gs_new_state.gs_state = hostname;

   for(; collide;) {
      gs_rc = ha_gs_change_state_value(
         provider_token,
         &proposal_info);
      if(gs_rc != HA_GS_OK) {
         if(gs_rc == HA_GS_COLLIDE) {
            printf("* warning * hostname registration is canceled - retry\n");
            collide = 1;
         } else {
            printf("*** ha_gs_change_state_value failed rc=%d ***\n", gs_rc);
            collide = 0;
         }
      } else {
         collide = 0;
      }
   }
   return;
}

/**************************************
* propose_message (ha_gs_send_message)
**************************************/
void propose_message() {
   char                     message[2048];

   strcpy(message, RNFS_MESSAGE);

   proposal_info.gs_message_request.gs_num_phases         = HA_GS_N_PHASE;
   proposal_info.gs_message_request.gs_time_limit         = RNFS_REPLICATE_TIME_LIMIT;
   proposal_info.gs_message_request.gs_message.gs_length  = strlen(message) + 1;
   proposal_info.gs_message_request.gs_message.gs_message = message;

   gs_rc = ha_gs_send_message(
      provider_token,
      &proposal_info);
   if(gs_rc != HA_GS_OK) {
      if(gs_rc == HA_GS_COLLIDE) {
         printf("* warning * replication is canceled\n");
      } else {
         printf("*** ha_gs_send_message failed rc=%d ***\n", gs_rc);
      }
   }
   return;
}

/**************************************
* propose_leave (ha_gs_leave)
**************************************/
void propose_leave() {

   proposal_info.gs_leave_request.gs_num_phases = HA_GS_N_PHASE;
   proposal_info.gs_leave_request.gs_time_limit = RNFS_SHUTDOWN_TIME_LIMIT;
   proposal_info.gs_leave_request.gs_leave_code = ima;

   imdoing = RNFS_LEAVING;

   gs_rc = ha_gs_leave(
      provider_token,
      &proposal_info);
   if(gs_rc != HA_GS_OK) {
      if(gs_rc == HA_GS_COLLIDE) {
         printf("* warning * leaving the group is canceled\n");
      } else {
         printf("*** ha_gs_leave failed rc=%d ***\n", gs_rc);
      }
   }
   return;
}

/**************************************
* quit_program (ha_gs_quit)
**************************************/
void quit_program() {

   gs_rc = ha_gs_quit();
   if (gs_rc != HA_GS_OK) {
      printf("*** ha_gs_quit failed rc=%d ***\n", gs_rc);
   } else {
      printf("[ server takeover ] umount network file system\n");
      if(rc = system(RNFS_UMOUNT)) {
         printf("\n*** system failed rc=%d ***\n", rc);
      }
      exit(0);
   }
   return;
}

/**************************************
* responsiveness notification
**************************************/
ha_gs_callback_rc_t ha_gs_responsiveness_callback(
   const ha_gs_responsiveness_notification_t *block) {

   return(HA_GS_CALLBACK_OK);
}
```

```
/*************************************
* n-phasse notification
*************************************/
void ha_gs_n_phase_callback(
   const ha_gs_n_phase_notification_t *block) {
   char                hostname[256];
   ha_gs_state_value_t host_name;
   char                mount_command[64];

