RS/6000 Cluster Technology

Group Services Programming Guide and Reference

SA22-7355-00

RS/6000 Cluster Technology

Group Services Programming Guide and Reference

SA22-7355-00

Note!

Before using this information and the product it supports, be sure to read the general information under “Notices” on page vii.

First Edition (October 1998)

This edition applies to:

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

Order publications through your IBM representative or the IBM branch office serving your locality. Publications are not stocked at the address below.

IBM welcomes your comments. A form for readers' comments may be provided at the back of this publication, or you may address your comments to the following address:

International Business Machines Corporation
Department 55JA, Mail Station P384
522 South Road
Poughkeepsie, NY 12601-5400
United States of America

FAX (United States & Canada): 1+914+432-9405
FAX (Other Countries):
Your International Access Code +1+914+432-9405

IBMLink (United States customers only): IBMUSM10(MHVRCFS)
IBM Mail Exchange: USIB6TC9 at IBMMAIL
Internet e-mail: mhvrcfs@us.ibm.com
World Wide Web: http://www.rs6000.ibm.com

If you would like a reply, be sure to include your name, address, telephone number, or FAX number.

Make sure to include the following in your comment or note:

- Title and order number of this book
- Page number or topic related to your comment

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

Contents

Figures

Notices

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program, or service is not intended to state or imply that only IBM's product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any of IBM's intellectual property rights may be used instead of the IBM product, program, or service. Evaluation and verification of operation in conjunction with other products, except those expressly designated by IBM, are the user's responsibility.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
500 Columbus Avenue
Thornwood, NY 10594
USA

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
Mail Station P300
522 South Road
Poughkeepsie, NY 12601-5400
USA
Attention: Information Request

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

Trademarks

The following terms are trademarks of the International Business Machines Corporation in the United States or other countries or both:

AIX
AIX/6000
DATABASE 2
DB2
ES/9000
ESCON
HACMP/6000
IBM
IBMLink
LoadLeveler
NQS/MVS
POWERparallel

POWERserver
POWERstation
RS/6000
RS/6000 Scalable POWERparallel Systems
Scalable POWERparallel Systems
SP
System/370
System/390
TURBOWAYS

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States and/or other countries.

UNIX is a registered trademark in the United States and/or other countries licensed exclusively through X/Open Company Limited.

Other company, product, and service names, which may be denoted by a double asterisk (**), may be trademarks or service marks of others.

Publicly Available Software

PSSP includes software that is publicly available:

expect Programmed dialogue with interactive programs

Kerberos Provides authentication of the execution of remote commands

NTP Network Time Protocol

Perl Practical Extraction and Report Language

SUP Software Update Protocol

Tcl Tool Command Language

TclX Tool Command Language Extended

Tk Tcl-based Tool Kit for X-windows

This book discusses the use of these products only as they apply specifically to the RS/6000 SP system. The distribution for these products includes the source code and associated documentation. (Kerberos does not ship source code.) **/usr/lpp/ssp/public** contains the compressed **tar** files of the publicly available software. (IBM has made minor modifications to the versions of Tcl and Tk used in the SP system to improve their security characteristics. Therefore, the IBM-supplied versions do not match exactly the versions you may build from the compressed **tar** files.) All copyright notices in the documentation must be respected. You can find version and distribution information for each of these products that are part of your selected install options in the **/usr/lpp/ssp/README/ssp.public.README** file.

About This Book

This book contains conceptual, guidance, and reference information to help you write programs that use the Group Services APIs that are part of RS/6000 Cluster Technology (RSCT). RSCT is provided by IBM Parallel System Support Programs for AIX (PSSP) and by the Enhanced Scalability feature of IBM High Availability Cluster Multi-Processing for AIX (HACMP/ES). Specifically, this book contains information to help you write Group Services client programs.

For a list of related books and information about accessing online information, see the bibliography in the back of the book.

Checking Which Level(s) of Code You Have

This book applies to PSSP Version 3 Release 1 and to HACMP/ES at the HACMP Version 4 Release 3 level. To find out which level(s) of code you have running on your system, follow the instructions in the sections below.

PSSP

To find out what version of PSSP is running on your control workstation (node 0), enter the following:

```
splst_versions -t -n0
```

In response, the system displays something similar to:

```
0 PSSP-3.1
```

If the response indicates **PSSP-3.1**, this book applies to the version of PSSP that is running on your system.

To find out what version of PSSP is running on the nodes of your system, enter the following from your control workstation:

```
splst_versions -t -G
```

In response, the system displays something similar to:

```
1 PSSP-3.1
2 PSSP-3.1
7 PSSP-2.4
8 PSSP-2.2
```

If the response indicates **PSSP-3.1**, this book applies to the version of PSSP that is running on your system.

If you are running mixed levels of PSSP, be sure to maintain and refer to the appropriate documentation for whatever versions of PSSP you are running.

HACMP

To find out which version of HACMP is running on a particular node, enter the following:

```
lslpp -L | grep cluster.es.server.rte
```

RSCT

To find out which code version of IBM RS/6000 Cluster Technology is running on a particular node, enter the following:

lslpp -L rsct.basic.rte

Who Should Use This Book

This book 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 book contains information for programmers who want to write new clients that use the GSAPI or add the use of Group Services to existing programs.

It assumes that you are an experienced C programmer and have a thorough understanding of RS/6000 SP (SP) hardware, IBM Parallel System Support Programs for AIX (PSSP) software and/or the Enhanced Scalability feature of IBM High Availability Cluster Multi-Processing for AIX (HACMP/ES) software, and AIX operating system fundamentals.

The commands and interfaces described in this book require that you have the appropriate privileges and authorizations. For example, you may need to be running with an effective user ID of **root**. The specific security requirements for each subroutine and command are listed in the reference material.

For information about managing the Group Services subsystem and its daemons, see *PSSP: Administration Guide*, SA22-7348. For information about Group Services diagnostic commands, see the IBM redbook *RS/6000 SP High Availability Infrastructure*, SG24-4838.

Typographic Conventions

This book uses the following typographic conventions:

Typographic	Usage
Bold	• **Bold** words or characters represent system elements that you must use literally, such as commands, flags, and path names.
Italic	• *Italic* words or characters represent variable values that you must supply. • *Italics* are also used for book titles and for general emphasis in text.
`Constant width`	Examples and information that the system displays appear in `constant width` typeface.
[ ]	Brackets enclose optional items in format and syntax descriptions.
{ }	Braces enclose a list from which you must choose an item in format and syntax descriptions.
|	A vertical bar separates items in a list of choices. (In other words, it means “or.”)
< >	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 Enter.
...	An ellipsis indicates that you can repeat the preceding item one or more times.
<**Ctrl-*x*>**	The notation <**Ctrl-*x*>** indicates a control character sequence. For example, <**Ctrl-c>** means that you hold down the control key while pressing <**c>**.

Chapter 1. Understanding Group Services

This chapter introduces you to the application programming interface (API) for Group Services that is provided by the IBM Parallel System Support Programs for AIX (PSSP), which runs on the IBM RS/6000 SP (SP), and by the Enhanced Scalability feature of the IBM High Availability Cluster Multi-Processing for AIX Licensed Program (HACMP/ES), which runs on IBM RS/6000 clusters.

It begins with a bit of background on why high availability is important in a multicomputer environment and how Group Services can help you achieve it.

It continues with an introduction to Group Services functions and how you can use them, followed by an explanation of the concepts and operation of the Group Services subsystem.

A section on design considerations helps you make design choices as you plan your application's use of the Group Services Application Programming Interface (GSAPI).

For the definitions of terms and concepts that are introduced in this chapter, see “Glossary of Terms and Abbreviations” on page 237.

For information about managing the Group Services subsystem, see *PSSP: Administration Guide*, SA22-7348.

High Availability in a Multicomputer Environment

Applications that can best exploit a multicomputer such as the RS/6000 SP (SP) typically consist of several cooperating processes running on multiple nodes. These applications, which may or may not be subsystems, can be structured as:

- Client/server processes

  Examples include a distributed file system and the SP's Virtual Shared Disk (VSD) recovery subsystem. For more information on the latter, see *PSSP: Managing Shared Disks*, SA22-7279 .

- Peer processes

  Many scientific applications are structured as peer processes.

Most multicomputer environments are 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, multicomputer applications must be highly available. This means that the application continues to execute after the failure, perhaps after a brief interruption of service to accommodate recovery from the failure, and perhaps also 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, using which 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), using which 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 the detection of component (process and node) failures, the recovery from communication partitions, the coordination of activities among the processes of an application, and the coordination of activities between applications. It is these additional requirements that the Group Services subsystem and its Application Programming Interface (GSAPI) are designed to satisfy.

Introduction to Group Services

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

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 buy into 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 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; it was not designed to provide a high-volume, high-bandwidth, general-purpose messaging facility. Also, Group Services does not perform application recovery per se; rather, it helps applications orchestrate their recovery.

Using Group Services

If you are writing a new application or are considering updating an existing application, you can use the GSAPI to improve SP system availability in a number of ways. You can use the GSAPI to:

- 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.

Overview of Group Services Concepts

An application may consist of multiple processes that run on multiple nodes of an SP system. The set of nodes that is defined to Group Services is called a Group Services domain. A Group Services domain consists of the set of nodes that makes up a system partition. If there is only one system partition in the SP system, the Group Services domain consists of all of the nodes in the SP system. From the point of view of the Group Services subsystem, the system partition in which it is running is "the system."

