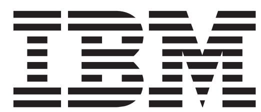
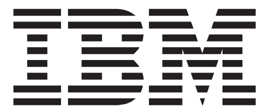
Reliable Scalable Cluster Technology
Version 3.1.5.0

Programming Group Services for RSCT



Reliable Scalable Cluster Technology
Version 3.1.5.0

Programming Group Services for RSCT



Note

Before using this information and the product it supports, read the information in “Notices” on page 133.

This edition applies to Reliable Scalable Cluster Technology Version 3.1.5.0 and to all subsequent releases and modifications until otherwise indicated in new editions.

© Copyright IBM Corporation 2012, 2014.
US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

About this document v
Highlighting v
Entering commands v
Case sensitivity in AIX. vi
ISO 9000 vi
RSCT versions vi
Related information vii

Programming Group Services for RSCT 1
What's new in Programming Group Services for RSCT 1
Group Services concepts 1
High availability in a cluster environment . . . 2
Synchronization within an application 3
Coordination among applications 3
Using Group Services 3
Migration and coexistence 4
Group membership 4
Group Services domains 5
Group creation 6
Responsiveness checks 7
Protocols and voting. 8
Expel protocol 14
Deactivate-on-failure handling 16
Notifications 18
Example: multi-phase protocol 19
Active protocol proposals. 22
Node, provider, or subsystem failure 24
Provider actions during voting 25
Subscribing to a group 26
Provider and subscriber tokens 27
Source-target group relationships 27
Host and adapter membership groups 29
Quorum 30
Sundered networks 30
Designing Group Services applications 31
IP addressing 33
Configuring Group Services 34
Enabling the collection of voting results 34
Connecting to Group Services domains 35
Managing Group Services 35
Deactivating scripts. 35
Changing membership and state value 36
Initiating group services protocols 37
Submitting changes with voting responses . . . 40
Ending a protocol 41
Accessing Group Services IP address 42
Receiving Group Services subscription special data 42
GSAPI library names 46
GSAPI subroutines 47
ha_gs_announcement_callback 48
ha_gs_change_attributes 50
ha_gs_change_state_value 54
ha_gs_delayed_error_callback 57
ha_gs_dispatch 59
ha_gs_expel 62
ha_gs_get_adapter_info 65
ha_gs_get_adapter_info_by_addr 67
ha_gs_get_adapter_info_by_id 70
ha_gs_get_ffdc_id 71
ha_gs_get_ipaddr_by_id 72
ha_gs_get_limits. 74
ha_gs_get_node_number 75
ha_gs_get_rsct_active_version 76
ha_gs_get_rsct_installed_version 77
ha_gs_goodbye 78
ha_gs_init 80
ha_gs_join 85
ha_gs_leave 91
ha_gs_n_phase_callback 94
ha_gs_protocol_approved_callback 102
ha_gs_protocol_rejected_callback 104
ha_gs_quit 107
ha_gs_responsiveness_callback 108
ha_gs_send_message 110
ha_gs_subscribe 112
ha_gs_subscriber_callback 118
ha_gs_unsubscribe. 122
ha_gs_vote 123
ha_gs.h header file 125
GSAPI return codes 126
Group Services sample programs. 129
The sample_schg.c sample program 129
The sample_test.c sample program 130
The Sample_Subscribe.C sample program . . . 130

Notices 133
Privacy policy considerations 135
Trademarks 135

Index 137

© Copyright IBM Corp. 2012, 2014

About this document

This document contains conceptual, guidance, and reference information about the Group Services APIs that are part of Reliable Scalable Cluster Technology (RSCT). Specifically, this document contains information to help you write Group Services client programs.

Highlighting

The following highlighting conventions are used in this document:

Table 1. Conventions

<table>
<tr><th>Convention</th><th>Usage</th></tr>
<tr><td>bold</td><td>Bold words or characters represent system elements that you must use literally, such as commands, flags, path names, directories, file names, values, PE component names (poe, for example), and selected menu options.</td></tr>
<tr><td><u>bold underlined</u></td><td><u>Bold underlined</u> keywords are defaults. These take effect if you do not specify a different keyword.</td></tr>
<tr><td><code>constant width</code></td><td>Examples and information that the system displays appear in <code>constant-width</code> typeface.</td></tr>
<tr><td>italic</td><td>Italic words or characters represent variable values that you must supply.

Italics are also used for information unit titles, for the first use of a glossary term, and for general emphasis in text.</td></tr>
<tr><td><<i>key</i>></td><td>Angle brackets (less-than and greater-than) enclose the name of a key on the keyboard. For example, <Enter> refers to the key on your terminal or workstation that is labeled with the word <i>Enter</i>.</td></tr>
<tr><td>\</td><td>In command examples, a backslash indicates that the command or coding example continues on the next line. For example:
<code>mkcondition -r IBM.FileSystem -e "PercentTotUsed > 90" \</code>
<code>-E "PercentTotUsed < 85" -m d "FileSystem space used"</code></td></tr>
<tr><td>{item}</td><td>Braces enclose a list from which you must choose an item in format and syntax descriptions.</td></tr>
<tr><td>[item]</td><td>Brackets enclose optional items in format and syntax descriptions.</td></tr>
<tr><td><Ctrl-<i>x</i>></td><td>The notation <Ctrl-<i>x</i>> indicates a control character sequence. For example, <Ctrl-c> means that you hold down the control key while pressing <c>.</td></tr>
<tr><td>item...</td><td>Ellipses indicate that you can repeat the preceding item one or more times.</td></tr>
<tr><td>|</td><td>• In synopsis or syntax statements, vertical lines separate a list of choices. In other words, a vertical line means Or.
• In the left margin of the document, vertical lines indicate technical changes to the information.</td></tr>
</table>

Entering commands

When you work with the operating system, you typically enter commands following the shell prompt on the command line. The shell prompt can vary. In the following examples, $ is the prompt.

To display a list of the contents of your current directory, you would type `ls` and press the **Enter** key:

```
$ ls
```

When you enter a command and it is running, the operating system does not display the shell prompt. When the command completes its action, the system displays the prompt again. This indicates that you can enter another command.

The general format for entering operating system commands is:

© Copyright IBM Corp. 2012, 2014

Command Flag(s) Parameter

The flag alters the way a command works. Many commands have several flags. For example, if you type the **-l** (long) flag following the **ls** command, the system provides additional information about the contents of the current directory. The following example shows how to use the **-l** flag with the **ls** command:

```
$ ls -l
```

A parameter consists of a string of characters that follows a command or a flag. It specifies data, such as the name of a file or directory, or values. In the following example, the directory named `/usr/bin` is a parameter:

```
$ ls -l /usr/bin
```

When entering commands in, it is important to remember the following items:

- Commands are usually entered in lowercase.
- Flags are usually prefixed with a - (minus sign).
- More than one command can be typed on the command line if the commands are separated by a ; (semicolon).
- Long sequences of commands can be continued on the next line by using the \ (backslash). The backslash is placed at the end of the first line. The following example shows the placement of the backslash:

  ```
  $ cat /usr/ust/mydir/mydata > \
  /usr/usts/yourdir/yourdata
  ```

When certain commands are entered, the shell prompt changes. Because some commands are actually programs (such as the telnet command), the prompt changes when you are operating within the command. Any command that you issue within a program is known as a subcommand. When you exit the program, the prompt returns to your shell prompt.

The operating system can operate with different shells (for example, Bourne, C, or Korn) and the commands that you enter are interpreted by the shell. Therefore, you must know what shell you are using so that you can enter the commands in the correct format.

Case sensitivity in AIX

Everything in the AIX® operating system is case sensitive, which means that it distinguishes between uppercase and lowercase letters. For example, you can use the **ls** command to list files. If you type `LS`, the system responds that the command `is not found`. Likewise, **FILEA**, **FiLea**, and **filea** are three distinct file names, even if they reside in the same directory. To avoid causing undesirable actions to be performed, always ensure that you use the correct case.

ISO 9000

ISO 9000 registered quality systems were used in the development and manufacturing of this product.

RSCT versions

This edition applies to RSCT version, release, modification, and fix number 3.1.5.0.

You can use the **ctversion** command to find out which version of RSCT is running on a particular AIX, Linux, Solaris, or Windows node. For example:

```
/usr/sbin/rsct/install/bin/ctversion
```

An example of the output follows:

```
# /usr/sbin/rsct/install/bin/ctversion
rlis1313a 3.1.5.0
```

where, `rlis1313a` is the RSCT build level.

On the AIX operating system, you can also use the **lslpp** command to find out which version of RSCT is running on a particular AIX node. For example:

```
lslpp -L rsct.core.utils
```

The following output is displayed:

```
  Fileset                      Level    State  Type  Description (Uninstaller)
  ----------------------------------------------------------------------------
  rsct.core.utils              3.1.5.0    C     F    RSCT Utilities

State codes:
 A -- Applied.
 B -- Broken.
 C -- Committed.
 E -- EFIX Locked.
 O -- Obsolete.  (partially migrated to newer version)
 ? -- Inconsistent State...Run lppchk -v.

Type codes:
 F -- Installp Fileset
 P -- Product
 C -- Component
 T -- Feature
 R -- RPM Package
```

On the Linux operating system, you can also use the **rpm** command to find out which version of RSCT is running on a particular Linux or Solaris node. For example:

```
rpm -qa | grep rsct.basic
```

On the Windows operating system, you can also perform the following steps to find out which version of RSCT is running on a particular Windows node:

1. Click the Windows **start** button.
2. Select **All Programs**.
3. Select **Tivoli SA MP Base**.
4. Click **SA MP Version**.

Related information

The following PDF documents that contain RSCT information can be found at RSCT PDFs:

- *Administering RSCT*
- *Messages for RSCT*
- *Programming RMC for RSCT*
- *Technical Reference: RSCT for AIX*
- *Technical Reference: RSCT for Multiplatforms*
- *Troubleshooting RSCT*

Programming Group Services for RSCT

This information is intended for programmers of applications that manage system resources, which may or may not be subsystems, who want to use Group Services to make their applications highly available. This publication contains information for programmers who want to write new clients that use the Group Services application programming interface (GSAPI) or who want to add the use of Group Services to existing programs.

It assumes that you are an experienced C programmer and have a thorough understanding of the fundamentals of the AIX, Linux, Solaris, or Windows operating system, depending on your system environment.

To use the subroutines described in this publication, you need appropriate privileges and authorizations. To use the GSAPI subroutines, the calling process needs to have an effective group ID (EGID) of **hagsuser**, an effective user ID (EUID) of **root**, or an EUID that is a member of the **hagsuser** group.

Group Services help in the design and implementation of fault-tolerant applications, and in the consistent recovery of multiple applications. They also accomplish these two distinct tasks in an integrated framework. They offer a simple programming model because it is based on a small number of core concepts: it is a cluster-wide group membership and synchronization service based on the publish/subscribe model, which maintains an application-specific state for each group.

For information about managing the Group Services subsystem and its daemons, see the *Administering RSCT* guide.

What's new in Programming Group Services for RSCT

Read about new or significantly changed information for the Programming Group Services for RSCT topic collection.

How to see what's new or changed

In this PDF file, you might see revision bars (|) in the left margin that identifies new and changed information.

June 2014

The following information is a summary of the updates made to this topic collection:
- Obsolete information was removed or changed in various topics.

October 2011

The following information is a summary of the updates made to this topic collection:
- Updated the **ha_gs_init** subroutine description with the **gs_responsiveness_type** field information.

Group Services concepts

The Group Services subsystem is a system-wide, fault-tolerant, and highly-available service that provides a general purpose facility for coordinating and monitoring changes to the state of an application that is running on a set of nodes.

© Copyright IBM Corp. 2012, 2014

Group Services help in the design and implementation of fault-tolerant applications, and in the consistent recovery of multiple applications. Group Services accomplishes these two distinct tasks in an integrated framework.

Group Services offer a simple programming model because it is based on a small number of core concepts: it is a cluster-wide group membership and synchronization service based on the publish/subscribe model, which maintains an application-specific state for each group.

An application can consist of multiple processes that run on multiple nodes of a system. The set of nodes that is defined to Group Services is called a Group Services domain or a Reliable Scalable Cluster Technology (RSCT) peer domain.

A Group Services domain is the set of nodes that makes up a system partition. If there is only one system partition in the system, the Group Services domain consists of all of the nodes in the system. From the standpoint of the Group Services subsystem, the system partition in which it is running is "the system."

An RSCT peer domain is a set of nodes that is configured for high availability by the configuration resource manager. Such a domain has no distinguished or master node. All nodes are aware of all other nodes, and administrative commands can be issued from any node in the domain. All nodes also have a consistent view of the domain membership.

You must understand certain concepts before you use the Group Services application programming interface (GSAPI). For example, you should understand why high availability is important in a cluster environment and how Group Services can help you achieve it. You must also understand how to use the Group Services functions that are available, and how the Group Services subsystem operates. Understanding these basic concepts help you make design choices as you plan your application's use of the GSAPI.

High availability in a cluster environment

Applications that can best exploit a cluster environment typically consist of several cooperating processes that are running on multiple nodes.

These applications, which may or may not be subsystems, can be structured as follows:

- Client/server processes, for example: a distributed file system and the virtual shared disk (VSD) recovery subsystem. For information about VSD, see the *Reliable Scalable Cluster Technology: Managing Shared Disks* manual.
- Peer processes. Many scientific applications are structured as peer processes.

Cluster environments can be subject to node, process, network, and device failures. Such failures arise from a combination of hardware failures, software failures, resource exhaustion, and operator error.

To be competitive, cluster applications must be highly available. This means that the application continues to run after a failure, perhaps after a brief interruption of service to accommodate error recovery, and perhaps with degraded performance. While dealing with a failure, an application should never break any correctness requirement. For example, a database should never violate the integrity of customer data.

Hardware techniques form an essential component of a comprehensive suite of system support for high availability. These include the use of multi-tailed disks, where a node can *take over* the data that was previously owned by a failed node, and the use of network address takeover (for example, IP takeover), where a node can assume the *identity* of a failed node.

Hardware techniques, however, make up only part of a complete solution. Additional aspects of the solution include detection of component (process and node) failures, recovery from communication partitions, coordination of activities among the processes of an application, and coordination of activities

between applications. The Group Services subsystem and its application programming interface (GSAPI) are designed to satisfy these additional requirements.

Synchronization within an application

Group Services encapsulates a collection of software abstractions that are commonly used in the design of fault-tolerant systems. Specifically, Group Services provides process groups (simply called groups), state data, barrier synchronization, one-phase protocol, and multi-phase commit.

By using the Group Services abstractions, application developers do not have to develop their own synchronization and commit protocols. This is important because such protocols tend to be complex, error-prone, and expensive to duplicate. Developers who use Group Services interfaces are free to use only those abstractions that are suitable for their application, and do not have to use the entire toolkit. For example, developers of applications that have historically relied on their own methods for fault-tolerance, such as database servers, are free to restrict their use of Group Services interfaces to interapplication coordination services.

Coordination among applications

The Group Services subsystem supports the following features that enhance its use for interapplication coordination.

Extensibility:

Suppose that a new application is being added to a functioning cluster. The synchronization of this application with respect to other applications can be seamlessly managed by Group Services without disturbing existing applications.

No required operator knowledge:

Using Group Services interfaces, the programmers of each application determine the appropriate degree of synchronization between that application and other applications in the cluster. There is no need for an operator to understand all of the interactions between all of the components, as may be the case with other facilities such as scripts.

Different synchronization at different times:

Group Services allows applications to have different dependencies on each other at different times. For example, these dependencies may change during initialization, recovery, and shutdown.

It is equally important to understand what Group Services is not. Group Services is not intended to replace the current collection of communications libraries and services. Group Services was not designed to provide a high-volume, high-bandwidth, general-purpose messaging facility. Group Services does not perform application recovery; rather, it helps applications orchestrate their own recovery.

Using Group Services

If you are writing a new application or are considering updating an existing application, you can use the Group Services application programming interface (GSAPI) to improve system availability in a number of ways.

You can use the GSAPI to do the following:

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

Migration and coexistence

Group Services clients that were written to any previous level of Group Services run without having to be recompiled, and without change on the latest level.

However, if you want your Group Services client to take advantage of the new support available in the latest level of Group Services, you must change the code to use the function, recompile it, and relink it using the latest Group Services library and header. In addition, your Group Services client must run on nodes at the latest level of RSCT, and all of the nodes in the domain must be running the latest level of RSCT. It is the responsibility of the Group Services client to ensure that all binaries are at the same level. Group Services clients can use group attributes, provider instance numbers, and group state values to ensure this.

Existing applications that already have some groups joined with old attributes and have just installed the latest version of Group Services will need to upgrade node by node. When the last node is upgraded, the entire peer domain will then have the latest level of RSCT. If an existing group wants to upgrade its attributes so it can collect the vote list, a provider in the group can use the one-phase protocol of the **ha_gs_change_attributes()** subroutine to upgrade the attributes. The one-phase protocol does not need to vote and will be proved automatically. The existing group needs to let other providers know and it needs the attributes that were updated without voting back and forth. If the nodes in a domain are at different levels of RSCT, the Group Services daemons will not enable the new functions and will return an error that these functions are not supported. The Group Services daemons will run at the lowest level of RSCT that is installed on the nodes in the peer domain.

Group membership

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

The term *GS client* refers to both providers and subscribers. A process that has registered to use the Group Services subsystem, but has not yet become a provider or subscriber, is also referred to as a GS client.

The domain control environment variable must be set and exported in a GS client's environment to the name of the system partition in which the GS client and the Group Services daemon are running. On a node, this is the system partition to which the node belongs. On the control workstation, there is a Group Services daemon for each system partition. The value of the domain control environment variable identifies the system partition and the particular Group Services daemon to which the GS client will connect.

A group may have members on multiple nodes in the domain, and each node may have multiple members. For each group, the Group Services subsystem maintains consistent group state data. A group's state consists of the membership list and the group state value:

Related concepts:

"Group Services domains" on page 5
The Group Services subsystem provides services within the boundaries of what it calls a domain. The three types of Group Services domains are: Group Services PSSP domain, Group Services HACMP/ES domain, and RSCT peer domain. For Linux, a Group Services domain can be an RSCT peer domain.

"Changing membership and state value" on page 36
Any provider in the group can ask Group Services to modify the state value and can also specify the level of consistency that is to be associated with the modification.

Related reference:

"ha_gs_init" on page 80

Membership list

A group's membership list contains 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 (that is, the first provider to join the group) 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 membership list is modified when providers join or leave the group. In addition to voluntarily leaving a group, a provider may leave involuntarily, due to the failure of the provider process itself or the failure of the node on which it is running. An involuntary leave is called a **failure leave** and is initiated by the Group Services subsystem. Finally, a provider may be expelled from the group at the request of a provider.

Group state value

The state value of a group is defined by the application that is using the GSAPI 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 Group Services subsystem does not interpret the state value.

The group state is available to surviving providers despite node, communication adapter, and network failures. However, the group state does not survive the dissolution of a group. If all of the providers fail, the group state is lost.

Group Services domains

The Group Services subsystem provides services within the boundaries of what it calls a domain. The three types of Group Services domains are: Group Services PSSP domain, Group Services HACMP/ES domain, and RSCT peer domain. For Linux, a Group Services domain can be an RSCT peer domain.

Related concepts:

"Group membership" on page 4
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. Any authorized process in the domain may ask to become a member of a group.

Group Services PSSP domains

An SP system is divided into one or more Group Services domains, based on the number of SP system partitions defined. Each of these domains is referred to as a *Group Services PSSP domain*.

A Group Services PSSP domain includes an SP system control workstation and the set of nodes within an SP system partition. An SP system control workstation can be within multiple Group Services domains.

To become a Group Services client on the control workstation or on an SP system node, an application must set one or more environment variables to ensure that it is able to connect to the proper Group Services domain. To connect to the Group Services PSSP domain, a Group Services client must ensure that the environment variable **HA_DOMAIN_NAME** is set to the name of the SP system partition in which the Group Services client is running prior to the Group Services client invoking the **ha_gs_init** subroutine. This must be done on a 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 more than one SP system partition is defined).

Group Services HACMP/ES domains

In addition to the Group Services PSSP domain, if HACMP/ES is installed on a node, that node will also be part of a Group Services HACMP/ES domain, which is separate from that node's Group Services PSSP domain.

A *Group Services HACMP/ES domain* consists of all nodes that are part of the HACMP/ES cluster, which may include SP system nodes, non-SP AIX workstations, and SP system nodes from a physically-separate SP system.

RSCT peer domains

A set of AIX systems can be configured as an RSCT peer domain using the configuration resource manager, as described in the Administering RSCT guide.

In this domain, the name of the Group Services subsystem is **cthags**. Similarly, the name of the Topology Services subsystem is **cthats**.

Group creation

Typically, an application defines one or more group names that are known to all of the processes that are part of the application. During initialization, each process in the application asks to join the group as it starts up. The first join request creates the group and defines its attributes.

The subsequent join requests result in new providers joining the group. Each subsequent join request also includes group attribute information, which must match the group's established attributes. Otherwise, the join request is rejected.

The attributes of a group are:

- The name of the group
- An application-defined version code
- The number of phases (one or multiple) for join and failure leave protocols.
- A time limit, in seconds, for voting in each phase of a join or failure leave n-phase protocol. If a time limit of 0 is specified, no limit is enforced.
- A default vote to use as a proxy for a provider that fails to vote or fails to vote in time. The default vote may be to either approve or reject. If none is specified, the default value is to reject.
- A batch control field that specifies how requests may be batched. Join requests may be batched with other join requests, and failure leave requests may be batched with other failure leave requests. Join requests are never batched with failure leave requests.
- Attributes related to a source-target relationship, if any, include:
  - The name of the source-group for this group. Specifying a source-group name defines this group as a target-group.
  - The number of phases to use for the source-reflection protocols, which run in the target-group when the source-group changes its state value.
  - The voting phase time limit for source-reflection protocols, if they are n-phase.

Mutability of group attributes

Certain group attributes are *mutable*. This means that they can be dynamically changed by the group's providers using the **ha_gs_change_attributes** asynchronous interface.

The following group attribute fields are mutable (can be dynamically changed):

gs_client_version
: Client-specified *version* number

gs_batch_control
: Batch control setting for membership (join and failure protocols)

gs_num_phases
Phase control setting for membership (join and failure protocols)

gs_source_reflection_num_phases
Phase control setting for source-state reflection protocols

gs_group_default_vote

Group's base default vote for all N-phase protocols

gs_merge_control
Behavior of group in a merge situation

gs_time_limit
Voting time limit for N-phase join and failure protocols

gs_source_reflection_time_limit
Voting time limit for N-phase source-state reflection protocols

The following group attribute fields are **not** mutable (cannot be dynamically changed). To change these group attribute fields, the providers must all leave the group, then rejoin the group with the desired new attribute field.

gs_group_name
The name of the provider's group

gs_source_group_name
The name of the source group for the provider's group

Related concepts:

"Source-target group relationships" on page 27
It is sometimes convenient to associate several groups with a single application, and to allow a process to be a member of multiple groups.

Related reference:

"ha_gs_change_attributes" on page 50

Responsiveness checks

Responsiveness checks allow the Group Services subsystem to periodically inspect the state of the GS client when there are no ongoing group activities.

Group Services always monitors the GS client for 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.

When the GS client initializes itself with Group Services, it must specify information about the protocol, if any, to be used to perform responsiveness checks for the GS client. It must also specify the path name of a callback routine to invoke if the GS client fails its responsiveness check

Responsiveness protocols

The GS client can specify one of several responsiveness protocols.

No protocol
In this case, Group Services acts only if the GS client process exits.

A ping-like protocol
In this case, Group Services periodically sends a responsiveness notification to the GS client and expects a response. The notification calls the responsiveness callback routine 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.

This protocol is available to both single-threaded and multi-threaded GS clients.

Responsiveness callback routines

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

Nonresponsive providers

Group Services (GS) perform responsiveness checks once the GS client has initialized.

If a responsiveness check fails and the GS client is a provider, Group Services places it in a list of nonresponsive providers. Then, Group Services sends an announcement notification that contains the list to all of the group's providers. Group Services takes no other direct action.

On receipt of the announcement notification, a provider could initiate an expel protocol to remove the nonresponsive providers from the group, if appropriate. Group Services tries to contact nonresponsive providers. If a previously nonresponsive provider responds, Group Services places it in a list of "rejuvenated" providers. Then, Group Services sends an announcement notification that contains 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.

Protocols and voting

The Group Services subsystem uses a variety of protocols. A *protocol* is the mechanism that coordinates membership and state value changes within a group.

The Group Services application programming interface (GSAPI) provides a flexible n-phase voting protocol to mediate provider joins and departures, and state value changes. Different applications have different synchronization and coordination requirements for membership and state changes. Programmers can customize their applications to meet these requirements by choosing the appropriate number of voting phases, as follows:

- A one-phase protocol is the special case in which no voting is allowed. Here, the proposed membership or state value change is automatically approved, without voting.
- An n-phase protocol puts the group through at least one phase of voting before the change is approved. The number of phases required is not specified in advance to the Group Services subsystem. Instead, in each phase of voting, the providers can request another phase of voting, or end the protocol by approving or rejecting the proposal.

The votes of the providers cause a proposed change to be approved or rejected. If it is approved, Group Services sends a final notification that describes the change to all of the group's providers and subscribers. If it is rejected, Group Services sends the final notification only to the providers. The group state reverts to its value at the beginning of the protocol. When a protocol is proposed, the proposal indicates whether it is one-phase or n-phase.

Protocol categories

There are four categories of protocols.

Protocols are grouped into four categories, as follows:

Membership change protocols

These protocols are used when a provider joins or leaves a group. If approved, the membership of the group changes. In addition, the group state value may also be changed during all phases of n-phase membership change protocols, as discussed in “Submitting changes with voting responses” on page 40.

Membership change protocols include:
- Join
- Leave (also called "voluntary leave")
- Expel
- Failure leave (including clients that invoke **ha_gs_goodbye**)
- Cast-out (a form of failure leave that is associated with source-target relationships. See “Source-target group relationships” on page 27).

The state value change protocol

A provider uses this protocol to change the state value of the group, but leave the membership unchanged. n-phase state value change protocols may also change the group state value during the voting phases. State value change protocols do not affect the group membership.

The provider-broadcast message protocol

If this protocol is one-phase, it allows a provider to broadcast a message to all other providers in the group, with no voting.

If this protocol is n-phase, it allows a provider to broadcast the message to the other providers in the group, and also initiates the standard voting phases. The group state value can be changed during each voting phase. Provider-broadcast message protocols do not affect the group membership.

The change-attributes protocol

If this protocol is one-phase, it allows a provider to change a group's attributes with no voting.

If this protocol is n-phase, it allows a provider to propose to change the group attributes and initiates the standard voting phases. Change-attributes protocols do not affect the group membership.

Voting on an n-phase protocol

When a provider receives an n-phase protocol proposal notification, it is asked to vote.

At the start of every phase of voting, the Group Services subsystem informs all of the providers in the group of the proposed state value change, and the current phase number. Each provider then votes either to approve the proposed change (APPROVE), to request another round of voting (CONTINUE), or to reject it and end the protocol (REJECT). Voting can occupy any number of phases, based on the wishes of the providers.

For each phase, each provider must provide one of the following vote values:

APPROVE

The provider approves the proposed change. If all providers vote to APPROVE the proposal in the same voting phase, the change is approved and the group state is changed accordingly. If the vote tally indicates that the protocol should continue, the provider must continue to vote in each subsequent phase.

CONTINUE

The provider conditionally approves the proposed change, but wants to continue to another phase of voting. If at least one provider votes to CONTINUE, the protocol continues to another voting phase.

REJECT

the provider rejects the proposed change. Like CONTINUE, only one provider needs to vote to

REJECT to reject the proposed change. A REJECT vote on a failure leave protocol requires special consideration, as described in “Rejection of the Group Services subsystem-initiated protocols” on page 39.

Voting can have one of the following outcomes:
- The proposed change is approved if every provider that was a member of the group at the start of the protocol votes to APPROVE the proposal, either explicitly or implicitly.
- The protocol continues for another round if no provider votes to REJECT, and at least one provider votes to CONTINUE. The proposed change remains pending.
- The proposed change is rejected if at least one provider that was a member of the group at the start of the protocol votes to REJECT the proposal, either implicitly or explicitly

Normally, providers vote explicitly by responding to the Group Services subsystem by calling the **ha_gs_vote** subroutine. However, if a provider fails before it submits a vote, or if it fails to vote within the group's voting time limit, the Group Services subsystem enters a default vote on behalf of that provider. The default vote is also called an implicit vote.

By default, the default vote is REJECT. However, the provider can set the default vote to APPROVE when it joins the group. The Group Services subsystem does not permit an implicit vote to CONTINUE, because it could lead to a non-terminating protocol.

After the proposal is approved or rejected, the Group Services subsystem notifies all of the providers of the outcome. The providers do not vote in this last phase of the protocol. Thus, unlike the other phases, in the last phase no information flows from the providers to the Group Services subsystem. Finally, and only if the proposal was approved, the Group Services subsystem informs the subscribers of the outcome of the vote.

In certain cases, an approved proposal also generates notifications related to source-target handling. For more information, see “Source-target group relationships” on page 27.

Specifying the provider's default vote value

By default, the Group Services subsystem assigns REJECT as the default value for each group.

As part of its request to join a group, a provider may specify a default vote value as part of the group attributes. It may specify either REJECT or APPROVE. All providers must specify the same value. During each voting phase, any provider may specify a new default vote to be used for the group if any provider fails during this voting phase. The provider may specify either REJECT or APPROVE.

If no new default vote is specified, the current default vote carries over to the next phase. At the end of the protocol, the default vote reverts to the original value that was specified during the join of the providers.

If more than one provider specifies an updated default vote value with its vote, the Group Services subsystem arbitrarily chooses one of them. If different values are specified by different providers, it is not possible to predict which one the Group Services subsystem will choose.

To ensure consistency, the group should ensure one of the following:
- All providers submit the same updated default vote value.
- Only one provider submits an updated default vote value.
- No providers submit an updated default vote value, allowing the current value to remain in effect.

Related concepts:

“Submitting changes with voting responses” on page 40
The voting response to each phase of an n-phase protocol may contain a proposed new group state value, a provider-broadcast message, and a proposed new default vote for the group.

Approving and rejecting protocols

Every protocol must be either approved or rejected as described, based on the desires of the providers in the group. A protocol is approved when the providers vote to approve it. A protocol is rejected when the protocol is voted down or is ended for some reason.

In summary, a provider or the Group Services subsystem proposes the protocol. If necessary, voting proceeds for the desired number of phases.

If the protocol is approved, the updated information is broadcast to all providers and subscribers, as well as any appropriate target-groups. If the protocol is rejected, a notice of the rejection is broadcast to all providers. Subscribers receive no notification of a rejected protocol.

A one-phase protocol proposal is automatically approved. It cannot be rejected.

If a voluntary leave is rejected, whether by an explicit or implicit vote to REJECT, the protocol ends.

Proposing, voting, and phases for protocols

A protocol starts with a proposal. Either a provider or the Group Services subsystem itself can initiate the proposal.

Every protocol takes place in some number of phases. A one-phase protocol is an atomic multicast to the group members. 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, 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.

Each protocol proposal indicates whether it is a one-phase or an n-phase protocol. A one-phase protocol is a nonvoting protocol and is completed in a single phase, as described below.

A protocol that requires one or more voting phases is defined as an n-phase protocol. The exact number of phases is not defined in advance. Instead, the providers determine by their votes the exact number of voting phases.

One-phase protocols:

A one-phase protocol is a notification that the change proposed by the protocol is automatically approved.

For a membership change proposal, all providers and subscribers are notified of the updated membership. This is the join of a new provider or the leave of an old provider. The list of providers that are notified includes the providers that just joined, but does not include any providers that just left. Joins and leaves are not batched together in any one membership change proposal.

For a state value change proposal, all providers and subscribers are notified of the updated state.

For a provider-broadcast message proposal, all providers are notified of the message.

If a provider fails during the protocol, all remaining members receive the protocol notification. The Group Services subsystem immediately proposes a membership change protocol to handle the failed provider.

All providers and subscribers see a series of one-phase protocol notifications in the same order. However, any individual recipient may see a second or subsequent notification before all recipients have seen the first.

N-phase protocols:

An n-phase protocol establishes a series of barrier-synchronization voting phases for the providers in the group. Any protocol may be proposed as an n-phase protocol.

The proposal indicates that the protocol requires one or more voting phases, but it does not specify the exact number of phases to use. The voting results determine the actual number of voting phases.

The provider that proposes the protocol may specify a time limit for each voting phase. Each provider must register its vote within the given time limit. If a provider fails to register its vote in time, the Group Services subsystem does the following:

- Applies the group's default vote value for that provider.
- Notifies all providers of the lateness and includes a list of any providers that failed to respond in time.

The Group Services subsystem starts the protocol by broadcasting the proposal to all providers, which starts the first phase, and ends each phase by tallying the votes. In response to the initial notification, each provider must vote. Each vote response contains:

- The actual vote value, as described in “Voting on an n-phase protocol” on page 9.
- Optionally, an updated state value or a provider-broadcast message. For details, see “Submitting changes with voting responses” on page 40.
- Optionally, a default vote to be used by the group if a provider fails during this voting phase. For details, see “Specifying the provider's default vote value” on page 10.

If at least one provider votes to REJECT the proposal, the Group Services subsystem broadcasts to all providers a notification that the proposal was rejected. If no provider votes to reject the proposal, but at least one provider votes to CONTINUE the voting, Group Services broadcasts a notification that another vote is expected on the proposal. Once again, each provider must respond by voting.

Once all providers vote in the same phase to approve the proposal, the Group Services subsystem broadcasts to all providers and subscribers a notification of the approved change. This final broadcast is equivalent to the one-phase notification.

Failure of a provider during any phase of voting is handled by using the group's default vote for the provider. The Group Services subsystem automatically includes the default vote (REJECT or APPROVE) in the vote tally. Once the protocol completes (that is, is either approved or rejected), the Group Services subsystem immediately proposes a membership change to handle the failed provider.

The final notification of the protocol's rejection or approval also indicates whether any default votes were used during the protocol.

The voting phase allows each provider to take any action desired, such as running scripts, issuing commands to manipulate resources, or displaying graphics on the screen. The provider then submits its vote. If a provider fails during a voting phase, the Group Services subsystem enters the default vote into the tally on behalf of the failed provider.

Once each provider has submitted its vote, the Group Services subsystem tallies the votes. If all of the providers voted to APPROVE the protocol in the same voting phase, the voting ends and the proposal is approved. During the protocol, the providers determine the number of voting phases that are used by voting to CONTINUE the protocol. This mechanism allows the providers to adapt to unexpected occurrences during each protocol, rather than having to know in advance the exact number of phases that will be required.

Voting phase time limit

The voting phase time limit allows the providers to determine if their peers are not responding quickly enough during voting protocols.

Once the Group Services subsystem has delivered its notification for each voting phase, it sets a timer. If it has not received a voting response from the provider within that time, the Group Services subsystem assumes that the provider is not going to respond, and applies the group's default vote for this provider. Note that 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 subsystem specifies that a default vote was applied because the time limit was exceeded, but does not specify, at this time, the providers that were slow. If the application of the default vote causes the protocol to be rejected, or the time limit is exceeded in the last voting phase of an approved protocol, Group Services sends a notification to the providers that lists the providers that exceeded the time limit. The Group Services subsystem takes no further action. However, a provider may initiate an expel protocol to remove any providers that exceeded the time limit, if appropriate.