   switch(latest_protocol = block->gs_protocol_type) {
   case HA_GS_JOIN:
      if(block->gs_proposal->gs_current_providers->gs_count == 0) {
         if(gethostname(hostname, 256)) {
            printf("*** gethostname failed ***\n");
         }
         host_name.gs_length = strlen(hostname) + 1;
         host_name.gs_state = hostname;
         vote_protocol(HA_GS_VOTE_APPROVE, &host_name);
      } else if(imdoing == RNFS_UNSTABLE) {
         vote_protocol(HA_GS_VOTE_REJECT, NULL);
      } else {
         vote_protocol(HA_GS_VOTE_APPROVE, NULL);
      }
      break;
   case HA_GS_FAILURE_LEAVE:
      if(ima != RNFS_SERVER) {
         if(serveris ==
            block->gs_proposal->gs_changing_providers->gs_providers->gs_node_number) {
            printf("[ server failure ] umount network file system\n");
            if(rc = system(RNFS_UMOUNT)) {
               printf("\n*** system failed rc=%d ***\n", rc);
            }
            imdoing = RNFS_UNSTABLE;
         }
      }
      vote_protocol(HA_GS_VOTE_APPROVE, NULL);
      break;
   case HA_GS_LEAVE:
      if(imdoing != RNFS_LEAVING) {
         if(block->gs_proposal->gs_leave_info->gs_leave_codes->gs_voluntary_leave_code
            == RNFS_SERVER) {
            printf("[ server shutdown ] replicate file system\n");
            if(rc = system(RNFS_REPLICATE)) {
               printf("\n*** system failed rc=%d ***\n", rc);
            }
            printf("[ server shutdown ] umount network file system\n");
            if(rc = system(RNFS_UMOUNT)) {
               printf("\n*** system failed rc=%d ***\n", rc);
            }
            imdoing = RNFS_UNSTABLE;
         }
         vote_protocol(HA_GS_VOTE_APPROVE, NULL);
      } else {
         quit_program();
      }
      break;
   case HA_GS_STATE_VALUE_CHANGE:
      printf("[ server takeover ] mount network file system from %s\n",
         block->gs_proposal->gs_proposed_state_value->gs_state);
      strcpy(mount_command, RNFS_MOUNT);
      strcat(mount_command,
         block->gs_proposal->gs_proposed_state_value->gs_state);
      if(rc = system(mount_command)) {
         printf("\n*** system failed rc=%d ***\n", rc);
      }
      serveris = block->gs_proposal->gs_proposed_by.gs_node_number;
      vote_protocol(HA_GS_VOTE_APPROVE, NULL);
      break;
   case HA_GS_PROVIDER_MESSAGE:
      if(strcmp(RNFS_MESSAGE,
         block->gs_proposal->gs_provider_message->gs_message)) {
         printf("*** provider-broadcast message is not expected ***\n");
         break;
      }
      if(ima == RNFS_CLIENT) {
         printf("[ replicate ] replicate file system\n");
         if(rc = system(RNFS_REPLICATE)) {
            printf("\n*** system failed rc=%d ***\n", rc);
         }
      }
      vote_protocol(HA_GS_VOTE_APPROVE, NULL);
      break;
   default:
      printf("*** n-phase notificaiton is not expected ***\n");
      break;
   }
   return;
}

/***************************************
* protocol approved notification
***************************************/
void ha_gs_protocol_approved_callback(
   const ha_gs_approved_notification_t *block) {
   char mount_command[64];

   switch(block->gs_protocol_type) {
   case HA_GS_JOIN:
      if(imdoing == RNFS_JOINING) {
         mynodeis = block->gs_proposal->gs_proposed_by.gs_node_number;
         serveris =
block->gs_proposal->gs_current_providers->gs_providers->gs_node_number;
         if(mynodeis == serveris) {
            ima = RNFS_SERVER;
            printf("[ joined as server ] ");
         } else {
            ima = RNFS_CLIENT;
            printf("[ joined as client ] ");
         }
         printf("mount network file system from %s\n",
            block->gs_proposal->gs_current_state_value->gs_state);
         strcpy(mount_command, RNFS_MOUNT);
         strcat(mount_command,
            block->gs_proposal->gs_current_state_value->gs_state);
         if(rc = system(mount_command)) {
            printf("\n*** system failed rc=%d ***\n", rc);
         }
         imdoing = RNFS_STABLE;
      }
      break;
   case HA_GS_FAILURE_LEAVE:
      if(ima != RNFS_SERVER) {
         if(mynodeis ==
            block->gs_proposal->gs_current_providers->gs_providers->gs_node_number) {
            ima = RNFS_SERVER;
            propose_state();
         }
      }
      break;
   case HA_GS_LEAVE:
      if(ima != RNFS_SERVER) {
         if(mynodeis ==
            block->gs_proposal->gs_current_providers->gs_providers->gs_node_number) {
            ima = RNFS_SERVER;
            propose_state();
         }
      }
      break;
   case HA_GS_STATE_VALUE_CHANGE:
      imdoing = RNFS_STABLE;
      break;
   case HA_GS_PROVIDER_MESSAGE:
      break;
   default:
      printf("*** protocol approved notification is not expected **\n");
      break;
   }
   return;
}

/****************************************
* protocol rejected notification
****************************************/
void ha_gs_protocol_rejected_callback(
   const ha_gs_rejected_notification_t *block) {

   switch(block->gs_protocol_type) {
   case HA_GS_JOIN:
      if(imdoing == RNFS_JOINING) {
         printf("* warning * the group is unstable, join later - exit\n");
         exit(-1);
      }
      break;
   default:
      printf("*** protocol rejected notification is not expected ***\n");
      break;
   }
   return;
}