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" is used to refer 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 **HA_SYSPAR_NAME** 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, and the setting of **HA_SYSPAR_NAME** 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 two pieces of information:

- The membership list

  The membership list of a group is the list of providers in the group. Each provider is identified by a provider identifier. The Group Services subsystem maintains the list in the following order: the oldest provider (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 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 state value is not interpreted by the Group Services subsystem.

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.

Providers cause the membership list to be modified by joining or leaving 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.

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.
- It may be expelled at the request of another provider.
- It may 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; that is, 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 execution, then as soon as the protocol completes, Group Services runs a failure leave protocol to remove the failed providers from the group.

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.

A Detailed Look at Group Services Concepts

This section expands on the concepts outlined in “Overview of Group Services Concepts” on page 3. It provides more information on the following topics:

Group Services Domains

The Group Services subsystem provides services within the boundaries of what it calls a **domain**. A Group Services domain includes an SP control workstation and the set of nodes defined to be within an SP Partition. This means that an SP control workstation can be within multiple Group Services domains. An application wishing to become a GS client on the control workstation or on an SP node must set one or more environment variables to ensure that it is able to connect to the proper Group Services domain.

An SP will be divided into one or more Group Services domains, based on the number of SP partitions defined. Each of these domains is referred to as a “Group Services PSSP domain.” To connect to the Group Services PSSP domain a GS client must ensure that the environment variable **HA_DOMAIN_NAME** is set to the name of the SP partition in which the GS client is executing prior to the GS client executing 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 there are multiple defined SP partitions).

Refer to the description of the **ha_gs_init** subroutine for more information about connecting to the Group Services subsystem and the meaning of error codes that may be returned.

In addition to the Group Services PSSP domain, if HACMP/ES is installed on a node, then that node will also be part of a Group Services HACMP/ES domain, which is separate from that node's Group Services PSSP domain. The Group Services HACMP/ES domain consists of all nodes that are part of the HACMP/ES cluster, which may include SP nodes, non-SP AIX workstations, and possibly even SP nodes from a physically separate SP.

A GS client may 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 GS client only gets information about the nodes and groups that are in the Group Services domain to which it is connected.

If a GS client wishes to connect to the Group Services HACMP/ES domain on its node, it needs to set two environment variables:

- **HA_DOMAIN_NAME** must be set to the name of the HACMP/ES cluster.
- **HA_GS_SUBSYS** must be set to **grpsvcs**.

These environment variables must be set before the GS client executes the **ha_gs_init** subroutine.

All of the Group Services interfaces and semantics work identically and are supported within both Group Services PSSP domains and Group Services HACMP/ES domains.

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 of the application, as each process in the application starts up, it asks to join the group. On receipt of the first join request, the Group Services subsystem creates the group. The subsequent join requests result in new providers joining the group.

The first join request creates the group and defines its attributes. All subsequent requests to join this group also include group attribute information, and the information must match the group's established attributes. Otherwise, the subsequent 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 of the default vote 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:
  - 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.

All of the topics related to these attributes are discussed in greater detail later in this chapter.

Mutability of Group Attributes

Certain group attributes are **mutable**; that is, they can be dynamically changed by the group's providers, using the **ha_gs_change_attributes** asynchronous interface. (See “ha_gs_change_attributes Subroutine” on page 52.)

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

gs_client_version

Client-specified “version” number

gs_batch_control

Batch control setting for membership (join/failure protocols)

gs_num_phases

Phase control setting for membership (join/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/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. (See “Source-Target Group Relationships” on page 39.)

Responsiveness Checks

Responsiveness checks allow the Group Services subsystem to inspect the state of the GS client periodically 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 the following information:

- Information about the protocol, if any, to be used to perform responsiveness checks for the GS client
- The path name of a callback routine to be called if the GS client fails its responsiveness check

The GS client can specify one of the following 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.

- A counter-checking protocol.

  In this case, Group Services periodically checks an arithmetic counter that is specified by a multi-threaded client. If the counter is changing, Group Services assumes that the GS client is responsive.

  If the counter does not change within a time limit specified by the GS client, Group Services calls the responsiveness callback routine before assuming that the GS client is nonresponsive.

  This protocol works properly only for multi-threaded GS clients, and a thread must be dedicated to performing Group Services operations. The dedicated thread should handle responsiveness checking as well as the receipt of notifications from Group Services.

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 any periodic validity checks on its own operation or its environment that may be needed.

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

On receipt of the announcement notification, a provider could initiate an expel protocol to remove the nonresponsive providers from the group, if appropriate. For more information, see “The Expel Protocol” on page 22. 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 containing the list to all of the group's providers.

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

Protocols and Voting

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

The 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 tailor 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 proposal puts the group through at least one phase of voting before the change can be approved. The number of phases required is not specified explicitly 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, a final notification describing the change is sent to all of the group's providers and subscribers. If it is rejected, only the providers receive the final notification, and 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.

Protocols can be grouped into four categories:

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 19.

Membership change protocols include:

- Join
- Leave (also called "voluntary leave")
- Expel
- Failure leave (including clients that execute **ha_gs_goodbye**)
- Cast-out, a form of failure leave that is associated with source-target relationships (see “Source-Target Group Relationships” on page 39).

The state value change protocol

This protocol is used when a provider wants to change the state value of the group, but wants to leave the membership unchanged.

As for n-phase membership change protocols, 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 one-phase, this protocol allows a provider to broadcast a message to all other providers in the group, with no voting.

If n-phase, this protocol 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 may be changed during each voting phase. Provider-broadcast message protocols do not affect the group membership.

The change-attributes protocol

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

If n-phase, this protocol 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.

The remainder of this section discusses general mechanisms for handling protocols and includes the following topics:

- Voting on an n-phase protocol
- Approving and rejecting protocols
- Proposing, voting, and phases for protocols
- Initiating protocols
- Submitting changes with voting responses
- Simultaneous protocols
- Ending a protocol

Voting on an n-Phase Protocol

When a provider receives an n-phase protocol proposal notification, it is being asked to vote to either approve or reject the proposed change. 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 proposed by the protocol is approved and the group state is changed accordingly. If the vote tally indicates 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 Protocols” on page 17.

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 subject the proposed change to another round of voting (CONTINUE), or to reject it and end the protocol (REJECT). The 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, when it joins the group, the provider can set the default vote to APPROVE instead. 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 of the vote. 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 39.

Specifying the Provider's Default Vote: All providers within a group have the same default vote. Its value is set as follows:

1. The Group Services subsystem assigns REJECT as the default value for each group.
2. As part of their request to join a group, the providers may specify a default vote value as part of the group attributes. They may specify either REJECT or APPROVE. All providers must specify the same value.
3. Finally, 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 unpredictable which one the Group Services subsystem will choose.

As discussed in “Submitting Changes with Voting Responses” on page 19, to ensure consistency, the group should ensure that:

- All providers submit the same updated default vote value, or
- Only one provider submits it, or
- No providers submit it, allowing the current value to carry over.

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 a nutshell, a protocol proceeds as follows:

1. The protocol is proposed by a provider or by the Group Services subsystem.
2. If necessary, voting proceeds for the desired number of phases.
3. If the protocol is approved, the updated information is broadcast to all providers and subscribers, as well as any appropriate target-groups.
4. 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. The proposal can be initiated by either a provider or the Group Services subsystem itself.

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.

The following sections describe the two types of protocols in more detail.

One-Phase Protocols: A one-phase protocol is simply 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 (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 it is detected that a provider has failed during the protocol, all remaining members receive the protocol notification and 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.

The proposal indicates that the protocol requires one or more voting phases, but it does not specify the exact number of phases that will be used. The actual number of voting phases that will be held is determined by the voting results.

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:

- 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.

Any protocol may be proposed as an n-phase protocol.

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 11.
- Optionally, an updated state value and/or a provider-broadcast message. For details, see “Submitting Changes with Voting Responses” on page 19.
- 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” on page 12.

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, the Group Services subsystem 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 phases end 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.

Initiating the Protocols

Proposals are initiated by either a provider or the Group Services subsystem itself.

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-group(s).

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, then, it is simply a matter of calling the proper GSAPI subroutine. The Group Services subsystem takes care of notifying 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 13, 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 being 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 provider(s) 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 execute 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.

Approval of the Protocols: In all cases, if the protocol is approved, the following actions are taken.

- All providers receive an updated membership list and/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 and/or state 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 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 execution of the protocol stops. The membership list is updated to show the removal of the targeted provider(s). 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 execution of 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.

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

Provider-Initiated Protocols: Provider-initiated proposals include proposals made by a provider to:

- 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

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 an executing protocol, the provider receives a synchronous error code that indicates a collision between competing protocols. (Only one protocol may execute 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 results of the protocol, that is, its approval or rejection.

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 and/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 execution of 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 provider(s). 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 “The Expel Protocol” on page 22.

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-group(s) 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:
  - If it is approved, and if the group state value was changed during the voting phases, the providers and subscribers receive the updated state value.
  - If it is 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.
  - If it is 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-group(s) also receive notification of the change.