The voting phase time limit is also used to time the invocation of deactivate scripts during expel protocols.

Simultaneous protocols

Because there may be multiple providers in a group, more than one provider may submit a proposal at the same time. However, the Group Services subsystem does not invoke more than one protocol at a time within a group. (Of course, multiple protocols may be running simultaneously in a domain, one for each of the groups in the domain.)

What are simultaneous proposals? There is always a delay time between the call of a GSAPI subroutine by a provider to initiate a protocol and the broadcast of any resultant notification for that subroutine. The lag time allows the Group Services subsystem to batch multiple join requests, because the Group Services subsystem may receive multiple such requests before it has actually broadcast a notification. In this case, the Group Services subsystem collects all of the joins and issues a single notification. Similarly, the Group Services subsystem batches together multiple failure leaves or cast-outs into a single protocol. In all other cases, it deals with proposals one at a time.

In general, the first proposal to be made after a running protocol completes is the one that is chosen to invoke next. If multiple providers all attempt to submit proposals, the Group Services subsystem chooses one arbitrarily.

For provider-initiated proposals, all proposals that are not chosen to be invoked immediately are returned to the providers, with an asynchronous collision error code. The notification of the collision may arrive before or after the protocol that was chosen begins. The provider may resubmit the proposal at a later time, if appropriate.

All the Group Services subsystem-initiated proposals remain pending until they have been invoked within the group. No provider-initiated proposals are accepted until all of the pending Group Services subsystem-initiated proposals have been invoked. A provider that attempts to submit a proposal receives a synchronous or asynchronous collision error code.

When choosing among multiple proposals, the Group Services subsystem chooses a proposal based on the following priority order:

1. Failure leaves and cast-outs
2. Source-state reflection
3. Joins
4. Leaves and expels
5. State value change, provider-broadcast message, or change-attributes protocols.

Within these categories, if there are multiple simultaneous proposals of the chosen type, the Group Services subsystem arbitrarily chooses one of them, excluding those that may be batched together.

If batching is allowed, membership changes are batched. Joins are batched only with joins and failure leaves are batched only with failure leaves.

No provider is allowed to cycle invisibly. If a provider should fail and then restart and try to join the group, the Group Services subsystem ensures that the leave of that provider is proposed before the subsequent join of that provider.

A running protocol is always completed. The protocol could complete successfully or unsuccessfully. An unsuccessful completion could be caused by a provider voting to REJECT the protocol, by an explicit or implicit vote. The protocol might also end unsuccessfully if one or more providers fail to submit their votes within the specified time limit.

A rejected provider-initiated protocol is not automatically resubmitted. The providers must resubmit the protocol, if it is required.

Expel protocol

The expel protocol allows a provider to propose the removal from the group of one or more providers.

Some situations in which this could be useful include:

- 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 during a previously completed n-phase protocol within the specified time limit.
- A provider has detected through some other means that another provider is not behaving as expected in the context of the application that the group is running.

During the invocation of the expel protocol, Group Services runs a **deactivate script** against each provider that is being expelled. The deactivate script, which is specified by each Group Services client on initialization, is used to perform any cleanup actions that may be required.

The deactivate script does not need to be a shell script but can be any kind of executable file. For each provider that is targeted for expulsion, the Group Services daemon forks a child process that attempts to invoke the deactivate script on the provider's node.

The expel protocol is a provider-initiated protocol. Therefore, if it collides with another already-running protocol, Group Services returns it to the proposer. The proposer must resubmit the protocol; the protocol is not automatically queued.

A provider uses the **ha_gs_expel** subroutine to request an expel protocol. On input, the provider specifies the following information:

- **The number of phases for the protocol**. An expel protocol may be either a one-phase or an n-phase protocol.
- **The voting time limit for each phase**. Providers that are not being expelled must vote within this time limit. For providers that are being expelled, the deactivate script must complete within this time limit, or be considered unsuccessful.
- **The list of providers to be expelled**. These providers do not take part in the protocol and receive no notice of it, unless it is approved. All providers that are not targeted for expulsion take part in running the protocol, even if they had been declared nonresponsive before the protocol began.
- **A deactivate phase specifier**. This value tells Group Services in which voting phase it should invoke the deactivate script. A value of 0 indicates that the deactivate script should not be invoked.
- **An expel flag**. This flag is passed to the deactivate script. A null value indicates that no flag should be passed to the deactivate script.

For each provider that is targeted for expulsion, Group Services runs the deactivate script that was specified by that provider when it initialized itself with Group Services. The deactivate script runs on the node on which the provider that is targeted for expulsion is running. It runs during the phase and uses the flag that was specified on the expel protocol. To be successful, the deactivate script must complete within the voting time limit for the phase. To invoke the deactivate script, Group Services acts as a substitute for each provider that is being expelled.

During the expel protocol, providers that are not being expelled treat this as a normal protocol and take any action they deem appropriate. If it is an n-phase protocol, their voting responses are tallied as if it were any other n-phase protocol.

If the value of the deactivate phase specifier is 0, no deactivate script is invoked during the protocol. If the protocol is approved, the providers that are targeted for expulsion are removed from the group. Because one-phase protocols are always approved, a one-phase expel protocol with a deactivate phase specifier of 0 simply removes the targeted providers from the group. If the protocol is rejected, the targeted providers are not removed from the group.

At the start of the voting phase given by a non-zero deactivate phase specifier, Group Services runs the deactivate script against each targeted provider. If at least one provider votes to reject the protocol before this phase, the targeted providers are not removed from the group and no deactivate scripts are invoked.

If the expel protocol is a one-phase protocol, and the value of the deactivate phase specifier is 1, the deactivate script is run immediately after the protocol begins running. Providers that are not targeted for expulsion receive the usual protocol approval notification, informing them that the targeted providers are now out of the group. Providers that are targeted for expulsion receive the protocol approval notification after the Group Services daemon has forked a child process to run the deactivate script. The Group Services daemon does not wait for the script to complete before it sends the notification. Therefore, it difficult to determine whether the provider will receive the notification before or after the script runs.

The exit code of the deactivate script is not inspected, and the result is not returned to the providers that remain in the group.

If a provider that is targeted for expulsion by a one-phase expel protocol fails after the protocol has begun, no failure protocol is initiated in the group for that provider.

When a deactivate script runs successfully, it is expected to exit with an exit code of 0. Group Services treats the successful completion of the deactivate script as a vote to approve the protocol. If the protocol requires more voting phases, Group Services continues to vote APPROVE for each subsequent voting phase.

When a deactivate script does not exit with a code of 0, group services enters the group's current default vote value as the provider's vote for the phase. If the protocol requires more voting phases, group services continues to enter the current default vote value as the provider's vote for each subsequent voting phase.

If the deactivate script is to be run in a future voting phase, Group Services enters a vote of CONTINUE as the provider's vote for each interim voting phase.

If one or more providers that are targeted for expulsion did not specify a deactivate script, or specified a script that could not be run, but a non-zero deactivate phase specifier was given, then for those providers, the group's default vote value is entered for this and each subsequent voting phase. However, for providers that did specify a valid deactivate script, the script is run and its result is used to drive the voting, as previously described.

When a provider fails after the expel protocol begins but before the Group Services daemon has forked a child process to run the deactivate script, Group Services passes a process ID of 0 to the deactivate script. The deactivate script is still run and the exit code is used to determine the vote for this provider, as previously described.

Group Services tallies the votes for voting phases in the normal manner. If the expel protocol is approved, the providers that are targeted for expulsion are removed from the group. Remaining providers and subscribers are notified.

Group Services sends the protocol approval notification to expelled providers that did not exit in the course of running the deactivate script. However, Group Services does not verify that such providers receive or process the notification. Because they are no longer in the group, expelled providers cannot submit protocols and do not receive notifications related to the group.

In the event that the protocol is rejected for any reason, the providers that are targeted for expulsion are not removed from the group. However, if the deactivate script causes a provider to exit, Group Services initiates a failure leave protocol for that provider.

When a single process is joined as providers 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.

If a single process is joined as providers to multiple groups, and more than one of the groups are simultaneously running expel protocols that target those providers (because the process is unresponsive, for example), the order in which deactivate scripts are run against the process is not defined by Group Services. Because each group's expel protocol proceeds independently, Group Services does not coordinate the invocation 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 invoked. If one or more groups reject their expel protocols, but the process is killed in the course of running the deactivate script, those groups initiate failure leave protocols to remove the failed provider.

Deactivate-on-failure handling

The same deactivate script will be run in the case of a local provider's process failure as well as in the case of the expel protocol.

When a provider is failing, its group is forced into a failure leave protocol. Deactivate-on-failure handling allows recovery and clean-up actions on the failed provider's node, although the failed provider's process no longer exists.

In the case of an n-phase protocol, the results of the script's invocation will be used in subsequent voting for the protocol. For a one-phase protocol, the results will not be relayed to the remaining group members. Unlike expel, the group cannot specify the voting phase in which the script will be invoked. It is always run in the first phase. The failure leave protocol, with deactivate-on-failure handling, operates as follows:

- If batching of failures *is not* allowed, the deactivate script is run for every provider.
- If batching of failures *is* allowed, and:
  - If there are multiple failed providers on one node in one protocol, the deactivate script will be run once.
  - If there are multiple failed providers in separate protocols, the deactivate script will be run once per protocol.

  The deactivate script is invoked on each node with a failed provider.

- In the case where a failed GS client process had been joined as providers to multiple groups, each group continues to run independent failure protocols.
  - If multiple groups specify deactivate-on-failure, then the deactivate script will be run during each group's failure protocol.
  - Group Services does not define the order in which the deactivate scripts will be run by each group, as the order in which the individual groups will run the failure protocols is not defined.
- If a group has enabled deactivate-on-failure, and a one or more providers are to be cast out, the decision will be:
  - If the targeted provider's process exists at the time the cast-out protocol begins running, the deactivate script will *not* be invoked.
  - If the targeted provider's process does not exist at the time the cast-out protocol begins running, the deactivate script *will* be run.
  - If the targeted provider's process exists at the time the cast-out protocol begins running, but fails during the cast-out protocol, the deactivate script will *not* be run.

Related concepts:

"Node, provider, or subsystem failure" on page 24
When a node fails, Group Services assumes that all providers on that node have also failed. The GSAPI supports process failure detection by detecting the loss of a socket connection.

Deactivate-on-failure handling with one-phase protocol

If the failure protocol is a one-phase protocol, the deactivate script is invoked immediately after the protocol begins running and the Group Services daemon does not wait for the script to complete.

Non-failed providers receive the usual protocol approval notification, informing them that the failed providers are now out of the group.

The exit code of the deactivate script is not inspected, and the result is not returned to the providers that remain in the group.

Deactivate-on-failure handling with n-phase protocol

If the failure protocol is an n-phase protocol, the results of the deactivate script will be used to guide the *vote* submitted for the failed providers.

Note: The deactivate script is always run during the **first** phase of the failure protocol.

If a deactivate script runs successfully, it is expected to exit with an exit code of 0. Group Services treats the successful invocation of the deactivate script as a vote to approve the protocol. If the protocol requires more voting phases, Group Services continues to vote APPROVE for each subsequent voting phase.

If a deactivate script does not exit with a code of 0, Group Services enters the group's current default vote value as the failure provider's vote for the phase. If the protocol requires more voting phases, Group Services continues to enter the current default vote value as the failed provider's vote for each subsequent voting phase.

If the group has specified a time limit for failure protocols, and the script does not complete within the time specified, the Group Services daemon treats this as a normal *voting time out* and applies the group's current default vote. If the voting phases continue in the protocol, the Group Services daemon will continue to apply the group's current default vote value for each subsequent voting phase.

Group Services tallies the votes for voting phases in the normal manner. If the failure protocol is approved, the failed providers are removed from the group. Remaining providers and subscribers are notified. If the failure protocol is rejected, there are special conditions that apply to rejection of any failure protocol.

- When the rejection is caused by either an explicit reject vote or a default reject vote, and batching of failures is not allowed, the protocol ends and the failed providers are removed from the group. Remaining providers and subscribers are notified.
- When the rejection is caused by an explicit reject vote, and batching of failures is allowed the protocol ends and the failed providers are removed from the group. Remaining providers and subscribers are notified.
- When the rejection is caused by a default reject vote, and batching of failures is allowed, the protocol ends, but the failed providers are not removed. The group is immediately put into a new failure protocol, with any newly-failed providers added to the list of already-failed providers from the previous protocol. A deactivate script is **run only once** against any single failed provider instance. As a result, during the subsequent failure protocols, only the newly-failed providers will have their deactivate scripts run, but no deactivate scripts will be run against the already-failed providers. During any subsequent failure protocols, the Group Services daemon votes APPROVE on behalf of the old failed providers. This prevents the group from being put into an infinitely-looping situation, where the failure protocol ends via a default REJECT vote caused by a failed deactivate script, and would otherwise be continually restarted.

Notifications

A GS client can receive several types of messages, called notifications, from Group Services.

These include notifications for:
- Protocol proposals and ongoing protocols
- Protocol approvals
- Protocol rejections
- Announcements
- Responsiveness checks

All messages are sent in a fault-tolerant-manner. That is, providers and subscribers are guaranteed to receive notifications despite failures.

Protocol proposal and ongoing protocol notifications

These notifications are sent to the providers of a group to indicate that an n-phase protocol has been proposed or is in progress. As a response to these notifications, the Group Services subsystem typically expects a vote.

Protocol proposal notifications are not sent for one-phase membership or state value changes because these proposals are automatically approved.

There are three types of proposals for which notifications are sent:

Membership change proposals
A membership change proposal notification is sent when a provider has requested to voluntarily join or leave a group, a provider has requested the expulsion of one or more providers from a group, a provider has left the group involuntarily either because the process itself failed, or because the node on which it was running failed. An involuntary leave is called a failure leave and is initiated by Group Services.

State value change proposals
A state value change proposal notification is sent when a provider has requested a change to the group's state value.

Provider-broadcast message proposals
A provider-broadcast message proposal notification is sent when a provider has issued a request to broadcast a message and may also initiate voting.

Attribute change proposals
Attribute change proposal notification is sent when a provider has issued a request to change the group's attribute.

The only GS clients that are concerned with protocol proposal and ongoing protocol notifications are providers. Subscribers do not participate in proposing, approving, or rejecting membership or state value changes for the group. Also, when subscribers join or leave the group, no notification is sent to any GS client.

Protocol approvals

Protocol approvals are sent to the providers of a group to indicate that a proposal has been approved. They are also sent to the subscribers of the group for membership changes and state value changes.

Note: A protocol approval notification is sent as the first and only notification for a one-phase protocol.

Protocol rejections

Protocol rejections are sent to the providers of a group to indicate that a proposed membership or state value change has been rejected. Subscribers are not notified when proposals are rejected.

Announcement notifications

Announcement notifications are 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 in time or not responded to a responsiveness check.

Responsiveness notifications

Responsiveness notifications are sent to of a group's providers to determine whether the provider is active. If a provider does not respond to this responsiveness check within the time limit it specified previously, Group Services sends an announcement notification to all providers.

Example: multi-phase protocol

The figures that follow illustrate a state change protocol for a group with two providers, **P1** and **P2**, and two subscribers, **S1** and **S2**.

P2 proposes a change to the group's state value, and specifies whether the change requires voting phases or is handled as a single broadcast. Figure 1 on page 20 shows the sequence of events for a one-phase protocol. Figure 2 on page 21 shows the sequence of events for a two-phase commit protocol. Figure 3 on page 22 shows the sequence of events for a three-phase protocol.

Upon receipt of the state change proposal from **P2**, the Group Services subsystem sends a notification to all of the providers in the group, namely **P1** and **P2**. If **P2** requested a one-phase protocol, the change is approved, **S1** and **S2** are notified, and the protocol terminates. If **P2** requested a multi-phase protocol, **P1** and **P2** are instructed to vote on the outcome of the protocol.

Figure 2 on page 21 shows the invocation of a multi-phase protocol in which the providers vote to approve the change after one round of voting. Figure 3 on page 22 shows the providers extending the voting to three rounds. When the change is approved, all of the providers and the subscribers, that is, **P1**, **P2**, **S1**, and **S2**, are informed of the change.

Note that if both **P1** and **P2** submit state change requests concurrently, the Group Services subsystem chooses one of the requests for invocation, and returns the other to its proposer.

The n-phase agreement protocol that the GSAPI provides is flexible and powerful enough to handle a variety of synchronization and coordination requirements:

- A one-phase protocol is invoked when a provider submits a state change requesting that there be no voting by the providers, and therefore another provider cannot stop this state change. The first phase of such a protocol is also the last phase of the protocol, as shown in Figure 1 on page 20.

- A two-phase state change protocol is essentially the well-understood two-phase commit protocol with a reliable *coordinator*.
- An n-phase state change protocol gives the providers the framework to perform n-1 rounds of barrier synchronization. For example, the four-phase protocol shown in Figure 3 on page 22 yields three rounds of barrier synchronization, at the end of voting phases one, two, and three.

If any provider is notified that the state change is approved, the GSAPI guarantees that all (non-failed) providers and subscribers are notified of the approved state change without regard to failures within the system.

It is the responsibility of the providers in a group to determine the level of consistency that is required for managing changes to the group membership and state value. As described previously, providers may use either one-phase or n-phase protocols. In all cases, all providers see all protocols in the same order. However, the level of consistency differs in an important way, as follows.

Assume that two proposals occur rapidly one after the other.
- For one-phase protocols, although all providers see both protocols in the same order, some providers may see both the first and the second protocol before another provider has seen the first. This leads to a loosely synchronous consistency level, because the providers loosely catch up to each other in seeing the "latest and greatest" group state.
- For n-phase protocols, the group state is managed in a strongly-consistent manner. Because an n-phase protocol forces all participating providers to submit votes, that is, to reach the barrier synchronization points, no provider can see the second protocol before all have seen and reacted to the first.

Subscribers have no choice but to receive the notifications of approved group changes in a loosely synchronous manner. The GSAPI guarantees that all subscribers to a group see the approved changes in the same order as do the group's providers. However, one subscriber may see multiple notifications before another subscriber has seen any.

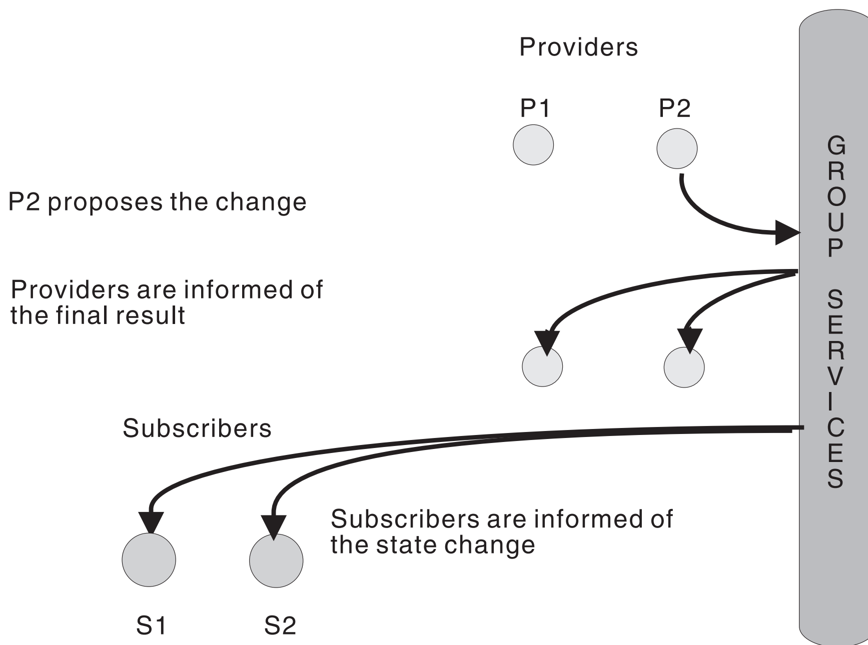


Figure 1. A One-Phase Protocol

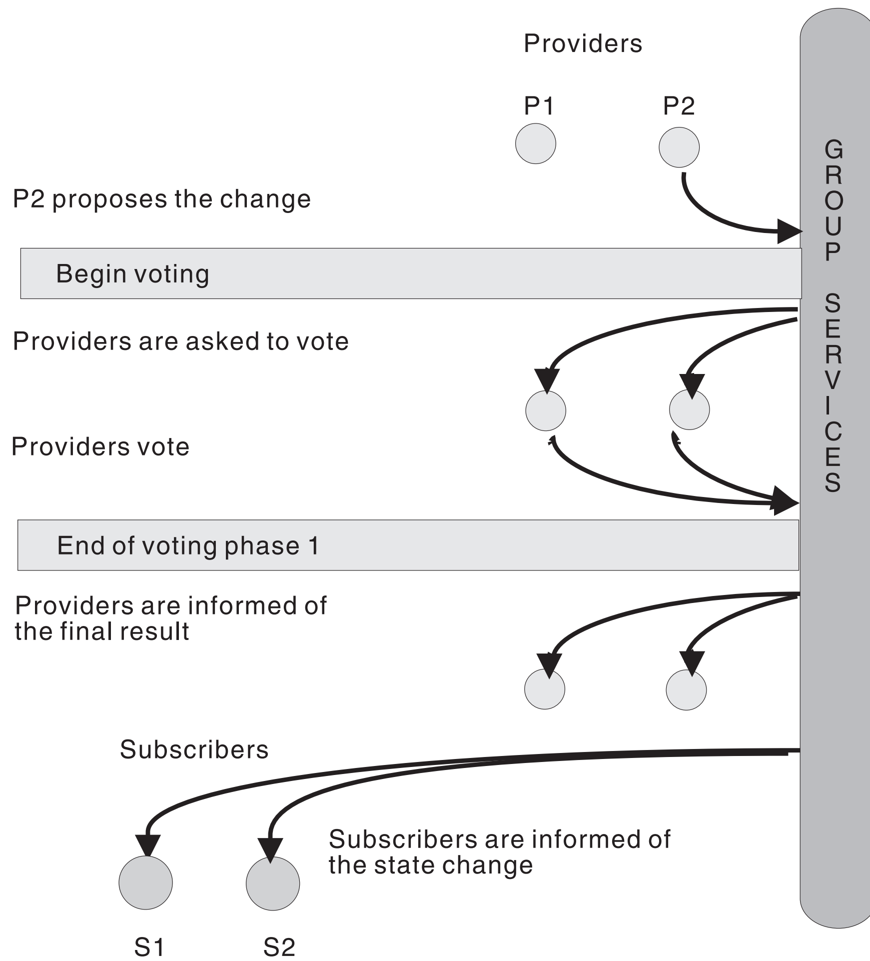


Figure 2. A Two-Phase Commit Protocol

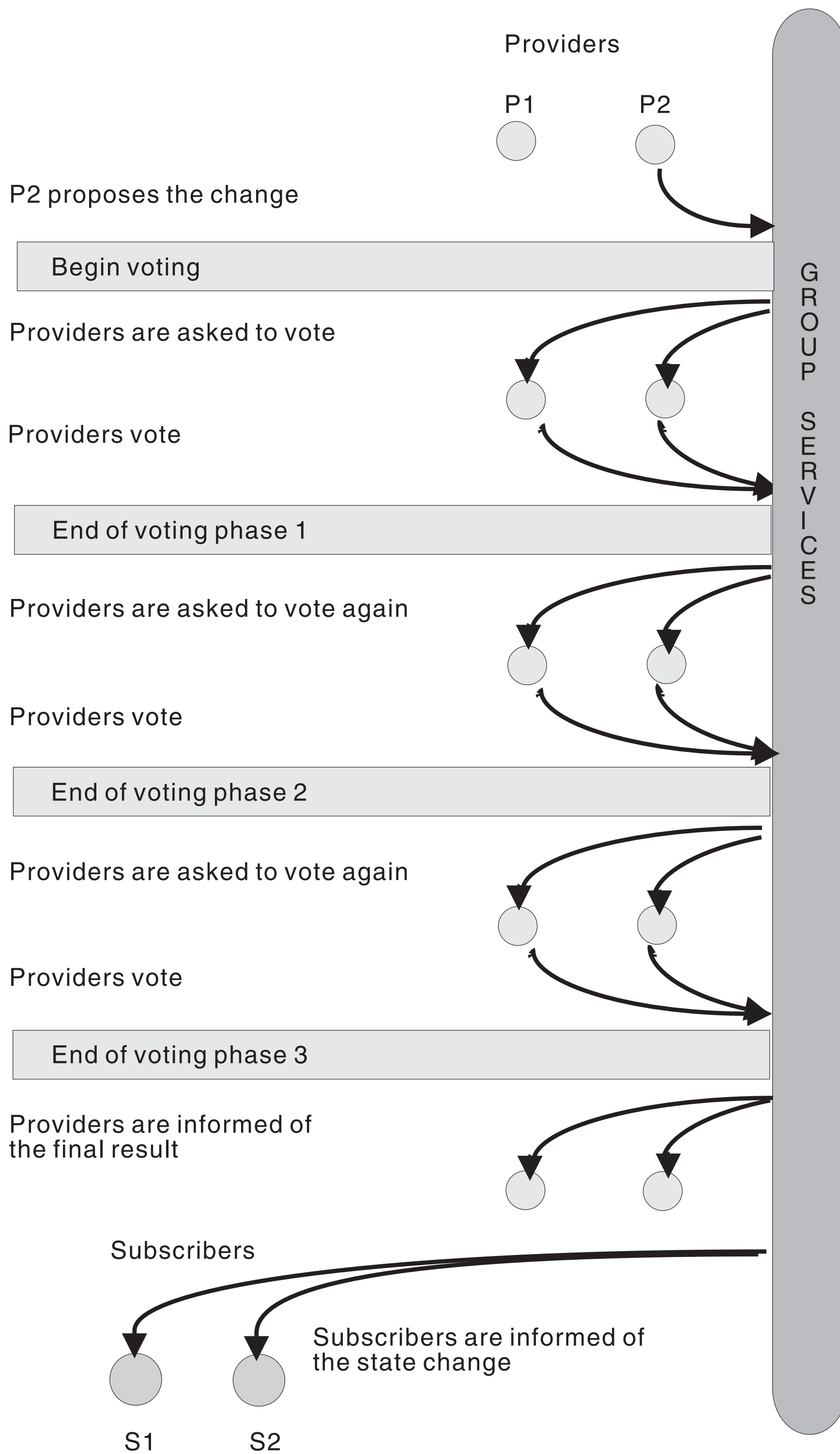


Figure 3. Barrier Synchronization in a Multi-Phase Protocol

Active protocol proposals

The GSAPI guarantees that only one protocol that affects the group's membership or state value is run at any time.

If more than one proposal is submitted within the group simultaneously, the Group Services subsystem chooses one for invocation and returns the others to the providers that submitted them. It is the responsibility of a provider that receives a returned proposal to resubmit it for invocation, if appropriate.

When providers join or involuntarily leave a group, this processing is modified. In these cases, the membership protocol to deal with the join or involuntary leave request is held until the currently running protocol has been approved or rejected. The membership change protocol is then started immediately. Figure 4 on page 24 shows how a new provider join request is delayed until the completion of an ongoing three-phase state change protocol.

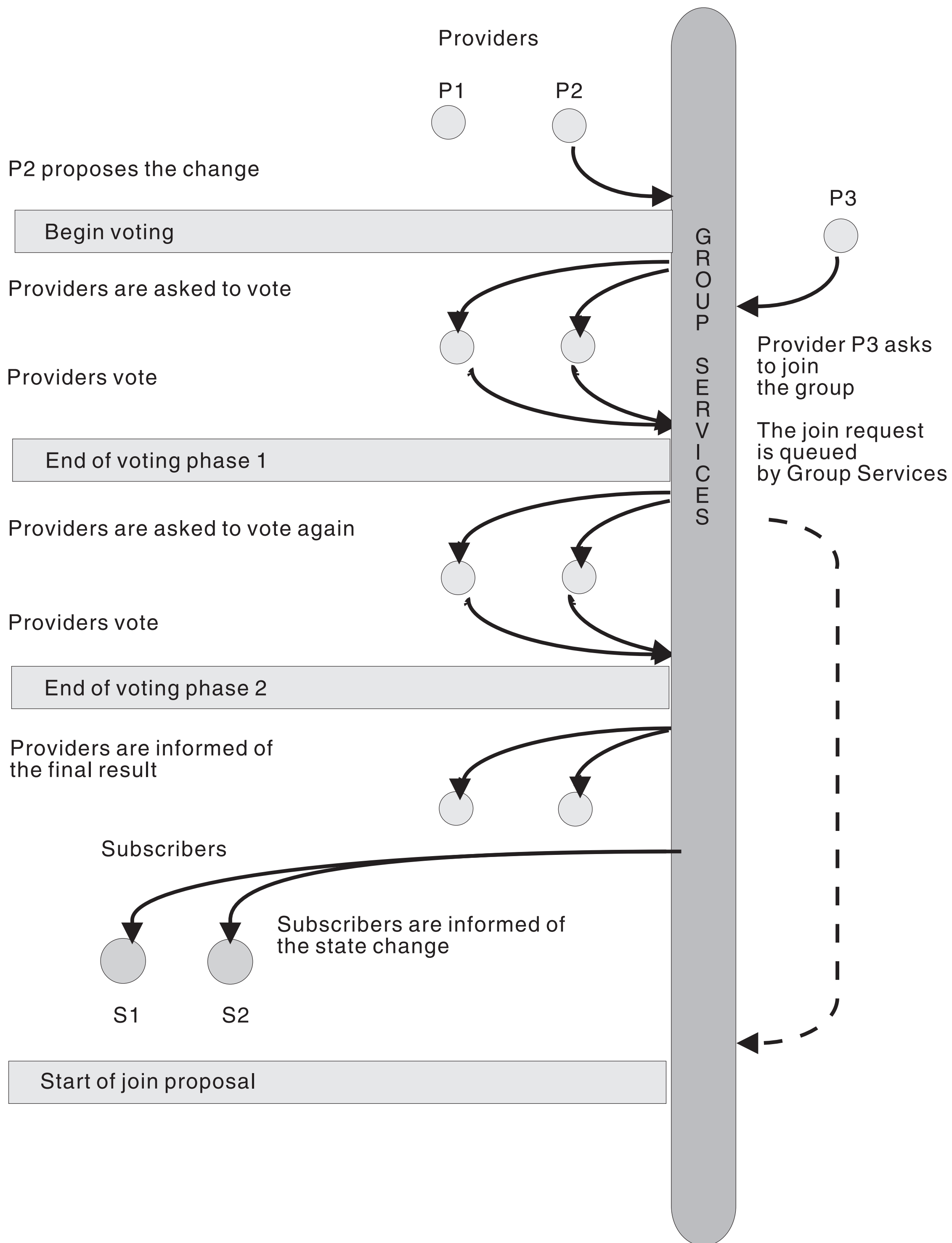


Figure 4. The Serialization of a Pending Join Request

Node, provider, or subsystem failure

When a node fails, Group Services assumes that all providers on that node have also failed. The GSAPI supports process failure detection by detecting the loss of a socket connection.

When a provider leaves due to either a node failure or the failure of the process itself, Group Services proposes a failure leave protocol for that provider. If the group had been using a one-phase protocol to handle joins, the failure leave is specified as one-phase. If the join had been n-phase, the failure leave is specified as n-phase.

As long as one provider is active, Group Services continues to keep the group going.

The Group Services subsystem itself has been designed to survive failures. These can be node failures, that lead to the loss of one or more Group Services processes, or network failures and communications adaptor failures, that hinder the communication between Group Services processes.

If the Group Services subsystem fails, any surviving GS client receives an announcement notification that the Group Services daemon has terminated suddenly and unexpectedly. The GS client can get the FFDC ID (First Failure Data Capture identifier) that is related to the cause of the Group Services subsystem failure when the FFDC ID exists. In addition, if a protocol is running, it is terminated.

Related concepts:

"Deactivate-on-failure handling" on page 16
The same deactivate script will be run in the case of a local provider's process failure as well as in the case of the expel protocol.

Provider actions during voting

A provider can perform any sequence of actions that it chooses between the time that it receives an ongoing protocol notification (that is, a request for a vote) and the time that it votes.

This is shown in Figure 5 on page 26. However:

- **Providers should submit their votes within the voting time limit.** When a provider is asked to vote on a proposed change, the proposal may include a time limit within which the vote must be submitted. The time limit includes any message delays. If any provider fails to submit its vote in time, the Group Services subsystem applies the group's current default vote in lieu of that provider's vote. The Group Services subsystem supplies a list of the providers that failed to vote in time to the other providers.
- **Providers should wait until a running protocol has completed before submitting a new proposal.** During the invocation of any protocol, no provider is allowed to submit another protocol proposal. The Group Services subsystem simply returns an error code to the provider if it tries to do so and ignores the new proposal. The provider must wait until the running protocol has completed before it resubmits the proposal.

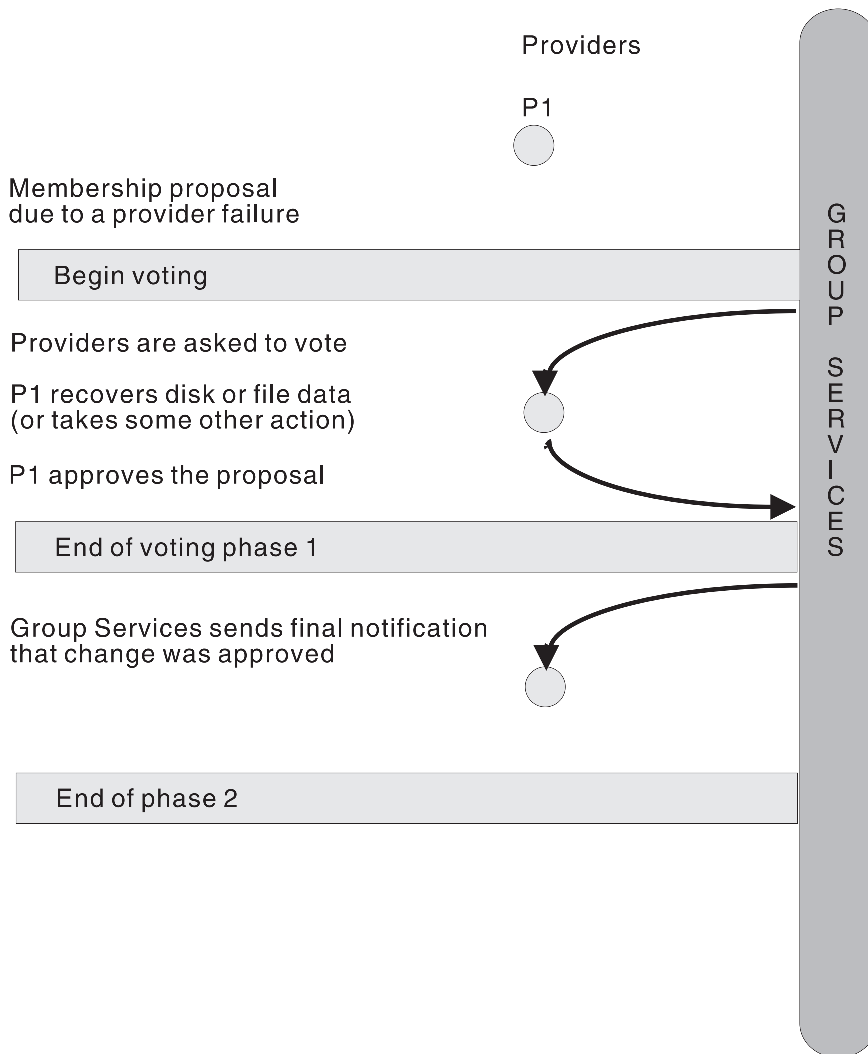


Figure 5. Actions during a Membership Change Protocol

Subscribing to a group

When it is desirable for a process to monitor a group without playing a part in the control of the group's state, the GSAPI allows the process to subscribe to the group. A subscriber may subscribe to receive approved membership changes, approved state changes, or both.