/****************************************
* announcement notification
****************************************/
void ha_gs_announcement_callback(
   const ha_gs_announcement_notification_t *block) {

   switch(block->gs_summary_code) {
   case HA_GS_RESPONSIVENESS_NO_RESPONSE:
      printf("* warning * responsiveness check has failed on node ");
      if(imdoing != RNFS_LEAVING) {
         imdoing = RNFS_UNSTABLE;
      }
      break;
   case HA_GS_RESPONSIVENESS_RESPONSE:
      printf("* warning * responsiveness check has recovered on node ");
      if(imdoing != RNFS_LEAVING) {
         imdoing = RNFS_STABLE;
      }
      break;
   case HA_GS_TIME_LIMIT_EXCEEDED:
      switch(latest_protocol) {
      case HA_GS_JOIN:
         printf("* warning * it might be very busy on node ");
         break;
      case HA_GS_FAILURE_LEAVE:
         printf("* warning * umounting file system might have failed on node ");
         break;
      case HA_GS_LEAVE:
         printf("* warning * replicating/umounting file system might have failed on node
");
         break;
      case HA_GS_STATE_VALUE_CHANGE:
         printf("* warning * mounting file system might have failed on node ");
         break;
      case HA_GS_PROVIDER_MESSAGE:
         printf("* warning * replicating file system might have failed on node ");
         break;
      default:
         printf("*** announcement notification is not expected ***\n");
         return;
      }
      break;
   case HA_GS_GROUP_SERVICES_HAS_DIED_HORRIBLY:
      printf("*** Group Services has died ***\n");
      break;
   case HA_GS_GROUP_DISSOLVED:
      printf("*** rnfs_group has dissolved ***\n");
      break;
   default:
      printf("*** announcement notification is not expected ***\n");
      return;
   }
   print_nodes(block->gs_announcement);
   return;
}

/****************************************
* delayed error notification
****************************************/
void ha_gs_delayed_error_callback(
   const ha_gs_delayed_error_notification_t *block) {

   switch(block->gs_protocol_type) {
   case HA_GS_JOIN:
      if(block->gs_delayed_return_code == HA_GS_DUPLICATE_INSTANCE_NUMBER) {
         printf("* warning * another rnfs is running on this node - exit\n");
         exit(-1);
      }
      break;
   case HA_GS_LEAVE:
      if(block->gs_delayed_return_code == HA_GS_COLLIDE) {
         printf("* warning * leaving the group is canceled\n");
      }
      break;
   case HA_GS_STATE_VALUE_CHANGE:
      if(block->gs_delayed_return_code == HA_GS_COLLIDE) {
         printf("* warning * hostname registration is canceled - retry\n");
         propose_state();
      }
      break;
   case HA_GS_PROVIDER_MESSAGE:
      if(block->gs_delayed_return_code == HA_GS_COLLIDE) {
         printf("* warning * replication is canceled\n");
      }
      break;
   default:
      printf("*** delayed error notification is not expected ***\n");
      break;
   }
   return;
}

/**************************************
* vote_protocol
**************************************/
void vote_protocol(
   ha_gs_vote_value_t vote_value,
   const ha_gs_state_value_t *host_name) {

   gs_rc = ha_gs_vote(
      provider_token,
      vote_value,
      host_name,
      NULL,
      HA_GS_NULL_VOTE);
   if(gs_rc != HA_GS_OK) {
      printf("*** ha_gs_vote failed rc=%d ***\n", gs_rc);
   }
   return;
}

/**************************************
* print_nodes
**************************************/
void print_nodes(ha_gs_membership_t *membership_list) {
   int              number_of_nodes;
   ha_gs_provider_t *member;

   member = membership_list->gs_providers;
   for(number_of_nodes = 0;
      number_of_nodes < membership_list->gs_count;
      number_of_nodes++, member++) {
      printf("%d ", member->gs_node_number);
   }
   printf("\n");
   return;
}
```

C.2 rnfs_deact.ksh

```
#!/usr/bin/ksh
#
# RSCT Group Services:
# Programming Cluster Applications
#
# SG24-5523-00
#
# Recoverable Network File System
#
# rnfs_deact.ksh
#

./rnfs_deact.perl $1 $2 $3 $4 $5
```

```
exit $?
```

C.3 rnfs_deact.perl

```
#!/usr/bin/perl
#
# RSCT Group Services:
# Programming Cluster Applications
#
# SG24-5523-00
#
# Recoverable Network File System
#
# rnfs_deact.perl
#

$FILE_SYSTEM = "/shared_nfs";
$LOG_FILE    = "./rnfs.log";

if($#ARGV == 4) {
   open(STDOUT, ">> $LOG_FILE");
   `date >> $LOG_FILE`;
   print "   deactivate script executed by the Group Services\n";
   print "      Process ID:         $ARGV[0]\n";
   print "      Voting Time Limit:  $ARGV[1]\n";
   print "      Failed group:       $ARGV[2]\n";
   print "      Deactivate flag:    $ARGV[3]\n";
   print "      Failed provider(s): $ARGV[4]\n";
   print "   umount $FILE_SYSTEM\n";
   close STDOUT;
} else {
   print "*** the number of parameters is not expected ***\n";
   exit -1;
}

# exit with return code 0

exit 0;
```

C.4 rnfs_mount

```
#! /usr/bin/ksh
#
# RSCT Group Services:
# Programming Cluster Applications
#
# SG24-5523-00
#
# Recoverable Network File System
#
# rnfs_mount
#

date >> ./rnfs.log
print "   mount $1:/local_nfs /shared_nfs" >> ./rnfs.log
```

C.5 rnfs_replicate