Submitting Changes with Voting Responses

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

- A proposed new group state value
- A provider-broadcast message
- 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 very 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 each 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:

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

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

The 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 provider (or 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, the Group Services subsystem delivers an announcement 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 execution of deactivate scripts during expel protocols. For more details, see “The Expel Protocol” on page 22.

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 execute 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 lag time between the call of a GSAPI subroutine by a provider to initiate a protocol and the actual 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.

The Group Services subsystems handles simultaneous proposals as follows:

- In general, the first proposal to be made after an executing protocol completes is the one that is chosen to execute 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 executed 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 executed within the group. No provider-initiated proposals are accepted until all of the pending Group Services subsystem-initiated proposals have been executed. 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, except for those that may be batched together.
- If batching is allowed, membership changes are batched. Joins are batched only with joins, 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.

Ending a Protocol

The end of the protocol is signalled by the end of the voting phases of 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 the following information:

- If there was a membership change, the notification contains the updated membership list. Both providers and subscribers receive this information.
- If there was a proposal to change the group state value, the notification contains the new group state value. Both providers and subscribers receive this information.
- If a provider-broadcast message was submitted on the final vote, the notification contains the message. Providers receive this information; subscribers do not.
- The notification contains a flag that specifies whether any default votes were used to approve the protocol. 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.

The 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 the group is running.

During the execution of the expel protocol, Group Services executes a **deactivate script** against each provider that is being expelled. The deactivate script, which is specified by each GS client when it initializes itself to use Group Services, can be 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, on the node on which the provider is running, the Group Services daemon forks a child process that attempts to execute the deactivate script. For details about the environment in which the deactivate script executes and the input and output specifications to which it must conform, see "Deactivate Scripts" on page 28 and "ha_gs_expel Subroutine" on page 67.

The expel protocol is a provider-initiated protocol. Therefore, if it collides with another already-executing 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 its execution 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 the execution of 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 execute the deactivate script. A value of 0 indicates that the deactivate script should not be executed.

- 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 executes the deactivate script that was specified by that provider when it initialized itself with Group Services. The deactivate script executes on the node on which the provider that is targeted for expulsion is running. It executes during the phase and using the flag that was specified on the expel protocol, and, to be successful, must complete within the voting time limit for the phase. To execute the deactivate script, Group Services acts as a "stand-in" for each provider that is being expelled.

The expel protocol operates as follows:

- During the protocol, providers that are not being expelled treat this as a normal protocol and take any action they deem appropriate. If this is an n-phase protocol, their voting responses are tallied as for any other n-phase protocol.
- If the value of the deactivate phase specifier is 0, no deactivate script is executed 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 executes the deactivate script against each targeted provider.

  If at least one voting provider votes to reject the protocol before this phase, the targeted providers are not removed from the group and no deactivate scripts are executed.
- If the expel protocol is a one-phase protocol and the value of the deactivate phase specifier is 1, the deactivate script is executed immediately after the protocol begins execution.

  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 are sent the protocol approval notification after the Group Services daemon has forked a child process to execute the deactivate script. The Group Services daemon does not wait for the script to complete execution before it sends the notification. Therefore, it is unpredictable whether the provider will receive the notification before or after the script executes.

  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 targeted for expulsion by a one-phase expel protocol fails after the the protocol has begun, no failure protocol is initiated in the group for that provider.
- If a deactivate script executes successfully, it is expected to exit with an exit code of 0. Group Services treats the successful execution 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 an exit 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 executed 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 was not executable, 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 an executable deactivate script, the script is executed and its result is used to drive the voting, as previously described.
- If a provider fails after the expel protocol begins but before the Group Services daemon has forked a child process to execute the deactivate script, Group Services passes a process ID of 0 to the deactivate script. The deactivate script is still executed and the exit code is used to determine the vote for this provider, as previously described.
- Group Services tallies the votes as usual for voting phases.
- If the expel protocol is approved, the providers that are targeted for expulsion are removed from the group. Remaining providers and subscribers are notified.

  Expelled providers that did not exit in the course of executing the deactivate script are sent the protocol approval notification by Group Services. 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.
- If 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.
- If 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 executing expel protocols that target those providers (because the process is unresponsive, for example), it is undefined by Group Services the order in which deactivate scripts are executed against the process.

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

Deactivate-On-Failure Handling

The same *deactivate script* will be executed 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 will allow recovery/clean-up actions on the failed provider's node, although the failed provider's process no longer exists.

If the protocol is an N-phase protocol, the results of the script's execution will be used in subsequent voting for the protocol. For a one-phase protocol, it will be a fire-and-forget situation, and the results will not be relayed to the remaining group members. Unlike expel, the group cannot specify in which voting phase the script will be executed. It will always be executed in the first phase. The failure leave protocol with the deactivate-on-failure operates as follows:

- If batching of failures *is not* allowed, the deactivate script is executed 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 executed once.
  - if there are multiple failed providers in separate protocols, the deactivate script will be executed once per protocol.

  The deactivate script will be executed on each node with a failed provider.
- If the failure protocol is a one-phase protocol:
  - The deactivate script is executed immediately after the protocol begins execution and the Group Services daemon does not wait for the script to complete execution. 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.
- 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 that the deactivate script will always be executed during the *first* phase of the failure protocol.
  - If a deactivate script executes successfully, it is expected to exit with an exit code of 0. Group Services treats the successful execution 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 an exit 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 if the script does not complete within the time specified by the voting phase time limit, Group Services daemon will treat this as a normal “voting time out” and apply the group's current default vote. If voting phases continue in the

protocol, Group Services daemon will continue to apply the group's current default vote value each subsequent voting phase.

  – Group Services tallies the votes as usual for voting phases.

  – 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.

    - If batching of failures is not allowed, and the rejection is caused by either an explicit reject vote or a default reject vote, the protocol ends and the failed providers are removed from the group. Remaining providers and subscribers will be notified.

    - If batching of failures is allowed, and the rejection is caused by an explicit reject vote, the protocol ends and the failed providers are removed from the group. Remaining providers and subscribers will be notified.

    - If batching of failures is allowed, and the rejection is caused by a default reject vote, the protocol ends, but the failed providers are not removed. The group will be 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 will be *executed only once* against any single failed provider instance. Thus, during the subsquent failure protocol(s), only the newly-failed providers will have their deactivate scripts executed, but no deactivate scripts will be executed against the already-failed providers. During any subsequent failure protocols, the Group Services daemon will vote APPROVE on behalf of the old failed providers. This avoids the group being put into an infinitely-looping situation, where the failure protocol is ended via a default REJECT vote caused by a failed deactivate script, and would otherwise be continually restarted.

- In the case where a failed GS client process had been joined as providers to multiple groups, each group continues to execute independent failure protocols.

  – If multiple groups specify deactivate-on-failure, then the deactivate script will be executed during each group's failure protocol.

  – Group Services does not define the order in which the deactivate scripts will be executed by each group, as the order in which the individual groups will execute the failure protocols is not defined.

- If a group has enabled deactivate-on-failure, and a provider (or more than one) is to be cast out, the decision will be:

  – If the targeted provider's process exists at the time the cast-out protocol begins execution, the deactivate script will *not* be executed.

  – If the targeted provider's process does not exist at the time the cast-out protocol begins execution, the deactivate script *will* be executed.

  – If the targeted provider's process exists at the time the cast-out protocol begins execution, but fails during the cast-out protocol, the deactivate script will *not* be executed.

Deactivate Scripts

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

Deactivate Scripts—Execution Environment

To handle a situation in which it 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 is executable until an expel protocol executes. If the specified deactivate script is not found or is not executable, Group Services applies the group's default vote value for the phase in which the deactivate script should have been executed, and for each subsequent voting phase, if there are any.

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

Effective uid and gid

The forked process executes with the effective uid and gid of the targeted provider that the provider 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 executes with those values.

Working directory

The forked process begins execution in the current working directory of the targeted provider that the provider had when it 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 execute in another directory must change to that directory.

Environment variables

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

STDIN, STDOUT, and STDERR file descriptors

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

Deactivate Scripts—Input Parameters

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

- The process ID parameter is always zero when the script is executed for deactivate-on-failure handling. (See “The Expel Protocol” on page 22.)
- 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 is the null-terminated string “providerdied.” The deactivate script can distinguish when it is called by checking this deactivate flag.
- The comma(,)-delimited list of failed provider's instance numbers

  This parameter will be presented only for deactivate-on-failure handling. When batching of failures is enabled, the deactivate script can be executed once for the multiple providers' failure. This fifth parameter will tell which providers were failing. Note that each provider instance number does not contain the node number.

Deactivate Scripts—Exit Codes

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 any.

If the deactivate script does not exit before the time limit expires, Group Services applies the group's default vote for this voting phase of the protocol, and each subsequent voting phase of the protocol, if any.

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.

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

These notifications are sent to the providers of a group to indicate that a proposal has been approved. It is also sent to the subscribers of the group.

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

Protocol Rejections

These notifications 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

These 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 responded to a responsiveness check.

Responsiveness Notifications

These notifications are sent to each of the providers of a group 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.

An Illustration of a 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 33 shows the sequence of events for a one-phase protocol. Figure 2 on page 33 shows the sequence of events for a two-phase commit protocol. Figure 3 on page 34 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**, informing them of the proposed change. 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 33 shows the execution of a multi-phase protocol in which the providers vote to approve the change after one round of voting. Figure 3 on page 34 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 execution 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 33.
- 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 34 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.