Subscribers do not participate in any of the voting protocols. In fact, they are not notified that any such activity is taking place. If a state or membership change is not approved, no notification is sent to the subscribers.

Notifications to all subscribers to any single group are serialized, so all subscribers receive all notifications in the same order. However, it is not guaranteed that all subscribers will receive any one notification before any other subscribers receive subsequent notifications. No notifications or protocol proposals are made when subscribers join or leave a group.

Subscription allows one group to maintain a loose synchronization with one or more other groups. For example, a subscriber could be used to monitor and display information about the state of a number of

groups and the members of those groups. As the subscribed-to groups change state or membership, the monitor can collect the changes and display or log the updated information.

Provider and subscriber tokens

The GSAPI uses integers that are called provider and subscriber tokens to identify providers and subscribers. These tokens are assigned, invalidated, and reassigned in a similar way in which file descriptors are assigned, invalidated, and reassigned as files are opened and closed.

For example, a GS client joins group **foo** and receives provider token 0. When the same client leaves group **foo**, Group Services invalidates provider token 0 and makes it available for reassignment. When the next GS client (which could be the same GS client or a different GS client) joins the next group (which could be the same group or a different group), Group Services assigns provider token 0 to that client.

As another example, a GS client subscribes to group **bar** and receives subscriber token 2. When the same client unsubscribes from group **bar** or becomes unsubscribed because group **bar** is dissolved, Group Services invalidates subscriber token 2 and makes it available for reassignment. When the next GS client (which could be the same GS client or a different GS client) subscribes to the next group (which could be the same group or a different group), Group Services assigns subscriber token 2 to that client.

Source-target group relationships

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 receive a membership change proposal 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, they 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 the target groups, the source group completes its membership change protocol before the target group begins its membership change protocol. Thus, the providers in the target group can run 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.

With source-target groups, join and leave protocols work a little differently than with other groups. Here are some key differences.

- A group defines itself as a target-group by listing a source-group name in the set of group attributes specified on the **ha_gs_join** subroutine by each target-group provider. A source-group is not notified that it has been *sourced* by any groups.
- 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 in the normal manner, that is, by a membership change proposal to the target-group.

- There may be multiple source-group or target-group providers on a node. A source-group may have any number of target-groups. A target-group may have only one source 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 leaves as a normal membership change proposal.

  Once the source-group has approved the leave protocol, a membership change is proposed to the target-group as a cast-out of the affected providers from the target-group. As a failure leave protocol, the cast-out protocol cannot be rejected. As part of the notification initiating this target-group membership change, the target-group receives the source-group's state value. If there is no target-group provider on that node, no notification is sent to the target-group providers.

  The providers 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.
- 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 could continue. However, once the source-group completes its leave protocol, the target-group provider may no longer validly belong to the target-group.

  As a result, the Group Services subsystem considers the target-group providers that will be cast-out as having failed during the protocol, and treats them as follows:
  - If the target-group's default vote is REJECT, the protocol is rejected, and the Group Services subsystem initiates 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 providers 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, the Group Services subsystem immediately initiates a cast-out protocol for the target-group.
- When a source-group leave protocol prevents the last target-group providers from invoking protocols, those providers are given a cast-out final notification and the target-group is, in effect, dissolved.
- If a node fails, rather than the last source-group provider on the node, it is handled in the same way as if the source-group provider itself had failed. The source-group completes its protocol before the target-group is notified. In this case, the target-group receives a cast-out protocol, rather than a failure leave protocol.
- If a source-group changes its state value during protocols that do not result in a target-group cast-out , its associated target-groups receive the committed state value. An example of this is a state value change protocol or a voting response during any other n-phase protocol.

  The notification appears to the target-group as a source-state reflection protocol. Values specified in the group attributes of the target-group control the number of phases and a voting time limit. The target-group treats this as a normal protocol and takes whatever actions are required.

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

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

Related concepts:

"Mutability of group attributes" on page 6
Certain group attributes are *mutable*. This means that they can be dynamically changed by the group's providers using the **ha_gs_change_attributes** asynchronous interface.

Host and adapter membership groups

The Group Services subsystem provides several system-defined groups to which GS clients can subscribe for keeping track of hardware status.

See "ha_gs_subscribe" on page 112 for a complete listing of the different system-defined groups that are available for subscription.

The host membership group

The Group Services subsystem keeps track of node status to determine when nodes are no longer reachable.

A node that is fully isolated due to network or communications adapter failures is not distinguishable from a node that has failed. Accordingly, a fully isolated or failed node triggers such actions as notifications to groups that have one or more providers on the failed node or nodes.

The state of the nodes is reflected by the Group Services subsystem in a special system-defined group called the host membership group. The host membership group is represented by **HA_GS_HOST_MEMBERSHIP**. By subscribing to this group, a GS client can obtain information about the nodes that are currently active and any transitions that occur as nodes become active or fail.

A node appears active in the host membership group when the Group Services subsystem is active on that node. All such active nodes that can communicate with each other appear in this group.

The adapter membership groups

The Group Services subsystem also keeps track of the status of Ethernet and SP Switch adapters. The state of the adapters is reflected by the Group Services subsystem in two system-defined groups called the Ethernet adapter membership group and the SP Switch adapter membership group.

By subscribing to these groups, a GS client can obtain adapter membership information which, for example, it uses to determine communication paths to nodes.

The view of adapter membership implies that all of the nodes in the membership are able to communicate with each other over the IP network to which the adapters are connected. On the SP, this means that for Ethernet membership (**HA_GS_ENET_MEMBERSHIP**), the view from any one node is the set of all other nodes reachable from that node via the SP Ethernet. If the SP Ethernet is sundered (broken such that only subsets of nodes can communicate with each other), a node's view of Ethernet membership will be only the subset of nodes with which it can communicate.

Within the Group Services PSSP domain, adapter membership information is available for only a single Ethernet adapter on the SP control workstation, even if multiple Ethernet adapters are connected to the SP nodes.

For the SP Switch (**HA_GS_CSS_MEMBERSHIP**), the hardware does not allow sundering; if a node is on the switch, it can communicate with any other node on the switch. Thus, a local node's view of cssMembership is the global view of all nodes that are on the switch. A node must look at the given membership to determine if it is listed.

If HACMP/ES is installed on a node, and a GS client is connected to the Group Services HACMP/ES domain, there may be more adapter membership groups than simply the two described above. This is because heartbeating may take place on additional networks for HACMP/ES if the networks are installed and are defined to HACMP/ES. In this case, the semantics for these adapter membership groups match those as described for the Ethernet membership group. When a GS client receives a subscription notification for an adapter membership group, it will indicate the set of nodes with which this node can communicate across the given adapter type.

In summary, the information presented to a GS client for adapter membership subscriptions is not globally consistent (except for the **HA_GS_CSS_MEMBERSHIP** group). Since each node is presented with the set of nodes to which it can communicate for the given adapter type, different nodes may see different views. This can occur if the different networks are not fully connected to all nodes in the domain, or if there are failures of various routers between sections of the networks.

The subscription group **HA_GS_ALL_ADAPTER_MEMBERSHIP_GROUP** is like other adapter groups, such as **HA_GS_ENET_MEMBERSHIP_GROUP**, which a client can subscribe to for information about the members of the group. The difference is that the members or providers of the **HA_GS_ALL_ADAPTER_MEMBERSHIP_GROUP** include **all** adapters which are reported from Topology Services as being in the "up" state, including pluggable adapters. The subscription of other adapter groups will get only the information for a specific kind of adapter group For example, **HA_GS_ENET_MEMBERSHIP_GROUP** is for subscription of Ethernet adapter information.

For more information, see “ha_gs_subscribe” on page 112.

Quorum

Many applications require a form of quorum to ensure that the proper resources are available before the application begins operation. For example, one application may require a certain percentage of nodes to be up and running before it begins, while another requires particular nodes.

Because groups have significantly different requirements for quorum, the GSAPI does not provide a predefined quorum as part of its support. It is the responsibility of the application that is using the GSAPI to form groups that define and implement required quorum mechanisms. By manipulating the state information of the group, an application can build the required quorum mechanism.

Sundered networks

The Group Services subsystem provides a single group namespace within each system partition. Given the right set of multiple network failures, a system partition 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 previously existed, 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 if it needs a form of quorum, and use it to guide when a group is ready to perform its services.

Although the Group Services subsystem does not provide a quorum mechanism, it does provide some assistance to groups when a network is sundered. When a system partition 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 run 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 terminated abnormally 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.

Designing Group Services applications

There are things to consider as you design an application to use the Group Services application programming interface (GSAPI).

As you design an application to use the GSAPI, you should consider:
- How you code the callback routines.
- How you can code to get the best performance from your application.
- The implications of running in a mixed environment of AIX, Linux, Solaris, and Windows nodes.

Coding callback routines

The GSAPI 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 two following 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 could 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 increases the amount of work each must do to determine the type of notification it has received.

Refer to the ha_gs_dispatch Subroutine Description for additional information related to invoking callback functions and handling notifications.

Coordination of multiple notifications:

These multiprocessing considerations apply to all callback routines.

The Group Services subsystem presents all notifications to all providers in a single group in the same order. The providers should try to invoke the same callback routines in the same order. However, the Group Services subsystem verifies that, for n-phase protocols, all of the group's providers have reached the same control point before continuing to the next notification. In other cases, the providers may not receive and react to the notifications at the same time. For example, a provider might not receive a notification immediately because it is busy and not reading the socket.

If Group Services clients are providers in multiple groups, there is no guarantee that every provider receives the notifications from different groups in the same order.

For multi-threaded clients, it is assumed that the callback routines are thread-safe and reentrant. If the same callback routines are specified for multiple groups, a multi-threaded client can process notifications

by invoking the callback routines for more than one group at a time. For single-threaded providers, if they are acting as providers for multiple groups, they must also be coded to handle simultaneously running protocols in all groups.

In all cases where Group Services clients are acting as providers in multiple groups, it is the responsibility of the providers to ensure that they do not create deadlock situations across groups. An example of a deadlock is when one provider blocks before voting, waiting for another provider to take some action. Meanwhile, the second provider is blocked on another group protocol, waiting for the first provider to take some action.

The Group Services subsystem invokes callback routines only on the same thread or threads that are used to call the **ha_gs_dispatch** subroutine.

Coding for performance

There are guidelines to follow to get the best performance from your application.

To get the best performance from your application, keep the following guidelines in mind:

- Minimize the number of groups, providers, and nodes your application requires. The performance of your application depends on its size and distribution in the system. The greater the number of groups, the greater the number of providers joined to those groups. The greater the number of nodes across which each group is spread, the longer it will take to coordinate your application's activities.

  To the extent that it is possible, keep your application as small as possible.
- Minimize the size of the provider-broadcast messages that your application uses. The larger the messages, the greater the load on the network, particularly when a message must be broadcast to every provider in a group and a large number of subscribers as well.
- If possible, select the batching option to allow the Group Services subsystem to batch multiple join requests together. During group initialization, when all of the providers are joining their groups, each join request requires the invocation of a separate protocol. To decrease the load on the system, batch them together whenever possible.
- The actions taken during the barriers that are imposed by an n-phase protocol should be idempotent (designed so that they can run one or more times with no loss of correctness).

  Suppose that the providers of an application have been taking external actions during the barriers that are imposed by a multi-phase protocol, and that these actions must be completed for the application to be operational. Now suppose that the protocol terminates because of a failure. If the actions that the providers have already taken are not designed to be idempotent, the providers must explicitly undo the actions before they restart the protocol. Such an undo phase can be time-consuming and may require additional phases of coordination among the providers. However, if the actions are idempotent, there is no need for an explicit undo phase. The protocol can simply be restarted.

Coding message data for a mixed environment

Group Services can run in a mixed environment consisting of both AIX and Linux nodes.

Furthermore, it may be running in a mixed endianness cluster in which some nodes employ *Little Endian* byte order (in which the low order byte of a number is stored in the lowest memory address, and the high-order byte is stored at the highest memory address), while other nodes employ a *Big Endian* byte order (in which the high-order byte of a number is stored at the lowest memory address, and the low-order byte is stored at the highest memory address).

Because your Group Services application may be running in a mixed environment, you need to make sure that:

- you ship message data in a network-independent format. If message data is not shipped in a network-independent format, your code will not work in a mixed environment, because Group Services will not have the information to convert the message data.

- mixed endianness clusters are running the new level of codes. Older codes have no endian decoding functions and cannot handle messages from nodes with differing endianness.

IP addressing

RSCT uses the **ha_gs_ip_addr** data structure for accessing the IP address of the adapter corresponding to the given subscriber token and the provider ID.

Related concepts:

"Accessing Group Services IP address" on page 42
For IPv4-mapped addresses, the IPv4 part is accessed using **ip4** (**ipv4_in_6.ip4x**).

ha_gs_ip_addr data structure

The format of the **ha_gs_ip_addr** data structure follows:

```
typedef union {
    struct {
        uint32_t        filler[3];
        struct in_addr  ip4x;
    }ipv4_in_6;
    struct in6_addr     ip6;
} ha_gs_ip_addr
#define ip4 ipv4_in_6.ip4x
```

All addresses in the **ha_gs_ip_addr** data structure are in IPv6 format. IPv4 addresses are represented in IPv4-mapped IPv6 address format. Use the **ip4x** field if the IPv6 address is an IPv4-mapped address. Clients can use the **ip4x** field to access (32-bit) IPv4 addresses, though new clients are expected to be able to handle IPv6 addresses. The addresses are assumed to be stored in host byte order.

Compatibility with previous versions of ha_gs_ip_addr

For clients compiled with previous versions (RSCT 2.5.2 or earlier) of the Group Services library header file, the previous version of the **ha_gs_ip_addr** data structure:

```
typedef union {
    struct in_addr      ip4;
    struct in6_addr     ip6;
} ha_gs_ip_addr;
```

is applied. In this case, the Group Services library casts a pointer to the previous version of the **ha_gs_ip_addr** data structure.

Using the HA_GS_ENABLE_IPV6 option

Use the **HA_GS_ENABLE_IPV6** option in the **ha_gs_socket_ctrl_t** data type to indicate that the client can handle IPv6 addresses.

The ha_gs_socket_ctrl_t data type

The **ha_gs_socket_ctrl_t** data type is defined as follows:

```
typedef enum
{
    HA_GS_SOCKET_NO_SIGNAL              = 0x0000000,
    HA_GS_SOCKET_SIGNAL                 = 0x00000001,
    HA_GS_ENABLE_ADAPTER_INFO           = 0x00000002,
    HA_GS_ENABLE_DOMAIN_EVENT           = 0x00001000,
    HA_GS_ENABLE_IPV6                   = 0x00002000,
    HA_GS_ENABLE_MIGRATION_CALLBACK     = 0x00004000,
    HA_GS_STREAM                        = 0x00800000,
    HA_GS_IMMEDIATE_DOMAIN_CONTROL      = 0x10000000
} ha_gs_socket_ctrl_t;
```

The **HA_GS_ENABLE_IPV6** option allows client programs that are compiled with the current Group Services library header file to work without source code changes, even in an environment where IPv6 addresses are present. Clients that use this option are expected to be able to handle IPv6 addresses. Clients that are compiled using the **ha_gs.h** file in RSCT version 2.5.3 or later and the **HA_GS_ENABLE_IPV6** option get an IPv6 address in the **.ipv6** field. If this option is not used, only IPv4 addresses will be returned by the GSAPI. Clients compiled using the **ha_gs.h** file in RSCT version 2.5.3 or later that do not use the **HA_GS_ENABLE_IPV6** option get IPv4 addresses in the **.ipv4** field.

Related reference:

"ha_gs_init" on page 80

Configuring Group Services

A set of AIX® systems can be configured as an RSCT peer domain using the configuration resource manager. All services provided by the Group Services subsystem are within a single domain only.

Enabling the collection of voting results

After a proposal is approved or rejected, the Group Services subsystem notifies all of the providers of the outcome.

Sometimes, however, it may be useful for a provider to have more details about the vote in addition to the outcome. When calling the **ha_gs_join** subroutine to join a group as a provider, the GS client can enable the collection of voting results. When collection of voting results is enabled, more details about the vote will be reported to the provider through the callback routines **ha_gs_n_phase_callback**, **hs_gs_protocol_approved_callback**, and **ha_gs_protocol_rejected_callback**. Specifically, the following voting result information, for each provider, will be available through the callback routines.

- The provider's vote.
- If the provider failed, the failure reason.
- If the provider proposed a new state value, the proposed state value.
- If the provider proposed a new broadcast message, the proposed message.

By providing the detailed voting result information through the callback routines, Group Services frees the application from having to call additional protocols to get the same information. However, this option collects voting results for all providers and all proposals and so increases network traffic. A GS client can enable voting result collection when it joins a group as a provider, and the collection is then carried out for all proposals. You should be aware, particularly in the case of very large clusters, that message traffic and size will be increased by voting result collection.

To enable the collection of voting results, include one of the following flags in the **gs_batch_control** field of the **ha_gs_join** subroutine.

- **HA_GS_COLLECT_VOTE_RESULT** (collect only the vote list)
- **HA_GS_COLLECT_MSG_RESULT** (collect only the message list)
- **HA_GS_COLLECT_STATEVALUE_RESULT** (collect only the state value list)
- **HA_GS_COLLECT_ALL_RESULT** (collect all lists)

The results will be reported in the **gs_vote_result** field of the callback routines **ha_gs_n_phase_callback**, **hs_gs_protocol_approved_callback**, and **ha_gs_protocol_rejected_callback**.

When one of the voting collection flags is set on a group attribute, the *n*-phase protocol notification will contain the voting results for all providers. As a result, for clusters with large numbers of nodes, the message size and the traffic between the daemon and the library will be large. This should be considered before using the voting collection flags on a group attribute.

Related reference:

“ha_gs_join” on page 85
“ha_gs_n_phase_callback” on page 94

Connecting to Group Services domains

A Group Services client can connect to only *one* domain, regardless of whether a node (or the control workstation) is part of multiple Group Services domains.

All services provided by the Group Services subsystem are within a single domain only. A Group Services client gets information only about the nodes and groups that are in the Group Services domain to which it is connected.

To connect to a Group Services PSSP domain on its node, a group services client must set the **HA_SYSPAR_NAME** (or **HA_DOMAIN_NAME**) and **HA_GS_SUBSYS** environment variables to the PSSP partition name and **hags** respectively, before it invokes the **ha_gs_init** subroutine.

To connect to the RSCT peer domain on its nodes, none of those environment variables (**HA_SYSPAR_NAME**, **HA_DOMAIN_NAME**, and **HA_GS_SUBSYS**) should be set. If anything is set, the subroutine **ha_gs_init** may try to connect to Group Services for the PSSP or HACMP/ES domain, and thus the subroutine **ha_gs_init** may fail.

All of the Group Services interfaces and semantics work identically and are supported within the three types of Group Services domains.

Managing Group Services

You can manage system resources, which may or may not be subsystems, that use Group Services to be highly available. This section contains information that helps you to write new clients that use the Group Services application programming interface (GSAPI) or add the use of Group Services to existing programs.

Deactivating scripts

To handle a situation in which a provider must be expelled, or in which it is failing, a provider can specify a deactivate script on the **ha_gs_init** subroutine when it first registers with Group Services. 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 can be run, until an expel protocol is invoked. If the specified deactivate script is not found or cannot be run, Group Services applies the group's default vote value for the phase in which the deactivate script should have been invoked, and for each subsequent voting phase, if there are any.

A valid deactivate script is run as follows. For each provider targeted by the expel protocol, the Group Services daemon on the provider's node forks a child process that tries to run the deactivate script, using the following environment:

Effective uid and gid

The forked process runs with the effective uid and gid of the targeted provider that it had when it registered with Group Services by its call to the **ha_gs_init** subroutine. If the provider changed its uid or gid after calling **ha_gs_init**, the deactivate script still uses the effective uid and gid from the time when **ha_gs_init** was called. A deactivate script with a set uid bit in its file permissions runs with those values.

Working directory

The forked process begins running in the current working directory of the targeted provider that it had when it registered with Group Services by its call to the **ha_gs_init** subroutine. If the

provider changed its current working directory after calling **ha_gs_init**, the deactivate script still uses the current working directory that existed when **ha_gs_init** was called. A deactivate script that wants to run 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.

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

- The process ID parameter is always zero when the script is run for deactivate-on-failure handling. (See “Expel protocol” on page 14.)
- The voting time limit of the expel protocol, in seconds, as an *int* (4 bytes) The deactivate script must complete and exit within this limit.
- The name of the failed provider's group, as a null-terminated string
- The deactivate flag, which is the null-terminated string *providerdied*. The deactivate script can distinguish when it is called by checking this deactivate flag.
- The list of failed provider's instance numbers, separated by commas. This parameter will be presented only for deactivate-on-failure handling. When batching of failures is enabled, the deactivate script can be run once for the multiple providers' failure. This fifth parameter tells which providers were failing. Note that each provider instance number does not contain the node number.

On output, the 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.

On receipt of an exit code indicating a successful completion before the time limit expires, Group Services votes APPROVE for this voting phase of the protocol. On receipt of an exit code indicating an unsuccessful completion 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 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.

Changing membership and state value

Any provider in the group can ask Group Services to modify the state value and can also specify the level of consistency that is to be associated with the modification.

Specifically, the provider can subject the proposed change to a voting protocol, or request that the change be approved without putting it to a vote. The voting protocol unifies the multi-phase commit and barrier synchronization abstractions.

A GS client that asks to become a provider of a group must have the correct authorization and must be admitted to the group by the current providers of the group. This is accomplished by the same voting protocol as the one used to mediate state changes.

A provider may leave a group in a number of ways. It may:

- Leave voluntarily

- Be expelled at the request of another provider
- Leave involuntarily when its process, or the node on which it is running, fails.

All changes to a group, in state value or in membership, appear to the providers and subscribers of the group to be logically serialized. This means that one change completes before another begins. The Group Services subsystem processes all outstanding membership changes before it accepts any proposals to change the state value. If one or more providers fail during an ongoing protocol invocation, Group Services runs a failure leave protocol to remove the failed providers from the group when the protocol completes.

Subscribers are notified of the approved results of a state value change or a provider membership change. However, they do not participate in approving a state value change, admitting a new provider, or removing a leaving provider. Thus, a subscriber cannot affect the group state. In addition, subscribers do not appear in any group membership lists; they are known only to the Group Services subsystem. The providers of a group and the other subscribers of the group are unaware of any of the subscribers to the group.

Related concepts:

"Group membership" on page 4
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. Any authorized process in the domain may ask to become a member of a group.

Initiating group services protocols

Either a provider or the Group Services subsystem itself initiates proposals.

The Group Services subsystem proposes protocols to handle a join, a failure leave or a cast-out of a provider, or a source-group reflection protocol when a source-group state value change needs to be reflected to its target-groups.

It is up to the providers to initiate proposals to voluntarily leave a group, to expel a provider, to change the state value of a group, or to initiate a provider-broadcast message.

Please note carefully the difference here between initiating the protocol proposal and notifying the providers that a protocol has been proposed. The Group Services subsystem always issues the notification. However, the Group Services subsystem actually initiates a proposal only for the cases listed above.

For a provider to initiate a protocol proposal, it is simply a matter of calling the proper GSAPI subroutine. The Group Services subsystem notifies the other providers in the group that a proposal has been made and proceeds as described in "Proposing, voting, and phases for protocols" on page 11, based on the number of phases and the nature of the proposal.

Group Services subsystem-initiated protocols

The protocols that the Group Services subsystem initiates are join, failure leave, and source-target proposal protocols. Once it initiates the protocols, the Group Services subsystem notifies the providers, and the protocols proceed in a manner quite similar to other proposals.

The protocols that the Group Services subsystem initiates cover the following situations:

- A membership change proposal to join a group by a potential provider
- A membership change proposal for a failure leave of one or more failed providers
- A membership change proposal to cast out one or more providers, due to source-target processing
- A source-state reflection proposal to reflect to a target-group when its source-group has changed its state value through a non-membership change protocol.

The number of phases for these protocols is determined as follows:

- For a join proposal, the provider must specify either a one-phase or an n-phase join protocol. The first join request to a group by the first provider places the phase setting in the **gs_num_phases** field of the group attributes block for the group. All subsequent membership change protocols must match the setting.
- For failure leaves and cast-out proposals, the Group Services subsystem uses the **gs_num_phases** setting from the group attributes block.
- For source-state reflection protocols, the Group Services subsystem uses the **gs_source_reflection_num_phases** setting from the group attributes block to control the number of phases.

Time limits for voting phases are determined as follows:

- For join, failure leave, and cast-out protocols, the **gs_time_limit** field in the group attributes block is used.
- For source reflection protocols, the **gs_source_reflection_time_limit** field in the group attributes block is used.

The notification procedure varies slightly, depending on the proposal that is made, as follows:

- For a join proposal, the Group Services subsystem notifies all of the providers, including the "old" providers that are already in the group, and the providers asking to join the group. The notification specifies the "old" and joining providers.
- For a failure leave proposal, the Group Services subsystem notifies the remaining providers of the protocol proposal.
- For a cast-out proposal, the Group Services subsystem notifies all of the providers except those that are being cast out. A provider that is being cast out receives a final notification but does not receive interim notifications that occur while the cast-out is being voted on.

Membership changes may be batched, which means that multiple providers can be handled in a single join or leave protocol.

Once the Group Services subsystem has initiated the protocol, the providers invoke it in the usual manner based on the number of phases. One-phase protocols are a single notification. n-phase protocols proceed to the first voting phase.

Once voting has completed, the providers are notified of the result.

A rejection of a source-state reflection protocol simply ends the protocol, and the providers are notified that the protocol is rejected.

Approval of the Group Services subsystem-initiated protocols:

In all cases, if the protocol is approved, Group Services performs these actions.

- All providers receive an updated membership list or state value. They may also receive a provider-broadcast message, if one was included in the final vote.
- Subscribers receive the updated membership list or status value, depending on their subscription request.
- An approved source-state reflection protocol results in a notification that the protocol has completed. The state value is updated only if a provider submitted a state value with a voting response.

Rejection of the Group Services subsystem-initiated protocols:

If a join is rejected for any reason, the rejected GS clients receive a notification that their application to join the group has been rejected. The existing providers also receive the notification. The subscribers receive no notification.

The failure leave and cast-out proposals require some special handling for rejecting the protocols, because the failing providers must be removed in any case.

If any provider **explicitly** votes to REJECT a failure leave or cast-out proposal, the protocol stops. The membership list is updated to show the removal of the targeted provider. The providers and subscribers receive this updated list. The state value reverts to its value at the beginning of the protocol.

If any provider **implicitly** votes to REJECT a failure leave or cast-out proposal, the protocol stops. If failure leave requests are allowed to be batched, the Group Services subsystem immediately proposes another failure leave protocol, adding the newly-failed provider into the list of leaving providers.

If failure leave requests are not allowed to be batched, the Group Services subsystem handles this as an explicit vote to REJECT. To handle the newly-failed provider, Group Services initiates a failure leave protocol.

Provider-initiated protocols

Provider-initiated proposals include proposals made by a provider to perform certain operations.

Provider-initiated proposals include proposals made by a provider to perform the following:
- Join a group
- Leave a group voluntarily
- Expel one or more providers from the group
- Change the group state value
- Broadcast a provider-broadcast message to all providers.
- Change attributes
- Goodbye

A provider calls a GSAPI subroutine to propose one of these protocols, specifying either a one-phase or an n-phase protocol. If an n-phase protocol is specified for one of these protocols, the provider must also specify a voting phase time limit as well. However, if no time limit is desired, a time limit of 0 may be specified.

The Group Services subsystem checks the proposal for errors. If the proposal is syntactically invalid, the provider receives a synchronous syntax error code. If the group currently has a running protocol, the provider receives a synchronous error code that indicates a collision between competing protocols. (Only one protocol may run at a time.)

If the synchronous checks pass, the Group Services subsystem tentatively accepts the proposal and the provider receives a synchronous successful return code. However, if collision errors are detected asynchronously (because other providers or the Group Services subsystem itself submits a proposal at the same time), the Group Services subsystem returns an error code asynchronously.

If multiple providers submit proposals at the same time, only one proposal is accepted by the Group Services subsystem. The other proposals are returned to the providers that made them, asynchronously returning a collision error code.

Whichever proposal the Group Services subsystem chooses, the providers are notified. If the protocol is a one-phase protocol, the proposal is automatically approved. If the protocol is an n-phase protocol, the proposal notification requests a vote from the providers.

At the end of the protocol, the Group Services subsystem notifies the providers of the protocol's result, that is, whether the protocol was approved or rejected.

For voluntary leave protocols:

- If a leave protocol is approved, all remaining providers receive the updated membership list and, if it changed, the updated state value. Subscribers receive the updated membership list or state value, based on their subscription. The provider targeted by the protocol is sent the initial notification that its leave protocol is running.
- If a leave is rejected, it ends the protocol. However, the provider who proposed the leave is still removed from the group. The membership list is updated to show the removal of the targeted providers. The providers and subscribers receive this updated list. The state value reverts to its value at the beginning of the protocol.

For expel protocols, see “Expel protocol” on page 14.

For state value change protocols:

- If a state value change is approved, the providers and subscribers receive the updated state value. If the group is a source-group, its target-groups also receive notification of the change.
- If a state value change protocol is rejected, the state value remains unchanged. The providers receive notification of the rejection. The subscribers receive no notification.

For provider-broadcast message protocols:

- If it is a one-phase protocol, the message contained in the proposal is broadcast to all providers. Subscribers receive no notification.
- If it is an n-phase protocol, and it is:
  - Approved, and if the group state value was changed during the voting phases, the providers and subscribers receive the updated state value.
  - Approved, but the group state value was not changed during the voting phases, the providers receive notice that the protocol is completed. Subscribers receive no notification.
  - Rejected, the state value remains unchanged. The providers receive notification of the rejection. The subscribers receive no notification.

For change-attributes protocols:

- If a change-attributes protocol is approved, the providers receive the updated group attributes. Subscribers receive no notification of the attribute change.
- If a change-attributes protocol is rejected, the group attributes remain unchanged. The providers receive notification of the rejection. The subscribers receive no notification.

Note that any n-phase protocol can propose a change to the group state value. If the group state value change is accepted:

- The providers and subscribers receive the updated state value.
- If the group is a source-group, its target-groups also receive notification of the change.

Submitting changes with voting responses

The voting response to each phase of an n-phase protocol may contain a proposed new group state value, a provider-broadcast message, and a proposed new default vote for the group.

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 during a voting response, the Group Services subsystem broadcasts it to all providers as part of the notification for the next phase of the protocol.

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

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

Because each provider must issue a vote response, each provider could submit with its vote a proposed updated state value, a provider-broadcast message, or a new default vote. In case of multiple submissions, the Group Services subsystem chooses only one of the values to propagate to the providers for the next phase notification. The Group Services subsystem considers only the providers who do not specify a null pointer to a state value or message. For these, the Group Services subsystem arbitrarily chooses one of the responses it receives from the group. Because the providers cannot control which response is chosen, they should guarantee that one of the following is true:

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

If these rules are not followed, it is not possible to determine which response will be chosen to be propagated for the next phase.

Related concepts:

"Specifying the provider's default vote value" on page 10
By default, the Group Services subsystem assigns REJECT as the default value for each group.

Ending a protocol

The end of the protocol is signaled by the end of the voting phases for the protocol. In the case of a one-phase protocol, the end phase is the only phase.

In the case of an n-phase protocol, the voting phases can end in one of the following ways:

- If any provider votes to REJECT the proposal in any voting phase, the proposal is rejected.
- If a default vote of REJECT is entered, the proposal is rejected.
- If all providers vote to APPROVE the proposal, or default votes of APPROVE are entered in the same voting phase, the proposal is approved.

In all cases, the end phase consists of a broadcast of the results of the protocol just processed.

For approved proposals, a notification is sent to all providers and subscribers. The notification contains a flag that specifies whether any default votes were used to approve the protocol. Providers receive this information, but subscribers do not. The contents of the notification is also determined as follows:

- If there was a membership change, it contains the updated membership list. Both providers and subscribers receive this information.
- If there was a proposal to change the group state value, it contains the new group state value. Both providers and subscribers receive this information.
- If a provider-broadcast message was submitted on the final vote, it contains the message. Providers receive this information; subscribers do not.

For rejected proposals, a notification is sent only to providers. The notification contains the following information:

- An indication that the proposal was rejected

- A flag that specifies why the proposal was rejected. Reasons include: there was an explicit vote to REJECT, a default vote to REJECT was submitted on behalf of a failed provider, or the protocol was ended because a provider exceeded the specified time limit.

Accessing Group Services IP address

For IPv4-mapped addresses, the IPv4 part is accessed using **ip4** (**ipv4_in_6.ip4x**).

To access the IPv4 and IPv6 addresses in **ha_gs_ip_addr**, enter:

```
ha_gs_ip_addr ip;

HA_GS_ENABLE_IPV6    is not set:

  IPv4:   ip.ip4
  IPv6:   ------

HA_GS_ENABLE_IPV6    is set:

  IPv4:   ip.ip4
  IPv6:   ip.ip6
```

To test which kind of address is present, if **HA_GS_ENABLE_IPV6** has been specified, use the **IN6_ISADDR_V4MAPPED()** function:

```
if(IN6_IS_ADDR_V4MAPPED(&ip.ip6))
```

A successful test indicates that **ha_gs_ip_addr** contains an IPv4 address.

Related concepts:

"IP addressing" on page 33
RSCT uses the **ha_gs_ip_addr** data structure for accessing the IP address of the adapter corresponding to the given subscriber token and the provider ID.

Receiving Group Services subscription special data

The subscription special data format is very flexible, due to the wide variety of data that is carried in this section.

The way to view this data is as a linked list of **ha_gs_special_block_t** elements, each containing a single type of special data related to the overall subscription notification. The other fields of the subscription notification carry the normal group state value and membership information. The special data carries additional identification information for the entries listed in the group state value and membership fields. The following subscription special data items have been added:

- Adapter death or deconfiguration indications
- SP Switch adapter alias information

The **ha_gs_subscribe** and **ha_gs_subscriber_callback** subroutines can also support IP-address-based membership lists, which makes programming easier.

To receive the special data for the adapter group, the **ha_gs_subscribe** request must enable **HA_GS_SUBSCRIBE_SPECIAL_DATA** in the **gs_subscription_control** field of the **ha_gs_subscribe_request_t** block. If it is not supported for a certain adapter group, **ha_gs_subscribe** fails with **HA_GS_NOT_SUPPORTED**.

If the subscription notification contains the special data block on the **subscriber_callback**, the **gs_subscription_type** field of the **ha_gs_subscription_notification_t** block will include the **HA_GS_SUBSCRIPTION_SPECIAL_DATA** flag. More detailed information is located in "ha_gs_subscribe" on page 112 and "ha_gs_subscriber_callback" on page 118.

Note: This special data block subscription is automatically enabled for the raw switch adapter membership, **HA_GS_CSSRAW_MEMBERSHIP**, for compatibility with older applications.