```
#! /usr/bin/ksh
#
# RSCT Group Services:
# Programming Cluster Applications
#
# SG24-5523-00
#
# Recoverable Network File System
#
# rnfs_replicate
#

date >> ./rnfs.log
print "   cp -R /shared_nfs/. /local_nfs" >> ./rnfs.log
```

C.6 rnfs_umount

```
#! /usr/bin/ksh
#
# RSCT Group Services:
# Programming Cluster Applications
#
# SG24-5523-00
#
# Recoverable Network File System
#
# rnfs_umount
#

date >> ./rnfs.log
print "   umount /shared_nfs" >> ./rnfs.log
```

C.7 rnfsm.c

```
/**************************************
*
* RSCT Group Services:
* Programming Cluster Applications
*
* SG24-5523-00
*
* Recoverable Network File System
*
* rnfsm.c
*
**************************************/

/**************************************
* header files
**************************************/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/select.h>
#include <ha_gs.h>
```

```
/**************************************
* definitions
**************************************/
void init_program(void);
void propose_subscribe(void);
void propose_unsubscribe(void);
void quit_program(void);
void suspend_program(void);
void ha_gs_subscriber_callback(
   const ha_gs_subscription_notification_t *);
void ha_gs_delayed_error_callback(
   const ha_gs_delayed_error_notification_t *);
void print_members(ha_gs_membership_t *);

#define RNFS_GROUP_NAME "rnfs_group"

/**************************************
* variables
**************************************/
int                      rc;
ha_gs_rc_t               gs_rc;

char                     domain_name[256];

ha_gs_descriptor_t       descriptor;
ha_gs_responsiveness_t   responsiveness;
ha_gs_token_t            subscriber_token;
ha_gs_proposal_info_t    proposal_info;

/**************************************
* main
**************************************/
int main(int argc, char **argv) {
   char           key;
   fd_set         my_fd;
   struct timeval timeout;

   if(argc != 2) {
      printf("Usage: %s domain_name\n", argv[0]);
      exit(argc);
   }
   strcpy(domain_name, "HA_DOMAIN_NAME=");
   strcat(domain_name, argv[1]);
   putenv(domain_name);
   printf("domain name: %s, ", getenv("HA_DOMAIN_NAME"));
   printf("group name: %s\n", RNFS_GROUP_NAME);

   descriptor = 0;
   timeout.tv_sec  = 1;
   timeout.tv_usec = 0;
   init_program();
   propose_subscribe();

   printf("hit <Enter> key to suspend\n");
   for(;;) {
      FD_ZERO(&my_fd);
      FD_SET(0, &my_fd);
      FD_SET(descriptor, &my_fd);
      rc = select(descriptor + 1, &my_fd, NULL, NULL, &timeout);
      if(rc < 0) {
         printf("*** select failed rc=%d ***\n", rc);
         exit(rc);
      }
      if(FD_ISSET(0, &my_fd)) {
         suspend_program();
      }
      if(descriptor && FD_ISSET(descriptor, &my_fd)) {
         gs_rc = ha_gs_dispatch(HA_GS_NON_BLOCKING);
         if(gs_rc != HA_GS_OK) {
            printf("*** ha_gs_dispatch failed rc=%d ***\n", gs_rc);
         }
      }
   }
}

/**************************************
* suspend_program
**************************************/
void suspend_program() {
   char proposal[32];

   gets(proposal); /* remove previously input strings */
   printf("[ program suspended ] u(nsubscribe) or r(esume)?: ");
   scanf("%s", proposal);
   switch((int)proposal[0]) {
   case 'u': case 'U':
      propose_unsubscribe();
      break;
   default:
      break;
   }
   gets(proposal); /* remove extra strings */
   return;
}

/**************************************
* init_program (ha_gs_init)
**************************************/
void init_program() {

   /* responsiveness control block */
   responsiveness.gs_responsiveness_type     = HA_GS_NO_RESPONSIVENESS;
   responsiveness.gs_responsiveness_interval = NULL;
   responsiveness.gs_responsiveness_response_time_limit = NULL;
   responsiveness.gs_counter_location        = NULL;
   responsiveness.gs_counter_length          = NULL;

   gs_rc = ha_gs_init(
      &descriptor,
      HA_GS_SOCKET_NO_SIGNAL,
      &responsiveness,
      NULL,
      NULL,
      ha_gs_delayed_error_callback,
      NULL);
   if(gs_rc != HA_GS_OK) {
      printf("*** ha_gs_init failed rc=%d ***\n", gs_rc);
   }
   return;
}