One-Phase and n-Phase Changes

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.

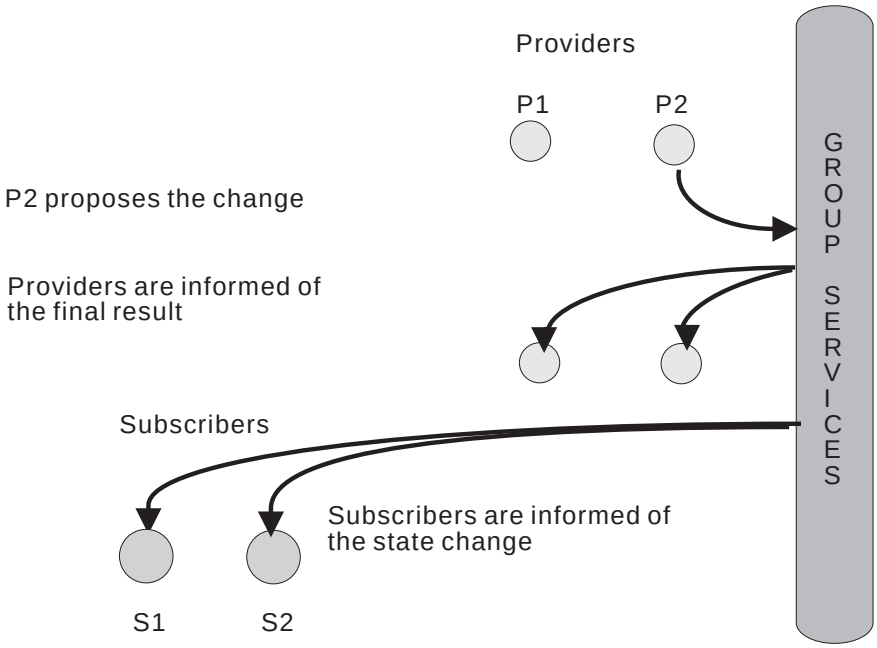

Figure 1. A One-Phase Protocol

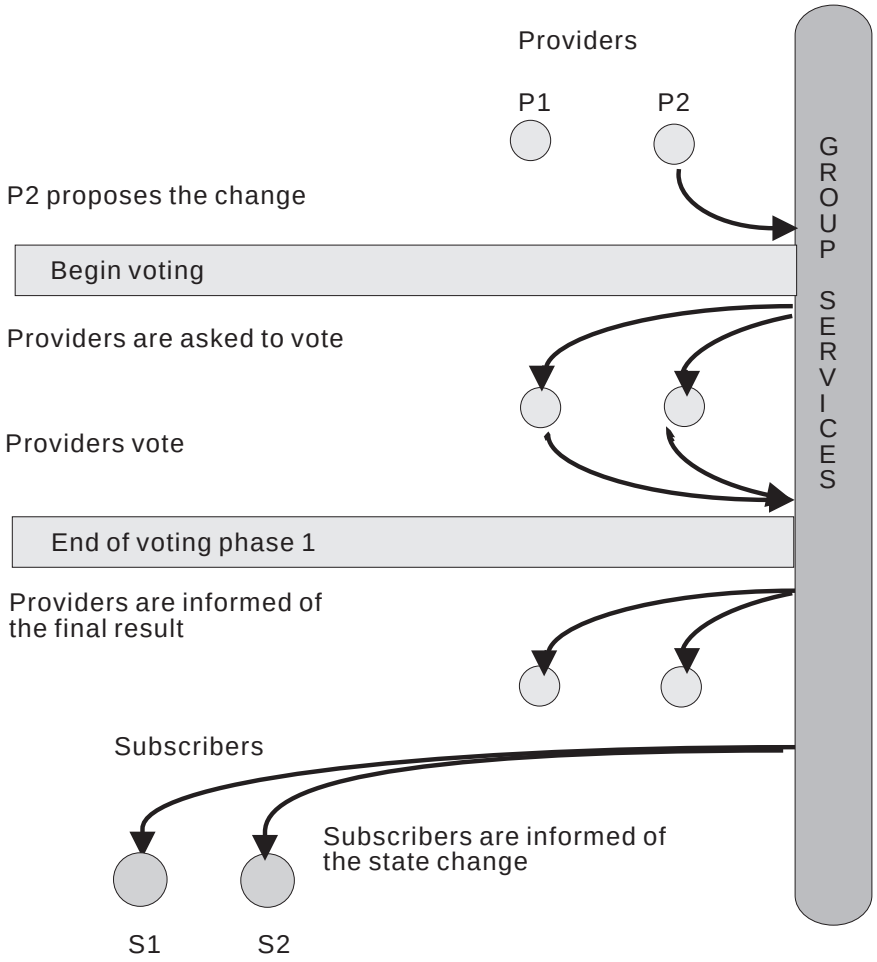

Figure 2. A Two-Phase Commit Protocol

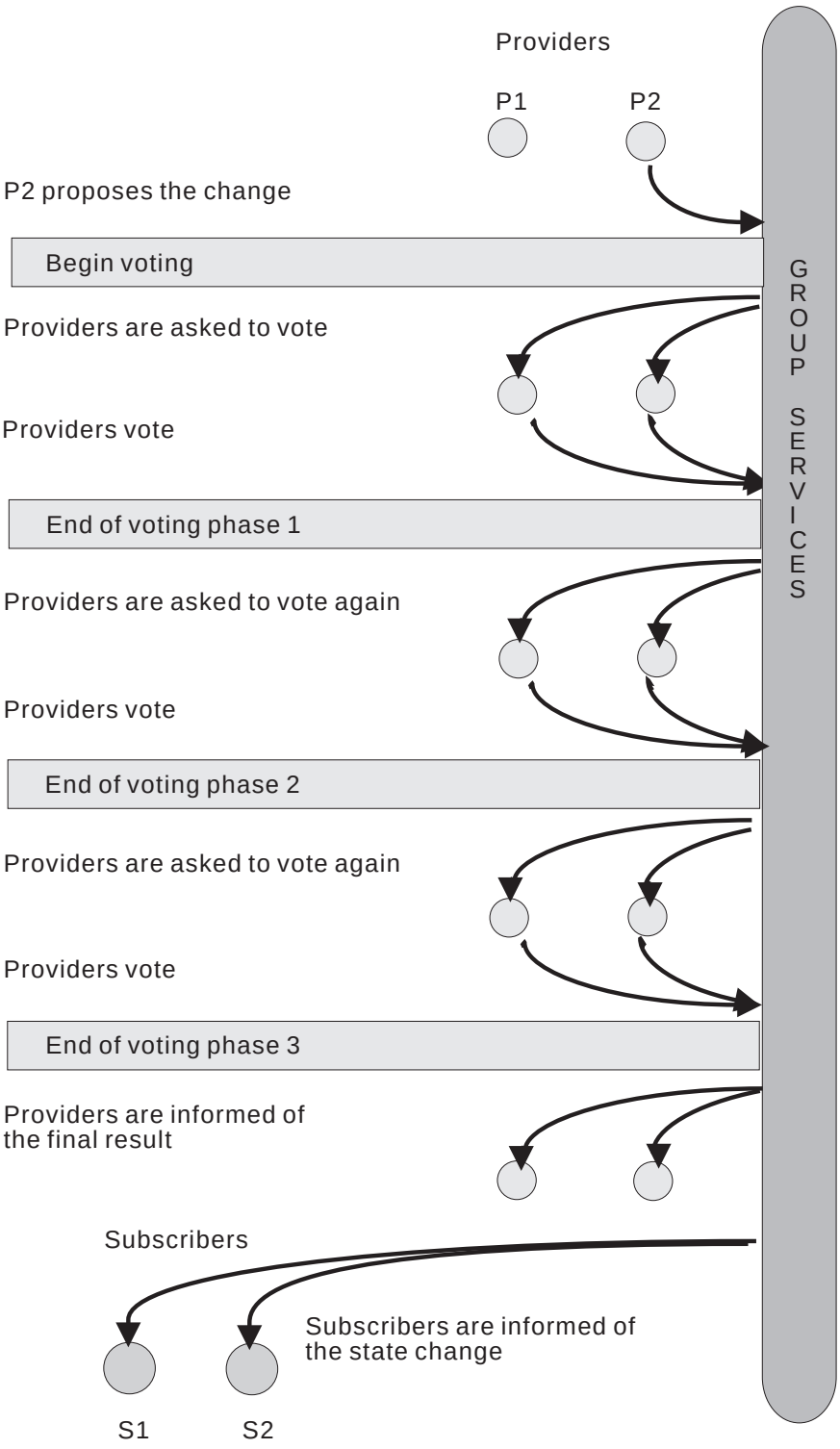

Figure 3. Barrier Synchronization in a Multi-Phase Protocol

Active Protocol Proposals

The GSAPI guarantees that only one protocol that affects the group's membership or state value is executed at any time. If more than one proposal is submitted within the group simultaneously, the Group Services subsystem chooses one for execution 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 execution, 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 and the membership change protocol is then started immediately. Figure 4 on page 36 shows how a new provider join request is delayed until the completion of an ongoing three-phase state change protocol.