Prerequisite system conditions for receiving subscription special data

At a minumum, a node must be at RSCT 1.2, PSSP 3.1, or HACMP™ 4.3, for both Group Services and Topology Services. This is because adapter membership reporting is a purely local issue for Group Services. However, if clients do not care about the information conveyed by the special data, they can simply ignore the special data. All functions should work exactly as they expect today.

General representation structure

If the **HA_GS_SUBSCRIPTION_SPECIAL_DATA** flag is set in the subscription notification, the **gs_subscription_special_data** field will point to a special data block.

The special data block will be of the following format:

```
typedef struct {
 int                    gs_length;
 unsigned int           gs_flag;
 ha_gs_special_block_t  *gs_special_data;
} ha_gs_special_data_t;
```

This block acts as a header to point to a linked list of **ha_gs_special_data_t** elements.

gs_length

The number of **ha_gs_special_data_t** elements in the list.

gs_flag

The result of OR'ing together all of the flags defining the types of **ha_gs_special_data_t** data included in the list. This allows the subscriber to quickly investigate this flag. If none of the included **ha_gs_special_data_t** types are of interest, it need not continue investigating the block any further.

gs_special_data

A pointer to the first **ha_gs_special_data_t** element in the list.

Each **ha_gs_special_block_t** contains a single type of subscription special data and looks like the following:

```
typedef struct {
 unsigned int           gs_special_flag;
 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;
```

This block contains the actual subscription special data.

gs_special_flag

The flag identifying the type of this block's special data.

gs_next_special_block

A pointer to the next **ha_gs_special_block_t** in the list. If this is the last **ha_gs_special_block_t**, this value will be NULL.

gs_special_num_entries

The number of entries contained in the data pointed to by the **gs_special** field for this **ha_gs_special_block_t**. If this entry is one, then the **gs_special_length** field is the total size of the data. If this entry is greater than one, the total number of bytes is this value multiplied by **gs_special_length**.

gs_special_length

The number of bytes in each of the entries in the block pointed to by the **gs_special** field for this **ha_gs_special_block_t**.

gs_special

A pointer to the actual data.

The **gs_flag** field in the **gs_special_data_t** block and the **gs_special_flag** field in each **ha_gs_special_block_t** may contain the following flags. The **gs_special_flag** field contains only one, whichever kind of data its **ha_gs_special_block_t** represents. The **gs_flag** will contain all of the individual **gs_special_flags** OR'ed together.

```
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;
```

These flags describe the contents of the **ha_gs_special_block_t** that may be included.

HA_GS_ADAPTER_DEATH_ARRAY

Indicates that there is a **ha_gs_special_block_t** containing the "death reasons" for each adapter listed in the **gs_changing_membership** field.

HA_GS_CURRENT_ADAPTER_ALIAS_ARRAY

Indicates that there is a **ha_gs_special_block_t** containing the alias IP addresses of each adapter listed in the **gs_current_membership** field.

HA_GS_CHANGING_ADAPTER_ALIAS_ARRAY

Indicates that there is a **ha_gs_special_block_t** containing the alias IP addresses of each adapter listed in the **gs_changing_membership** field.

Reporting adapter death events

For dealing with adapter death versus deconfiguration, the **ha_gs_special_block_t**, if included, will contain an array of **ha_gs_adapter_death_t** entries, each containing flags with the death status of that adapter.

The ordering of the array entries will match that in the **gs_changing_membership** field.

If all adapters listed in the notification suffered a normal failure (that is, each adapter is still part of the configuration but is no longer operational):

There will be no change from the current subscription notifications, and **HA_GS_SUBSCRIPTION_SPECIAL_DATA** will not be set into the **gs_subscription_type** field in the subscription notification.

- The client should simply assume that all reported adapter deaths were natural; none failed due to being removed from the configuration.
- The **gs_subscription_special_data** field will be NULL.

If one or more adapters listed in the notification failed due to being removed from the configuration:

- **HA_GS_SUBSCRIPTION_SPECIAL_DATA** will be OR'ed into the **gs_subscription_type** field in the subscription notification.
- The **gs_subscription_special_data** field in the subscription notification will point to an **ha_gs_special_data** block.
  - The **gs_special_block_count** field will contain the count of **ha_gs_special_block_t** elements.
  - The **gs_special_block_flags** field will contain **HA_GS_ADAPTER_DEATH_ARRAY**.
  - The **gs_subscription_special_block** field will point to the **ha_gs_special_block_t**.

- The **ha_gs_special_block_t** fields will contain the following values:

  gs_special_flag field

  Will contain **HA_GS_ADAPTER_DEATH_ARRAY**.

  gs_special_num_entries field

  Will contain the same number of entries as the number of adapters that were listed in the **gs_changing_membership** field.

  gs_special_length field

  Will contain the number 4, to indicate that each entry is four bytes long.

  gs_special field

  Will point to an array of four-byte entries, listed in the order matching the order of adapters in the **gs_changing_membership** field. Each entry will contain one of the following flags:

```
typedef enum
{
 HA_GS_ADAPTER_DEAD     = 0x0001,
 HA_GS_ADAPTER_REMOVED  = 0x0002
} ha_gs_adapter_death_t;
```

  – If an adapter died of natural causes, the flag will be **HA_GS_ADAPTER_DEAD**.

  – If an adapter was removed from the configuration, the flag will be **HA_GS_ADAPTER_REMOVED.**

Dealing with adapter events with multiple aliases

The representation of adapters in subscription notifications is to use the **provider_id** and use the instance number to represent the physical adapter name.

This allows the subscriber to associate the list of adapters from the notification with the physical adapters to which the events relate. However, this support is not sufficient in certain cases, where a single adapter is represented multiple times due to the use of IP alias addresses.

To allow subscribers to differentiate among the different aliases, if a subscription notification includes one or more adapters with multiple aliases, a **ha_gs_special_block_t** will be included that has an array of the IP alias address associated with the adapter listed in the changing membership. The ordering of the array entries will match that in the **gs_current_membership** and **gs_changing_membership** fields.

If the network *is not* **defined as supporting multiple aliases:**

The **HA_GS_SUBSCRIPTION_SPECIAL_DATA** will not be set into the **gs_subscription_type** field in the subscription notification.

- The client should simply act as it currently does, and use the instance number to associate the listed adapter with the physical adapter.
- The **gs_subscription_special_data** field will be NULL.

If the network *is* **defined as supporting multiple aliases:**

- **HA_GS_SUBSCRIPTION_SPECIAL_DATA** will be OR'ed into the **gs_subscription_type** field in the subscription notification.
- The **gs_subscription_special_data** field in the subscription notification will point to an **ha_gs_special_data** block.
  – The **gs_special_block_count** field will contain the count of **ha_gs_special_block_t** elements.
  – The **gs_special_block_flags** field will contain one or both of the following: **HA_GS_CURRENT_ADAPTER_ALIAS_ARRAY** and **HA_GS_CHANGING_ADAPTER_ALIAS_ARRAY**.

- The **gs_subscription_special_data** field in the subscription notification will point to one or more **ha_gs_special_block_t** elements.
  - If the subscription notification contains a list of providers in the **gs_current_membership** field, there will be a **ha_gs_special_block_t** with its **gs_special_flag** field containing **HA_GS_CURRENT_ADAPTER_ALIAS_ARRAY**.
  - If the subscription notification contains a list of providers in the **gs_changing_membership** field, there will be a **ha_gs_special_block_t** with its **gs_special_flag** field containing **HA_GS_CHANGING_ADAPTER_ALIAS_ARRAY**.
- For each **ha_gs_special_block_t**, the following will be true:
  - The **gs_special_num_entries** field will contain the same number of entries as the number of adapters that were listed in the **gs_changing_membership** field.
  - The **gs_special_length** field will contain 4.
  - The **gs_special** field will point to the array of entries, listed in the order matching the order of adapters in the **gs_current_membership** and **gs_changing_membership** fields. Each entry will contain the IP address of the adapter's alias affected by the notification (death or join).

Multiple ha_gs_special_block_t elements

It is possible for a single subscription notification to contain adapters which were deconfigured and which are mapped with multiple alias addresses.

In this case, both of the above types of special data will be included. In this situation, the following conditions hold:

- **HA_GS_SUBSCRIPTION_SPECIAL_DATA** will be OR'ed into the **gs_subscription_type** field in the subscription notification.
- The **gs_subscription_special_data** field in the subscription notification will point to an **ha_gs_special_data** block.
  - The **gs_special_block_count** field will contain the count of **ha_gs_special_block_t** elements, which in this case may be two (2) or three (3).
  - The **gs_special_block_flags** field will contain a value of the OR'ed together flags **HA_GS_CURRENT_ADAPTER_ALIAS_ARRAY**, **HA_GS_CHANGING_ADAPTER_ALIAS_ARRAY**, and **HA_GS_ADAPTER_DEATH_ARRAY**.
  - The **gs_subscription_special_data** field in the subscription notification will point to the first **ha_gs_special_block_t**.
    - It is unpredictable which **ha_gs_special_block_t** will be first in the list.
    - The **gs_next_special_block** of each **ha_gs_special_block_t** will point to the next **ha_gs_special_block_t**.
    - The **gs_next_special_block** of the last **ha_gs_special_block_t** will be NULL.

GSAPI library names

One of several libraries should be linked for AIX applications that use the Group Services subroutines.

For AIX applications that use the Group Services subroutines, one of these libraries should be linked:
- GSAPI thread-safe library (**libha_gs_r.a**)
- GSAPI non-thread-safe library (**libha_gs.a**)

For Linux applications that use the Group Services subroutines, one of these libraries should be linked:
- GSAPI thread-safe library (**libha_gs_r.so**)
- GSAPI non-thread-safe library (**libha_gs.so**)

GSAPI subroutines

GSAPI subroutines request specific actions from the Group Services subsystem.

The GSAPI contains these types of subroutines:
- Subroutines to issue commands that request an action from the Group Services subsystem.
- Subroutines that define callback routines that handle notifications from the Group Services subsystem.
- Deactivation scripts that the Group Services subsystem runs against providers that are targeted for expulsion from a group.
- A first failure data capture subroutine.

Group Services supports both the 32-bit and the 64-bit operating environments.

The following GSAPI subroutines request an action from the Group Services subsystem:

Subroutine	Action
ha_gs_change_attributes	Dynamically change certain group attributes.
ha_gs_change_state_value	Propose a change to the group state value.
ha_gs_dispatch	Check for notifications.
ha_gs_expel	Expel one or more providers from the group.
ha_gs_get_adapter_info	Get the node number, adapter interface name, hb_network_name and hb_network_type corresponding to a given IP address. For compatibility purposes, will return the node number and adapter interface name when invoked by an older application.
ha_gs_get_adapter_info_by_addr	Get the node number, adapter interface name, hb_network_name and hb_network_type corresponding to a given IP address.
ha_gs_get_adapter_info_by_id	Get adapter information, including the IP address, node number, adapter interface name, network name, and network type, corresponding to a given subscriber token and member ID of a subscribed adapter group.
ha_gs_get_ffdc_id	Obtain an FFDC ID to find the cause when the **hags** daemon stops suddenly and unexpectedly.
ha_gs_get_ipaddr_by_id	Get the IP address corresponding to a given subscriber token and a provider ID of a subscribed adapter group.
ha_gs_get_limits	Get the maximum limits of the current Group Services capacity.
ha_gs_get_node_number	Get the local node number.
ha_gs_get_rsct_active_version	Get the current active version of RSCT.
ha_gs_get_rsct_installed_version	Get the local installed version of RSCT.
ha_gs_goodbye	Leave a group immediately.
ha_gs_init	Register with Group Services.
ha_gs_join	Join a group as a provider.
ha_gs_leave	Leave a group (as a provider).
ha_gs_quit	Terminate the connection to the Group Services subsystem.
ha_gs_send_message	Send data to all of the providers in the group.
ha_gs_subscribe	Subscribe to a group.
ha_gs_unsubscribe	Unregister as a subscriber to a group.
ha_gs_vote	Vote on a proposed change to a group membership, state value, attribute, and so forth, by approving, rejecting, or continuing.

The following GSAPI subroutines define callback routines to handle notifications from the Group Services subsystem that something is happening:

Subroutine	Response
ha_gs_announcement_callback	Respond to an announcement that: • One or more providers failed a responsiveness check. • One or more providers that previously failed responsiveness checks are now responding successfully. • The Group Services daemon has died or is about to die. • The voting time limit has expired.
ha_gs_delayed_error_callback	Handle an asynchronously presented error.
ha_gs_n_phase_callback	Respond to a request for a vote on a proposed join, leave, expel, cast-out, failure leave, change attribute, or group state change request.
ha_gs_protocol_approved_callback	Respond to a notification that a proposal has been approved.
ha_gs_protocol_rejected_callback	Respond to a notification that a proposal has been rejected.
ha_gs_responsiveness_callback	Respond to a responsiveness check.
ha_gs_subscriber_callback	Receive a notification that a subscribed-to group membership or state has been changed.

For the specifications of deactivation scripts to use with expel protocols and with deactivate-on-failure handling, see "Deactivate-on-failure handling" on page 16.

The **ha_gs_get_ffdc_id** GSAPI subroutine retrieves a related first failure data capture (FFDC) ID from the Group Services subsystem, provided it is available, when the Group Services daemon fails.

ha_gs_announcement_callback

Purpose

ha_gs_announcement_callback – A callback routine that the Group Services subsystem calls to deliver an announcement notification to a GS client

Library

See "GSAPI library names" on page 46.

Syntax

```
#include <ha_gs.h>

void
    ha_gs_announcement_callback(
        const   ha_gs_announcement_notification_t       *notification)
```

Parameters

notification
: A pointer to an announcement notification block.

Description

The **ha_gs_announcement_callback** subroutine defines a GS client's announcement callback routine. The GS client uses it to handle announcement notifications from the Group Services subsystem. The process provides the address of the announcement callback routine to the Group Services subsystem on the **ha_gs_join** subroutine when it joins the group as a provider. The Group Services subsystem then calls the announcement callback routine when it has an announcement notification to deliver to the GS client. Announcements provide detailed information about abnormal conditions, other than complete failure, that affect one or more providers in the group.

On input, the announcement callback routine receives a pointer to the announcement notification block. The announcement notification block has the following definition:

```
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;
```

The **gs_notification_type** field contains the type of notification. For an announcement notification, it contains a value of **HA_GS_ANNOUNCEMENT_NOTIFICATION**.

The **gs_provider_token** field contains 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.

The **gs_summary_code** field contains one or more flags that indicate the type of announcement that is being delivered. It can contain one or more of the following flags:

HA_GS_TIME_LIMIT_EXCEEDED
: This flag is set for an announcement notification when one or more providers failed to vote in time during a voting protocol. The **gs_announcement** field of the announcement notification block points to the list of providers that failed to vote in time.

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.

The **gs_announcement** field points to a list of providers that are affected by the condition that is being reported by this announcement. It has the following definition:

```
typedef struct {
    ulong               gs_count;
    ha_gs_provider_t    *gs_providers;
} ha_gs_membership_t;
```

The **gs_count** field contains the number of providers in the list.

The **gs_providers** field points to the list of providers. Each provider is described by a provider information block, which is defined in the **ha_gs_n_phase_callback** man page.

Restrictions

For important information about multiprocessing considerations that apply to all callback routines, see the **ha_gs_n_phase_callback** man page.

Return values

None.

Error values

None.

Asynchronous errors

None.

Files

ha_gs.h

Related reference:

"ha_gs_init" on page 80
"ha_gs_join" on page 85
"ha_gs_change_state_value" on page 54
"ha_gs_send_message" on page 110
"ha_gs_leave" on page 91
"ha_gs_expel" on page 62
"ha_gs_change_attributes"

ha_gs_change_attributes

Purpose

ha_gs_change_attributes – Called by a provider of a group to propose a change to the group's attributes

Library

See "GSAPI library names" on page 46.

Syntax

```
#include <ha_gs.h>

ha_gs_rc_t
    ha_gs_change_attributes(
                ha_gs_token_t          provider_token,
        const   ha_gs_proposal_info_t   *proposal_info)
```

Parameters

provider_token
: 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.

proposal_info
: A pointer to a buffer that contains a proposal information block, which describes the proposed attribute change request.

Description

The **ha_gs_change_attributes** subroutine is used by a provider of a Group Services group to propose a change to the group's attribute value.

If the request is specified as a one-phase protocol, and Group Services chooses to run this protocol, the group's providers are notified using normal protocol approval procedures.

If the request is specified as an n-phase protocol, and Group Services chooses to run this protocol, the group's providers are notified using normal n-phase voting procedures.

If the Group Services subsystem chooses not to run this protocol (because another protocol is already in progress), the **HA_GS_COLLIDE** error number is returned either synchronously or asynchronously, depending on when the error is detected. Asynchronous errors are delivered through the delayed error callback routine. Otherwise, the proposal will initiate a protocol within the group.

Information about the attribute change request is supplied through the attribute change request block, which is a type of proposal information block. On the **ha_gs_change_attributes** subroutine, specify the proposal information block as an attribute change request block. For the definition of the proposal information block, see the **ha_gs_delayed_error_callback** man page.

The change attributes request block has the following definition:

```
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_change_attributes_t;
```

The **gs_num_phases** field specifies whether the attribute change protocols are to be n-phase protocols or one-phase protocols. It can take one of the following values:

HA_GS_1_PHASE
: The protocols are to be one-phase protocols. One-phase protocols are automatically approved.

HA_GS_N_PHASE
: The protocols are to be n-phase protocols, which put the group into multi-phase voting.

The **gs_time_limit** field contains the voting phase time limit, in seconds. This is the number of seconds within which each provider must register its vote for each phase of an n-phase protocol. If the field has a value of 0, no limit is enforced.

The group attributes block describes the attributes of the group, including the group's name. The group attribute block is specified as input to a change attribute request using the **ha_gs_change_attributes** subroutine. It has the following definition:

```
typedef char    *ha_gs_group_name_t;

typedef struct {
    short                  gs_version;
    short                  gs_sizeof_group_attributes;
    unsigned               gs_client_version;
    ha_gs_group_name_t     gs_group_name;
    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_source_group_name;
} ha_gs_group_attributes_t;
```

The group attributes block contains the name of the group and the set of group attributes that are passed to the Group Services subsystem on the **ha_gs_change_attributes** subroutine call.

The **ha_gs_change_attributes** subroutine can change the following attributes:

- **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 attribute descriptions, see the complete list of group attributes under the ha_gs_join Subroutine.

The **gs_backlevel_providers** field in the change attributes request block should be set to NULL when it is submitted to the **ha_gs_change_attributes** subroutine.

If the request is returned with an asynchronous error of **HA_GS_BACKLEVEL_PROVIDERS**, the **gs_backlevel_providers** field will point to a list of providers. For Group Services PSSP domains on RS/6000® SP systems, these providers are in the group that was compiled and linked against a pre-3.1 version of PSSP. For Group Services HACMP/ES domains on RS/6000 workstation clusters, these providers are in the group that was compiled and linked against a pre-4.3 version of HACMP. In this case, the request will be returned asynchronously via the **ha_gs_delayed_error_callback** function.

For the group to successfully use the **ha_gs_change_attributes** subroutine to dynamically change the group's attributes, all providers in the group must be compiled (or recompiled) against the proper level of the Group Services subsystem library, as described above.

Restrictions

The calling process must be a provider. The group must not already be running an n-phase protocol.

Return values

If the **ha_gs_change_attributes** subroutine is successful, it returns a value of 0 (**HA_GS_OK**). Group Services has accepted the request and will asynchronously attempt to run the proposed protocol.

Error values

If the **ha_gs_change_attributes** subroutine is unsuccessful, it returns an error number. If the error is detected immediately, an error is returned synchronously. If the error is detected after the call has been accepted, an error is returned asynchronously.

The GSAPI error numbers are defined in the **ha_gs.h** header file. For more information on GSAPI errors, see “GSAPI return codes” on page 126.

Synchronous errors

The following errors may be returned synchronously by the **ha_gs_change_attributes** subroutine:

HA_GS_BAD_GROUP_ATTRIBUTES
: One or more of the fields specified in the **proposal_info** block contain invalid values for a group attribute value.

HA_GS_BAD_PARAMETER
: The number of phases specified for the protocol is not allowable; it must be **HA_GS_1_PHASE** or **HA_GS_N_PHASE**.

HA_GS_BAD_MEMBER_TOKEN
The given **provider_token** does not specify a valid provider joined to a group.

HA_GS_COLLIDE
The provider's group is already running a protocol or this provider has already submitted a protocol request.

HA_GS_NO_INIT
The GS client has not yet successfully initialized itself with Group Services by calling **ha_gs_init**.

HA_GS_NOT_A_MEMBER
The given **provider_token** does not specify a valid group.

HA_GS_NOT_OK
The connection to the GS daemon has been lost. The GS client needs to reinitialize (via **ha_gs_init**).

HA_GS_NOT_SUPPORTED
The GS client was not compiled against the proper level of the GS library to use this function.

Asynchronous errors

The following errors may be returned asynchronously by the **ha_gs_change_attributes** subroutine:

HA_GS_BACKLEVEL_PROVIDERS

For Group Services PSSP domains on RS/6000 SP systems, one or more providers in the group was compiled and linked against a pre-3.1 version of PSSP. For Group Services HACMP/ES domains on RS/6000 workstation clusters, one or more providers in the group was compiled and linked against a pre-4.3 version of HACMP. This error code is sent with a delayed-error notification to the provider proposing to change the attributes. The delayed-error notification contains a list of the back-level providers, to identify which processes need to be upgraded.

In this case, to change the attributes for a group, all providers must leave the group and rejoin with the new attributes.

HA_GS_COLLIDE

Another protocol is already active for this group. In this case, to change the attributes for a group, the provider must resubmit the request.

HA_GS_NOT_SUPPORTED

The Group Services subsystem on one or more machines in the domain does not support the appropriate functionality. This error code is sent with a delayed-error notification to the provider proposing to change the group attributes.

In this case, to change the attributes for a group, all providers must leave the group and rejoin with the new attributes.

Files

ha_gs.h

Related concepts:

"Mutability of group attributes" on page 6
Certain group attributes are *mutable*. This means that they can be dynamically changed by the group's providers using the **ha_gs_change_attributes** asynchronous interface.

Related reference:

"ha_gs_announcement_callback" on page 48

"ha_gs_init" on page 80

"ha_gs_join" on page 85

"ha_gs_send_message" on page 110
"ha_gs_leave" on page 91
"ha_gs_expel" on page 62
"ha_gs_subscribe" on page 112
"ha_gs_change_state_value"
"ha_gs_goodbye" on page 78
"ha_gs_delayed_error_callback" on page 57
"ha_gs_protocol_approved_callback" on page 102
"ha_gs.h header file" on page 125
ha_gs.h is a header file that provides datatypes and structures for use with the GSAPI subroutines.

ha_gs_change_state_value

Purpose

ha_gs_change_state_value – Called by a provider of a group to propose a change to the group's state value

Library

See "GSAPI library names" on page 46.

Syntax

```
#include <ha_gs.h>

ha_gs_rc_t
    ha_gs_change_state_value(
                ha_gs_token_t          provider_token,
        const   ha_gs_proposal_info_t  *proposal_info)
```

Parameters

provider_token
: 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.

proposal_info
: A pointer to a buffer that contains a proposal information block, which describes the proposed state change request.

Description

The **ha_gs_change_state_value** subroutine is used by a provider of a Group Services group to propose a change to the group's state value.

If the request is specified as a one-phase protocol, and Group Services chooses to run this protocol, the group's providers are notified using normal protocol approval procedures.

If the request is specified as an n-phase protocol, and Group Services chooses to run this protocol, the group's providers are notified using normal n-phase voting procedures.

If the Group Services subsystem chooses not to run this protocol (because another protocol is already in progress), the **HA_GS_COLLIDE** error number is returned either synchronously or asynchronously, depending on when the error is detected. Asynchronous errors are delivered through the delayed error callback routine. Otherwise, the proposal will initiate a protocol within the group.

Information about the state change request is supplied through the state change request block, which is a type of proposal information block. On the **ha_gs_change_state_value** subroutine, specify the proposal information block as a state change request block. For the definition of the proposal information block, see the **ha_gs_delayed_error_callback** man page.

The state change request block has the following definition:

```
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;
```

The **gs_num_phases** field specifies whether the state change protocols are to be n-phase protocols or one-phase protocols. It can take one of the following values:

HA_GS_1_PHASE
The protocols are to be one-phase protocols. One-phase protocols are automatically approved.

HA_GS_N_PHASE
The protocols are to be n-phase protocols, which put the group into multi-phase voting.

The **gs_time_limit** field contains the voting phase time limit, in seconds. This is the number of seconds within which each provider must register its vote for each phase of an n-phase protocol. If the field is set to a value of 0, no limit is enforced.

The **gs_new_state** field points to a buffer that contains the proposed new value for the group's state. The group state value has the following definition:

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

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

The **gs_state** field points to a buffer that contains the actual state value bytes. The state value of a group is defined by the application that is using the GSAPI. The state value is controlled by the providers in a way that is meaningful to the application. The state value is not interpreted by the Group Services subsystem.

Restrictions

The calling process must be a provider. The group must not already be running an n-phase protocol.

Return values

If the **ha_gs_change_state_value** subroutine is successful, it returns a value of 0 (**HA_GS_OK**). Group Services has accepted the request and will asynchronously attempt to run the proposed protocol.

Error values

If the **ha_gs_change_state_value** subroutine is unsuccessful, it returns an error number. If the error is detected immediately, an error is returned synchronously. If the error is detected after the call has been accepted, an error is returned asynchronously.

The GSAPI error numbers are defined in the **ha_gs.h** header file. For more information on GSAPI errors, see "GSAPI return codes" on page 126.

Synchronous errors

The following errors may be returned synchronously by the **ha_gs_change_state_value** subroutine:

HA_GS_BAD_MEMBER_TOKEN
The given **provider_token** does not specify a valid provider joined to a group.

HA_GS_BAD_PARAMETER
The number of phases specified for the protocol is not allowable; it must be **HA_GS_1_PHASE** or **HA_GS_N_PHASE**.

HA_GS_COLLIDE
The provider's group is already running a protocol or this provider has already submitted a protocol request.

HA_GS_NO_INIT
The GS client has not yet successfully initialized itself with Group Services by calling **ha_gs_init**.

HA_GS_NOT_A_MEMBER
The given **provider_token** does not specify a valid group.

HA_GS_NOT_OK
The connection to the GS daemon has been lost. The GS client needs to reinitialize (via **ha_gs_init**).

Asynchronous errors

The following errors may be returned asynchronously by the **ha_gs_change_state_value** subroutine:

HA_GS_NOT_A_MEMBER
The provider that is proposing the protocol is no longer a provider for the specified group.

HA_GS_BAD_PARAMETER
The specified parameter was not valid.

HA_GS_COLLIDE
Another protocol is already active for this group.

Files

ha_gs.h

Related reference:

"ha_gs_announcement_callback" on page 48
"ha_gs_change_attributes" on page 50
"ha_gs_init" on page 80
"ha_gs_goodbye" on page 78
"ha_gs_expel" on page 62
"ha_gs_subscribe" on page 112
"ha_gs_delayed_error_callback" on page 57
"ha_gs_join" on page 85
"ha_gs_leave" on page 91
"ha_gs_protocol_approved_callback" on page 102
"ha_gs.h header file" on page 125
ha_gs.h is a header file that provides datatypes and structures for use with the GSAPI subroutines.

ha_gs_delayed_error_callback

Purpose

ha_gs_delayed_error_callback – A callback routine that the Group Services subsystem calls to deliver an asynchronous error notification to a GS client

Library

See "GSAPI library names" on page 46.

Syntax

```
#include <ha_gs.h>

void
    ha_gs_delayed_error_callback(
        const   ha_gs_delayed_error_notification_t      *notification)
```

Parameters

notification
: A pointer to a delayed error notification block.

Description

Delayed errors are asynchronous errors. They occur when a GS client has submitted a proposal, (such as to join a group, broadcast a message, or change the group's state value) and the Group Services subsystem later discovers a problem with the proposal. When such an error occurs, Group Services delivers it to the GS client by invoking a callback routine.

The **ha_gs_delayed_error_callback** subroutine defines a GS client's delayed error callback routine. The GS client uses it to handle delayed error notifications from the Group Services subsystem. The process provides the address of the delayed error callback routine to the Group Services subsystem on the **ha_gs_init** subroutine during GSAPI initialization. The Group Services subsystem then calls the delayed error callback routine when it has a delayed error notification to deliver to the GS client.

On input, the delayed error callback routine receives a pointer to the delayed error notification block. The delayed error notification block has the following definition:

```
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;
```

The **gs_notification_type** field contains the type of notification. For a delayed error notification, it contains a value of **HA_GS_DELAYED_ERROR_NOTIFICATION**.

The **gs_protocol_type** field contains the type of request for which this delayed error is being delivered.

HA_GS_JOIN
: This provider's join request is invalid because its group attributes do not match those that were specified by the group.

HA_GS_LEAVE
: A provider is voluntarily leaving the group. If the error is **HA_GS_COLLIDE**, then the Group Services subsystem has chosen another protocol to run instead of this one.

HA_GS_EXPEL
A provider is attempting to expel one or more providers from the group. If the error is **HA_GS_COLLIDE**, then the Group Services subsystem has chosen another protocol to run instead of this one.

HA_GS_STATE_VALUE_CHANGE
A provider is trying to change the group's state value. If the error is **HA_GS_COLLIDE**, then the Group Services subsystem has chosen another protocol to run instead of this one.

HA_GS_PROVIDER_MESSAGE
A provider is broadcasting a message to the group. If the error is **HA_GS_COLLIDE**, then the Group Services subsystem has chosen another protocol to run instead of this one.

HA_GS_SUBSCRIBE
This subscriber's request is invalid because the group specified on the request does not exist.

HA_GS_GROUP_ATTRIBUTE_CHANGE
This provider's group attribute change request is invalid or not supported.

The **gs_delayed_return_code** field contains error number of the delayed error. The GSAPI error numbers are defined in the **ha_gs.h** header file. For more information on GSAPI errors, see "GSAPI return codes" on page 126.

The **gs_failing_request** field points to the proposal information block for the proposal that is in error.

The proposal information block has the following definition:

```
#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;
```

For details on the block that defines each type of proposal, see the subroutine that is used to initiate the proposal, as follows:

Proposal	Subroutine
Joining a group	ha_gs_join
Changing the group's attributes	ha_gs_change_attributes
Changing the group's state value	ha_gs_change_state_value
Broadcasting a message to all of a group's providers	ha_gs_send_message
Leaving a group	ha_gs_leave
Expelling one or more providers from the group	ha_gs_expel
Subscribing to a group	ha_gs_subscribe

Restrictions

For important information about multiprocessing considerations that apply to all callback routines, see the **ha_gs_n_phase_callback** man page.

Return values

None.

Error values

None.

For information about GSAPI synchronous and asynchronous errors, see "GSAPI return codes" on page 126.

Synchronous errors

None.

Asynchronous errors

None.

Files

ha_gs.h

Related reference:

"ha_gs_join" on page 85
"ha_gs_change_state_value" on page 54
"ha_gs_change_attributes" on page 50

ha_gs_dispatch

Purpose

ha_gs_dispatch – Called by a GS client to handle messages that have arrived from the Group Services Application Programming Interface (GSAPI)

Library

See "GSAPI library names" on page 46.

Syntax

```
#include <ha_gs.h>

ha_gs_rc_t
    ha_gs_dispatch(
        const   ha_gs_dispatch_flag_t   dispatch_flags)
```

Parameters

dispatch_flags
A flag that indicates how messages from the Group Services application programming interface (GSAPI) are to be processed. It can be one of the following values:

HA_GS_NON_BLOCKING
The GSAPI should check for messages that have arrived on the GSAPI socket. If any messages have arrived, the GSAPI should call the appropriate callback routines. If no messages have arrived, the GSAPI should return control immediately.

HA_GS_BLOCKING
The GSAPI should check for messages that have arrived on the GSAPI socket. As messages arrive, the GSAPI will call the appropriate callback routines, and it will continue to do so until an error occurs or the connection is broken.

Description

The **ha_gs_dispatch** subroutine is used by a process to handle messages from the Group Services Application Programming Interface (GSAPI). It is vital that the GS client calls this subroutine on a regular basis to be able to receive messages from the GSAPI for delivery as notifications by invoking the appropriate callback routines. This allows the GS client to respond to any protocols that may be running in the group. The parameter to this routine controls the behavior of **ha_gs_dispatch** once all outstanding messages have been delivered.

Although **ha_gs_dispatch** needs to be called regularly, exactly how often it is necessary will differ for each GS client. The most important factor is that the GS client should be ready to respond to arriving messages as quickly as possible. This allows it to respond to changes in its group or the system as quickly as possible. Once **ha_gs_dispatch** is called, it will process all messages that have arrived, which may result in multiple GS client callback routines being invoked. The *dispatch_flags* parameter controls the behavior of **ha_gs_dispatch** once all messages have been processed.

To assist in this, there are two general models of execution for a GS client. The different settings for the *dispatch_flags* parameter to **ha_gs_dispatch** correlate to these:

1. The first model can be loosely described as the "non-blocking" model. This matches to the flag **HA_GS_NON_BLOCKING**, and is most appropriate to singly-threaded (or non-threaded) GS clients. In this model, it is expected that the GS client will remain responsive to arriving messages by using one of the following mechanisms:
   a. The system subroutine **select()**. In this case, the GS client should set up a select mask containing the file descriptor returned during the **ha_gs_init** subroutine in the **ha_gs_descriptor** field. When **select()** indicates that a message has arrived on this file descriptor from Group Services, the GS client should run **ha_gs_dispatch** with the **HA_GS_NON_BLOCKING** flag.
   b. Although it is not recommended, the GS client may set a fixed timer and simply call **ha_gs_dispatch** after some fixed time interval. This is not recommended. This is because the call may be more frequent than is necessary, and in most cases will result in no actions being taken because no messages have arrived. This may also cause the GS client to be very slow to respond to messages if the time period is too long, or if a number of messages arrive quickly.
2. The second model is, alternately, a "blocking" model, and is indicated with the **HA_GS_BLOCKING** flag. This model is normally just appropriate to multi-threaded GS clients, although it may be used by singly-threaded (or non-threaded) GS clients.

   The **HA_GS_BLOCKING** flag causes all threads that enter **ha_gs_dispatch** to never return from **ha_gs_dispatch**, unless they encounter an error. In this case, they will return from **ha_gs_dispatch** with an **HA_GS_NOT_OK** return code. These threads simply remain in **ha_gs_dispatch**. As soon as a message arrives from Group Services, a thread will read the message and run the appropriate callback function.

   If the GS client has multiple threads call **ha_gs_dispatch**, all of these threads will queue up to read messages as they arrive in Group Services. The GSAPI library will handle synchronization of these threads to assure that they properly read and process messages. The GS client need only have as many threads as desired call **ha_gs_dispatch** with the **HA_GS_BLOCKING** flag.

A multi-threaded GS client may certainly use the non-blocking model described above, using **select()** or signals as desired. In this case, it is still valid for multiple threads to run **ha_gs_dispatch**. If a thread calls **ha_gs_dispatch** with the **HA_GS_NON_BLOCKING** flag and there are no pending messages to process, it will return from **ha_gs_dispatch** with an **HA_GS_OK** return code.

A key concern for multi-threaded GS clients is in dealing with parallel execution of callback functions. Since it is possible for a single GS client process to join multiple groups, subscribe to multiple groups, or both, the GSAPI library will enforce the following synchronization rules:

1. There will never be more than one active callback function being invoked for any *one* group in parallel.
2. However, where appropriate, callback functions for different groups will be invoked in parallel if there are multiple threads that have run **ha_gs_dispatch** and messages have arrived at the GS client for the different groups.