/**************************************
* propose_subscribe (ha_gs_subacribe)
**************************************/
void propose_subscribe() {
   proposal_info.gs_subscribe_request.gs_subscription_control
      = HA_GS_SUBSCRIBE_STATE_AND_MEMBERSHIP;
   proposal_info.gs_subscribe_request.gs_subscription_group
      = RNFS_GROUP_NAME;
   proposal_info.gs_subscribe_request.gs_subscription_callback
      = ha_gs_subscriber_callback;

   gs_rc = ha_gs_subscribe(
      &subscriber_token,
      &proposal_info);
   if(gs_rc != HA_GS_OK) {
      printf("*** ha_gs_subscribe failed rc=%d **\n", gs_rc);
   }
   return;
}

/***************************************
* propose_unsubscribe (ha_gs_unsubscribe)
***************************************/
void propose_unsubscribe() {

   gs_rc = ha_gs_unsubscribe(
      subscriber_token);
   if(gs_rc != HA_GS_OK) {
      printf("*** ha_gs_unsubscribe failed rc=%d ***\n", gs_rc);
   } else {
      quit_program();
   }
   return;
}

/***************************************
* quit_program (ha_gs_quit)
***************************************/
void quit_program() {

   gs_rc = ha_gs_quit();
   if (gs_rc != HA_GS_OK) {
      printf("*** ha_gs_quit failed rc=%d ***\n", gs_rc);
   } else {
      exit(0);
   }
   return;
}

/***************************************
* subscriber notification
***************************************/
void ha_gs_subscriber_callback(
   const ha_gs_subscription_notification_t *block) {

   printf("\n");
   if(block->gs_subscription_type & HA_GS_SUBSCRIPTION_DISSOLVED) {
      printf("*** no RNFS node is available any more ***\n");
      quit_program();
   } else {
      if(block->gs_subscription_type & HA_GS_SUBSCRIPTION_STATE) {
         printf("server node hostname: %s\n", block->gs_state_value->gs_state);
      }
      if(block->gs_subscription_type & HA_GS_SUBSCRIPTION_MEMBERSHIP) {
         printf("client nodes: ");
         print_members(block->gs_full_membership);
      }
   }
   return;
}

/**************************************
* delayed error notification
**************************************/
void ha_gs_delayed_error_callback(
   const ha_gs_delayed_error_notification_t *block) {

   switch(block->gs_protocol_type) {
   case HA_GS_SUBSCRIPTION:
      if(block->gs_delayed_return_code == HA_GS_UNKNOWN_GROUP) {
         printf("*** no RNFS node is available currently ***\n");
         quit_program();
      }
      break;
   default:
      printf("*** delayed error notification is not expected ***\n");
      break;
   }
   return;
}

/**************************************
* print_members
**************************************/
void print_members(ha_gs_membership_t *membership_list) {
   int              i;
   ha_gs_provider_t *member;

   if(membership_list->gs_count) {
      member = membership_list->gs_providers;
      for(i = 0; i < membership_list->gs_count; i++, member++){
         printf("%d ", member->gs_node_number);
      }
   }
   printf("\n");
   return;
}
```

C.8 makefile

```
#
# RSCT Group Services:
# Programming Cluster Applications
#
# SG24-5523-00
#
# Recoverable Network File System
#
# makefile
#


all:   rnfs rnfsm

rnfs:  rnfs.c
       cc -o rnfs -g -qnofold -DBSD -bloadmap:rnfs.map -lha_gs -lbsd rnfs.c
```

```
rnfsm: rnfsm.c
       cc -o rnfsm -g -qnofold -DBSD -bloadmap:rnfsm.map -lha_gs -lbsd rnfsm.c