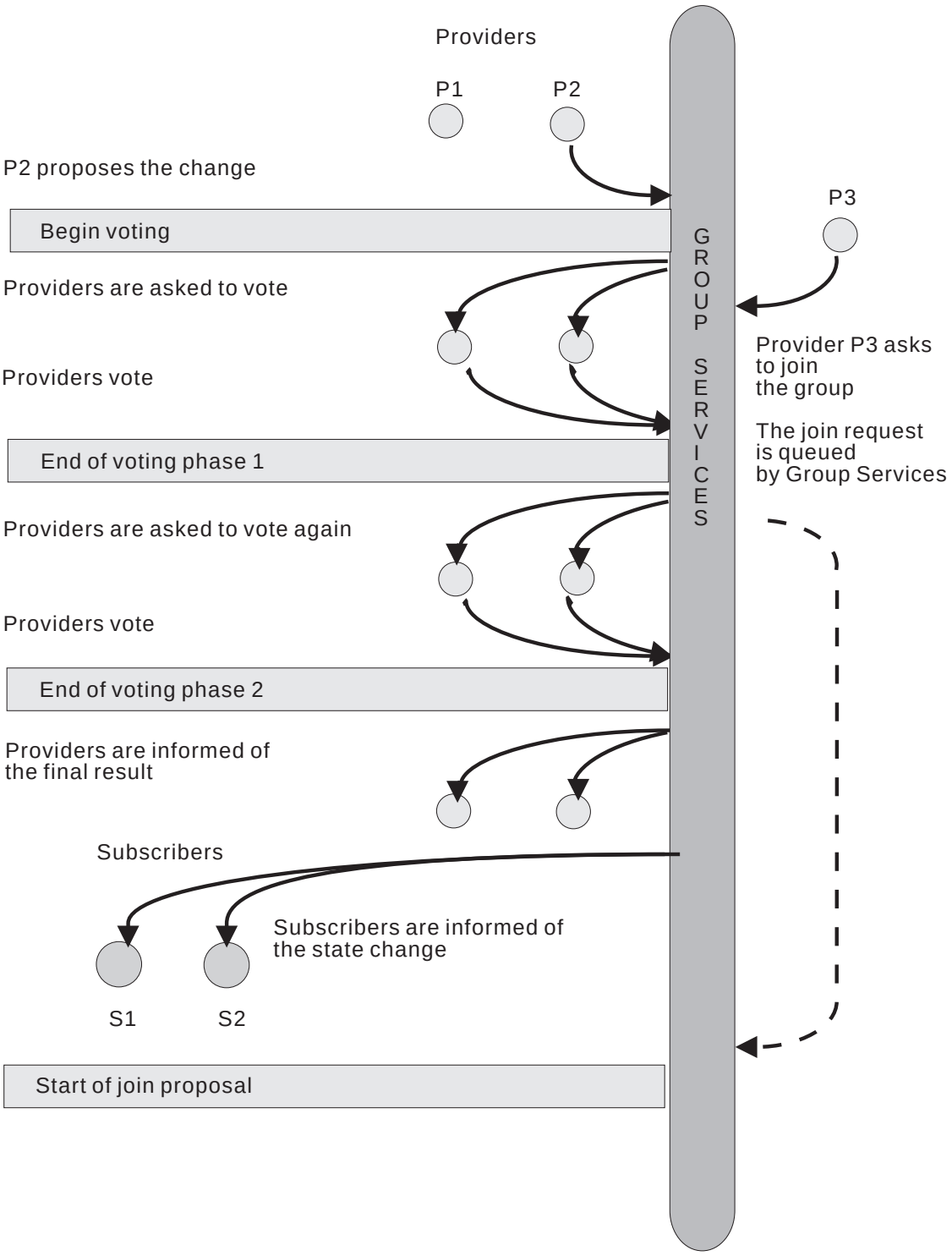

Figure 4. The Serialization of a Pending Join Request

Failures

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 a failure, 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, such as node failures, that lead to the loss of one or more Group Services processes, and network failures and communications adaptor failures, that can hinder the communication between Group Services processes.

If the Group Services subsystem fails, any surviving GS client receives an announcement notification that Group Services has died horribly. In addition, if a protocol is running, it is terminated.

See “Deactivate-On-Failure Handling” on page 26.

Provider Actions during Voting

As shown in Figure 5 on page 38, a provider can perform any sequence of actions that it chooses between the time that it receives a ongoing protocol notification (that is, a request for a vote) and the time that it votes.

However:

- Providers should submit their votes within the voting time limit.

  When a provider has been 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 an executing protocol has completed before submitting a new proposal.

  During the execution 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 executing protocol has completed before it resubmits the proposal.

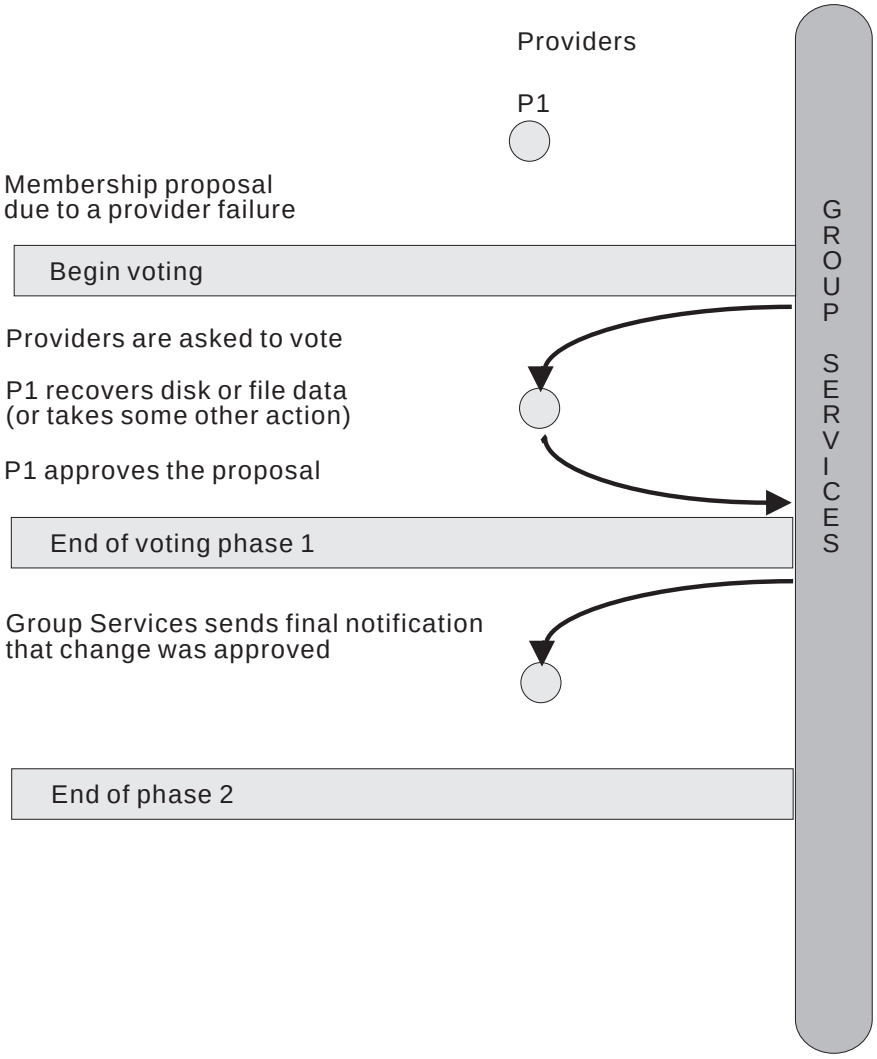

Figure 5. Actions during a Membership Change Protocol

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 called provider and subscriber tokens to identify providers and subscribers. These 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 are some examples.

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.

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 (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.

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 are sent 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 after another group has completed processing this change. Such a relationship might exist, for example, between a disk recovery subsystem and a distributed database application. If the database is on a disk on the failed node, the database application must wait for the disk recovery subsystem to recover from the node failure before it can begin its recovery.

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

The source-target facility allows a target group to tie itself to a source group as follows. If a failure leads to the failure of a provider in both the source and the target groups, then 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 execute with the knowledge that the providers in the source group have already handled the failure. This knowledge is particularly useful when the recovery of the target group depends on the completion of recovery by the source group.

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

With source-target groups, joins and leaves operate 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 and/or target-group providers on a node.

  A source-group may have any number of target-groups. A target-group may source only one group.

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

  The source-group processes the leave(s) as a normal 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(s) from

the target-group. As a failure leave, 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 provider(s) that are being cast out receive a notification that they have been cast out of the group. They do not otherwise participate in the cast-out protocol.

- 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.

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

  – If the protocol continues, the "failed" target-group provider(s) are no longer allowed to participate. Instead, the default vote (APPROVE in this case) is registered for them for each voting phase.

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

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

- 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.

- If a source-group changes its state value during protocols that do not result in a target-group cast-out (for example, through a state value change protocol or a voting response during any other n-phase protocol), its associated target-group(s) receive the committed state value.

  The notification appears to the target-group as a source-state reflection protocol. The number of phases and a voting time limit are controlled by values specified in the group attributes of the target-group.

  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.