This leads to a general rule that a multi-threaded GS client that deals with multiple groups should be prepared to dedicate the same number of threads to **ha_gs_dispatch** as the number of groups with which the GS client is concerned. This keeps such a GS client as responsive as possible for all of its groups.

A singly-threaded (or non-threaded) GS client may also deal with multiple groups, although it will only be able invoke a single callback function at any one time. This may keep it from being able to be more responsive if messages arrive for multiple groups within a small amount of time.

A limitation for threaded GS clients is that any thread that has run **ha_gs_dispatch** should never be cancelled (via the **pthread_cancel** subroutine) or killed (via the **pthread_kill** subroutine) until after the thread has returned from **ha_gs_dispatch**. The behavior of the client will be unpredictable if such a thread is killed or cancelled.

A multi-threaded client must issue **ha_gs_quit** to disconnect from Group Services, and it must then wait for any threads that had invoked **ha_gs_dispatch** to return before they can be killed or cancelled.

Restrictions

The caller must be a GS client.

Return values

If the **ha_gs_dispatch** subroutine is successful, it returns a value of 0 (**HA_GS_OK**).

Error values

If the **ha_gs_dispatch** subroutine is unsuccessful, it returns an error number synchronously

The GSAPI error numbers are defined in the **ha_gs.h** header file. For more information on GSAPI errors, see “GSAPI return codes” on page 126.

Synchronous errors

The following errors may be returned synchronously by the **ha_gs_dispatch** subroutine:

HA_GS_BAD_PARAMETER

The given parameter must be **HA_GS_BLOCKING** or **HA_GS_NON_BLOCKING**.

HA_GS_NOT_OK

The connection to the GS daemon has been lost, or user authentication has failed. The GS client needs to reinitialize (via **ha_gs_init**).

Asynchronous errors

None.

Files

ha_gs.h

Related reference:

"ha_gs_init" on page 80
"ha_gs_get_ffdc_id" on page 71
"ha_gs_quit" on page 107
"ha_gs_send_message" on page 110

ha_gs_expel

Purpose

ha_gs_expel – Called by a provider of a group to request the expulsion of one or more providers.

Library

See "GSAPI library names" on page 46.

Syntax

```
#include <ha_gs.h>

ha_gs_rc_t
    ha_gs_expel(
                ha_gs_token_t          provider_token,
        const   ha_gs_proposal_info_t  *proposal_info)
```

Parameters

provider_token
: A token that identifies the caller as a provider of the group. This token was previously initialized when the provider joined the group by using the **ha_gs_join** subroutine.

proposal_info
: A pointer to a buffer that contains a proposal information block, which describes the proposed expel request.

Description

The **ha_gs_expel** subroutine is used by a provider of a Group Services group to propose the expulsion of one or more providers. A provider may propose the expulsion of itself.

A group might decide that a provider should be expelled for a number of reasons. Examples include:

- The provider is not responsive or has detected an internal error, as determined by a responsiveness check.
- The provider failed to submit a vote during a previously completed n-phase protocol within the specified time limit.
- The provider is not behaving as expected in the context of the application the group is running.

If the request is specified as a one-phase protocol, and Group Services chooses to run this protocol, the group's providers are notified using normal protocol approval procedures.

If the request is specified as an n-phase protocol, and Group Services chooses to run this protocol, the group's providers are notified using normal n-phase voting procedures.

If the Group Services subsystem chooses not to run this protocol (because another protocol is already in progress), the **HA_GS_COLLIDE** error number is returned either synchronously or asynchronously, depending on when the error is detected. Asynchronous errors are delivered through the delayed error callback routine. Otherwise, the proposal will initiate a protocol within the group.

Information about the expel request is supplied through the expel request block, which is a type of proposal information block. On the **ha_gs_expel** subroutine, specify the proposal information block as an expel request block. For the definition of the proposal information block, see the **ha_gs_delayed_error_callback** man page.

The expel request block has the following definition:

```
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;
    unsigned int            gs_deactivate_phase;
    char                    *gs_deactivate_flag;
} ha_gs_expel_request_t;
```

The **gs_num_phases** field specifies whether the expel protocols are to be n-phase protocols or one-phase protocols. It can take one of the following values:

HA_GS_1_PHASE
: The protocols are to be one-phase protocols. One-phase protocols are automatically approved.

HA_GS_N_PHASE
: The protocols are to be n-phase protocols, which put the group into multi-phase voting.

The **gs_time_limit** field contains the voting phase time limit, in seconds.

For providers that are not being expelled, this is the number of seconds within which each provider must register its vote for each phase of an n-phase protocol. If the field is set to a value of 0, no limit is enforced.

For providers that are being expelled, this is the number of seconds within which the deactivate script must complete. Otherwise, the deactivate script is considered unsuccessful. The deactivate script that is invoked is the one that was specified on the **ha_gs_init** subroutine when the provider that is being expelled first registered with Group Services.

The **gs_expel_list** field is a list of providers that are targeted to be expelled. It has the following definition:

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

The **gs_count** field contains the number of providers in the list.

The **gs_providers** field points to the list of providers. Each provider is described by a provider information block, which is defined in the **ha_gs_n_phase_callback** man page.

The providers in the expel list do not take part in the protocol and receive no notice of it, unless it is approved.

All of the providers that are not targeted for expulsion take part in the protocol, even if they were declared nonresponsive before the protocol began.

The **gs_deactivate_phase** field contains the phase number in which the deactivate script should be run against the providers that are being expelled. If this field contains 0, no deactivate script will be invoked.

The **gs_deactivate_flag** 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 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.

Deactivate Scripts

For information about deactivate scripts, see “Deactivating scripts” on page 35.

Restrictions

The calling process must be a provider. The group must not already be running an n-phase protocol.

Return values

If the **ha_gs_expel** subroutine is successful, it returns a value of 0 (**HA_GS_OK**). Group Services has accepted the request and will asynchronously attempt to run the proposed protocol.

Error values

If the **ha_gs_expel** subroutine is unsuccessful, it returns an error number. If the error is detected immediately, an error is returned synchronously. If the error is detected after the call has been accepted, an error is returned asynchronously.

The GSAPI error numbers are defined in the **ha_gs.h** header file. For more information on GSAPI errors, see “GSAPI return codes” on page 126.

Asynchronous errors

The following errors may be returned asynchronously by the **ha_gs_expel** subroutine:

HA_GS_NOT_A_MEMBER
: The provider that is proposing the protocol is no longer a provider for the specified group.

HA_GS_BAD_PARAMETER
: The specified parameter was not valid.

HA_GS_COLLIDE
: Another protocol is already active for this group.

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.

Synchronous errors

The following errors may be returned synchronously by the **ha_gs_expel** subroutine:

HA_GS_BAD_MEMBER_TOKEN
: The given **provider_token** does not specify a valid provider joined to a group.

HA_GS_BAD_PARAMETER
: The number of phases specified for the protocol is not allowable (it must be **HA_GS_1_PHASE** or **HA_GS_N_PHASE**); or no providers were specified to be expelled.

HA_GS_COLLIDE
: The provider's group is already running a protocol, or this provider has already submitted a protocol request.

HA_GS_INVALID_DEACTIVATE_PHASE
: The specified **gs_deactivate_phase** is less than zero; or the protocol is specified as a one phase (**gs_num_phases** is **HA_GS_1_PHASE**) and **gs_deactivate_phase** is greater than one.

HA_GS_NO_INIT
: The GS client has not yet successfully initialized itself with Group Services by calling **ha_gs_init**.

HA_GS_NOT_A_MEMBER
: The given **provider_token** does not specify a valid group.

HA_GS_NOT_OK
: The connection to the GS daemon has been lost. The GS client needs to reinitialize (via **ha_gs_init**).

HA_GS_NOT_SUPPORTED
: The GS client was not compiled against the proper level of the GS library to use this function.

HA_GS_PROVIDER_APPEARS_TWICE
: A provider to be expelled is listed twice in the given **gs_expel_list**.

Files

ha_gs.h

Related reference:

"ha_gs_announcement_callback" on page 48
"ha_gs_change_attributes" on page 50
"ha_gs_change_state_value" on page 54
"ha_gs_goodbye" on page 78
"ha_gs_n_phase_callback" on page 94
"ha_gs_responsiveness_callback" on page 108
"ha_gs_send_message" on page 110
"ha_gs.h header file" on page 125
ha_gs.h is a header file that provides datatypes and structures for use with the GSAPI subroutines.

ha_gs_get_adapter_info

Purpose

ha_gs_get_adapter_info – gets the node number, adapter interface name, hb_network_name, and hb_network_type that correspond to a given IP address. For compatibility purposes, returns the node number and adapter interface name when invoked by an older application.

Library

See "GSAPI library names" on page 46.

Syntax

```
#include <ha_gs.h>
#include <netinet/in.h>

ha_gs_rc_t ha_gs_get_adapter_info( ha_gs_adapter_info *adapter)

typedef struct {
    ha_gs_ip_addr             ip_addr;
```

```
    short                      node_number;
    const char                 *interface_name;
    ha_gs_provider_t           gs_member_id;
    const char                 *gs_group_name;
    const char                 *hb_network_name;
    const char                 *hb_network_type;
    void                       *__reserved_1__;
    void                       *__reserved_2__;
} ha_gs_adapter_info;
```

For information about the **ha_gs_ip_addr** data structure, see "IP addressing" on page 33

Parameters

Input `adapter->ip_addr`, an input IP address parameter. The input IP address must be network-byte-ordered.

Output
`adapter->node_number`, `adapter->interface_name`, `adapter->hb-network_name`, and `adapter->hb_network_type`. These are the node number, adapter interface name, Topology Services network name of the adapter, and Topology Services network type of the adapter returned from this function call corresponding to the given adapter IP address.

Description

This subroutine gives the node number, adapter interface name, *hb_network_name*, and *hb_network_type* corresponding to a given IP address. When this function is called, the Group Services library will send back the information corresponding to the given IP address, if the mapping is available to the caller. The function will return **HA_GS_OK**.

In order to use this subroutine, the user must include the **netinet/in.h** header file.

Before calling this subroutine, the user must first call **ha_gs_init** with a flag set as in the following description:

```
ha_gs_rc_t ha_gs_init(ha_gs_descriptor              *ha_gs_descriptor,
                      const ha_gs_socket_ctrl       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)

typedef enum
{
    HA_GS_SOCKET_NO_SIGNAL              = 0x0000000,
    HA_GS_SOCKET_SIGNAL                 = 0x00000001,
    HA_GS_ENABLE_ADAPTER_INFO           = 0x00000002,
    HA_GS_ENABLE_DOMAIN_EVENT           = 0x00001000,
    HA_GS_ENABLE_IPV6                   = 0x00002000,
    HA_GS_ENABLE_MIGRATION_CALLBACK     = 0x00004000,
    HA_GS_STREAM                        = 0x00800000,
    HA_GS_IMMEDIATE_DOMAIN_CONTROL      = 0x10000000
} ha_gs_socket_ctrl_t;
```

If the adapter information is needed, **HA_GS_ENABLE_ADAPTER_INFO** must be OR'd in addition to the existing option **HA_GS_SOCKET_NO_SIGNAL** such as:

```
socket_options = HA_GS_SOCKET_NO_SIGNAL|HA_GS_ENABLE_ADAPTER_INFO
```

When the GS daemon receives the **ha_gs_init** call message, it will check this flag. If **HA_GS_ENABLE_ADAPTER_INFO** is set, it will send the adapter information table to the GSAPI library. Otherwise, the daemon will not send the table to the GSAPI library. This is to minimize the traffic between the daemon and the library.

In order to be compatible with previous releases, for old applications programs the new Group Services library will not touch the new entry fields of the **ha_gs_adapter_info** structure and **ha_gs_subscription_notification_t** structure. This means that Group Services will not deliver the new entry fields if the application is an old version. If it is the new release, group services will deliver the new entry fields to clients.

If the adapter information is not available yet, the function will return **HA_GS_NULL_ADAPTER_INFO**.

If the global adapter configuration is available, but the adapter information corresponding to the specified IP address is not found, it will return **HA_GS_ADAPTER_INFO_NOT_FOUND**.

If the library is not initialized yet, it will return **HA_GS_NO_INIT**.

If there is an internal error in the library, it will return **HA_GS_NOT_OK**.

If the user passed a NULL pointer to the function, (fox example, adapter = NULL), the function also returns **HA_GS_NOT_OK**.

Return values

HA_GS_OK — The function call is successful.

HA_GS_NOT_OK— The library is not initialized well, or the user input a NULL pointer.

HA_GS_NO_INIT— The library is not initialized yet. Call **ha_gs_init** first.

HA_GS_NULL_ADAPTER_INFO— The GS daemon has sent a NULL adapter table since there is no adapter configuration information available yet.

HA_GS_ADAPTER_INFO_NOT_FOUND— The adapter information corresponding to the given IP address is not found in the current adapter table.

Asynchronous errors

None.

Files

ha_gs.h

Related reference:

"ha_gs_get_node_number" on page 75
"ha_gs_get_adapter_info" on page 65

ha_gs_get_adapter_info_by_addr

Purpose

ha_gs_get_adapter_info_by_addr – gets the node number, adapter interface name, *hb_network_name*, and *hb_network_type* that correspond to a given IP address.

Library

See "GSAPI library names" on page 46.

Syntax

```
#include <ha_gs.h>
ha_gs_rc_t ha_gs_get_adapter_info_by_addr(
       const ha_gs_ip_addr *ip,
             ha_gs_adapter_info *adapter)

typedef struct {
    ha_gs_ip_addr            ip_addr;
    short                    node_number;
    const char               *interface_name;
    ha_gs_provider_t         gs_member_id;
    const char               *gs_group_name;
    const char               *hb_network_name;
    const char               *hb_network_type;
    void                     *__reserved_1__;
    void                     *__reserved_2__;
} ha_gs_adapter_info;
```

For information about the **ha_gs_ip_addr** data structure, see "IP addressing" on page 33

Parameters

Input `ip`, a pointer to an input IP address parameter. The input IP address must be network-byte-ordered.

Output
`adapter->node_number`, `adapter->interface_name`, `adapter->hb-network_name`, and `adapter->hb_network_type`. These are the node number, adapter interface name, Topology Services network name of the adapter, and Topology Services network type of the adapter returned from o the given adapter IP address.

Description

This subroutine gives the node number, adapter interface name, *hb_network_name*, and *hb_network_type* corresponding to a given IP address. When this function is called, the Group Services library will send back the information if the mapping is available to the caller. The subroutine will return **HA_GS_OK**.

Before calling this subroutine, the user must call **ha_gs_init** with a flag set as in this example:

```
ha_gs_rc_t ha_gs_init(ha_gs_descriptor             *ha_gs_descriptor,
                      const ha_gs_socket_ctrl      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)

typedef enum
{
    HA_GS_SOCKET_NO_SIGNAL              = 0x0000000,
    HA_GS_SOCKET_SIGNAL                 = 0x00000001,
    HA_GS_ENABLE_ADAPTER_INFO           = 0x00000002,
    HA_GS_ENABLE_DOMAIN_EVENT           = 0x00001000,
    HA_GS_ENABLE_IPV6                   = 0x00002000,
    HA_GS_ENABLE_MIGRATION_CALLBACK     = 0x00004000,
    HA_GS_STREAM                        = 0x00800000,
    HA_GS_IMMEDIATE_DOMAIN_CONTROL      = 0x10000000
} ha_gs_socket_ctrl_t;
```

If the user wants the adapter information, then in addition to setting the socket_options as:

```
socket_options = HA_GS_SOCKET_NO_SIGNAL;
```

the user must "OR" this flag with **HA_GS_ENABLE_ADAPTER_INFO**, as follows:

```
socket_options = HA_GS_SOCKET_NO_SIGNAL|HA_GS_ENABLE_ADAPTER_INFO;
```

- When this function is called, the Group Services library will first check if the client has called **ha_gs_init** successfully. If the GS library is not yet initialized, this function will return **HA_GS_NO_INIT**.
- If the user inputs a NULL *adapter* pointer to the function, that is, adapter = NULL, the function returns **HA_GS_NOT_OK**.
- If the user inputs a NULL *ip* pointer to the function, that is, ip = NULL, the function returns **HA_GS_NOT_OK**.
- When calling **ha_gs_init**, if the flag **HA_GS_ENABLE_ADAPTER_INFO** is not set, this API will return an error code of **HA_GS_NOT_OK**.
- If an adapter IP address is given, Group Services will send back the node number, the adapter interface name, hb_network_name, and hb_network_type corresponding to the given IP address provided the mapping is available to the caller. The function will return the code **HA_GS_OK**.
- If the adapter IP address to node number mapping table is not available, that is, it is not yet sent by the GS daemon, the function will return the code **HA_GS_NULL_ADAPTER_INFO** and set the node_number to a value of -1, the adapter interface name to an empty string, that is, interface_name[0]='\0', the hb_network_name to an empty string, and the hb_network_type to an empty string.
- If the adapter, associated with the adapter IP address is not configured in Topology Services, it will return the code **HA_GS_ADAPTER_INFO_NOT_FOUND**.
- If there is an internal error, it will return the code **HA_GS_NOT_OK**.

Return values

HA_GS_OK — The function call is successful.

HA_GS_NOT_OK— The inputs *ip* or *adapter* are null pointers, or there is an internal error. This error code may also be returned if the flag **HA_GS_ENABLE_ADAPTER_INFO** is not set on when calling **ha_gs_init**.

HA_GS_NO_INIT— The library is not yet initialized. Call **ha_gs_init** first.

HA_GS_NULL_ADAPTER_INFO— When the **HA_GS_ENABLE_ADAPTER_INFO** flag is set, the GS daemon has sent a NULL adapter table since the adapter configuration information is not yet available.

HA_GS_ADAPTER_INFO_NOT_FOUND— The adapter information corresponding to the given IP address is not found in the current adapter table. This situation may happen when the input is an incorrect IP address, or the adapter corresponding this IP address is not in the configuration.

Asynchronous errors

None.

Files

ha_gs.h

Related reference:

"ha_gs_get_adapter_info_by_addr" on page 67

"ha_gs_get_adapter_info_by_id" on page 70

ha_gs_get_adapter_info_by_id

Purpose

ha_gs_get_adapter_info_by_id – Get adapter information, including the IP address, node number, adapter interface name, network name, and network type, corresponding to a given subscriber token and member ID of a subscribed adapter group.

Library

See "GSAPI library names" on page 46.

Syntax

```
#include <ha_gs.h>
ha_gs_rc_t ha_gs_get_adapter_info_by_id(
            ha_gs_token_t _subscriber_token,
      const ha_gs_provider_t *_id,
            ha_gs_adapter_info *adapter)
```

Parameters

Input *subscriber_token* and *provider_id*, which are a subscriber token and a pointer to a provider ID of a subscribed adapter group.

Output
adapter, a pointer to a structure **ha_gs_adapter_info**. This structure includes information about the adapter: IP address, node_number, adapter interface name, network name, and network type, which are returned from this function call corresponding to the given subscriber token and the provider ID.

The **ha_gs_adapter_info** structure is defined in "Parameters" on page 68.

Description

This function gives adapter information, including the IP address, node number, adapter interface name, network name, and and network type corresponding to a given subscriber token and provider ID of a subscribed adapter group.

- When this function is called, the Group Services library will first check if the client has called **ha_gs_init** successfully. If the GS library is not yet initialized, this function will return **HA_GS_NO_INIT**.
- The Group Services library will then check if *provider_id* or *adapter* are null pointers. If either of them is a null pointer, the function will return **HA_GS_NOT_OK**.
- If the flag **HA_GS_SUBSCRIBE_ADAPTER_INFO** is not set on when calling **ha_gs_subscribe**, then when this function is called with a valid subscriber token and provider ID, it will return an error code **HA_GS_NOT_OK**.
- If a subscriber token and a provider ID of the group are given, and Group Services finds the corresponding adapter information, it will return the IP address, node number, adapter interface name, network name, and network type corresponding to the input subscriber token and a provider ID. The function will return **HA_GS_OK**.
- If GS cannot find a subscriber token matching the input subscriber token, it will return **HA_GS_BAD_MEMBER_TOKEN**.
- If GS cannot find a matching provider ID corresponding to the input provider ID, it will return **HA_GS_NOT_A_MEMBER**.
- If the adapter information is not available yet, the function will return code **HA_GS_NULL_ADAPTER_INFO**. This means that Topology Services has not yet found any adapter information, so the GS daemon sent a empty adapter table to the library.

- If the adapter, associated with the input subscriber token and provider ID is not configured in Topology Services, this function will return code **HA_GS_ADAPTER_INFO_NOT_FOUND**.
- If there is an internal error, this function will return code **HA_GS_NOT_OK**.

Before calling this function, the user must call **ha_gs_init**. The user can optionally specify the **HA_GS_ENABLE_ADAPTER_INFO** option when calling **ha_gs_init**.

Before calling this function, the user must also have successfully subscribed to the adapter group by calling **ha_gs_subscribe** with the flag **HA_GS_SUBSCRIBE_ADAPTER_INFO** set on.

Return values

HA_GS_OK — The function call is successful.

HA_GS_NOT_OK— The user input a Null pointer to any one of the function parameters: provider_id, adapter, or an internal error has occurred. If when calling **ha_gs_subscribe**, the flag **HA_GS_SUBSCRIBE_ADAPTER_INFO** is not set on, then when calling this API, it will return this error code. Note for a **HA_GS_ALL_ADAPTER_MEMBERSHIP_GROUP** subscription, it will automatically turn on the flag **HA_GS_SUBSCRIBE_ADAPTER_INFO**.

HA_GS_NO_INIT — The library is not initialized yet. Call **ha_gs_init** first.

HA_GS_NULL_ADAPTER_INFO — The GS daemon has sent a Null adapter table since adapter configuration information is not yet available.

HA_GS_ADAPTER_INFO_NOT_FOUND — The adapter information corresponding to the given subscriber token and the provider ID is not found in the current adapter table.

HA_GS_BAD_MEMBER_TOKEN — The given subscriber token is not found.

HA_GS_NO_IPV4_ADDRESS_FOR_THE_ID — There is no IPv4 address for this ID.

HA_GS_NOT_A_MEMBER The given provider ID is incorrect.

Asynchronous errors

None.

Files

ha_gs.h

Related reference:

“ha_gs_get_adapter_info_by_addr” on page 67
“ha_gs_get_ipaddr_by_id” on page 72

ha_gs_get_ffdc_id

Purpose

ha_gs_get_ffdc_id – Retrieve an FFDC ID that is associated with a Group Services subsystem failure

Library

See “GSAPI library names” on page 46.

Syntax

```
#include <ha_gs.h>

ha_gs_rc_t
    ha_gs_get_ffdc_id(fc_eid_t fid)
```

Parameters

fid A pointer to the FFDC ID that will be copied from the GSAPI.

Description

The **ha_gs_get_ffdc_id** subroutine is provided to retrieve a First Failure Data Capture (FFDC) identifier that can help determine the root cause of a Group Services subsystem failure. The FFDC ID is reset if **ha_gs_init** is called again. For more information about FFDC, see the *First Failure Data Capture Programming Guide and Reference*.

Restrictions

In certain situations, this function may not provide an FFDC ID even if the Group Services subsystem fails.

Return values

HA_GS_NOT_OK— GSAPI did not receive an FFDC ID from the Group Services subsystem.

HA_GS_OK— GSAPI received an FFDC ID from the Group Services subsystem.

Asynchronous errors

None.

Files

ha_gs.h

Related reference:

"ha_gs_dispatch" on page 59

ha_gs_get_ipaddr_by_id

Purpose

ha_gs_get_ipaddr_by_id – Get the IP address corresponding to a given subscriber token and a provider ID of a subscribed adapter group.

Library

See "GSAPI library names" on page 46.

Syntax

```
#include <ha_gs.h>
ha_gs_rc_t ha_gs_get_ipaddr_by_id(
           ha_gs_token_t _subscriber_token,
     const ha_gs_provider_t *_id,
           ha_gs_ip_addr *_ip)
```

Parameters

Input *subscriber_token* and *provider_id*, which are a subscriber token and a pointer to a provider ID of a subscribed adapter group.

Output
ip, which is a pointer to the IP address of the adapter corresponding to the given subscriber token and the provider ID.

For information about the **ha_gs_ip_addr** data structure, see "IP addressing" on page 33

Description

The **ha_gs_get_ipaddr_by_id** subroutine returns the IP address corresponding to a given subscriber token and a provider ID of a subscribed adapter group.

- When this function is called, the Group Services library will first check if the client has called **ha_gs_init** successfully. If the GS library is not yet initialized, this function will return **HA_GS_NO_INIT**.
- This function then checks if either *provider_id* or *ip* is a null pointer. If either of them is a null pointer, the function returns error code **HA_GS_NOT_OK**.
- If the flag **HA_GS_SUBSCRIBE_ADAPTER_INFO** is not set on when calling **ha_gs_subscribe**, then when this function is called with a valid subscriber token and a valid provider ID, it will return an error code of **HA_GS_NOT_OK**.
- If a subscriber token and a provider ID of the group are given, and the corresponding adapter IP address is found, Group Services will send back the IP address corresponding to the given subscriber token and provider ID. The function will return **HA_GS_OK**.
- If GS cannot find a subscriber token matching the given subscriber token, it will return **HA_GS_BAD_MEMBER_TOKEN**.
- If GS cannot find a matching provider ID corresponding to the given ID, it will return **HA_GS_NOT_A_MEMBER**.
- If the IP address information is not yet available, the function will return code **HA_GS_NULL_ADAPTER_INFO**.
- If the adapter associated with the given subscriber token and the provider ID is not configured in Topology Services, it will return code **HA_GS_ADAPTER_INFO_NOT_FOUND**.
- If there is an internal error, this function will return code **HA_GS_NOT_OK**.

Before calling this function, the user must call **ha_gs_init**. The user can specify the **HA_GS_ENABLE_ADAPTER_INFO** socket option when calling **ha_gs_init**. See "Description" on page 68.

Before calling this function, the user must also have successfully subscribed to the adapter group by calling **ha_gs_subscribe** with the flag **HA_GS_SUBSCRIBE_ADAPTER_INFO** set on.

Return values

HA_GS_OK — The function call is successful.

HA_GS_NOT_OK — The user input a Null pointer for either of the function parameters, *provider_id* or *ip*, or an internal error has occurred. If when calling **ha_gs_subscribe**, the flag **HA_GS_SUBSCRIBE_ADAPTER_INFO** is not set on, then when calling this API, it will return this error code. Note that for a **HA_GS_ALL_ADAPTER_MEMBERSHIP_GROUP** subscription, it will automatically turn on the flag **HA_GS_SUBSCRIBE_ADAPTER_INFO**.

HA_GS_NO_INIT — The library is not yet initialized. Call **ha_gs_init** first.

HA_GS_NULL_ADAPTER_INFO — The GS daemon has sent a Null adapter table because the adapter configuration information is not yet available.

HA_GS_ADAPTER_INFO_NOT_FOUND — The IP address information corresponding to the given subscriber token and the provider ID is not found.

HA_GS_BAD_MEMBER_TOKEN — The given subscriber token is not found.

HA_GS_NO_IPV4_ADDRESS_FOR_THE_ID — There is no IPv4 address for this ID.

HA_GS_NOT_A_MEMBER — The given provider ID is incorrect.

Asynchronous errors

None.

Files

ha_gs.h

Related reference:

"ha_gs_get_adapter_info_by_id" on page 70
"ha_gs_subscribe" on page 112

ha_gs_get_limits

Purpose

ha_gs_get_limits – Get the maximum limits of the current Group Services capacity.

Library

See "GSAPI library names" on page 46.

Syntax

```
#include <ha_gs.h>
ha_gs_rc_t ha_gs_get_limits(ha_gs_limits *limits)
```

Parameters

Input A pointer to the data structure for the limits.

```
typedef struct {
    int max_provider_message_length;
    int max_state_value_length;
    int max_group_name_length;
} ha_gs_limits;
```

where:

- **max_provider_message_length** is the maximum number of bytes that can be sent at a time via the provider message protocol (that is, **ha_gs_send_message** or **ha_gs_vote**).
- **max_state_value_length** is the maximum number of bytes for the state value. It is currently 256 bytes.
- **max_group_name_length** is the maximum number of bytes for the group name. It is currently 32 bytes.

Output
: The contents of the data structure containing the limits which are currently supported by Group Services.

Description

Using this API, applications do not need to use predefined constant values (which may be changed later). This permits applications to maintain portability with respect to the limits of Group Services.

Return values

HA_GS_OK— The function call is successful.

HA_GS_NOT_OK— Internal error, or the Group Services limits are not available.

HA_GS_NO_INIT— The library is not yet initialized. Call **ha_gs_init** first.

Asynchronous errors

None.

Files

ha_gs.h

Related reference:

"ha_gs_init" on page 80

ha_gs_get_node_number

Purpose

ha_gs_get_node_number – Get the local node number.

Library

See "GSAPI library names" on page 46.

Syntax

```
#include <ha_gs.h>

ha_gs_rc_t ha_gs_get_node_number( int *node_number )
```

Parameters

node_number
: An output parameter which is a pointer to an integer which is the local node number returned from this function call.

Description

This function gives the local node number. When this function is called, the Group Services library will send back the local node number to the caller through the *node_number* parameter.

Return values

HA_GS_OK— The function call is successful.

HA_GS_NOT_OK— The GS library is not initialized successfully, or the local node number is not yet obtained from the Group Services. If the user inputs a NULL pointer for the parameter *node_number*, the function also returns **HA_GS_NOT_OK**.

HA_GS_NO_INIT— The GS library is not initialized yet. The user must call **ha_gs_init** first.

Asynchronous errors

None.

Files

ha_gs.h

Related reference:

"ha_gs_get_adapter_info" on page 65

ha_gs_get_rsct_active_version

Purpose

ha_gs_get_rsct_active_version – Get the current active version of RSCT.

Library

See "GSAPI library names" on page 46.

Syntax

```
#include <ha_gs.h>

ha_gs_rc_t
    ha_gs_get_rsct_active_version( ha_gs_rsct_version_t  *aVn)
```

Parameters

aVn A pointer to the **ha_gs_rsct_version_t** structure which gives the current active version of RSCT.

Description

This API is used to get the current RSCT active version for a peer domain. Since this is the active version for an RSCT peer domain, it will be NULL in all environments other than an RSCT peer domain. The API will return NULL if the current active version is not available. It will return **HA_GS_OK** if the version is available.

The **ha_gs_rsct_version_t** is defined as:

```
typedef struct
{
    struct
    {
        uint16_t       version;
        uint16_t       release;
        uint16_t       modlevel;
        uint16_t       fixlevel;
    } vrmf;
    uint32_t    quick_version;   /* combined VRMF */
} ha_gs_rsct_version_t;
```

Return values

If the **ha_gs_get_rsct_active_version** subroutine is successful, it returns a value of 0 (**HA_GS_OK**).

Error values

If the **ha_gs_get_rsct_active_version** subroutine is unsuccessful, it returns an error number synchronously.

The GSAPI error numbers are defined in the **ha_gs.h** header file. For more information on GSAPI errors, see "GSAPI return codes" on page 126.

Synchronous errors

The following errors may be returned synchronously by the **ha_gs_get_rsct_active_version** subroutine:

HA_GS_NO_INIT
The GS client has not yet successfully initialized itself with Group Services by calling **ha_gs_init**.

HA_GS_NOT_OK
The input parameter *aVn* is a null pointer, or there is an internal error.

Asynchronous errors

None

Files

ha_gs.h

Related reference:

"ha_gs_get_rsct_installed_version"

ha_gs_get_rsct_installed_version

Purpose

ha_gs_get_rsct_installed_version – Get the local installed RSCT version.

Library

See "GSAPI library names" on page 46.

Syntax

```
#include <ha_gs.h>

    ha_gs_rc_t
        ha_gs_get_rsct_installed_version( ha_gs_rsct_version_t  *iVn)
```

Parameters

iVn A pointer to the **ha_gs_rsct_version_t** structure which gives the local installed version of RSCT.

Description

This API is used to get the local installed RSCT version for a peer domain. Since this is the local version for an RSCT peer domain, it will be NULL in all environments other than an RSCT peer domain. The API will return NULL if the local installed version is not available. It will return **HA_GS_OK** if the version is available.

The **ha_gs_rsct_version_t** is defined as:

```
typedef struct
{
    struct
    {
        uint16_t       version;
        uint16_t       release;
        uint16_t       modlevel;
        uint16_t       fixlevel;
    } vrmf;
    uint32_t    quick_version;   /* combined VRMF */
} ha_gs_rsct_version_t;
```

Return values

If the **ha_gs_get_rsct_installed_version** subroutine is successful, it returns a value of 0 (**HA_GS_OK**).

Error values

If the **ha_gs_get_rsct_installed_version** subroutine is unsuccessful, it returns an error number synchronously.

The GSAPI error numbers are defined in the **ha_gs.h** header file. For more information on GSAPI errors, see "GSAPI return codes" on page 126.

Synchronous errors

The following error may be returned synchronously by the **ha_gs_get_rsct_installed_version** subroutine:

HA_GS_NOT_OK
: The input parameter *iVn* is a null pointer, or there is an internal error.

Asynchronous errors

None

Files

ha_gs.h

Related reference:

"ha_gs_get_rsct_active_version" on page 76

ha_gs_goodbye

Purpose

ha_gs_goodbye – Called by a provider of a group to immediately leave the group

Library

See "GSAPI library names" on page 46.

Syntax

```
#include <ha_gs.h>
 ha_gs_rc_t    ha_gs_goodbye(ha_gs_token_t provider_token
```

Parameters

provider_token
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.

Description

The **ha_gs_goodbye** subroutine enables a provider to immediately leave its group as if it had failed, while informing the group that it "failed" voluntarily. This is a synchronous interface.

If this call returns with an **HA_GS_OK** return code, then the calling provider is no longer in the group.

If the group is in an n-phase protocol, the provider issuing the **ha_gs_goodbye** protocol is considered to have failed. The Group Services subsystem will apply the group's default vote for any subsequent voting phases in that protocol.

The result of a provider successfully issuing this call is that when the remaining providers in the group see this provider's failure protocol, the leave reason for this provider will be set to **HA_GS_PROVIDER_SAID_GOODBYE**. (See the discussion of **gs_leave_info** in "ha_gs_n_phase_callback" on page 94.)

Restrictions

The calling process must be a provider.

Return values

If the **ha_gs_goodbye** subroutine is successful, it returns a value of 0 (**HA_GS_OK**).

Error values

If the **ha_gs_goodbye** subroutine is unsuccessful, it returns an error number. If the error is detected immediately, an error is returned synchronously.

The GSAPI error numbers are defined in the **ha_gs.h** header file. For more information on GSAPI errors, see "GSAPI return codes" on page 126.

Synchronous errors

The following errors may be returned synchronously by the **ha_gs_goodbye** subroutine:

HA_GS_BACKLEVEL_PROVIDERS
The group contains providers who were compiled against an older level of the GS library. They must leave the group before **ha_gs_goodbye** can be used by this provider.

HA_GS_BAD_MEMBER_TOKEN
The given **provider_token** does not specify a valid provider joined to a group.

HA_GS_NOT_SUPPORTED
The GS client was not compiled against the proper level of the GS library to use this function.

HA_GS_NO_INIT
The GS client has not yet successfully initialized itself with Group Services by calling **ha_gs_init**.