clean:
       rm -f rnfs rnfsm *.map core
```

Appendix D. Using the additional material

This redbook contains additional material in the form of Web material. See the following section for instructions on downloading or using this material.

D.1 Downloading the additional material on the Internet

The Web material associated with this redbook is available in softcopy on the Internet from the IBM Redbooks Web server. Point your Web browser to:

`ftp://www.redbooks.ibm.com/redbooks/SG245523`

Alternatively, you can go to the IBM Redbooks Web site at:

`http://www.redbooks.ibm.com/`

Select the **Additional materials** and open the directory that corresponds with the redbook form number.

D.2 Using the Web material

The additional Web material that accompanies this redbook includes the following:

File name	*Description*
sg245523.zip	Code samples (using zip)
sg245523.tar.Z	Code samples (using tar and compress)

The contents of these two files are identical. They use different tools for packing the code samples. Use a zip tool for the sg245523.zip file or use the `tar` and `compress` command for the sg245523.tar.Z file.

Appendix E. Special notices

This publication is intended to help programmers to learn how to use the RSCT Group Services Application Programming Interfaces to develop highly available cluster applications. The information in this publication is not intended as the specification of any programming interfaces that are provided by the RSCT Group Services. See the PUBLICATIONS section of the IBM Programming Announcement for RSCT Group Services for more information about what publications are considered to be product documentation.

References in this publication to IBM products, programs or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program, or service is not intended to state or imply that only IBM's product, program, or service may be used. Any functionally equivalent program that does not infringe any of IBM's intellectual property rights may be used instead of the IBM product, program or service.

Information in this book was developed in conjunction with use of the equipment specified, and is limited in application to those specific hardware and software products and levels.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to the IBM Director of Licensing, IBM Corporation, North Castle Drive, Armonk, NY 10504-1785.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact IBM Corporation, Dept. 600A, Mail Drop 1329, Somers, NY 10589 USA.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The information contained in this document has not been submitted to any formal IBM test and is distributed AS IS. The information about non-IBM ("vendor") products in this manual has been supplied by the vendor and IBM assumes no responsibility for its accuracy or completeness. The use of this information or the implementation of any of these techniques is a customer responsibility and depends on the customer's ability to evaluate and integrate them into the customer's operational environment. While each item may have

AFP	AIX
AS/400	IBM
Micro Channel	Netfinity
RS/6000	SP
System/390	

Appendix F. Related publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this redbook.

F.1 IBM Redbooks

For information on ordering these publications see “How to get IBM Redbooks” on page 287.

- *HACMP Enhanced Scalability Handbook*, SG24-5328
- *RS/6000 SP High Availability Infrastructure*, SG24-4838

F.2 IBM Redbooks collections

Redbooks are also available on the following CD-ROMs. Click the CD-ROMs button at http://www.redbooks.ibm.com/ for information about all the CD-ROMs offered, updates, and formats.

CD-ROM Title	**Collection Kit Number**
System/390 Redbooks Collection	SK2T-2177
Networking and Systems Management Redbooks Collection	SK2T-6022
Transaction Processing and Data Management Redbooks Collection	SK2T-8038
Lotus Redbooks Collection	SK2T-8039
Tivoli Redbooks Collection	SK2T-8044
AS/400 Redbooks Collection	SK2T-2849
Netfinity Hardware and Software Redbooks Collection	SK2T-8046
RS/6000 Redbooks Collection (BkMgr Format)	SK2T-8040
RS/6000 Redbooks Collection (PDF Format)	SK2T-8043
Application Development Redbooks Collection	SK2T-8037
IBM Enterprise Storage and Systems Management Solutions	SK3T-3694

F.3 Other resources

These publications are also relevant as further information sources:

- *HACMP V4.3 AIX: Enhanced Scalability Installation & Administration Guide*, SC23-4284
- *PSSP: Administration Guide*, SA22-7348
- *RSCT: Group Services Programming Guide and Reference*, SA22-7355

How to get IBM Redbooks

This section explains how both customers and IBM employees can find out about IBM Redbooks, redpieces, and CD-ROMs. A form for ordering books and CD-ROMs by fax or e-mail is also provided.

- **Redbooks Web Site** http://www.redbooks.ibm.com/

  Search for, view, download, or order hardcopy/CD-ROM Redbooks from the Redbooks Web site. Also read redpieces and download additional materials (code samples or diskette/CD-ROM images) from this Redbooks site.

  Redpieces are Redbooks in progress; not all Redbooks become redpieces and sometimes just a few chapters will be published this way. The intent is to get the information out much quicker than the formal publishing process allows.