Host and Adapter Membership Groups

The Group Services subsystem provides several system-defined groups to which GS clients can subscribe to keep track of hardware status. Refer to “Description” on page 112 for a complete listing of the different system-defined groups 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, which is represented by **HA_GS_HOST_MEMBERSHIP**. By subscribing to this group, a GS client can obtain information about which nodes are currently active, and about any transitions that occur as nodes become active or fail.

A node will appear 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 will thus 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 it can use to determine communications paths to nodes, for example.

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 for ethernet membership (**HA_GS_ENET_MEMBERSHIP**) that 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 (i.e., 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, then 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, then different nodes will see different views 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.

See “ha_gs_subscribe Subroutine” 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 different 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 any 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 this case of a sundered namespace, the nodes become split in such a way that they can no longer communicate with any nodes on the other side of the split. However, it is possible for each sundered portion to maintain enough information to reconstruct the groups that were in

existence previously, at least those groups that still have members within any particular portion.

When a namespace is sundered, it is possible to get two instances of what should be one group. For example, in a sundered network, two nodes that own the two tails of a twin-tailed disk could end up on separate sides of the split. Because the processes of the subsystem coordinating the disk on each node would believe that the other process had disappeared, the process might want to activate its tail, which could lead to data corruption. As this example shows, it is important that each group determine 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 execute those protocols as they normally would, taking into account such factors as quorum to protect resources as necessary.

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

GSAPI Design Considerations

This section includes some things to think about as you design your application to use the GSAPI. It includes guidelines for:

- Coding callback routines
- Coding for performance
- Migration and coexistence

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 two strategies, or a combination of the two:

- Code a number of specialized callback routines.

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

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

  This reduces the number of callback routines you need to code, but 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 on page 63 for additional information related to executing callback functions and handling notifications.

Coordination of Multiple Notifications

The following discussion of multiprocessing considerations applies 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 execute 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 executing 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 executes callback routines only on the same thread or threads that are used to call the **ha_gs_dispatch** subroutine.

Coding for Performance

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

  So, to the extent 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 execution 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, that is, they should be designed so that they can execute 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 is terminated 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 expensive 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.

Migration and Coexistence

This book documents the level of Group Services provided by PSSP 3.1 and by HACMP/ES at the HACMP 4.3 level. The changes from the previous level (provided by PSSP 2.3 and by HACMP/ES at the HACMP 4.2.2 level) are:

- Addition of the change-attributes protocol
- Support of the ability for a provider to exit the group immediately with the **ha_gs_goodbye** subroutine
- Ability to do cleanup on provider failure (deactivate on failure support)

GS clients that were written to any previous level of Group Services execute without recompilation and without change on the latest level.

However, if you wish your GS 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 GS client must run on nodes at the latest level of PSSP or HACMP/ES , and all of the nodes in the system partition must be running the latest level of PSSP or HACMP/ES. It is the responsibility of the GS client to ensure that all binaries are at the same level. GS clients can use group attributes, provider instance numbers, and group state values to ensure this.

Chapter 2. Group Services Subroutine Reference

This chapter contains the reference material for each of the subroutines in the Group Services Application Programming Interface (GSAPI). The chapter begins with a summary of GSAPI routines by function. Following the summary, the man pages for the subroutines and callback routines appear in alphabetical order.

GSAPI Summary

The GSAPI contains these types of routines:

- Routines to issue commands that request an action from the Group Services subsystem
- Routines that define callback routines that handle notifications from the Group Services subsystem.
- Deactivate scripts that the Group Services subsystem executes against providers that are targeted for expulsion from a group.

GSAPI Commands

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

Subroutine	Action
ha_gs_change_state_value	Propose a change to the group's state value
ha_gs_change_attributes	Dynamically changes certain group attributes
ha_gs_dispatch	Check for notifications
ha_gs_expel	Expel one or more providers from the group
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's membership or state value by approving, rejecting, or continuing

GSAPI Routines for Handling Notifications

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, 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's membership or state has been changed.

GSAPI Deactivate Scripts

For the specifications of deactivate scripts to use with expel protocols and with deactivate-on-failure handling, see “ha_gs_expel Subroutine” on page 67 and “Deactivate-On-Failure Handling” on page 26.

ha_gs_announcement_callback Subroutine

Purpose

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

Library

GSAPI Thread-Safe Library (**libha_gs_r.a**)

GSAPI Library (not thread-safe) (**libha_gs.a**)

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.

HA_GS_VOTING_TIME_LIMIT_EXCEEDED
: This flag is set for an announcement notification when the voting time limit has expired.

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

Prerequisite Information

Chapter 1, “Understanding Group Services” on page 1.

Related Information

Subroutines: **ha_gs_init**, **ha_gs_join**, **ha_gs_change_state_value**, **ha_gs_send_message**, **ha_gs_leave**, **ha_gs_expel**, **ha_gs_change_attributes**

ha_gs_change_attributes Subroutine

Purpose

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

Library

GSAPI Thread-Safe Library (**libha_gs_r.a**)

GSAPI Library (not thread-safe) (**libha_gs.a**)

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 state 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 state value.

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

If the Group Services subsystem chooses not to execute 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 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 group attributes block describes the attributes of the group, including the group's name, and is specified as input to a change attribute request using the **ha_gs_change_attribute** 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 following attributes can be changed via an **ha_gs_change_attributes** subroutine call:

- **gs_client_version**
- **gs_batch_control**
- **gs_num_phases**
- **gs_source_reflection_num_phases**
- **gs_group_default_vote**
- **gs_merge_control**

- **gs_time_limit**
- **gs_source_reflection_time_limit**

For attribute descriptions, see the complete list of group attributes under the ha_gs_join Subroutine on page 80.

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 that are in the group that were compiled and linked against a version of PSSP earlier than 3.1 (for RS/6000 SP systems) or against a version of HACMP/ES earlier than HACMP 4.3 (for RS/6000 workstation clusters). 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 execute 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 Errors (err_gsapi)” on page 130.

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 executing 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

One or more providers in the group were compiled and linked against a version of PSSP earlier than PSSP 3.1 (for RS/6000 SP systems) or against a version of HACMP/ES earlier than HACMP 4.3 (for RS/6000 workstation clusters); 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 is not at the PSSP 3.1 level (for RS/6000 SP systems) or is not at the HACMP 4.3 level of HACMP/ES (for RS/6000 workstation clusters); 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

Prerequisite Information

Chapter 1, “Understanding Group Services” on page 1.

Related Information

Subroutines: **ha_gs_init**, **ha_gs_join**, **ha_gs_send_message**, **ha_gs_leave**, **ha_gs_expel**, **ha_gs_subscribe**, **ha_gs_change_state_value**, **ha_gs_goodbye**

ha_gs_change_state_value Subroutine

Purpose

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

Library

GSAPI Thread-Safe Library (**libha_gs_r.a**)

GSAPI Library (not thread-safe) (**libha_gs.a**)

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 execute 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 execute this protocol, the group's providers are notified using normal n-phase voting procedures.

If the Group Services subsystem chooses not to execute 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 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 and 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 execute 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 Errors (err_gsapi)” on page 130.

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 executing 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

Prerequisite Information

Chapter 1, “Understanding Group Services” on page 1.

Related Information

Subroutines: **ha_gs_init**, **ha_gs_join**, **ha_gs_send_message**, **ha_gs_leave**, **ha_gs_expel**, **ha_gs_subscribe**, **ha_gs_change_attributes**, **ha_gs_goodbye**

ha_gs_delayed_error_callback Subroutine

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

GSAPI Thread-Safe Library (**libha_gs_r.a**)

GSAPI Library (not thread-safe) (**libha_gs.a**)

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 that 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_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 execute 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 execute 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 execute 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 execute instead of this one.

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

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 Errors (err_gsapi)” on page 130.

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 Errors (err_gsapi)” on page 130.

Synchronous Errors

None.

Asynchronous Errors

None.

Files

ha_gs.h

Prerequisite Information

Chapter 1, “Understanding Group Services” on page 1.

Related Information

Subroutines: **ha_gs_init**, **ha_gs_join**, **ha_gs_change_state_value**, **ha_gs_send_message**, **ha_gs_leave**, **ha_gs_expel**, **ha_gs_subscribe**, **ha_gs_change_attributes**

ha_gs_dispatch Subroutine

Purpose

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

Library

GSAPI Thread-Safe Library (**libha_gs_r.a**)

GSAPI Library (not thread-safe) (**libha_gs.a**)

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 call this subroutine on a regular basis to be able to receive messages from the GSAPI for delivery as notifications by executing the appropriate callback routines. This allows the GS client to respond to any protocols that may be executing in the group. The parameter to this 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, to allow 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 executed. 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, and 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 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 execute **ha_gs_dispatch** with the **HA_GS_NON_BLOCKING** flag.

   b. The GS client may set the **socket_options** flag on the **ha_gs_init** call to **HA_GS_SOCKET_SIGNAL**. In this case, any messages arriving on the file descriptor from Group Services will cause a SIGIO signal to be delivered to the client process. The client process must have established a handler to deal with this signal. The GS client should *not* call **ha_gs_dispatch** directly from the signal handler, but must remember to call it as soon as possible after returning from the signal handler.

   c. 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, as it may be much more often than necessary, and most times will result in no actions being taken as no messages have arrived; or it may 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 which case they will return from **ha_gs_dispatch** with an **HA_GS_NOT_OK** return code. These threads simply remain in **ha_gs_dispatch**, and as soon as a message arrives from Group Services a thread will read the message and execute the appropriate callback function.