HA_GS_NOT_OK
The connection to the GS daemon has been lost. The GS client needs to reinitialize (via **ha_gs_init**).

Asynchronous errors

None.

Files

ha_gs.h

Related reference:

"ha_gs_change_attributes" on page 50
"ha_gs_change_state_value" on page 54
"ha_gs_expel" on page 62
"ha_gs_subscribe" on page 112
"ha_gs_join" on page 85
"ha_gs_leave" on page 91

ha_gs_init

Purpose

ha_gs_init – Called by a process to register with the Group Services application programming interface (GSAPI).

Library

See "GSAPI library names" on page 46.

Syntax

```
#include <ha_gs.h>

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)
```

Parameters

ha_gs_descriptor
: Is a pointer to a buffer in which the Group Services subsystem returns the file descriptor that the process uses to communicate with the GSAPI.

  The process itself must *not* read or write directly on this file descriptor.

socket_options
: Specifies one or more socket options. The **HA_GS_SOCKET_NO_SIGNAL** option must be specified. The valid options are:

  HA_GS_ENABLE_ADAPTER_INFO
  : Enables some GSAPI subroutines to obtain adapter information. In addition to the required **HA_GS_SOCKET_NO_SIGNAL** option, the caller must specify the **HA_GS_ENABLE_ADAPTER_INFO** option to use the **ha_gs_get_adapter_info** and **ha_gs_get_adapter_info_by_addr** subroutines, and optionally, the **ha_gs_get_adapter_info_by_id** subroutine.

HA_GS_ENABLE_DOMAIN_EVENT
Receives domain events.

HA_GS_ENABLE_IPV6
Allows client programs that are compiled with the current group services library header file to work without source code changes, even in an environment where IPv6 addresses are present. Clients that use this flag are expected to be able to handle IPv6 addresses. Clients that are compiled by using the **ha_gs.h** file in RSCT version 2.5.3, or later and the **HA_GS_ENABLE_IPV6** flag get an IPv6 address in the **.ipv6** field. If this flag is not used, only IPv4 addresses are returned by the GSAPI. Clients compiled by using the **ha_gs.h** file in RSCT version 2.5.3, or later that do not use the **HA_GS_ENABLE_IPV6** flag get IPv4 addresses in the **.ipv4** field.

HA_GS_ENABLE_MIGRATION_CALLBACK
Enables migration to Cluster-Aware AIX (CAA).

HA_GS_IMMEDIATE_DOMAIN_CONTROL
This option is reserved for IBM use.

HA_GS_SOCKET_NO_SIGNAL
Indicates that no signals are generated when messages arrive.

HA_GS_SOCKET_SIGNAL
This option is not supported.

HA_GS_STREAM
This option is reserved for IBM use.

responsiveness_control
Is a pointer to a buffer that contains a responsiveness control structure. The responsiveness control structure specifies the method, if any, that is used to perform responsiveness checks for this process.

deactivate_script
Is the path name to a "deactivate script" to be called during an expel protocol. This field is optional. For more information on deactivate scripts, see the **ha_gs_expel** man page.

responsiveness_callback
Is a pointer to a callback routine that is called when the GSAPI delivers a responsiveness notification to the Group Services client. For information on the responsiveness callback routine, see the **ha_gs_responsiveness_callback** man page.

delayed_error_callback
Is a pointer to a callback routine that is called when the GSAPI needs to deliver a delayed error number for a request that is discovered asynchronously to be invalid. A delayed error number can be delivered in response to any protocol or subscription request.

query_callback
This field is reserved for IBM use.

Description

The **ha_gs_init** subroutine is used by a process to register itself with the Group Services application programming interface (GSAPI). The subroutine allows the GSAPI to initialize itself as necessary and establishes a connection between the GSAPI and the Group Services client. This subroutine returns synchronously.

Only processes that meet one of the following conditions are allowed to initialize themselves with the GSAPI:

1. The effective group ID (EGID) is an AIX group called **hagsuser**.
2. The effective user ID (EUID) is **root**.

3. The EUID is a member of the **hagsuser** group.

See the *Administering RSCT* guide for information about setting up the Group Services subsystem to use the **hagsuser** group.

A Group Services client must indicate the Group Services domain to which it wants to attach by setting the following environment variables before calling **ha_gs_init**. The Group Services domain can be a PowerHA® cluster or an RSCT peer domain.

- If the Group Services client is connecting to a PowerHA cluster, **HA_GS_SUBSYS** should be set to **grpsvcs**.
- For a Group Services PowerHA domain, the **HA_DOMAIN_NAME** environment variable must be set and exported in a Group Services client environment to the name of the domain in which the Group Services client and the Group Services daemon are running. On a node, this is the domain to which the node belongs. The setting of **HA_DOMAIN_NAME** identifies the domain and the particular Group Services daemon to which the group services client connects.

  To connect to a Group Services PowerHA domain, **HA_DOMAIN_NAME** should be set to the name of the PowerHA cluster.

  Note: Earlier releases of Group Services supported the **HA_SYSPAR_NAME** environment variable. **HA_SYSPAR_NAME** is still supported for compatibility, but all new applications should use the **HA_DOMAIN_NAME** environment variable. **HA_DOMAIN_NAME** can be used to refer to a domain on a PowerHA cluster. On a PowerHA cluster, a domain is the entire cluster.
- For an RSCT peer domain, the **HA_DOMAIN_NAME**, **HA_GS_SUBSYS**, and **HA_SYSPAR_NAME** environment variables should *not* be set. If the **CT_DOMAIN** environment variable is set to **1**, these three environment variables are ignored and a Group Services client always tries to connect to the RSCT peer domain.

The format of the **ha_gs_socket_ctrl_t** datatype follows:

```
typedef enum
{
    HA_GS_SOCKET_NO_SIGNAL             = 0x0000000,
    HA_GS_SOCKET_SIGNAL                = 0x00000001,
    HA_GS_ENABLE_ADAPTER_INFO          = 0x00000002,
    HA_GS_ENABLE_DOMAIN_EVENT          = 0x00001000,
    HA_GS_ENABLE_IPV6                  = 0x00002000,
    HA_GS_ENABLE_MIGRATION_CALLBACK    = 0x00004000,
    HA_GS_STREAM                       = 0x00800000,
    HA_GS_IMMEDIATE_DOMAIN_CONTROL     = 0x10000000
} ha_gs_socket_ctrl_t;
```

For more information about the **ha_gs_socket_ctrl_t** datatype, see "IP addressing" on page 33.

The format of the **ha_gs_responsiveness_t** data structure follows:

```
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;
```

The **ha_gs_responsiveness_t** structure is used to specify whether the process wants the GSAPI to check it periodically for responsiveness and, if so, the responsiveness check to be used. The GSAPI can always detect the actual exit (intentional or otherwise) of all Group Services clients. However, this check allows the GSAPI to determine if the Group Services client is able to respond in a reasonable fashion. It also allows the Group Services client to perform any periodic validity checks on its own operation or environment that might be needed.

If the Group Services client fails a responsiveness check and it is joined to any groups as a provider, the other providers in the groups receive an announcement that a provider has failed its responsiveness check.

The responsiveness check is run only when the Group Services client is idle. If the group services client is involved in group actions (for example, running protocols) and it is responding as expected, the GSAPI does not perform the responsiveness check.

The **gs_responsiveness_type** field contains the type of responsiveness check that is to be performed for this Group Services client. The valid options are:

HA_GS_NO_RESPONSIVENESS
: Indicates that the GSAPI must not perform a responsiveness check for this Group Services client. The remaining fields in the structure are ignored.

HA_GS_PING_RESPONSIVENESS
: Indicates that the GSAPI must "ping" the Group Services client periodically, by delivering a notification to the Group Services client and expecting a response. The notification calls the responsiveness callback routine specified on input by the Group Services client.

HA_GS_COUNTER_RESPONSIVENESS
: Indicates that the GSAPI must periodically check an arithmetic counter specified by a multi-threaded Group Services client. If the counter is changing, the Group Services client is assumed to be responsive. If the counter does not change, the GSAPI calls the responsiveness callback routine specified by the Group Services client before assuming that the Group Services client is unresponsive.

  This option is not supported.

The **gs_responsiveness_type** field includes the **HA_GS_CRITICAL_CLIENT** value along with any of the previous values (for example, **HA_GS_NO_RESPONSIVENESS** and **HA_GS_CRITICAL_CLIENT**) to indicate that the process that is being registered with the Group Services is a critical client. For the AIX operating system, the resulting behavior is that if the Group Services end before all of its critical clients are disconnected, the kernel crashes. The other operating systems have no significant behavioral changes.

The **gs_responsiveness_interval** field contains the number of seconds that the GSAPI should wait between invocations of the specified responsiveness protocol.

The **gs_responsiveness_response_time_limit** field contains the number of seconds that the GSAPI should wait for a return from the responsiveness callback routine. If the routine fails to return, the GSAPI assumes that the Group Services client has become nonresponsive.

The **gs_counter_location** field points to the counter that the GSAPI should monitor for the **HA_GS_COUNTER_RESPONSIVENESS** protocol. The counter must reside in the Group Services client's address space. If the **HA_GS_PING_RESPONSIVENESS** protocol is specified, this field is ignored. This field is not supported.

The **gs_counter_length** field contains the length in bytes, of the counter to be monitored. It can be a value of **2**, **4**, or **8**. If the **HA_GS_PING_RESPONSIVENESS** protocol is specified, this field is ignored.

Security

To use the GSAPI subroutines, the calling process needs to have an effective group ID (EGID) of **hagsuser**, an effective user ID (EUID) of **root**, or an EUID that is a member of the **hagsuser** group.

Return values

If the **ha_gs_init** subroutine is successful, it returns a value of 0 (**HA_GS_OK**), and the **ha_gs_descriptor** field contains the file descriptor of the GSAPI socket.

Error values

If the **ha_gs_init** subroutine is unsuccessful, it returns an error number synchronously. The contents of the **ha_gs_descriptor** field are undefined.

If the calling process fails to connect the Group Services subsystem because of insufficient authority, the **ha_gs_init** subroutine may be successful, but **ha_gs_dispatch** returns an error synchronously.

The GSAPI error numbers are defined in the **ha_gs.h** header file. For more information on GSAPI errors, see "GSAPI return codes" on page 126.

Synchronous errors

The following errors may be returned synchronously by the **ha_gs_init** subroutine:

HA_GS_BAD_PARAMETER
: A Null delayed error callback value was specified, non-valid responsiveness control information was specified, or a non-valid socket option was specified. The **HA_GS_SOCKET_NO_SIGNAL** must be specified, by itself, or with one or more other options.

HA_GS_EXISTS
: This GS client has already successfully invoked **ha_gs_init**.

HA_GS_NO_MEMORY
: The GS library is unable to allocate memory. The GS client should retry the request.

HA_GS_NOT_OK
: The connection to the GS daemon has been lost. The GS client needs to reinitialize (via **ha_gs_init**).

HA_GS_NOT_SUPPORTED
: The GS client was compiled against a newer version of the GS library than is currently installed on this node.

HA_GS_CONNECT_FAILED
: The GS daemon on this node is not running; or this GS client does not have appropriate authority to connect the GS daemon.

HA_GS_SOCK_CREATE_FAILED
: Internal error. Retry request.

HA_GS_SOCK_INIT_FAILED
: Internal error. Retry request.

Asynchronous errors

None.

Files

ha_gs.h

Related concepts:

"Using the HA_GS_ENABLE_IPV6 option" on page 33
Use the **HA_GS_ENABLE_IPV6** option in the **ha_gs_socket_ctrl_t** data type to indicate that the client

can handle IPv6 addresses.

"Group membership" on page 4
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. Any authorized process in the domain may ask to become a member of a group.

Related reference:

"ha_gs_announcement_callback" on page 48

"ha_gs_change_attributes" on page 50

"ha_gs_change_state_value" on page 54

"ha_gs_dispatch" on page 59

"ha_gs_get_limits" on page 74

"ha_gs_send_message" on page 110

"ha_gs_protocol_rejected_callback" on page 104

ha_gs_join

Purpose

ha_gs_join – Called by a GS client to join a group as a provider

Library

See "GSAPI library names" on page 46.

Syntax

```
#include <ha_gs.h>

ha_gs_rc_t
    ha_gs_join(
                ha_gs_token_t          *provider_token,
        const   ha_gs_proposal_info_t  *proposal_info)
```

Parameters

provider_token
: A pointer to a token that will be returned by the Group Services subsystem. This token is used to identify the membership of the GS client in the group as a provider.

  The GS client must pass a copy of the token on all subsequent GSAPI calls that refer to the group. The GSAPI passes a copy of the token to the GS client on all subsequent callbacks that refer to the group.

proposal_info
: A pointer to a buffer that contains a proposal information block, which describes the proposed join.

Description

The **ha_gs_join** subroutine is used by a GS client to join a group as a provider. If the named group does not already exist, it is created.

This section describes:

- The input to the **ha_gs_join** subroutine
- The **ha_gs_join** subroutine's synchronous and asynchronous operation.

On input, information about the join request is supplied through the join request block, which is a type of proposal information block. On the **ha_gs_join** subroutine, specify the proposal information block as a join request block. For the definition of the proposal information block, see the **ha_gs_delayed_error_callback** man page.

The join request block has the following definition:

```
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;
```

The **gs_group_attributes** field contains a pointer to a group attributes block, which is described later in this section.

The gs_provider_instance field contains the instance number to be used by this provider. If it is not unique on this node for this group, the join is rejected asynchronously with an error number of **HA_GS_DUPLICATE_INSTANCE_NUMBER**.

The **gs_provider_local_name** field points to an optional byte string that contains a local "name" for the provider. The name is used only locally, when logging errors or messages related to the provider. The provider local name is not distributed to the other providers.

The **gs_n_phase_protocol_callback** field points to the callback routine that is to be called during each voting phase of any n-phase protocol.

The **gs_protocol_approved_callback** field points to the callback routine that is to be called when the Group Services subsystem has an announcement to deliver that a protocol (one-phase or n-phase) has been approved in this group.

The **gs_protocol_rejected_callback** field points to the callback routine that is to be called when the Group Services subsystem has an announcement to deliver that a protocol (one-phase or n-phase) has been rejected in this group.

The **gs_announcement_callback** field points to the callback routine that is to be called when the Group Services subsystem has other announcements to deliver that are related to this group.

All of the above fields that point to callback routines should point to valid functions. If a NULL pointer is given in any of them, a warning will be issued to the GS client's STDERR. If a pointer is NULL, or if it is not NULL but does not point to an executable function, unpredictable results (such as fatal program failure) may occur during execution of your program.

The **gs_merge_callback** field is reserved for IBM use. It should be set to NULL.

The group attributes block describes the attributes of the group, including the group's name, and is specified as input to a join request using the **ha_gs_join** subroutine. It has the following definition:

```
typedef char    *ha_gs_group_name_t;

typedef struct {
    short               gs_version;
    short               gs_sizeof_group_attributes;
    unsigned            gs_client_version;
    ha_gs_group_name_t  gs_group_name;
```

```
    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_source_group_name;
} ha_gs_group_attributes_t;
```

The group attributes block contains the name of the group and the set of group attributes that are passed to the Group Services subsystem on the **ha_gs_join** subroutine call.

If the group does not already exist, a new group is defined with the specified attributes.

If the group already exists, the specified attributes must match the group's existing attributes. All of the attributes must match except those that are contained in the **gs_version** and **gs_sizeof_group_attributes** fields. Otherwise, the call fails with an error number of **HA_GS_BAD_GROUP_ATTRIBUTES**.

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

The **gs_sizeof_group_attributes** field contains the size of the group attributes block.

The **gs_client_version** field contains a user-defined version code.

The **gs_group_name** field points to a string that contains the name of the group. Its maximum length is 32 bytes, which can obtained by invoking the subroutine **ha_gs_get_limits**.

The **gs_batch_control** field controls the batching of multiple group joins and failure leaves. Either alone may be batched, or both. A failure leave occurs when a provider process is forced to leave a group because the process, or the node on which it is running, fails.

The **gs_batch_control** field can take one of the following values:

HA_GS_NO_BATCHING
: No batching is allowed. Joins and failure leaves are serialized and presented to the group one at a time.

HA_GS_BATCH_JOINS
: Any number of joins may be batched with other joins. Failure leaves are not batched.

HA_GS_BATCH_LEAVES
: Any number of failure leaves may be batched with other failure leaves. Joins are not batched.

HA_GS_BATCH_BOTH
: Any number of joins may be batched with other joins, and any number of failure leaves may be batched with other failure leaves.

HA_GS_DEACTIVATE_ON_FAILURE
: Enables the invocation of a deactivate script when the provider is failing.

See “Deactivate-on-failure handling” on page 16.

HA_GS_COLLECT_VOTE_RESULT
: Enables the collection of voting results made on a proposal. **HA_GS_COLLECT_VOTE_RESULT** collects only a vote list. A message list can be collected with **HA_GS_COLLECT_MSG_RESULT**, a state value list can be collected with **HA_GS_COLLECT_STATEVALUE_RESULT**, and all results can be collected with **HA_GS_COLLECT_ALL_RESULT**.

Each provider's vote is collected and reported through the callback routines **ha_gs_n_phase_callback**, **ha_gs_protocol_approved_callback**, and **ha_gs_protocol_rejected_callback**.

For more information, see "Enabling the collection of voting results" on page 34.

HA_GS_COLLECT_MSG_RESULT
Enables the collection of voting results made on a proposal. **HA_GS_COLLECT_MSG_RESULT** collects only a message list. A vote list can be collected with **HA_GS_COLLECT_VOTE_RESULT**, a state value list can be collected with **HA_GS_COLLECT_STATEVALUE_RESULT**, and all results can be collected with **HA_GS_COLLECT_ALL_RESULT**.

Each provider's vote is collected and reported through the callback routines **ha_gs_n_phase_callback**, **ha_gs_protocol_approved_callback**, and **ha_gs_protocol_rejected_callback**.

For more information, see "Enabling the collection of voting results" on page 34.

HA_GS_COLLECT_STATEVALUE_RESULT
Enables the collection of voting results made on a proposal. **HA_GS_COLLECT_STATEVALUE_RESULT** collects only a state value. A vote list can be collected with **HA_GS_COLLECT_VOTE_RESULT**, a message list can be collected with **HA_GS_COLLECT_MSG_RESULT**, and all results can be collected with **HA_GS_COLLECT_ALL_RESULT**.

Each provider's vote is collected and reported through the callback routines **ha_gs_n_phase_callback**, **ha_gs_protocol_approved_callback**, and **ha_gs_protocol_rejected_callback**.

For more information, see "Enabling the collection of voting results" on page 34.

HA_GS_COLLECT_ALL_RESULT
Enables the collection of voting results made on a proposal. **HA_GS_COLLECT_ALL_RESULT** collect the vote list, message list, and state value list. You can use **HA_GS_COLLECT_VOTE_RESULT**, **HA_GS_COLLECT_MSG_RESULT**, and **HA_GS_COLLECT_STATEVALUE_RESULT** to collect the lists individually.

Each provider's vote is collected and reported through the callback routines **ha_gs_n_phase_callback**, **ha_gs_protocol_approved_callback**, and **ha_gs_protocol_rejected_callback**.

For more information, see "Enabling the collection of voting results" on page 34.

The **gs_num_phases** field specifies whether join protocols and failure leave (including cast-out) protocols are to be n-phase protocols or one-phase protocols. It can take one of the following values:

HA_GS_1_PHASE
The protocols are to be one-phase protocols. One-phase protocols are automatically approved.

HA_GS_N_PHASE
The protocols are to be n-phase protocols, which put the group into multi-phase voting.

The **gs_source_reflection_num_phases** field contains the number of phases for source-reflection protocols, which are run in the target-group when the source-group changes its state value. If no **gs_source_group_name** is given, this field is ignored.

The **gs_source_reflection_num_phases** field can take one of the following values:

HA_GS_1_PHASE
The protocols are to be one-phase protocols. One-phase protocols are automatically approved.

HA_GS_N_PHASE
The protocols are to be n-phase protocols, which put the group into multi-phase voting.

The **gs_group_default_vote** field contains the default vote to use 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.

The **gs_merge_control** field specifies how the merging of groups should be handled. It must be set to a value of **HA_GS_DISSOLVE_MERGE**.

The **gs_time_limit** field contains the voting phase time limit, in seconds. This is the number of seconds within which each provider must register its vote for each phase of an n-phase join or failure leave protocol. If the field is set to a value of 0, no limit is enforced.

The **gs_source_reflection_time_limit** field contains the time limit, in seconds, for each voting phase of a source-reflection protocol. It is run in the target-group when the source-group changes its state value. 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.

The **gs_source_group_name** 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.

The **ha_gs_join** subroutine operates as follows.

First, it verifies that all of the required fields have been specified. If this checking succeeds, it submits a join request to initiate a join protocol within the specified group and returns synchronously with a successful return value (**HA_GS_OK**).

The join request is processed asynchronously. If errors are detected asynchronously, they are returned through the delayed error callback routine that was previously specified on the call to the **ha_gs_init** subroutine.

Upon receipt of the join request, Group Services checks the group attributes that were specified on input. If the named group already exists, it checks to see that the input group attributes match those that have already been established for the group. If they do not match, the **HA_GS_BAD_GROUP_ATTRIBUTES** error number is returned asynchronously by the delayed error callback routine.

If the join request is for a new group, Group Services uses the attributes specified on the join request to establishes the new group's attributes.

If the asynchronous checks succeed, Group Services initiates a membership change protocol within the group to enable the provider to join. Each provider in the group, including the joiner, is notified. The appropriate callback is invoked, based on the number of phases that are required for the join request.

Restrictions

The calling process must be a GS client.

Return values

If the **ha_gs_join** subroutine is successful, it returns a value of 0 (**HA_GS_OK**) and the **provider_token** field is set to the token that identifies this provider's connection to the group.

Note that provider tokens are assigned, invalidated, and reassigned in a manner similar to the way in which file descriptors are assigned, invalidated, and reassigned as files are opened and closed.

Here is an example.

A GS client joins group **foo** and receives provider token 0. When the same client leaves group **foo**, Group Services invalidates provider token 0 and makes it available for reassignment. When the next GS client (it

could be the same or a different GS client) joins the next group (it could be the same or a different group), Group Services assigns provider token 0 to that client.

If the **ha_gs_join** subroutine returns **HA_GS_OK** synchronously, that means only that the Group Services subsystem has accepted the join request. The GS client is not fully joined to the group as a provider until the join protocol has run and has been approved by the group.

Error values

If the **ha_gs_join** subroutine is unsuccessful, it returns an error number. The contents of the **provider_token** field are undefined.

If either an asynchronous error is reported by the delayed error callback routine, or the join protocol is rejected by the group, the GS client is not a provider in the group and the provider token that was returned is not valid for any requests.

The GSAPI error numbers are defined in the **ha_gs.h** header file. For more information on GSAPI errors, see "GSAPI return codes" on page 126.

Synchronous errors

The following errors may be returned synchronously by the **ha_gs_join** subroutine:

HA_GS_BAD_PARAMETER
: A null pointer was given for the merge callback but the merge control setting is not **HA_GS_DISSOLVE_MERGE**.

HA_GS_GROUP_ATTRIBUTES
: Values specified for the group attributes are not valid.

HA_GS_INVALID_GROUP
: The name of the group to be joined was NULL or zero length; or the name of the host membership group (**HA_GS_HOST_MEMBERSHIP_GROUP**) or an adapter membership group was given as the group to be joined.

HA_GS_INVALID_SOURCE_GROUP
: The name of the source group to be joined was NULL or zero length; or the name of the host membership group (**HA_GS_HOST_MEMBERSHIP_GROUP**) or an adapter membership group was given as the source group.

HA_GS_NAME_TOO_LONG
: The name of the group to be joined or the source group is too long.

HA_GS_NO_INIT
: The GS client has not yet successfully initialized itself with Group Services by calling **ha_gs_init**.

HA_GS_NO_MEMORY
: The GS library is unable to allocate memory. The GS client should re-try the request.

HA_GS_NOT_OK
: The connection to the GS daemon has been lost. The GS client needs to reinitialize (via **ha_gs_init**).

Asynchronous errors

The following errors may be returned asynchronously by the **ha_gs_join** subroutine:

HA_GS_BAD_PARAMETER
: The specified parameter was not valid.

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 number 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.

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.

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.

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.

Files

ha_gs.h

Related concepts:

"Enabling the collection of voting results" on page 34
After a proposal is approved or rejected, the Group Services subsystem notifies all of the providers of the outcome.

Related reference:

"ha_gs_announcement_callback" on page 48
"ha_gs_change_attributes" on page 50
"ha_gs_delayed_error_callback" on page 57
"ha_gs_goodbye" on page 78
"ha_gs_change_state_value" on page 54
"ha_gs_protocol_rejected_callback" on page 104
"ha_gs_quit" on page 107
"ha_gs_send_message" on page 110
"ha_gs_vote" on page 123

ha_gs_leave

Purpose

ha_gs_leave – Called by a provider of a group to leave the group voluntarily

Library

See "GSAPI library names" on page 46.

Syntax

```
#include <ha_gs.h>

ha_gs_rc_t
    ha_gs_leave(
              ha_gs_token_t           provider_token,
        const   ha_gs_proposal_info_t   *proposal_info)
```

Parameters

provider_token
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.

proposal_info
A pointer to a buffer that contains a proposal information block, which describes the proposed leave request.

Description

The **ha_gs_leave** subroutine is used by a provider of a Group Services group to leave the group.

If the request is specified as a one-phase protocol, and Group Services chooses to run this protocol, the group's providers are notified using normal protocol approval procedures.

If the request is specified as an n-phase protocol, and Group Services chooses to run this protocol, the group's providers are notified using normal n-phase voting procedures.

If the Group Services subsystem chooses not to run this protocol (because another protocol is already in progress), the **HA_GS_COLLIDE** error number is returned either synchronously or asynchronously, depending on when the error is detected. Asynchronous errors are delivered through the delayed error callback routine. Otherwise, the proposal will initiate a protocol within the group.

Information about the leave request is supplied through the leave request block, which is a type of proposal information block. On the **ha_gs_leave** subroutine, specify the proposal information block as a leave request block. For the definition of the proposal information block, see the **ha_gs_delayed_error_callback** man page.

The leave request block has the following definition:

```
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;
```

The **gs_num_phases** field specifies whether the leave protocols are to be n-phase or one-phase protocols. It can take one of the following values:

HA_GS_1_PHASE
The protocols are to be one-phase protocols. One-phase protocols are automatically approved.

HA_GS_N_PHASE
The protocols are to be n-phase protocols, which put the group into multi-phase voting.

The **gs_time_limit** field contains the voting phase time limit, in seconds. This is the number of seconds within which each provider must register its vote for each phase of an n-phase protocol. If the field is set to a value of 0, no limit is enforced.

The **gs_leave_code** field contains a four-byte value that is defined by the application using the GSAPI, 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 leave protocol notification. Leave codes are not interpreted by the Group Services subsystem.

Restrictions

The calling process must be a provider.

Return values

If the **ha_gs_leave** subroutine is successful, it returns a value of 0 (**HA_GS_OK**). Group Services has accepted the request and will asynchronously attempt to run the proposed protocol.

Once a voluntary leave protocol is started within the group by the Group Services subsystem, the provider who proposed the leave will receive only the first notification of the protocol. The first notification is the n-phase notification of an n-phase protocol, or the approved notification of a one-phase protocol. After this point, this provider is removed from the group and receives no more notifications. Even if the protocol is rejected, the provider is still removed from the group.

Error values

If the **ha_gs_leave** subroutine is unsuccessful, it returns an error number. If the error is detected immediately, an error is returned synchronously. If the error is detected after the call has been accepted, an error is returned asynchronously.

The GSAPI error numbers are defined in the **ha_gs.h** header file. For more information on GSAPI errors, see “GSAPI return codes” on page 126.

Synchronous errors

The following errors may be returned synchronously by the **ha_gs_leave** subroutine:

HA_GS_BAD_MEMBER_TOKEN
: The given **provider_token** does not specify a valid provider joined to a group.

HA_GS_BAD_PARAMETER
: The number of phases specified for the protocol is not allowable; it must be **HA_GS_1_PHASE** or **HA_GS_N_PHASE**.

HA_GS_NO_INIT
: The GS client has not yet successfully initialized itself with Group Services by calling **ha_gs_init**.

HA_GS_COLLIDE
: The provider's group is already running a protocol; or this provider has already submitted a protocol request.

HA_GS_NOT_A_MEMBER
: The given **provider_token** does not specify a valid group.

HA_GS_NOT_OK
: The connection to the GS daemon has been lost. The GS client needs to reinitialize (via **ha_gs_init**).

Asynchronous errors

The following errors may be returned asynchronously by the **ha_gs_leave** subroutine:

HA_GS_NOT_A_MEMBER
: The provider that is proposing the protocol is no longer a provider for the specified group.

HA_GS_BAD_PARAMETER
: The specified parameter was not valid.

HA_GS_COLLIDE
: Another protocol is already active for this group.

Files

ha_gs.h

Related reference:

"ha_gs_announcement_callback" on page 48
"ha_gs_change_attributes" on page 50
"ha_gs_goodbye" on page 78
"ha_gs_change_state_value" on page 54
"ha_gs.h header file" on page 125

ha_gs.h is a header file that provides datatypes and structures for use with the GSAPI subroutines.

ha_gs_n_phase_callback

Purpose

ha_gs_n_phase_callback – A callback routine that the Group Services subsystem calls to deliver an n-phase notification to a GS client

Library

See "GSAPI library names" on page 46.

Syntax

```
#include <ha_gs.h>

void
    ha_gs_n_phase_callback(
        const   ha_gs_n_phase_notification_t           *notification)
```

Parameters

notification
A pointer to an n-phase notification block.

Description

The **ha_gs_n_phase_callback** subroutine defines a GS client's n-phase callback routine. The GS client uses it to handle n-phase notifications from the Group Services subsystem. The process provides the address of the n-phase callback routine to the Group Services subsystem on the **ha_gs_join** subroutine when it joins the group as a provider. The Group Services subsystem then calls the n-phase callback routine when it has an n-phase notification to deliver to the GS client. This occurs during each voting phase of an n-phase protocol.

On input, the n-phase callback routine receives information that specifies the proposed changes to the group, such as its membership or its state value, as well as other control information for the protocol, such as the phase number and the voting time limit.

In response to this notification, it is expected that the provider will vote on the proposal by calling the **ha_gs_vote** subroutine. The call to submit the vote may be made either before or after the callback routine returns. The notification contains an identifying token that must be passed on the call to the **ha_gs_vote** subroutine so that the Group Services subsystem can match the vote to the protocol.

On input, the n-phase callback routine receives a pointer to the n-phase notification block. The n-phase notification block has the following definition:

```
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;
```

The **gs_notification_type** field contains the type of notification. For an n-phase notification, it contains a value of **HA_GS_N_PHASE_NOTIFICATION**.

The **gs_provider_token** field contains a token that identifies the caller as a provider of the group. This token was previously initialized when the provider joined the group by using the **ha_gs_join** subroutine.

The **gs_protocol_type** field contains the type of request for which this n-phase notification is being delivered.

HA_GS_JOIN
: One or more providers are attempting to join the group.

HA_GS_FAILURE_LEAVE
: One or more providers has failed and is leaving the group.

HA_GS_LEAVE
: A provider is voluntarily leaving the group.

HA_GS_EXPEL
: A provider is attempting to expel one or more providers from the group.

HA_GS_STATE_VALUE_CHANGE
: A provider is trying to change the group's state value.

HA_GS_PROVIDER_MESSAGE
: A provider is broadcasting a message to the group.

HA_GS_CAST_OUT
: One or more source-group providers have left the source-group, requiring these target-group providers to leave also. The source-group's state value is contained in the **gs_source_state_value** field of the proposal information block.

HA_GS_SOURCE_STATE_REFLECTION
: The source-group has modified its state value, and this is being reflected to the target-groups. The source-group's state value is contained in the **gs_source_state_value** field of the proposal information block.

HA_GS_MERGE
: This value is reserved for IBM use.

HA_GS_GROUP_ATTRIBUTE_CHANGE
: A provider has requested to change the group's attributes via the **ha_gs_change_attributes** interface.

The **gs_summary_code** field contains one or more flags that indicate whether any default votes were recorded during any previous voting phase. It can contain one or more of the following flags:

HA_GS_DEFAULT_APPROVE
: This flag is set for a protocol approved notification 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 flags are also set.

HA_GS_DEFAULT_REJECT
This flag is set for a protocol rejected notification 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 flags 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.

The **gs_time_limit** field contains the time limit, in seconds, within which the provider must submit its vote for this voting phase. If the field is set to a value of 0, no limit is enforced.

The **gs_proposal** field points to the proposal block for the proposal on which the vote is requested.

The proposal block is common to the notifications that carry proposal information for one-phase and n-phase protocols. It is provided on input to protocol callback functions and has the following definition:

```
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_merge_info_t          *gs_merge_info;
    ha_gs_vote_result_array_t   *gs_current_vote_results;
    ha_gs_vote_result_array_t   *gs_changing_vote_results;
} ha_gs_proposal_t;
```

The **gs_phase_info** field contains information about the type of protocol that is running and the phase number to which this notification applies. It has the following definition:

```
typedef struct {
    ha_gs_num_phases_t      gs_num_phases;
    ha_gs_num_phases_t      gs_phase_number;
} ha_gs_phase_info_t;
```

The **gs_num_phases** field contains:

HA_GS_1_PHASE
The executing protocol is a one-phase protocol.

HA_GS_N_PHASE
The executing protocol is an n-phase protocol.

The **gs_phase_number** field contains the phase number to which this notification applies.

The **gs_proposed_by** field contains the provider information block that identifies the provider (or the Group Services subsystem itself) that initiated the executing proposal. The provider information block is defined later in this section.

On all join protocols, this field always contains the provider information block for the GS client that is running the callback rather than the provider that initiated the join. This allows each provider to capture its own provider information block. For protocols (such as the cast-out protocol) that are initiated by the Group Services subsystem itself, this field contains the values that so identify it. For more details, see the description of the provider information block later in this section.

The **gs_whats_changed** field contains one or more flags that indicate whether the membership or the state values contained in the proposal are changes from the base group values at the beginning of the protocol, and if the notification contains a provider-broadcast message. It can contain one or more of the following flags:

HA_GS_NO_CHANGE
: No fields have been updated from a previous voting phase 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 running. 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 state 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 voting-phase notifications and final notifications. 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 state value change protocol. The source-group's state value is presented only with the first notification that is given to the target-groups. 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 via an **ha_gs_change_attributes** interface call.

HA_GS_ONGOING_GROUP_ATTRIBUTES
: The **gs_new_group_attributes** field contains the new group attributes that were proposed via an **ha_gs_change_attributes** interface call, and these are unchanged from a previous notification.

HA_GS_UPDATED_GROUP_ATTRIBUTES
: This flag is set on the final notification for an approved change attributes protocol, proposed via an **ha_gs_change_attributes** interface call. The **gs_new_group_attributes** field contains the new group attributes.

HA_GS_REJECTED_GROUP_ATTRIBUTES
: This flag is set on the final notification for a rejected change attributes protocol, proposed via an **ha_gs_change_attributes** interface call. The **gs_new_group_attributes** field contains the rejected group attributes.

HA_GS_UPDATED_MEMBERSHIP
: Membership has changed. The **gs_current_providers** field points to a list of the current members of the group.