- **E-mail Orders**

  Send orders by e-mail including information from the IBM Redbooks fax order form to:

	e-mail address
In United States or Canada	pubscan@us.ibm.com
Outside North America	Contact information is in the “How to Order” section at this site: http://www.elink.ibmlink.ibm.com/pbl/pbl

- **Telephone Orders**

United States (toll free)	1-800-879-2755
Canada (toll free)	1-800-IBM-4YOU
Outside North America	Country coordinator phone number is in the “How to Order” section at this site: http://www.elink.ibmlink.ibm.com/pbl/pbl

- **Fax Orders**

United States (toll free)	1-800-445-9269
Canada	1-403-267-4455
Outside North America	Fax phone number is in the “How to Order” section at this site: http://www.elink.ibmlink.ibm.com/pbl/pbl

This information was current at the time of publication, but is continually subject to change. The latest information may be found at the Redbooks Web site.

IBM intranet for Employees

IBM employees may register for information on workshops, residencies, and Redbooks by accessing the IBM Intranet Web site at http://w3.itso.ibm.com/ and clicking the ITSO Mailing List button. Look in the Materials repository for workshops, presentations, papers, and Web pages developed and written by the ITSO technical professionals; click the Additional Materials button. Employees may access MyNews at http://w3.ibm.com/ for redbook, residency, and workshop announcements.

IBM Redbooks fax order form

Please send me the following:

Title	Order Number	Quantity

First name	Last name	
Company		
Address		
City	Postal code	Country
Telephone number	Telefax number	VAT number

☐ Invoice to customer number

☐ Credit card number

Credit card expiration date	Card issued to	Signature

We accept American Express, Diners, Eurocard, Master Card, and Visa. Payment by credit card not available in all countries. Signature mandatory for credit card payment.

Glossary

ABI Application Binary Interface.

ACL Access Control List.

AFPA Adaptive Fast Path Architecture.

AH Authentication Header.

ANSI American National Standards Institute.

API Application Programming Interface.

ARP Address Resolution Protocol.

ASRAddress Space Register.

ATM Asynchronous Transfer Mode.

AUI Attached Unit Interface.

AWT Abstract Window Toolkit.

BIND Berkeley Internet Name Daemon.

BOS Base Operating System.

BLOB Binary Large Object.

BSC Binary Synchronous Communications.

CDE Common Desktop Environment.

CDLI Common Data Link Interface.

CD-R CD Recordable.

CE Customer Engineer.

CEC Central Electronics Complex.

CGE Common Graphics Environment.

CHRP Common Hardware Reference Platform.

CISPR International Special Committee on Radio Interference.

CLVM Concurrent LVM.

CMOS Complimentary Metal-Oxide Semiconductor.

COFF Common Object File Format.

CORBA Common Object Request Broker.

CSID Character Set ID.

DAD Duplicate Address Detection.

DASD Direct Access Storage Device.

DBE Double Buffer Extension.

DBCS Double Byte Character Set.

DCE Distributed Computing Environment.

DES Data Encryption Standard.

DHCP Dynamic Host Configuration Protocol.

DIT Directory Information Tree.

DMA Direct Memory Access.

DN Distinguished Name.

DNS Domain Naming System.

DS Differentiated Service.

DSE Diagnostic System Exerciser.

DSMIT Distributed SMIT.

DTE Data Terminating Equipment.

EA Effective Address.

ECC Error Checking and Correcting.

EIA Electronic Industries Association.

EMU European Monetary Union.

EOF End of File.

ESID Effective Segment ID.

ESP Encapsulating Security Payload.

FCAL Fibre Channel Arbitrated Loop.

FCC Federal Communication Commission.

FDDI Fiber Distributed Data Interface.

FDPR Feedback Directed Program Restructuring.

FIFO First In/First Out.

FLASH EPROM Flash Erasable Programmable Read-Only Memory.

FLIH First Level Interrupt Handler.

FRCA Fast Response Cache Architecture.

GAI Graphic Adapter Interface.

GPR General Purpose Register.

GUI Graphical User Interface.

HACMP High Availability Cluster Multi-Processing.

HCON IBM AIX Host Connection Program/6000.

HFT High Function Terminal.

IAR Instruction Address Register.

ICCCM Inter-Client Communications Conventions Manual.

ICE Inter-Client Exchange.

ICElib Inter-Client Exchange library.

ICMP Internet Control Message Protocol.

IETF Internet Engineering Task Force.

IHV Independent Hardware Vendor.

IIOP Internet Inter-ORB Protocol.

IJG Independent JPEG Group.

IKE Internet Key Exchange.

ILS International Language Support.

IM Input Method.

INRIA Institut National de Recherche en Informatique et en Automatique.

IPL Initial Program Load.

IPSec IP Security.

IS Integrated Service.

ISA Industry Standard Architecture.

ISAKMP/Oakley Internet Security Association Management Protocol.

ISNO Interface Specific Network Options.

ISO International Organization for Standardization.

ISV Independent Software Vendor.

ITSO International Technical Support Organization.

I/O Input/Output.

JDBC Java Database Connectivity.

JFC Java Foundation Classes.

JFS Journaled File System.

LAN Local Area Network.

LDAP Lightweight Directory Access Protocol.

LDIF LDAP Directory Interchange Format.

LFT Low Function Terminal.

LID Load ID.

LP Logical Partition.

LPI Lines Per Inch.

LPP Licensed Program Products.

LPR/LPD Line Printer/Line Printer Daemon.