   If the GS client has multiple threads call **ha_gs_dispatch**, then 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 execute **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 simply 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, or to subscribe to multiple groups, or to both join and subscribe to groups, the GSAPI library will enforce the following synchronization rules:

1. There will never be more than one active callback function being executed for any *one* group in parallel.
2. However, where appropriate, callback functions for different groups will be executed in parallel if there are multiple threads that have executed **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 execute 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 executed **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 executed **ha_gs_dispatch** to return from **ha_gs_dispatch** 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 Errors (err_gsapi)” on page 130.

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. The GS client needs to reinitialize (via **ha_gs_init()**).

Asynchronous Errors

None.

Files

ha_gs.h

Prerequisite Information

Chapter 1, “Understanding Group Services” on page 1.

Related Information

Subroutine: **ha_gs_init**

ha_gs_expel Subroutine

Purpose

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

Library

GSAPI Thread-Safe Library (**libha_gs_r.a**)

GSAPI Library (not thread-safe) (**libha_gs.a**)

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 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 of the group. 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 execute 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 execute this protocol, the group's providers are notified using normal n-phase voting procedures.

If the Group Services subsystem chooses not to execute 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 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 its execution. Otherwise, the deactivate script is considered unsuccessful. The deactivate script that is executed 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 points to a list of providers that are targeted to be expelled. 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.

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 non-responsive before the protocol began.

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

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 “Deactivate Scripts” on page 28.

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 execute 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 Errors (err_gsapi)” on page 130.

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 executing 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

Prerequisite Information

Chapter 1, “Understanding Group Services” on page 1.

Related Information

Subroutines: **ha_gs_init**, **ha_gs_join**, **ha_gs_change_state_value**, **ha_gs_send_message**, **ha_gs_subscribe**, **ha_gs_change_attributes**, **ha_gs_goodbye**

ha_gs_goodbye Subroutine

Purpose

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

Library

GSAPI Thread-Safe Library (**libha_gs_r.a**)

GSAPI Library (not thread-safe) (**libha_gs.a**)

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, then the provider issuing the **ha_gs_goodbye** protocol is considered to have failed, and 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 the section about the **ha_gs_n_phase_callback** subroutine on page 92.)

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 Errors (err_gsapi)" on page 130.

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

Prerequisite Information

Chapter 1, "Understanding Group Services" on page 1.

Related Information

Subroutines: **ha_gs_init**, **ha_gs_join**, **ha_gs_send_message**, **ha_gs_leave**, **ha_gs_expel**, **ha_gs_subscribe**, **ha_gs_change_attributes**

ha_gs_init Subroutine

Purpose

ha_gs_init – Called by a process to register with the Group Services Application Programming Interface (GSAPI)

Library

GSAPI Thread-Safe Library (**libha_gs_r.a**)

GSAPI Library (not thread-safe) (**libha_gs.a**)

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
A pointer to a buffer in which the Group Services subsystem will return the file descriptor that the process will use to communicate with the Group Services Application Programming Interface (GSAPI).

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

socket_options A flag that indicates whether the process will allow **SIGIO** signals to be generated when messages arrive. It must be one of the following values:

HA_GS_SOCKET_NO_SIGNAL
Do not generate **SIGIO** signals. The process will use non-signal I/O mechanisms, such as the **select** or **poll** system calls.

HA_GS_SOCKET_SIGNAL
Generate **SIGIO** signals when messages arrive. The process will catch **SIGIO** and call the **ha_gs_dispatch** subroutine.

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

deactivate_script
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
A pointer to a callback routine that will be called when the GSAPI delivers a responsiveness notification to the GS client. For information on the responsiveness callback routine, see the **ha_gs_responsiveness_callback** man page.

delayed_error_callback
A pointer to a callback routine that will be 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 GS client. This subroutine returns synchronously.

Only processes with **root** authority or whose user IDs belong to an AIX group called **hagsuser** are allowed to initialize themselves with the GSAPI. Refer to *PSSP: Administration Guide*, SA22-7348 for information about setting up the Group Services subsystem to enable support for use of **hagsuser**.

As described in “Group Services Domains” on page 6, a GS client must indicate the Group Services domain to which it wants to attach by setting the following environment variables prior to executing **ha_gs_init**. Normally this domain will be the group Services PSSP domain; however, it may also be a Group Services HACMP/ES domain if HACMP/ES is installed.

- **HA_DOMAIN_NAME** should be the name of the SP partition, to connect to the Group Services PSSP domain, or the name of the HACMP/ES cluster to connect to the Group Services HACMP/ES domain.

  Note: Earlier releases of Group Services supported the **HA_SYSPAR_NAME** environment variable, and this is still supported for compatibility, but all new applications should use **HA_DOMAIN_NAME**.

- **HA_GS_SUBSYS** should be set to **grpsvcs** if the GS client is connecting to the Group Services HACMP/ES domain. It does not need to be set if connecting to the Group Services PSSP domain, althrough if desired in this case it should be set to **hags**.

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

Note: On the RS/6000 SP, the **HA_SYSPAR_NAME** environment variable is still supported; however, the **HA_DOMAIN_NAME** environment variable is more versatile because it can be used to refer to a domain either on an RS/6000 SP or on an HACMP/ES cluster. (On an RS/6000 SP, a domain is a system partition; on an HACMP/ES cluster, a domain is the entire cluster.)

The responsiveness control structure has the following definition:

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

The responsiveness control structure is used to specify whether the process wants the GSAPI to check it periodically for responsiveness and, if so, the protocol to be used. It should be noted that the GSAPI will always be able to detect the actual exit (intentional or otherwise) of all GS clients. However, this check allows the GSAPI to determine if the GS client is able to respond in a reasonable fashion. It also allows the GS client to perform any periodic validity checks on its own operation or environment that might be needed.

If the GS 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 protocol.

The responsiveness protocol is executed only when the GS client is idle. If the GS client is involved in group actions (for example, executing protocols) and it is responding as expected, the GSAPI does not perform the responsiveness protocol.

The **gs_responsiveness_type** field contains the type of responsiveness protocol that is to be performed for this GS client. It may take one of the following values:

HA_GS_NO_RESPONSIVENESS
: The GSAPI should not perform a responsiveness protocol for this GS client. The remaining fields in the structure are ignored.

HA_GS_PING_RESPONSIVENESS
: The GSAPI should "ping" the GS client periodically, by delivering a notification to the GS client and expecting a response. The notification will call the responsiveness callback routine specified on input by the GS client.

HA_GS_COUNTER_RESPONSIVENESS
: The GSAPI should periodically check an arithmetic counter specified by a multithreaded GS client. If the counter is changing, the GS client is assumed to be responsive. If the counter does not change, the GSAPI will call the responsiveness callback routine specified by the GS client before assuming that the GS client is nonresponsive.

The **gs_responsiveness_interval** field contains the number of seconds that the GSAPI should wait between executions 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 GS 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 GS client's address space. If the **HA_GS_PING_RESPONSIVENESS** protocol is specified, this field is ignored.

The **gs_counter_length** field contains the length, in bytes, of the counter to be monitored. It may be a value of 2, 4 or 8. If the **HA_GS_PING_RESPONSIVENESS** protocol is specified, this field is ignored.

Security

The effective user ID of the calling process must be **root** or the user ID must belong to an AIX group called **hagsuser**.

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.

The GSAPI error numbers are defined in the **ha_gs.h** header file. For more information on GSAPI errors, see “GSAPI Errors (err_gsapi)” on page 130.

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 given; or invalid responsiveness control information was given; or an invalid socket control flag was given (it must be **HA_GS_SOCKET_SIGNAL** or **HA_GS_SOCKET_NO_SIGNAL**).

HA_GS_EXISTS
: This GS client has already successfully executed **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()**).

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 is not a root process; or this client is not a root process, but has an effective group ID of “hagsuser” and the GS daemon is not set up to allow access by non-root hagsuser processes.

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

Prerequisite Information

Chapter 1, “Understanding Group Services” on page 1.

Related Information

Structures in the **ha_gs.h** header file: **ha_gs_responsiveness_t** type definition

Subroutines: **ha_gs_init**, **ha_gs_join**, **ha_gs_send_message**, **ha_gs_leave**, **ha_gs_expel**, **ha_gs_subscribe**, **ha_gs_change_attributes**, **ha_gs_change_state_value**, **ha_gs_goodbye**

ha_gs_join Subroutine

Purpose

ha_gs_join – Called by a GS client to join a group as a provider

Library

GSAPI Thread-Safe Library (**libha_gs_r.a**)

GSAPI Library (not thread-safe) (**libha_gs.a**)

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 that 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, then 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, as defined by the **HA_GS_MAX_GROUP_NAME_LENGTH** constant.

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_FAILURES
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 execution of a deactivate script when the provider is failing.

See “Deactivate-On-Failure Handling” on page 26.

The **gs_num_phases** field specifies whether join protocols and failure leave (including cast-out) 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_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, which 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 and, 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 and the appropriate callback is executed, 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 executed and has been approved by the group.