The **gs_current_providers** field points to a list of providers that currently belong to the group. It has the following definition:

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

The **gs_count** field contains the number of providers in the list.

The **gs_providers** field points to the list of providers. Each provider is described by a provider information block, which is defined later in this section.

The **gs_changing_providers** field points to a list of providers that are joining or leaving the group through this protocol. If none are joining or leaving, the field is null.

The **gs_leave_info** field points to an array that contains the reason codes for each provider specified in the **gs_changing_providers** field that is leaving the group. This leave information field, which is used for voluntary and failure leave protocols only, has the following definition:

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

The **gs_count** field contains the number of providers that are leaving.

The **gs_leave_codes** field points to an entry for each provider that is leaving the group. This entry specifies whether it is a voluntary or failure leave, and the reason or reasons for the leave. The leave reason entries are in the same order in which the providers are listed in the **gs_changing_providers** list.

The leave reason entries have the following definition:

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

The **gs_voluntary_or_failure** field contains one or more of the following flags:

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 **voluntary_leave_code** field contains the application-defined leave code that was specified on input to the **ha_gs_leave** subroutine.

HA_GS_PROVIDER_FAILURE
: The provider is leaving the group because its process has failed.

  If the Group Services subsystem detected that the provider's process failed, and its node also failed before the process failure could be reported, this flag could be set with the **HA_GS_HOST_FAILURE** flag.

  If this flag is set, the **gs_voluntary_leave_code** field is not used and is undefined.

HA_GS_HOST_FAILURE
: The provider is leaving the group because its node has failed.

If the Group Services subsystem detected that the provider's process failed, and its node also failed before the process failure could be reported, this flag could be set with the **HA_GS_PROVIDER_FAILURE** flag.

If this flag is set, the **gs_voluntary_leave_code** field is not used and is undefined.

HA_GS_PROVIDER_EXPELLED
The provider is leaving the group because a provider has requested its expulsion by the **ha_gs_expel** subroutine.

If this flag is set, the **gs_voluntary_leave_code** field is not used and is undefined.

HA_GS_SOURCE_PROVIDER_LEAVE
The provider is being cast out of the group because it belongs to a target-group and the source-group provider on its node has left the source-group. 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.

If this flag is set, the **gs_voluntary_leave_code** field is not used and is undefined.

HA_GS_PROVIDER_SAID_GOODBYE
The provider issued the **ha_gs_goodbye** interface and has left the group.

The **gs_expel_info** field points to a structure that contains expel information. This expel information field, which is used for expel protocols only, has the following definition:

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

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

The **gs_expel_flag_length** field contains the length of the expel flag.

The **gs_expel_flag** 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.

The **gs_current_state_value** field points to a buffer that contains the current state value of the group. This is the latest approved state value of the group, which is the state value as it was at the beginning of the protocol. For the definition of the group state value, see the **ha_gs_change_state_value** man page.

The **gs_proposed_state_value** field points to a buffer that contains the proposed new value for the group's state. 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 this protocol, this field is null. For the definition of the group state value, see the **ha_gs_change_state_value** man page.

The **gs_source_state_value** field points to a buffer that contains the updated state value of this group's source-group, if this 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. For the definition of the group state value, see the **ha_gs_change_state_value** man page.

The **gs_provider_message** field points to a buffer 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. For information on the definition of the provider-broadcast message, see the **ha_gs_send_message** man page.

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 running. It has the following definition:

```
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;
```

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

When Group Services itself is acting as a "provider," the **gs_instance_number** field contains a value of **HA_GS_instance_number**.

The **gs_node_number** field contains the node number of the provider. This node number is specified by the Group Services subsystem.

When Group Services itself is acting as a "provider," the **gs_node_number** field contains a value of **HA_GS_node_number**.

The **gs_provider_id** field contains the **gs_instance_number** and the **gs_node_number** in a single word.

The **gs_new_group_attributes** field contains the new group attributes that were proposed by calling the **ha_gs_change_attributes** subroutine.

The **gs_merge_info** field is unsupported.

The **gs_current_vote_results** and **gs_changing_vote_results** fields point to an array that contains the voting results for each provider. It has the following definition:

```
typedef struct
{
     ha_gs_provider_t            gs_voter;
     ha_gs_vote_value_t          gs_vote_value;
     ha_gs_summary_code_t        gs_summary_code;
     ha_gs_leave_reasons_t       gs_leave_code;
     ha_gs_state_value_t         *gs_proposed_state_value;
     ha_gs_provider_message_t    *gs_provider_message;
} ha_gs_vote_result_t;

typedef ha_gs_vote_result_t
*ha_gs_vote_result_ptr_t;

typedef struct
{
     unsigned int               gs_count;
     ha_gs_vote_result_ptr_t    *gs_vote_results;
} ha_gs_vote_result_array_t;
```

In this definition:

gs_voter
indicates a member provider.

gs_vote_value
the provider's vote value.

gs_summary_code
contains one or more flags that indicate whether any default votes were recorded during any previous voting phase. For more information, refer to the explanation of **gs_summary_code** in “ha_gs_protocol_approved_callback” on page 102.

gs_leave_code
gives the failure reason if the **gs_summary_code** indicates that the provider failed. Otherwise its value will be 0.

gs_proposed_state_value
If the provider proposed a new state value, this field will indicate the proposed new state value. Otherwise this field will be NULL.

gs_provider_message
If the provider proposed a new broadcast message, this field will indicate the proposed message. Otherwise this field will be NULL.

Restrictions

The following discussion of multiprocessing considerations applies to all callback routines, not just those for handling n-phase notifications.

The Group Services subsystem presents all notifications to all providers in a single group in the same order. The providers should try to invoke the same callback routines in the same order.

However, only for n-phase protocols does the Group Services subsystem verify that all of the group's providers have reached the same execution point before continuing to the next notification. In other cases, the providers may not receive and react to the notifications at the same time. For example, a provider might not receive a notification immediately because it is busy and not reading the socket.

If GS clients are providers in multiple groups, there is no guarantee that every provider will receive the notifications from different groups in the same order.

For multi-threaded clients, it is assumed that the callback routines are thread-safe and reentrant. If the same callback routines are specified for multiple groups, a multi-threaded client can process notifications by invoking the callback routines for more than one group at a time. For single-threaded providers, if they are acting as providers for multiple groups, they must also be coded to handle simultaneously executing protocols in all groups.

In all cases where GS clients are acting as providers in multiple groups, it is the responsibility of the providers to ensure that they do not create deadlock situations across groups. An example of a deadlock that could occur is when one provider blocks before voting, waiting for another provider to take some action; and the second provider is blocked on another group protocol, waiting for the first provider to take some action.

All of that said, the Group Services subsystem invokes callback routines only on the same thread (or threads) that are used to call the **ha_gs_dispatch** subroutine.

Return values

None.

Error values

None.

Asynchronous errors

None.

Files

ha_gs.h

Related concepts:

"Enabling the collection of voting results" on page 34
After a proposal is approved or rejected, the Group Services subsystem notifies all of the providers of the outcome.

Related reference:

"ha_gs_expel" on page 62

ha_gs_protocol_approved_callback

Purpose

ha_gs_protocol_approved_callback – A callback routine that the Group Services subsystem calls to deliver a protocol approved notification to a GS client

Library

See "GSAPI library names" on page 46.

Syntax

```
#include <ha_gs.h>

void
    ha_gs_protocol_approved_callback(
        const   ha_gs_protocol_approved_notification_t      *notification)
```

Parameters

notification
: A pointer to a protocol approved notification block.

Description

The **ha_gs_protocol_approved_callback** subroutine defines a GS client's protocol approved callback routine. The GS client uses it to handle protocol approved notifications from the Group Services subsystem. The process provides the address of the protocol approved callback routine to the Group Services subsystem on the **ha_gs_join** subroutine when it joins the group as a provider.

The Group Services subsystem then calls the protocol approved callback routine when it has a protocol approved notification to deliver to the GS client. For an n-phase protocol, this notification is delivered after the protocol has been approved by voting. All one-phase protocols are automatically approved.

On input, the protocol approved callback routine receives information that specifies the changes that have been made to the group, such as its membership or its state value, as well as other control information for the protocol, such as the number of phases that were run.

On input, the protocol approved callback routine receives a pointer to the protocol approved notification block. The protocol approved notification block has the following definition:

```
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;
```

The **gs_notification_type** field contains the type of notification. For a protocol approved notification, it contains a value of **HA_GS_APPROVED_NOTIFICATION**.

The **gs_provider_token** field contains 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.

The **gs_protocol_type** field contains the type of request for which this protocol approved notification is being delivered.

HA_GS_JOIN
: One or more providers are attempting to join the group.

HA_GS_FAILURE_LEAVE
: One or more providers has failed and is leaving the group.

HA_GS_LEAVE
: A provider is voluntarily leaving the group.

HA_GS_EXPEL
: A provider is attempting to expel one or more providers from the group.

HA_GS_STATE_VALUE_CHANGE
: A provider is trying to change the group's state value.

HA_GS_PROVIDER_MESSAGE
: A provider is broadcasting a message to the group.

HA_GS_CAST_OUT
: One or more source-group providers have left the source-group, requiring these target-group providers to leave also. The source-group's state value is contained in the **gs_source_state_value** field of the proposal information block.

HA_GS_SOURCE_STATE_REFLECTION
: The source-group has modified its state value, and this is being reflected to the target-groups. The source-group's state value is contained in the **gs_source_state_value** field of the proposal information block.

HA_GS_GROUP_ATTRIBUTE_CHANGE
: A provider has requested to change the group's attributes via the **ha_gs_change_attributes** interface.

The **gs_summary_code** field contains one or more flags that indicate whether any default votes were recorded during any previous voting phase. It can contain one or more of the following flags:

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_DEFAULT_APPROVE
: This flag is set for a protocol approved notification 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 flags are also set.

HA_GS_TIME_LIMIT_EXCEEDED
: This flag is set when a default approval 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 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_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.

The **gs_proposal** field points to the proposal block for the proposal on which the vote is requested. For information about this block, see the **ha_gs_n_phase_callback** man page.

Restrictions

For important information about multiprocessing considerations that apply to all callback routines, see the **ha_gs_n_phase_callback** man page.

Return values

None.

Error values

None.

Synchronous errors

None.

Asynchronous errors

None.

Files

ha_gs.h

Related reference:

"ha_gs_change_state_value" on page 54
"ha_gs_change_attributes" on page 50

ha_gs_protocol_rejected_callback

Purpose

ha_gs_protocol_rejected_callback – A callback routine that the Group Services subsystem calls to deliver a protocol rejected notification to a GS client

Library

See "GSAPI library names" on page 46.

Syntax

```
#include <ha_gs.h>

void
    ha_gs_protocol_rejected_callback(
        const   ha_gs_rejected_notification_t      *notification)
```

Parameters

notification
A pointer to a protocol rejected notification block.

Description

The **ha_gs_protocol_rejected_callback** subroutine defines a GS client's protocol rejected callback routine. The GS client uses it to handle protocol rejected notifications from the Group Services subsystem. The process provides the address of the protocol rejected callback routine to the Group Services subsystem on the **ha_gs_join** subroutine when it joins the group as a provider. The Group Services subsystem then calls the protocol rejected callback routine when it has a protocol rejected notification to deliver to the GS client. This occurs after a n-phase protocol has been rejected by voting. One-phase protocols cannot be rejected; they are all automatically approved.

On input, the protocol rejected callback routine receives information that specifies the proposed changes to the group, such as its membership or its state value, as well as other control information for the protocol, such as the reason for the rejection.

A protocol can be rejected for several reasons, which include:
- At least one of the providers explicitly voted to reject it
- A default reject vote was recorded for a failing provider
- A default reject vote was recorded because a provider failed to vote within the specified time limit.

When a protocol is rejected because a provider failed, the Group Services subsystem initiates a separate failure leave protocol to allow the group to handle the failure. The failure leave protocol specifies the list of failing providers.

When a protocol is rejected because votes were not submitted within the voting time limit, the Group Services subsystem delivers an announcement notification that lists the tardy providers.

On input, the protocol rejected callback routine receives a pointer to the protocol rejected notification block. The protocol rejected notification block has the following definition:

```
typedef struct  {
    ha_gs_notification_type_t   ha_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;
```

The **gs_notification_type** field contains the type of notification. For a protocol rejected notification, it contains a value of **HA_GS_REJECTED_NOTIFICATION**.

The **gs_provider_token** field contains 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.

The **gs_protocol_type** field contains the type of request for which this protocol rejected notification is being delivered.

HA_GS_JOIN
One or more providers is attempting to join the group.

HA_GS_FAILURE_LEAVE
One or more providers has failed and is leaving the group.

HA_GS_LEAVE
A provider is voluntarily leaving the group.

HA_GS_EXPEL
A provider is attempting to expel one or more providers from the group.

HA_GS_STATE_VALUE_CHANGE
A provider is trying to change the group's state value.

HA_GS_PROVIDER_MESSAGE
A provider is broadcasting a message to the group.

HA_GS_CAST_OUT
One or more source-group providers have left the source-group, requiring these target-group providers to leave also. The source-group's state value is contained in the **gs_source_state_value** field of the proposal information block.

HA_GS_SOURCE_STATE_REFLECTION
The source-group has modified its state value, and this is being reflected to the target-groups. The source-group's state value is contained in the **gs_source_state_value** field of the proposal information block.

HA_GS_GROUP_ATTRIBUTE_CHANGE
A provider has requested to change the group's attributes via the **ha_gs_change_attributes** interface.

The **gs_summary_code** field contains one or more flags that indicate whether any default votes were recorded during any previous voting phase. It can contain one or more of the following flags:

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_REJECT
This flag is set for a protocol rejected notification 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 flags are also set.

HA_GS_TIME_LIMIT_EXCEEDED
This flag is set when 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 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_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.

The **gs_proposal** field points to the proposal block for the proposal on which the vote is requested. For information about this block, see the **ha_gs_n_phase_callback** man page.

Restrictions

For important information about multiprocessing considerations that apply to all callback routines, see the **ha_gs_n_phase_callback** man page.

Return values

None.

Error values

None.

Synchronous errors

None.

Asynchronous errors

None.

Files

ha_gs.h

Related reference:

"ha_gs_init" on page 80
"ha_gs_join" on page 85

ha_gs_quit

Purpose

ha_gs_quit – Called by a GS client to terminate its connection to the Group Services subsystem

Library

See "GSAPI library names" on page 46.

Syntax

```
#include <ha_gs.h>

void
    ha_gs_quit(void)
```

Parameters

None.

Description

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

If the GS client is 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 invoke a failure leave protocol. If the GS client wants to leave a group without terminating its connection, it should use the **ha_gs_leave** subroutine or the **ha_gs_goodbye** subroutine.

See the discussion of multi-threaded GS clients under .

Restrictions

The calling process must be a GS client.

Return values

None.

Error values

None.

Synchronous errors

None.

Asynchronous errors

None.

Files

ha_gs.h

Related reference:

"ha_gs_join" on page 85

"ha_gs_dispatch" on page 59

ha_gs_responsiveness_callback

Purpose

ha_gs_responsiveness_callback – A callback routine that the Group Services subsystem calls to deliver a responsiveness notification to a GS client.

Library

See "GSAPI library names" on page 46.

Syntax

```
#include <ha_gs.h>

ha_gs_callback_rc_t
    ha_gs_responsiveness_callback(
        const   ha_gs_responsiveness_notification_t     *notification)
```

Parameters

notification
: A pointer to a responsiveness notification block.

Description

The **ha_gs_responsiveness_callback** subroutine defines a GS client's responsiveness callback routine. The GS client uses it to handle responsiveness notifications from the Group Services subsystem. The process provides the address of the responsiveness callback routine to the Group Services subsystem on the **ha_gs_init** subroutine during GSAPI initialization. The Group Services subsystem then calls the responsiveness callback routine when it has a responsiveness notification to deliver to the GS client.

The responsiveness callback routine is called at intervals. Therefore, in addition to responding to the Group Services subsystem, the GS client can also use the routine to perform validity checks on its own operation or its environment.

On input, the responsiveness callback routine receives a pointer to the responsiveness notification block. The responsiveness notification block has the following definition:

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

The **gs_notification_type** field contains the type of notification. For a responsiveness notification, it contains a value of **HA_GS_RESPONSIVENESS_NOTIFICATION**.

The **gs_responsiveness_information** field contains the responsiveness control structure that was specified on input to the **ha_gs_init** subroutine when this process initialized itself with the Group Services subsystem. This structure specifies the method, if any, to be used to perform responsiveness checks for this process, and how frequently the checks should be performed. For details on the responsiveness control structure, see **ha_gs_init**.

Restrictions

For important information about multiprocessing considerations that apply to all callback routines, see the **ha_gs_n_phase_callback** man page.

Return values

On output, the Group Services subsystem expects the responsiveness callback routine to return a code that indicates whether the GS client is operational. If it is, the **ha_gs_responsiveness_callback** subroutine should return a value of **HA_GS_CALLBACK_OK**.

Error values

If the GS client has detected an internal problem that prevents its correct operation, the **ha_gs_responsiveness_callback** subroutine should return a value of **HA_GS_CALLBACK_NOT_OK**. In response, the Group Services subsystem considers the process to be nonresponsive and sends an announcement notification to the group's providers that lists the nonresponsive providers.

Synchronous errors

None.

Asynchronous errors

None.

Files

ha_gs.h

Related reference:

"ha_gs_expel" on page 62

ha_gs_send_message

Purpose

ha_gs_send_message – Called by a provider of a group to broadcast message data to all of the providers in the group

Library

See "GSAPI library names" on page 46.

Syntax

```
#include <ha_gs.h>

ha_gs_rc_t
    ha_gs_send_message(
                ha_gs_token_t           provider_token,
        const   ha_gs_proposal_info_t   *proposal_info)
```

Parameters

provider_token
: 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.

proposal_info
: A pointer to a buffer that contains a proposal information block, which describes the proposed provider-broadcast message request.

Description

The **ha_gs_send_message** subroutine is used by a provider of a Group Services group to broadcast message data to all of the providers of the group.

If the request is specified as a one-phase protocol, and Group Services chooses to run this protocol, the group's providers are notified using normal protocol approval procedures.

If the request is specified as an n-phase protocol, and Group Services chooses to run this protocol, the group's providers are notified using normal n-phase voting procedures.

If the Group Services subsystem chooses not to run this protocol (because another protocol is already in progress), the **HA_GS_COLLIDE** error number is returned either synchronously or asynchronously, depending on when the error is detected. Asynchronous errors are delivered through the delayed error callback routine. Otherwise, the proposal will initiate a protocol within the group.

Information about the provider-broadcast message request is supplied through the provider-broadcast message request block, which is a type of proposal information block. On the **ha_gs_send_message** subroutine, specify the proposal information block as a provider-broadcast message request block. For the definition of the proposal information block, see the **ha_gs_delayed_error_callback** man page.

The provider-broadcast message request block has the following definition:

```
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;
```

The **gs_num_phases** field specifies whether the provider-broadcast message protocols are to be n-phase protocols or one-phase protocols. It can take one of the following values:

HA_GS_1_PHASE
The protocols are to be one-phase protocols. One-phase protocols are automatically approved.

HA_GS_N_PHASE
The protocols are to be n-phase protocols, which put the group into multi-phase voting.

The **gs_time_limit** field contains the voting phase time limit, in seconds. This is the number of seconds within which each provider must register its vote for each phase of an n-phase protocol. If the field is set to a value of 0, no limit is enforced.

The **gs_message** field points to a buffer that contains the message that is to be broadcast to providers by a message-with-voting proposal. The provider-broadcast message has the following definition:

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

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

The **gs_message** field points to a buffer that contains the message. Provider-broadcast messages are defined by the application that is using the GSAPI 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 subsystem.

Restrictions

The calling process must be a provider.

Return values

If the **ha_gs_send_message** subroutine is successful, it returns a value of 0 (**HA_GS_OK**). Group Services has accepted the request and will asynchronously attempt to run the proposed protocol.

Error values

If the **ha_gs_send_message** subroutine is unsuccessful, it returns an error number. If the error is detected immediately, an error is returned synchronously. If the error is detected after the call has been accepted, an error is returned asynchronously.

The GSAPI error numbers are defined in the **ha_gs.h** header file. For more information on GSAPI errors, see "GSAPI return codes" on page 126.

Synchronous errors

The following errors may be returned synchronously by the **ha_gs_send_message** subroutine:

HA_GS_BAD_MEMBER_TOKEN
The given **provider_token** does not specify a valid provider joined to a group.

HA_GS_BAD_PARAMETER
The number of phases specified for the protocol is not allowable; it must be **HA_GS_1_PHASE** or **HA_GS_N_PHASE**.

HA_GS_NO_INIT
The GS client has not yet successfully initialized itself with Group Services by calling **ha_gs_init**.

HA_GS_COLLIDE
The provider's group is already running a protocol; or this provider has already submitted a protocol request.

HA_GS_NOT_A_MEMBER
The given **provider_token** does not specify a valid group.

HA_GS_NOT_OK
The connection to the GS daemon has been lost. The GS client needs to reinitialize (via **ha_gs_init**).

Asynchronous errors

The following errors may be returned asynchronously by the **ha_gs_send_message** subroutine:

HA_GS_NOT_A_MEMBER
The provider that is proposing the protocol is no longer a provider for the specified group.

HA_GS_BAD_PARAMETER
The specified parameter was not valid.

HA_GS_COLLIDE
Another protocol is already active for this group.

Files

ha_gs.h

Related reference:

"ha_gs_announcement_callback" on page 48
"ha_gs_change_attributes" on page 50
"ha_gs_init" on page 80
"ha_gs_expel" on page 62
"ha_gs_join" on page 85
"ha_gs_dispatch" on page 59
"ha_gs_vote" on page 123
"ha_gs.h header file" on page 125
ha_gs.h is a header file that provides datatypes and structures for use with the GSAPI subroutines.

ha_gs_subscribe

Purpose

ha_gs_subscribe – Called by a GS client to join a group as a subscriber.

Library

See "GSAPI library names" on page 46.

Syntax

```
#include <ha_gs.h>

ha_gs_rc_t
    ha_gs_subscribe(
                ha_gs_token_t           *subscriber_token,
        const   ha_gs_proposal_info_t   *proposal_info)
```

Parameters

subscriber_token
: A pointer to a token that will be returned by the Group Services subsystem that is used to identify the membership of the GS client in the group as a subscriber. The GS client must pass a copy of the token on all subsequent GSAPI calls that refer to the group. The GSAPI passes a copy of the token to the GS client on all subsequent callbacks that refer to the group.

proposal_info
: A pointer to a buffer that contains a proposal information block, which describes the proposed subscribe request.

Description

The **ha_gs_subscribe** subroutine is used by a GS client to register as a subscriber for a Group Services group. If the named group does not already exist, the **HA_GS_UNKNOWN_GROUP** error number is returned asynchronously by the delayed error callback routine.

Note that subscribers are known only to the Group Services subsystem. The providers of the group and the other subscribers of the group are unaware of any of the subscribers to the group.

In addition to groups defined by providers, a GS client can subscribe to a number of system-defined groups. These provide status information about nodes and various communications adapters. A listing of these groups appears later in this section. For additional information about these concepts, refer to "Host and adapter membership groups" on page 29.

Information about the subscribe request is supplied through the subscribe request block, which is a type of proposal information block. On the **ha_gs_subscribe** subroutine, specify the proposal information block as a subscribe request block. For the definition of the proposal information block, see the **ha_gs_delayed_error_callback** man page.

The subscribe request block has the following definition:

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

The **gs_subscription_control** field contains 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's state, its membership list, or both.

The subscription control block has the following definition:

```
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_SUBSCRIBE_ADAPTER_INFO = 0x10,
HA_GS_SUBSCRIBE_SPECIAL_DATA = 0x40
HA_GS_SUBSCRIBE_PERSISTENCE = 0x200
} ha_gs_subscription_ctrl_t;
```

The flags are not exclusive and may be specified in any combination by OR'ing the individual flags together. The flags that may be specified are:

HA_GS_SUBSCRIBE_STATE
: The subscriber wants to receive the group's state value whenever the state value is updated.

HA_GS_SUBSCRIBE_DELTA_JOINS
: The subscriber wants to receive the set of providers that are joining the group, whenever a join occurs.

HA_GS_SUBSCRIBE_DELTA_LEAVES
: 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
: The subscriber wants to receive both the set of providers that are joining the group, whenever a join occurs, and the set of providers that are leaving the group, whenever a leave occurs.

HA_GS_SUBSCRIBE_MEMBERSHIP
: The subscriber wants to receive a full list of providers in the group, whenever a membership change (either 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
: The subscriber wants to receive on all subscription notifications that contain membership information both the full set of providers in the group and the delta list of joining or leaving providers.

HA_GS_SUBSCRIBE_STATE_AND_MEMBERSHIP
: The subscriber wants to receive all of the information described by the other flags.

HA_GS_SUBSCRIBE_ADAPTER_INFO
: When this flag is set on, the socket control flag **HA_GS_ENABLE_ADAPTER_INFO** will be implicitly enabled for the adapter group.

HA_GS_SUBSCRIBE_SPECIAL_DATA
: If this flag is set on when a client calls **ha_gs_subscribe** to subscribe an adapter group, such as **HA_GS_ENET_MEMBERSHIP_GROUP**, Group Services will enable "special data" for the adapter group. See “Receiving Group Services subscription special data” on page 42.

  The relationship between the subscription control flags **HA_GS_SUBSCRIBE_SPECIAL_DATA**, **HA_GS_SUBSCRIBE_ADAPTER_INFO**, and the socket control flag **HA_GS_ENABLE_ADAPTER_INFO** has these characteristics:

  - The **HA_GS_SUBSCRIBE_SPECIAL_DATA** flag is automatically set on if **HA_GS_SUBSCRIBE_ADAPTER_INFO** is set on.
  - The **HA_GS_SUBSCRIBE_ADAPTER_INFO** flag is always set on for the "all adapter" group, **HA_GS_ALL_ADAPTER_MEMBERSHIP_GROUP**.

- It is not necessary to turn on the flag **HA_GS_ENABLE_ADAPTER_INFO** if the **HA_GS_SUBSCRIBE_SPECIAL_DATA** flag is set on.
- The **HA_GS_ENABLE_ADAPTER_INFO** flag is automatically set on if **HA_GS_SUBSCRIBE_ADAPTER_INFO** is set on.

HA_GS_SUBSCRIBE_PERSISTENCE

If this flag is set on, the GS client is registered as a persistent subscriber. The difference between a persistent subscription and a regular subscription is that:

- if the group that the GS client has subscribed to is dissolved, the subscription will not be dissolved. The Group Services subsystem will notify the GS client, through the subscription notification callback, that the group has 0 members.
- If the group that the GS client is subscribing to does not exist, the Group Services subsystem will not send a delayed error callback to notify the client. Instead, the subscription will be valid. The GS client will receive a subscription notification that the group has 0 members.

Note: The Group Services subsystem can enable a persistent subscription only if all nodes in the domain have the version of Group Services provided with version 2.3.1.0 (or later) of RSCT. In any of the nodes in the domain have a back-level version of Group Services, persistent subscription cannot be enabled.

One exception is adapter groups. Since these subscriptions are local, the persistent subscription will be enabled on nodes running the new version of RSCT. The persistent subscription will not be enabled on nodes that have the older release installed.

The **gs_subscription_group** field points to a string that contains the name of the group to which the caller wishes to subscribe.

The **gs_subscription_callback** field points to the callback routine that is to be called when Group Services has a notification to deliver that contains data that satisfies this subscription request. A pointer to a valid function must be given in the **gs_subscription_callback** field. If a NULL pointer is given, a synchronous error of **HA_GS_BAD_PARAMETER** will be returned on the call to **ha_gs_subscribe**. If the pointer is not NULL but does not point to an executable function, then unpredictable results, such as fatal program failure, may occur during execution of your program.

A process can also subscribe for host or adapter membership information as follows.

These groups are specified for subscription by using the following constants:

- **HA_GS_ALL_ADAPTER_MEMBERSHIP_GROUP** constant, for subscription for all adapter membership information.
- The **HA_GS_BOND_MEMBERSHIP_GROUP** constant, for subscriptions for channel bonding adapter information. The channel bonding interface names must follow the convention **bond***n* or **BOND***n*, where *n* is an integer (for example *bond1* or *BOND1*).
- The **HA_GS_EIP_MEMBERSHIP_GROUP** constant, for subscriptions for Ethernet Interface Processor (EIP) adapter information. The EIP interface names must follow the convention **eip***n*, where *n* is an integer (for example *eip1*).
- The **HA_GS_FC_MEMBERSHIP_GROUP** constant, for subscriptions for Fiber Channel adapter information. The Fiber Channel interface names must follow the convention **fc***n*, where *n* is an integer (for example *fc1*).
- The **HA_GS_HOST_MEMBERSHIP_GROUP** constant, for subscriptions for host membership information.
- The **HA_GS_ENET_MEMBERSHIP_GROUP** constant, for subscriptions for Ethernet adapter information.
- The **HA_GS_CSS0_MEMBERSHIP_GROUP** constant, for subscriptions for globally-consistent SP Switch adapter information for **css0** devices.

HA_GS_CSS0_MEMBERSHIP_GROUP is aliased to **HA_GS_CSS_MEMBERSHIP_GROUP** for compatibility.

- The **HA_GS_CSS1_MEMBERSHIP_GROUP** constant, for subscriptions for globally-consistent SP Switch adapter information for **css1** devices.
- The **HA_GS_CSSRAW_MEMBERSHIP_GROUP** constant, for subscriptions for non–globally-consistent SP Switch adapter information.
- The **HA_GS_ML0_MEMBERSHIP_GROUP** constant, for subscriptions for globally-consistent aggregate SP Switch adapter information between **HA_GS_CSS0_MEMBERSHIP_GROUP** and **HA_GS_CSS1_MEMBERSHIP_GROUP**.
- If a GS client is running in the Group Services HACMP/ES domain (see 'ha_gs_init Subroutine' for more information), the following system-defined groups may also be available in addition to the above:
  - The **HA_GS_TOKENRING_MEMBERSHIP_GROUP** constant, for subscriptions for token-ring adapter information
  - The **HA_GS_ATM_MEMBERSHIP_GROUP** constant, for subscriptions for ATM adapter information
  - The **HA_GS_FDDI_MEMBERSHIP_GROUP** constant, for subscriptions for fddi adapter information
  - The **HA_GS_RS232_MEMBERSHIP_GROUP** constant, for subscriptions for **rs232** heartbeating adapter information
  - The **HA_GS_TMSCSI_MEMBERSHIP_GROUP** constant, for subscriptions for target-mode SCSI heartbeating adapter information

If a GS client is running on an SP system that does not have an SP Switch, or if the SP Switch is not currently active on any nodes, a request to subscribe to **HA_GS_CSS0_MEMBERSHIP_GROUP** or **HA_GS_CSS1_MEMBERSHIP_GROUP** will result in an asynchronous **HA_GS_UNKNOWN_GROUP** delayed error.

If the node on which a GS client is running does not have an active SP Switch adapter, a request to subscribe to **HA_GS_CSSRAW_MEMBERSHIP_GROUP** will result in an asynchronous **HA_GS_UNKNOWN_GROUP** delayed error. It is also possible to receive subscription “special” data on notifications for this group. (See “Receiving Group Services subscription special data” on page 42.)

If a GS client is running in a Group Services PSSP domain and tries to subscribe to one of the Group Services HACMP/ES domain system-defined groups, it will result in an asynchronous **HA_GS_UNKNOWN_GROUP** delayed error.

The availability of information from the adapter membership groups supported a Group Services HACMP/ES domain depends upon the set of networks defined to HACMP/ES for heartbeating. You can specify any of the above-mentioned groups; however, the GS client will receive an asynchronous **HA_GS_UNKNOWN_GROUP** delayed error in the following cases:

1. The requested adapter type is not installed on any of the nodes in this HACMP/ES cluster.
2. The requested adapter type is installed on one or more of the nodes in this HACMP/ES cluster, but the adapter type is not specified to HACMP/ES for use in heartbeating.
3. The requested adapter type is installed on one or more of the nodes in this HACMP/ES cluster and the adapter type is specified to HACMP/ES for use in heartbeating, but one of the following two cases exist:
   a. None of the adapters of that type on any nodes in the domain are currently active.
   b. None of the adapters of that type are active on this node.

In any case where the GS client receives an asynchronous delayed error on a subscription request, it may want to wait a period of time and reissue the request, as adapters may have become active in the interim.

For all subscriptions to system-defined groups, all controls and notifications act very much like those for subscriptions to user-defined groups. The subscription control flags in the **gs_subscription_control** field may be used to control the level of information received by the subscriber for the host or adapter membership group notifications.

Notifications for host or adapter membership look the same as notifications for any other group; each active node or adapter is represented as a provider. The instance number for each node "provider" is a value of 0 and the node number for each node "provider" is its node number. The instance number for each adapter "provider" is its adapter interface number (for example, a value of 2 for en2) and the node number for each adapter "provider" is the node on which it is installed.

Nodes and adapters are ordered from "oldest" to "youngest," The oldest node or adapter (that is, the first node or adapter to join its membership group) is at the head of the list and the youngest is at the end.

Restrictions

The calling process must be a GS client.

Return values

If the **ha_gs_subscribe** subroutine is successful, it returns a value of 0 (**HA_GS_OK**) and the **subscriber_token** field is set to the token that identifies this subscriber's connection to the group.

Note that subscriber tokens are assigned, invalidated, and reassigned in a manner similar to the way in which file descriptors are assigned, invalidated, and reassigned as files are opened and closed.

Here is an example.

A GS client subscribes to group **bar** and receives subscriber token 2. When the same client unsubscribes from group **bar** or becomes unsubscribed because group **bar** is dissolved, Group Services invalidates subscriber token 2 and makes it available for reassignment. When the next GS client (it could be the same or a different GS client) subscribes to the next group (it could be the same or a different group), Group Services assigns subscriber token 2 to that client.

Error values

If the **ha_gs_subscribe** subroutine is unsuccessful, it returns an error number and the contents of the **subscriber_token** field are undefined. If the error is detected immediately, an error is returned synchronously. If the error is detected after the call has been accepted, an error is returned asynchronously.

The GSAPI error numbers are defined in the **ha_gs.h** header file. For more information on GSAPI errors, see “GSAPI return codes” on page 126.

Synchronous errors

The following errors may be returned synchronously by the **ha_gs_subscribe** subroutine:

HA_GS_BAD_PARAMETER
: The subscription control flags contain invalid values; or the group name is not specified or is zero length.

HA_GS_NAME_TOO_LONG
: The group name specified is too long.

HA_GS_NO_INIT
: The GS client has not yet successfully initialized itself with Group Services by calling **ha_gs_init**.

HA_GS_NO_MEMORY
The GS library is unable to allocate memory. The GS client should retry the request.

HA_GS_NOT_OK
The connection to the GS daemon has been lost. The GS client needs to reinitialize (via **ha_gs_init**).

HA_GS_NOT_SUPPORTED
Special data is not supported for the particular adapter group.

Asynchronous errors

The following errors may be returned asynchronously by the **ha_gs_subscribe** subroutine:

HA_GS_BAD_PARAMETER
The specified parameter was not valid.

HA_GS_UNKNOWN_GROUP
The group that was specified on the call to the **ha_gs_subscribe** subroutine does not exist.