LP64 Long-Pointer 64.

LRU Least Recently Used.

LTG Logical Track Group.

LV Logical Volume.

LVCB Logical Volume Control Block.

LVD Low Voltage Differential.

LVM Logical Volume Manager.

L2 Level 2.

MBCS MultiByte Character Support.

MCA Micro Channel Architecture.

MDI Media Dependent Interface.

MII Media Independent Interface.

MODS Memory Overlay Detection Subsystem.

MP Multiple Processor.

MPOA Multiprotocol Over ATM.

MST Machine State.

NBC Network Buffer Cache.

ND Neighbor Discovery.

NDP Neighbor Discovery Protocol.

NFS Network File System.

NHRP Next Hop Resolution Protocol.

NIM Network Installation Management.

NIS Network Information System.

NL National Language.

NLS National Language Support.

NTF No Trouble Found.

NVRAM Non-Volatile Random Access Memory.

OACK Option Acknowledgment.

ODBC Open DataBase Connectivity.

ODM Object Data Manager.

OEM Original Equipment Manufacturer.

OLTP Online Transaction Processing.

ONC+ Open Network Computing.

OOUI Object-Oriented User Interface.

OSF Open Software Foundation, Inc..

PCI Peripheral Component Interconnect.

PDT Paging Device Table.

PEX PHIGS Extension to X.

PFS Perfect Forward Security.

PHB Processor Host Bridges.

PHY Physical Layer Device.

PID Process ID.

PII Program Integrated Information.

PMTU Path MTU.

PPC PowerPC.

PSE Portable Streams Environment.

PTF Program Temporary Fix.

PV Physical Volume.

QoS Quality of Service.

RAID Redundant Array of Independent Disks.

RAN Remote Asynchronous Node.

RAS Reliability Availability Serviceability.

RDB Relational DataBase.

RDISC ICMP Router Discovery.

RDN Relative Distinguished Name.

RDP Router Discovery Protocol.

RFC Request for Comments.

RIO Remote I/O.

RIP Routing Information Protocol.

RPA RS/6000 Platform Architecture.

RPC Remote Procedure Call.

RPL Remote Program Loader.

RSVP Resource Reservation Protocol.

SA Secure Association.

SACK Selective Acknowledgments.

SBCS Single-Byte Character Support.

SCB Segment Control Block.

SCSI Small Computer System Interface.

SCSI-SE SCSI-Single Ended.

SDRAM Synchronous DRAM.

SE Single Ended.

SGID Set Group ID.

SHLAP Shared Library Assistant Process.

SID Segment ID.

SIT Simple Internet Transition.

SKIP Simple Key Management for IP.

SLB Segment Lookaside Buffer.

SLIH Second Level Interrupt Handler.

SM Session Management.

SMIT System Management Interface Tool.

SMB Server Message Block.

SMP Symmetric Multiprocessor.

SNG Secured Network Gateway.

SP Service Processor.

SPCN System Power Control Network.

SPI Security Parameter Index.

SPM System Performance Measurement.

SPOT Shared Product Object Tree.

SRC System Resource Controller.

SRN Service Request Number.

SSA Serial Storage Architecture.

SSL Secure Socket Layer.

STP Shielded Twisted Pair.

SUID Set User ID.

SVC Supervisor or System Call.

SYNC Synchronization.

TCE Translate Control Entry.

TCP/IP Transmission Control Protocol/Internet Protocol.

TOS Type Of Service.

TTL Time To Live.

UCS Universal Coded Character Set.

UIL User Interface Language.

ULS Universal Language Support.

UP Uni-Processor.

USLA User-Space Loader Assistant.

UTF UCS Transformation Format.

UTM Uniform Transfer Model.

UTP Unshielded Twisted Pair.

VFB Virtual Frame Buffer.

VG Volume Group.

VGDA Volume Group Descriptor Area.

VGSA Volume Group Status Area.

VHDCI Very High Density Cable Interconnect.

VMM Virtual Memory Manager.

VP Virtual Processor.

VPD Vital Product Data.

VPN Virtual Private Network.

VSM Visual System Manager.

WLM Workload Manage.

XCOFF Extended Common Object File Format.

XIE X Image Extension.

XIM X Input Method.

XKB X Keyboard Extension.

XOM X Output Method.

XPM X Pixmap.

XVFB X Virtual Frame Buffer.

Index

IBM Redbooks review

Your feedback is valued by the Redbook authors. In particular we are interested in situations where a Redbook "made the difference" in a task or problem you encountered. Using one of the following methods, **please review the Redbook, addressing value, subject matter, structure, depth and quality as appropriate.**

- Use the online **Contact us** review redbook form found at http://www.redbooks.ibm.com/
- Fax this form to: USA International Access Code + 1 914 432 8264
- Send your comments in an Internet note to redbook@us.ibm.com

Document Number **Redbook Title**	SG24-5523-00 RSCT Group Services: Programming Cluster Applications
Review	
What other subjects would you like to see IBM Redbooks address?	
Please rate your overall satisfaction:	O Very Good O Good O Average O Poor
Please identify yourself as belonging to one of the following groups:	O Customer O Business Partner O Solution Developer O IBM, Lotus or Tivoli Employee O None of the above
Your email address: The data you provide here may be used to provide you with information from IBM or our business partners about our products, services or activities.	 O Please do not use the information collected here for future marketing or promotional contacts or other communications beyond the scope of this transaction.
Questions about IBM's privacy policy?	The following link explains how we protect your personal information. http://www.ibm.com/privacy/yourprivacy/

SG24-5523-00

Printed in the U.S.A.

RSCT Group Services: Programming Cluster Applications

SG24-5523-00