Files

ha_gs.h

Related reference:
"ha_gs_change_attributes" on page 50
"ha_gs_change_state_value" on page 54
"ha_gs_get_ipaddr_by_id" on page 72
"ha_gs_goodbye" on page 78
"ha_gs_unsubscribe" on page 122
"ha_gs_subscriber_callback"

ha_gs_subscriber_callback

Purpose

ha_gs_subscriber_callback – A callback routine that the Group Services subsystem calls to deliver a subscription notification to a GS client.

Library

See "GSAPI library names" on page 46.

Syntax

```
#include <ha_gs.h>

void
    ha_gs_subscriber_callback(
        const   ha_gs_subscription_notification_t *notification)
```

Parameters

notification
A pointer to a subscription notification block.

Description

The **ha_gs_subscriber_callback** subroutine defines a Group Services client's subscriber callback routine. The Group Services client uses it to handle subscription notifications from the Group Services subsystem.

The process provides the address of the subscriber callback routine to the Group Services subsystem on the **ha_gs_subscribe** subroutine when it joins the group as a subscriber. The Group Services subsystem then calls the subscriber callback routine when it has a subscription notification to deliver to the Group Services client. A subscription notification is delivered when a protocol is approved in a group to which the process is subscribed and the protocol modifies the group's membership or state value.

If the **gs_subscription_type** field has a value of **HA_GS_SUBSCRIPTION_DISSOLVED**, the subscriber's connection to the Group Services subsystem will be closed as soon as the subscriber callback routine returns.

On input, the subscriber callback routine receives a pointer to the subscription notification block. The subscription notification block has the following definition:

```
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_adapter_ip_membership_t      *gs_full_ip_membership;
                ha_gs_adapter_ip_membership_t      *gs_changing_ip_membership;
                ha_gs_adapter_death_t              *gs_reason_flags;
} ha_gs_subscription_notification_t;
```

where **ha_gs_adapter_ip_membership_t** is defined as:

```
typedef struct {
                unsigned int                        gs_count;
                ha_gs_ip_addr                       *gs_ip_members;
} ha_gs_adapter_ip_membership_t;
```

For information about the **ha_gs_ip_addr** data structure, see “IP addressing” on page 33

The **gs_notification_type** field contains the type of notification. For a subscription notification, it contains a value of **HA_GS_SUBSCRIPTION_NOTIFICATION**.

The **gs_subscriber_token** field contains a token that identifies the caller as a subscriber of the group. This token was previously initialized when the subscriber joined the group using the **ha_gs_subscribe** subroutine. If the **gs_subscription_type** field has a value of **HA_GS_SUBSCRIPTION_DISSOLVED**, this token no longer specifies a valid subscription.

The **gs_subscription_type** field contains one or more flags that indicate the type of change for which this subscription notification is being delivered. It can contain one or more of the following flags:

HA_GS_SUBSCRIPTION_STATE
: The notification contains the updated group state value. This flag may appear with any of the other flags.

HA_GS_SUBSCRIPTION_DELTA_JOIN
: 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
: 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
The notification contains the complete updated membership list. This flag may appear with either the **HA_GS_SUBSCRIPTION_DELTA_JOIN** and **HA_GS_SUBSCRIPTION_DELTA_LEAVE** flags.

HA_GS_SUBSCRIPTION_DISSOLVED
The group that was subscribed to has dissolved; all providers have left the group. This flag may appear with any of the other flags.

The subscription is deactivated. To start receiving notifications again, the subscriber must subscribe to the group again. 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**.

HA_GS_SUBSCRIPTION_GS_HAS_DIED
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.

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 reinitialize the connection to Group Services by calling the **ha_gs_init** subroutine, and then take any other necessary actions to re-subscribe to the group.

HA_GS_SUBSCRIPTION_PERSISTENT
The subscription is persistent. In other words, the subscription will not be automatically dissolved if the group is dissolved. Also, if the GS client has subscribed to a group that does not exist, the subscription will still be valid.

If the **HA_GS_SUBSCRIPTION_STATE** flag is set in the **gs_subscription_type** field, the **gs_state_value** field points to a buffer that contains the new value for the group's state. For the definition of the group state value, see the **ha_gs_change_state_value** man page.

If the **HA_GS_SUBSCRIBE_MEMBERSHIP** flag is set in the **gs_subscription_type** field, the **gs_full_membership** field points to the full updated list of providers that currently belong to the group. It has the following definition:

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

The **gs_count** field contains the number of providers in the list.

The **gs_providers** field points to the list of providers. Each provider is described by a provider information block, which is defined in the **ha_gs_n_phase_callback** man page.

If the **HA_GS_SUBSCRIBE_DELTA_JOIN** or **HA_GS_SUBSCRIBE_DELTA_LEAVE** flag is set in the **gs_subscription_type** field, the **gs_changing_membership** field points to the list of changing (either joining or leaving) providers rather than the full membership list. The membership list has the following definition:

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

The **gs_count** field contains the number of providers in the list.

The **gs_providers** field points to the list of providers. Each provider is described by a provider information block, which is defined in the **ha_gs_n_phase_callback** man page.

The **gs_subscription_special_data** field contains the special group-specific subscription data; it is valued only if **gs_subscription_type** has **HA_GS_SUBSCRIPTION_SPECIAL_DATA** set on.

The **gs_full_ip_membership** field is a list of adapter IP addresses in the order matching the order of the adapters in the **gs_full_membership** list. The field **gs_changing_ip_membership** is a list of adapter IP addresses in the order matching the order of the adapters in the **gs_changing_membership** list. These two entries will only be valid for the subscription of an adapter group. For other subscribed groups, these two entries will be assigned NULL.

The **gs_reason_flags** is a list of adapter death reasons in the order matching the order of the adapters in the **gs_changing_membership**. If an adapter died of natural causes, the flag will be **HA_GS_ADAPTER_DEAD**. If an adapter was removed from the configuration, the flag will be **HA_GS_ADAPTER_REMOVED**.

The adapter death block has the following definition:

```
typedef enum
{
    HA_GS_ADAPTER_DEAD  = 0x0001,
    HA_GS_ADAPTER_REMOVED = 0x0002
} ha_gs_adapter_death_t;
```

In order to be compatible with previous releases, for old programs, the new Group Services library will not touch the new entry fields of the **ha_gs_adapter_info** structure and **ha_gs_subscription_notification_t** structure. This means that Group Services will not deliver the new entry fields if the application is an old version. If it is the new release, group services will deliver the new entry fields to clients.

Also, these new entries will be filled in only if the subscription enables **HA_GS_SUBSCRIBE_SPECIAL_DATA**. Otherwise, they will be Null.

For more information, see "Receiving Group Services subscription special data" on page 42.

Restrictions

For important information about multiprocessing considerations that apply to all callback routines, see the **ha_gs_n_phase_callback** man page.

Return values

None.

Error values

None.

Synchronous errors

None.

Asynchronous errors

None.

Files

ha_gs.h

Related reference:

"ha_gs_subscribe" on page 112

ha_gs_unsubscribe

Purpose

ha_gs_unsubscribe – Called by a subscriber to unregister as a subscriber for a group

Library

See "GSAPI library names" on page 46.

Syntax

```
#include <ha_gs.h>

ha_gs_rc_t
   ha_gs_unsubscribe(
             ha_gs_token_t          subscriber_token)
```

Parameters

subscriber_token
: A token that identifies the subscription to be removed. This token was previously initialized when the subscriber requested the subscription using the **ha_gs_subscribe** subroutine.

Description

The **ha_gs_unsubscribe** subroutine is used by a subscriber to unregister as a subscriber for a group.

Note that subscribers are known only to the Group Services subsystem. The providers of the group and the other subscribers of the group are unaware of any of the subscribers to the group.

Restrictions

The calling process must be a subscriber.

Return values

If the **ha_gs_unsubscribe** subroutine is successful, it returns a value of 0 (**HA_GS_OK**).

Error values

If the **ha_gs_unsubscribe** subroutine is unsuccessful, it returns an error number.

The GSAPI error numbers are defined in the **ha_gs.h** header file. For more information on GSAPI errors, see "GSAPI return codes" on page 126.

Synchronous errors

The following errors may be returned synchronously by the **ha_gs_unsubscribe** subroutine:

HA_GS_BAD_MEMBER_TOKEN
: The given **subscriber_token** does not specify a valid subscription to a group.

HA_GS_NO_INIT
: The GS client has not yet successfully initialized itself with Group Services by calling **ha_gs_init**.

HA_GS_NOT_OK
: The connection to the GS daemon has been lost. The GS client needs to reinitialize (via **ha_gs_init**).

Asynchronous errors

None.

Files

ha_gs.h

Related reference:

"ha_gs_subscribe" on page 112

"ha_gs.h header file" on page 125

ha_gs.h is a header file that provides datatypes and structures for use with the GSAPI subroutines.

ha_gs_vote

Purpose

ha_gs_vote – Called by a provider of a group to submit its vote on a proposal during a voting phase of an executing protocol

Library

See "GSAPI library names" on page 46.

Syntax

```
#include <ha_gs.h>

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)
```

Parameters

provider_token
: 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
: A field that contains the value of the vote. It can take one of the following values:

  HA_GS_VOTE_APPROVE
  : Approve the proposal.

HA_GS_VOTE_CONTINUE
Neither approve nor reject the proposal right now, but propose another voting phase (continue to another voting phase).

HA_GS_VOTE_REJECT
Reject the proposal.

proposed_state_value
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 information on the definition of a group's state value, see the **ha_gs_change_state_value** man page.

provider_message
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 information on the definition of this message, see the **ha_gs_send_message** man page.

default_vote_value
The default vote value to be used by the group if any provider fails to vote during this phase. It can take one of the following values:

HA_GS_NULL_VOTE
A null vote. This value keeps the default vote at its previous value.

HA_GS_VOTE_APPROVE
Approve the proposal.

HA_GS_VOTE_REJECT
Reject the proposal.

Description

The **ha_gs_vote** subroutine is used by a provider of a Group Services group to submit its vote on a proposal during a voting phase of an executing protocol.

When an application has selected an n-phase protocol, providers will be expected to vote on proposed changes to the group. A change in either the group's state or its membership may be voted on. When a vote is requested, the appropriate callback routine is called for each of the providers, and each of the providers is expected to return a vote using this subroutine within the time limit previously established by the group.

This subroutine operates synchronously.

Voting results are tallied as follows:
- If all providers vote to APPROVE the proposal in the same voting phase, the protocol is approved.
- If any one provider votes to CONTINUE to another voting phase, the protocol proceeds to another voting phase.
- If any one provider votes to REJECT the proposal, the protocol is rejected and ends, regardless of any other votes.

Default votes are applied as if they were specified by the providers, except in the case of failure leave protocols.

If multiple providers, in the same voting phase, submit state value changes or provider messages, the Group Services subsystem chooses only one of each. Therefore, if different providers submit different values, it is arbitrary which values the Group Services subsystem will choose.

- If all providers submit the same values, then it does not matter which values are chosen.
- If only one provider submits values (that is, all of the other providers submit null values), then the Group Services subsystem chooses that provider's submission.
- If different providers can submit different values, they must be prepared to see other values chosen.

Restrictions

The calling process must be a provider. The group must be running an n-phase protocol.

Return values

If the **ha_gs_vote** subroutine is successful, it returns a value of 0 (**HA_GS_OK**).

Error values

If the **ha_gs_vote** subroutine is unsuccessful, it returns an error number synchronously.

The GSAPI error numbers are defined in the **ha_gs.h** header file. For more information on GSAPI errors, see "GSAPI return codes" on page 126.

Synchronous errors

The following errors may be returned synchronously by the **ha_gs_vote** subroutine:

HA_GS_BAD_MEMBER_TOKEN
The given **provider_token** does not specify a valid provider joined to a group.

HA_GS_NO_INIT
The GS client has not yet successfully initialized itself with Group Services by calling **ha_gs_init**.

HA_GS_BAD_PARAMETER
The given vote value is invalid (it must be one of **HA_GS_VOTE_APPROVE**, **HA_GS_VOTE_CONTINUE**, or **HA_GS_VOTE_REJECT**); or the given default vote value is invalid (it must be one of **HA_GS_NULL_VOTE**, **HA_GS_VOTE_APPROVE**, or **HA_GS_VOTE_REJECT**).

HA_GS_NOT_OK
The connection to the GS daemon has been lost. The GS client needs to reinitialize (via **ha_gs_init**).

HA_GS_VOTE_NOT_EXPECTED
The group to which this provider is joined is not currently running a protocol.

Asynchronous errors

None.

Files

ha_gs.h

Related reference:

"ha_gs_join" on page 85
"ha_gs_send_message" on page 110

ha_gs.h header file

ha_gs.h is a header file that provides datatypes and structures for use with the GSAPI subroutines.

Purpose

ha_gs.h – Header file for the group services application programming interface (GSAPI).

Description

The **ha_gs.h** header file provides data types and structures for use with the group services application programming interface (GSAPI) subroutines, which reside in the libraries listed in "GSAPI library names" on page 46. Any program that uses the GSAPI subroutines must include this file, which resides in the **/usr/include** directory.

Related reference:

"ha_gs_change_state_value" on page 54

"ha_gs_send_message" on page 110

"ha_gs_leave" on page 91

"ha_gs_expel" on page 62

"ha_gs_change_attributes" on page 50

"ha_gs_unsubscribe" on page 122

GSAPI return codes

The Group Services application programming interface (GSAPI) provides return codes that are returned synchronously or asynchronously. Synchronous information is returned immediately from a subroutine when it is unsuccessful. Asynchronous information is returned through the delayed error callback routine, when the error condition is detected at a later time.

The GSAPI error numbers and return codes are defined in the **ha_gs_rc_t** datatype of the **ha_gs.h** header file, as follows:

HA_GS_ADAPTER_INFO_NOT_FOUND

The given adapter is not found in the current adapter table.

HA_GS_ADAPTER_INFO_NOT_SENT

For internal use.

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 client interface library. This level does not support the new protocol request.

This error number 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 number is returned either synchronously or asynchronously, depending on when the error was detected.

HA_GS_BAD_MEMBER_TOKEN

The specified token does not represent a valid provider or subscriber instance for this client.

This error number is returned synchronously.

HA_GS_BAD_PARAMETER

The specified parameter was not valid.

This error number is returned either synchronously or asynchronously, depending on when the error was detected.

HA_GS_COLLIDE
Another protocol is already active for this group.

This error number can be returned either synchronously or asynchronously, depending on when the error was detected. This error number is returned in response to the protocol requests resulting from calls to the following subroutines: **ha_gs_change_state_value**, **ha_gs_change_attributes**, **ha_gs_expel**, **ha_gs_send_message**, and **ha_gs_leave**.

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 number is returned synchronously.

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 number is returned asynchronously through the delayed error callback routine.

HA_GS_EXISTS
The GSAPI has already been initialized by a previous call to the **ha_gs_init** subroutine.

This error number is returned synchronously.

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 number is returned synchronously.

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 number is 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 number is returned asynchronously through the delayed error callback routine.

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 number is 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 number 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 number 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 number is returned synchronously.

HA_GS_NO_IPV4_ADDRESS_FOR_THE_ID
There is no IPv4 address for this ID.

HA_GS_NO_MEMORY
The Group Services subsystem could not allocate required memory.

This error number is returned synchronously.

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 number is returned asynchronously through the delayed error callback routine.

HA_GS_NOT_A_MEMBER
The provider that is proposing the protocol is no longer a provider for the specified group.

This error number is returned asynchronously through the delayed error callback routine. It is returned in response to the protocol requests resulting from calls including the following subroutines: **ha_gs_change_state_value**, **ha_gs_send_message**, and **ha_gs_leave**.

HA_GS_NOT_OK
An error occurred.

This error number is returned synchronously.

HA_GS_NOT_SUPPORTED
The requested function is not currently supported.

This error number is returned synchronously.

HA_GS_NULL_ADAPTER_INFO

Group Services does not yet have the information about the network adapters.

HA_GS_OK
The subroutine was successful.

This return code is returned synchronously.

HA_GS_SOCK_CREATE_FAILED
The Group Services subsystem could not create a socket for communication.

This error number is returned synchronously.

HA_GS_SOCK_INIT_FAILED
The Group Services subsystem could not initialize the socket for communication.

This error number is returned synchronously.

HA_GS_UNKNOWN_GROUP
The group that was specified on the call to the **ha_gs_subscribe** subroutine does not exist.

This error number is returned asynchronously through the delayed error callback routine.

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 number can be returned either synchronously or asynchronously, depending on when the error was detected. This error number is returned in response to the protocol requests resulting from calls to the **ha_gs_expel** subroutine.

HA_GS_VOTE_NOT_EXPECTED
A vote was received that was not expected. Either no protocol was in progress, or the Group Services subsystem already received a vote for this protocol.

This error number is returned synchronously.

HA_GS_VOTE_VALUE_NOT_ALLOWED
For internal use.

HA_GS_WRITE_SOCK_ERROR
For internal use.

Group Services sample programs

Various Group Services sample programs are supplied.

The directory **/usr/sbin/rsct/samples/hags** contains three sample programs **— sample_schg.c**, **sample_test.c**, and **Sample_Subscribe.C** that illustrate how you can use the group services application programming interface (GSAPI) subroutines.

- **sample_schg.c** is a simple, non-interactive, program. It initializes itself as a Group Services client, joins the group as a provider, proposes a group state value change (at an interval the user had input to the program), and continues running until the program is killed.
- **sample_test.c** is a more complex, interactive, program that shows how an application can join groups on one or more nodes, subscribe to groups, propose various protocols, and respond with votes.
- **Sample_Subscribe.C** provides an example of a simple C++ language client. It illustrates how such a client can subscribe to a given group to get desired data.

The **/usr/sbin/rsct/samples/hags** directory also contains a Makefile (**Makefile.sample**) that you can use to build the sample programs.

Please note the, while the sample programs are intended to illustrate the various GSAPI subroutines, they are not intended to be an authoritative description of the best programming practices to employ when writing a Group Services application. Various aspects of the programs (in particular, their handling of screen input and output) are neither robust nor foolproof.

The sample_schg.c sample program

The **sample_schg.c** program is a simple GS Client.

This program performs the following functions:

- Reads command-line arguments to set various options, including the group name, protocol controls, and timing options
- Initializes itself as a Group Services client
- Joins the group as a provider
- Proposes a group state value change at the interval specified when the program was invoked
- Continues running until the program is killed (through the kill command or by otherwise sending it a signal).

For complete details on setting up the environment and starting the program, see the comments in the program listing.

The following table describes the files required by this sample program.

Table 2. Files required by the sample_schg.c sample program

File:	Description:
sample_schg.c	Contains the main() function, all of the necessary callback functions, and some of the utility functions for the application.
sample_utility.c	Provides the definitions for the utility functions used by the **sample_schg.c** program. (These definitions are also used by the **sample_test.c** program. Refer to "The sample_test.c sample program" on page 130 for more information.)
sample_utility.h	Header file containing the declarations for the utility functions contained in **sample_utility.c**.

The sample_test.c sample program

The **sample_test.c** program provides an interactive interface to the Group Services subsystem.

It provides the following functions:
- Reads command-line arguments and initializes itself as a Group Services client
- Enables user to interactively perform a number of tests. It enables the user to:
  - Display help data
  - Join a predefined group or interactively define and join a group
  - Propose a state value change protocol
  - Propose to have the provider leave a group voluntarily
  - Expel one or more providers from a group
  - Propose to change the attributes of a group
  - Leave a group immediately
  - Subscribe to a group
  - Unsubscribe from a group
  - Check for notifications
  - Close the socket connection to Group Services

For complete details on setting up the environment and starting the program, see the comments in the program listing.

The following table describes the files required by this sample program.

Table 3. Files required by the sample_test.c sample program

File:	Description:
sample_test.c	Supports interaction with the user, and most calls to the Group Services interfaces.
sample_callbacks.c	Provides the definitions for the callback functions used by the groups created by this program.
sample_callbacks.h	Header file containing the declarations for the callback functions contained in **sample_callbacks.c**.
sample_utility.c	Provides the definitions for the utility functions used by the **sample_test.c** program. (These definitions are also used by the **sample_schg.c** program. Refer to "The sample_schg.c sample program" on page 129 for more information.)
sample_utility.h	Header file containing the declarations for the utility functions contained in **sample_utility.c**.

The Sample_Subscribe.C sample program

The **Sample_Subscribe.C** program is a sample C++ language client.

It performs the following functions:
- Reads command-line arguments to set various options, including a group name
- Connect to Group Services
- Subscribe to the specified group
- Get the desired data and either wait for further data, or unsubscribe and exit.

For complete details on setting up the environment and starting the program, see the comments in the program listing.

The following table describes the files required by this sample program.

Table 4. Files required by the Sample_Subscribe.C sample program

File:	Description:
Sample_Subscribe.C	Contains the main() function for a sample C++ language client that subscribes with Group Services for a user-specified group. It takes as input a group name and some optional flags.
Sample_Subscribe.h	Header file containing the declarations for the functions defined in **Sample_Subscribe.C**.
Sample_Subscription.C	Represents a subscription to a group. It takes as input the targeted group name and other control information, sets up the subscription, and handles notifications for Group Services.
Sample_Subscription.h	Header file containing the declarations for the functions defined in **Sample_Subscription.C**.
Sample_ProviderTable.C	Builds the sorted list of providers in the system, sorted by node number.
Sample_ProviderTable.h	Header file containing the declarations for the functions defined in **Sample_ProviderTable.C**.
Sample_FrameTable.C	Builds a table of frame/node pairs for displaying the list of providers in a group.
Sample_FrameTable.h	Header file containing the declarations for the functions defined in **Sample_FrameTable.C**.
Sample_Node.C	Builds the sorted list of providers in the system, sorted by node number.
Sample_Node.h	Header file containing the declarations for the functions defined in **Sample_Node.C**.
Sample_Frame.C	Defines a frame. Contains a slot for each possible node that may be in the frame, the node in turn contains the providers.
Sample_Frame.h	Header file containing the declarations for the functions defined in Sample_Frame.C.

Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described 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:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte character set (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan, Ltd.
19-21, Nihonbashi-Hakozakicho, Chuo-ku
Tokyo 103-8510, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM websites are provided for convenience only and do not in any manner serve as an endorsement of those websites. The materials at those websites are not part of the materials for this IBM product and use of those websites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

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

© Copyright IBM Corp. 2012, 2014

one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
Dept. LRAS/Bldg. 903
11501 Burnet Road
Austin, TX 78758-3400
U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

All IBM prices shown are IBM's suggested retail prices, are current and are subject to change without notice. Dealer prices may vary.

This information is for planning purposes only. The information herein is subject to change before the products described become available.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

© (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs. © Copyright IBM Corp. _enter the year or years_.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Privacy policy considerations

IBM Software products, including software as a service solutions, ("Software Offerings") may use cookies or other technologies to collect product usage information, to help improve the end user experience, to tailor interactions with the end user or for other purposes. In many cases no personally identifiable information is collected by the Software Offerings. Some of our Software Offerings can help enable you to collect personally identifiable information. If this Software Offering uses cookies to collect personally identifiable information, specific information about this offering's use of cookies is set forth below.

This Software Offering does not use cookies or other technologies to collect personally identifiable information.

If the configurations deployed for this Software Offering provide you as the customer the ability to collect personally identifiable information from end users via cookies and other technologies, you should seek your own legal advice about any laws applicable to such data collection, including any requirements for notice and consent.

For more information about the use of various technologies, including cookies, for these purposes, see IBM's Privacy Policy at http://www.ibm.com/privacy and IBM's Online Privacy Statement at http://www.ibm.com/privacy/details the section entitled "Cookies, Web Beacons and Other Technologies" and the "IBM Software Products and Software-as-a-Service Privacy Statement" at http://www.ibm.com/software/info/product-privacy.

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the web at Copyright and trademark information at www.ibm.com/legal/copytrade.shtml.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Windows is a trademark of Microsoft Corporation in the United States, other countries, or both.

Index

Numerics

64–bit operating environment support 47

A

adapter alias 42
  multiple 45, 46
adapter death 42, 44
adapter deconfiguration 42
adapter information 47
adapter membership group
  subscription to Ethernet adapter membership group 29
  subscription to SP Switch adapter membership group 29
  system-defined group 29
addressing
  IP 33
announcement notification
  definition 19
application program design
  for a cluster environment 2
application programming interface
  GSAPI 2
application size
  minimizing for performance 32
applications
  in cluster environment 2
approval of a protocol
  notification 19
atomic multicast
  and voting 36
attribute change proposal
  definition 19
attributes of a group, changing 47
attributes, group
  changing 6
  defining 6
  definition 6
  mutability 6
audience 1

B

barrier synchronization
  and voting 36
  use in Group Services 3
batch control
  definition 6
batching requests
  using for performance 32

C

callback routine (Group Services)
  avoiding deadlock 31
  coordinating multiple notifications 31
  designing 31
  performance considerations 31
  responsiveness 7
callback subroutines
  ha_gs_announcement_callback 48
  ha_gs_delayed_error_callback 57
  ha_gs_n_phase_callback 94
  ha_gs_protocol_approved_callback 102
  ha_gs_protocol_rejected_callback 104
  ha_gs_responsiveness_callback 108
  ha_gs_subscriber_callback 118
change-attributes protocol
  introduction 9
changing group attributes 47
checking for notifications 47
client/server applications
  in a cluster environment 2
cluster environment
  design of applications for 2
  failure 2
  recovery 2
  role of high availability 2
COLLIDE error code
  for competing protocols 39
collision error code
  for competing protocols 39
concepts
  Group Services 2
consistency
  loosely synchronous 20
  strong 20
coordination
  in multinode environment 3
  of applications using GSAPI 2, 3
  of processes using GSAPI 2
counter-like responsiveness protocol
  definition 7

D

deactivate script 14
deactivate scripts 35
Deactivate-on-failure 16
deadlock
  avoiding in callback routines 31
default vote
  definition 6
  when used 124
design considerations
  actions during voting 25
  coding callback routines 31
  for a cluster environment 2
  Group Services 31
  migration and coexistence 4
  multi-threaded applications 31
domain, Group Services
  definition 5

E

Ethernet adapter membership group
  system-defined group 29

© Copyright IBM Corp. 2012, 2014

expel protocol
    deactivate script 14
    description 14
expelling providers 47
expulsion
    for a provider 36

F

failure
    causes in a cluster 2
    detection using GSAPI 2
    quick recovery 32
    undoing actions 32
failure leave
    definition 18
    for a provider 36
failure leave protocol
    removing failed providers 37
failure to responsiveness check
    Group Services action 8
failure, node
    Group Services handling 25
failure, process
    failure leave protocol 25
    responsiveness checking 25
fault-tolerant system
    design 3
final notification
    approval 19
    rejection 19
first failure data capture (FFDC) ID 47

G

group attributes
    changing 6
    defining for a group 6
    definition 6
    mutability 6
group creation
    definition 6
group dissolution
    announcement 19
group name
    definition 6
group quorum
    use with Group Services 30
group services
    creation 6
    defining attributes 6
Group Services
    application programming interface
        return values 126
    configuring 34
    managing 35
    subroutine summary 47
Group Services domain
    definition 5
Group Services subsystem
    concepts 2
    coordination among applications 3
    design considerations 31
    protocol priorities 37
    synchronization within an application 3
    use of for coordination 3
group state data
    definition 4
group state value
    definition 4
group, Group Services
    changing attributes 6
    definition 4
GS client
    definition 4
    failure to responsiveness check 8
GSAPI
    return values 126
GSAPI (Group Services application programming interface)
    functions supporting high availability 2
GSAPI deactivation scripts 47
GSAPI first failure data capture subroutine 47
GSAPI subroutines 47

H

ha_gs_announcement_callback subroutine 48
ha_gs_change_attributes subroutine
    reference 50
ha_gs_change_state_value subroutine
    reference 54
ha_gs_delayed_error_callback subroutine 57
ha_gs_dispatch subroutine 59
HA_GS_ENABLE_IPV6 33
ha_gs_expel subroutine 62
ha_gs_get_adapter_info subroutine
    reference 65
ha_gs_get_adapter_info_by_addr subroutine 67
ha_gs_get_adapter_info_by_id subroutine 70
ha_gs_get_ffdc_id subroutine
    reference 71
ha_gs_get_ipaddr_by_id subroutine
    reference 72
ha_gs_get_limits subroutine 74
ha_gs_get_node_number 75
ha_gs_get_rsct_active_version subroutine
    reference 76
ha_gs_get_rsct_installed_version subroutine
    reference 77
ha_gs_goodbye subroutine 78
ha_gs_init subroutine 80
ha_gs_ip_addr 33
ha_gs_join subroutine 85
ha_gs_leave subroutine 91
ha_gs_n_phase_callback subroutine
    reference 94
ha_gs_protocol_approved_callback subroutine 102
ha_gs_protocol_rejected_callback subroutine 104
ha_gs_quit subroutine 107
ha_gs_responsiveness_callback subroutine 108
ha_gs_send_message subroutine
    reference 110
ha_gs_socket_ctrl_t data type 33
ha_gs_special_block_t 46
ha_gs_special_data_t 43
ha_gs_subscribe subroutine 112
ha_gs_subscriber_callback subroutine
    reference 118
ha_gs_unsubscribe subroutine
    reference 122
ha_gs_vote subroutine 123
ha_gs.h header file
    reference 126

hardware techniques
  for high availability 2
header files
  ha_gs.h 126
high availability
  cluster environment 2
  definition 2
  hardware techniques 2
  software techniques
    Group Services subsystem 2
host membership group
  subscription to host membership group 29
  system-defined group 29

I

idempotent actions
  definition 32
interapplication dependency
  managing with source-target groups 27
IP address 47
IP addressing 33
  accessing 42
IP take over
  for high availability 2

J

join request
  definition 4
joining a group
  as a provider 36
  membership list change 5

L

leaving a group
  as a provider 36
  membership list change 5
loosely synchronous consistency
  definition 20

M

maximum
  length of group name 87
membership change proposal
  definition 18
membership change protocol
  introduction 9
membership in multiple groups
  coordination of members 27
membership list
  definition 4
merging
  of sundered networks 31
message size
  minimizing for performance 32
migration
  design considerations 4
multi-phase commit
  use in Group Services 3
multi-tailed disks
  for high availability 2
multi-threaded application
  designing callback routines 31
multi-threaded GS client
  responsiveness protocol 7
multinode environment
  coordination 3

N

n-phase protocol
  definition 8
  example 19
n-phase state change protocol
  example 20
network address takeover
  for high availability 2
networks, sundered
  handling of 30
  merging of 31
node failure
  Group Services handling 25
non-voting protocol
  one-phase protocol 36
nonresponsive process
  handling 7, 14
  notification 19
notification
  announcement 19
  checking for 47
  coordinating in callback routines 31
  definition 18
  for provider 19
  for subscriber 19
  ongoing protocol 18
  protocol approval 19
  protocol proposal 18
  protocol rejection 19
  responsiveness 7, 19

O

one-phase protocol
  definition 8
  example 19
  use in Group Services 3
ongoing protocol notification
  definition 18

P

peer process applications
  in a cluster environment 2
performance considerations
  batching multiple requests 32
  callback routines 31
  minimizing application size 32
  minimizing message size 32
ping-like responsiveness protocol
  definition 7
prerequisite knowledge 1
primary group
  definition 6
process failure
  failure leave protocol 25
  responsiveness checking 25

process group
  use in Group Services 3
process termination
  and expel protocol 14
  and Group Services 7
  notification 19
proposal
  attribute change 19
  membership change 18
  provider-broadcast message 18
  serialization 23
  state value change 18
proposing a state change 47
protocol
  approval 38
  change-attributes 9
  definition 8
  ending 41
  expel 14
  initiating 37
  membership change 9
  n-phase 9, 12
    definition 8
    example 19
  one-phase 8, 11
  provider-broadcast message 9
  provider-initiated 39
  rejection 39
  simultaneous 13
  source-state reflection protocol 28
  state value change 9
  system-initiated 37
  voting 9
protocol approval notification
  definition 19
protocol proposal notification
  definition 18
protocol rejection notification
  definition 19
protocols
  approving 11
  rejecting 11
provider
  default vote 10
  definition 4
  expelling 47
  failure to responsiveness check 8
  joining a group 36
  leaving a group 36
  n-phase 10
  proposing 11
  rejecting 11
  tokens 27
  voting 10
provider failure
  announcement 19
provider-broadcast message proposal
  definition 18
provider-broadcast message protocol
  introduction 9

Q

quorum
  use with Group Services 30

R

recovery
  in a cluster environment 2
  using GSAPI 2
requests, Group Services
  serialization 37
responsiveness callback routine
  initializing 7
  purpose 7
responsiveness check
  definition 19
  description 7
  initializing for 7
responsiveness notification
  definition 19
responsiveness protocol
  counter-checking protocol 7
  no protocol 7
  ping-like protocol 7
return values
  GSAPI 126

S

script, deactivate 14
serialization
  Group Services requests 37
  notifications to subscribers 26
  of proposals 23
single-threaded GS client
  responsiveness protocol 7
size
  application
    minimizing for performance 32
  message
    minimizing for performance 32
software abstractions, Group Services
  barrier synchronization 3
  multi-phase commit 3
  one-phase protocol 3
  process group 3
  state data 3
software techniques
  for high availability
    Group Services subsystem 2
source-state reflection protocol
  definition 28
source-target groups
  choosing over subscription 27
  managing interapplication dependency 27
  source-state reflection protocol 28
source-target relationship
  attributes 6
SP Switch adapter alias 42
SP Switch adapter membership group
  system-defined group 29
state data
  use in Group Services 3
state value change proposal
  definition 18
state value change protocol
  introduction 9
state value, proposing a change 47
strongly-consistent synchronization
  definition 20

subroutines
adapter information 47
changing a group's attributes 47
changing a group's state value 47
checking for notifications 47
expelling providers 47
getting an FFDC ID 47
getting an IP address 47
ha_gs_announcement_callback 48
ha_gs_change_attributes 50
ha_gs_change_state_value 54
ha_gs_delayed_error_callback 57
ha_gs_dispatch 59
ha_gs_expel 62
ha_gs_get_adapter_info 65
ha_gs_get_adapter_info_by_addr 67
ha_gs_get_adapter_info_by_id 70
ha_gs_get_ffdc_id 71
ha_gs_get_ipaddr_by_id 72
ha_gs_get_limits 74
ha_gs_get_node_number 75
ha_gs_get_rsct_active_version 76
ha_gs_get_rsct_installed_version 77
ha_gs_goodbye 78
ha_gs_init 80
ha_gs_join 85
ha_gs_leave 91
ha_gs_n_phase_callback 94
ha_gs_protocol_approved_callback 102
ha_gs_protocol_rejected_callback 104
ha_gs_quit 107
ha_gs_responsiveness_callback 108
ha_gs_send_message 110
ha_gs_subscribe 112
ha_gs_subscriber_callback 118
ha_gs_unsubscribe 122
ha_gs_vote 123
mutability of group attributes 47
summary 47
subscribe request
definition 4
subscriber
definition 4
participation in group 37
roles 26
tokens 27
subscribing to a group
notifications 26
overview 26
using for interapplication coordination 26
subscription
choosing source-target groups 27
subscription notification 43
subscription special data 42
formats 43
prerequisites 43
subsystems
in cluster environment 2
sundered networks
handling of 30
merging of 31
synchronization, intra-application
Group Services subsystem 3

T

tallying votes 124
time limit
voting 25
tokens
provider 27
reuse 27
subscriber 27
two-phase commit protocol
example 20

U

undoing actions
definition 32

V

version code
definition 6
votes
default 124
tallying 124
voting
approval 8
phases 6
protocol 8
protocol notification 18
rejection 8
time limit 6, 13, 25
voting protocol
n-phase protocol 36

Printed in USA