Error Values

If the **ha_gs_join** subroutine is unsuccessful, it returns an error number and 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 Errors (err_gsapi)” on page 130.

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

Prerequisite Information

Chapter 1, “Understanding Group Services” on page 1.

Related Information

Subroutines: **ha_gs_init**, **ha_gs_change_state_value**, **ha_gs_send_message**, **ha_gs_leave**, **ha_gs_expel**, **ha_gs_send_message**, **ha_gs_subscribe**, **ha_gs_change_attributes**, **ha_gs_goodbye**

ha_gs_leave Subroutine

Purpose

ha_gs_leave – Called by a provider of a group to leave the group voluntarily

Library

GSAPI Thread-Safe Library (**libha_gs_r.a**)

GSAPI Library (not thread-safe) (**libha_gs.a**)

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 execute 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 execute this protocol, the group's providers are notified using normal n-phase voting procedures.

If the Group Services subsystem chooses not to execute 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 that is 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 execute 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 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 Errors (err_gsapi)” on page 130.

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 executing 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

Prerequisite Information

Chapter 1, “Understanding Group Services” on page 1.

Related Information

Subroutines: **ha_gs_init**, **ha_gs_join**, **ha_gs_change_state_value**, **ha_gs_send_message**, **ha_gs_subscribe**, **ha_gs_expel**, **ha_gs_change_attributes**, **ha_gs_goodbye**

ha_gs_n_phase_callback Subroutine

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

GSAPI Thread-Safe Library (**libha_gs_r.a**)

GSAPI Library (not thread-safe) (**libha_gs.a**)

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, which 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 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 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-group(s). 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_proposal_t;
```

The **gs_phase_info** field contains information about the type of protocol that is executing 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 executing 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 executing. The **gs_changing_providers** field points to a list of joining or leaving providers and this field will not change during the protocol.

HA_GS_PROPOSED_STATE_VALUE
A change to the group state value is proposed. The **gs_proposed_state_value** field points to a proposed new group state value. If providers submit 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-group(s). It is the responsibility of the target-group providers to remember it, if it is necessary for their correct operation.

HA_GS_PROPOSED_GROUP_ATTRIBUTES
The **gs_new_group_attributes** field contains the new group attributes that were proposed 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.

The **gs_current_providers** field points to a list of providers that currently belong to the group. 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 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 long           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 that 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        voluntary_or_failure;
    unsigned int        voluntary_leave_code;
} ha_gs_leave_info_t;
```

The **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 **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 **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 **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 **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 **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 executed against any providers that are being expelled. If this field contains 0, no deactivate script is executed.

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 executing. 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.

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 execute 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 executing 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 executes 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

Prerequisite Information

Chapter 1, “Understanding Group Services” on page 1.

Related Information

Subroutines: **ha_gs_init**, **ha_gs_join**, **ha_gs_change_state_value**, **ha_gs_send_message**, **ha_gs_leave**, **ha_gs_expel**, **ha_gs_change_attributes**, **ha_gs_goodbye**

ha_gs_protocol_approved_callback Subroutine

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

GSAPI Thread-Safe Library (**libha_gs_r.a**)

GSAPI Library (not thread-safe) (**libha_gs.a**)

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, which occurs after a protocol has been approved. 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 executed.

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 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-group(s). 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

Prerequisite Information

Chapter 1, “Understanding Group Services” on page 1.

Related Information

Subroutines: **ha_gs_init**, **ha_gs_join**, **ha_gs_change_state_value**, **ha_gs_send_message**, **ha_gs_leave**, **ha_gs_expel**, **ha_gs_change_attributes**, **ha_gs_goodbye**

ha_gs_protocol_rejected_callback Subroutine

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

GSAPI Thread-Safe Library (**libha_gs_r.a**)

GSAPI Library (not thread-safe) (**libha_gs.a**)

Syntax

```
#include <ha_gs.h>

void
    ha_gs_protocol_rejected_callback(
        const   ha_gs_protocol_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, which 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-group(s). 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_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.

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

Prerequisite Information

Chapter 1, “Understanding Group Services” on page 1.

Related Information

Subroutines: **ha_gs_init**, **ha_gs_join**, **ha_gs_change_state_value**, **ha_gs_send_message**, **ha_gs_leave**, **ha_gs_expel**, **ha_gs_change_attributes** **ha_gs_goodbye**

ha_gs_quit Subroutine

Purpose

ha_gs_quit – Called by a GS client to terminate its connection to the Group Services subsystem

Library

GSAPI Thread-Safe Library (**libha_gs_r.a**)

GSAPI Library (not thread-safe) (**libha_gs.a**)

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 execute 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 “ha_gs_dispatch Subroutine” on page 63.

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

Prerequisite Information

Chapter 1, “Understanding Group Services” on page 1.

Related Information

Subroutines: **ha_gs_init**, **ha_gs_join**, **ha_gs_dispatch**

ha_gs_responsiveness_callback Subroutine

Purpose

ha_gs_responsiveness_callback – A callback routine that the Group Services subsystem calls to deliver a responsiveness notification to a GS client

Library

GSAPI Thread-Safe Library (**libha_gs_r.a**)

GSAPI Library (not thread-safe) (**libha_gs.a**)

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, that will 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

Prerequisite Information

Chapter 1, “Understanding Group Services” on page 1.

Related Information

Subroutine: **ha_gs_init**, **ha_gs_expel**

ha_gs_send_message Subroutine

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

GSAPI Thread-Safe Library (**libha_gs_r.a**)

GSAPI Library (not thread-safe) (**libha_gs.a**)

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 execute 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 execute this protocol, the group's providers are notified using normal n-phase voting procedures.

If the Group Services subsystem chooses not to execute 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 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 execute 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 Errors (err_gsapi)” on page 130.

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 executing 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

Prerequisite Information

Chapter 1, “Understanding Group Services” on page 1.

Related Information

Subroutines: **ha_gs_init**, **ha_gs_join**, **ha_gs_change_state_value**, **ha_gs_leave**, **ha_gs_expel**, **ha_gs_subscribe**, **ha_gs_change_attributes**, **ha_gs_goodbye**

ha_gs_subscribe Subroutine

Purpose

ha_gs_subscribe – Called by a GS client to join a group as a subscriber

Library

GSAPI Thread-Safe Library (**libha_gs_r.a**)

GSAPI Library (not thread-safe) (**libha_gs.a**)

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. Additional information about these concepts is in the section “Host and Adapter Membership Groups” on page 42 .

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_requested;
    ha_gs_group_name_t          gs_subscription_group;
    ha_gs_subscriber_cb_t       *gs_subscription_callback;
} ha_gs_subscribe_request_t;
```

The **gs_subscription_requested** 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 flags are not exclusive and may be specified in any combination by ORing 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.

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:

- If a GS client is executing in the Group Services PSSP domain (refer to 'ha_gs_init Subroutine' on page 73 for more information), the following four system-defined groups may be specified for subscription:
- 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_CSS_MEMBERSHIP_GROUP** constant, for subscriptions for globally-consistent SP Switch adapter information
- The **HA_GS_CSSRAW_MEMBERSHIP_GROUP** constant, for subscriptions for non–globally consistent SP Switch adapter information
- If a GS client is executing within the HACMP/ES Group Services domain (refer to 'ha_gs_init Subroutine' on page 73 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
  - The **HA_GS_TMSSA_MEMBERSHIP_GROUP** constant, for subscriptions for target-mode SSA heartbeating adapter information

If the SP on which the GS client is executing does not have an SP Switch, or the SP Switch is not currently active on any nodes, a request to subscribe to **HA_GS_CSS_MEMBERSHIP_GROUP** will result in an asynchronous **HA_GS_UNKNOWN_GROUP** delayed error.

If the node on which the GS client is executing does not have an active SP Switch adapter, then 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 Appendix A, “Subscription Special Data” on page 225.)

If a GS client is executing in the Group Services PSSP domain and attempts 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 within the HACMP/ES Group Services 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 within 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_requested** 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 Errors (err_gsapi)” on page 130.

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 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_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

Prerequisite Information

Chapter 1, “Understanding Group Services” on page 1.

Related Information

Subroutines: **ha_gs_init**, **ha_gs_unsubscribe**

ha_gs_subscriber_callback Subroutine

Purpose

ha_gs_subscriber_callback – A callback routine that the Group Services subsystem calls to deliver a subscription notification to a GS client

Library

GSAPI Thread-Safe Library (**libha_gs_r.a**)

GSAPI Library (not thread-safe) (**libha_gs.a**)

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 GS client's subscriber callback routine. The GS 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 GS 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_subscription_notification_t;
```

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 resubscribe to the group. If the group does not exist because providers have not rejoined it, each subscription request receives an asynchronous error code of **HA_GS_UNKNOWN_GROUP**.

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 resubscribe to the group.

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 {
    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.

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 {
    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.

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.

For more details, see Appendix A, “Subscription Special Data” on page 225.

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

Prerequisite Information

Chapter 1, “Understanding Group Services” on page 1.

Related Information

Subroutines: **ha_gs_init**, **ha_gs_subscribe**

ha_gs_unsubscribe Subroutine

Purpose

ha_gs_unsubscribe – Called by a subscriber to unregister as a subscriber for a group

Library

GSAPI Thread-Safe Library (**libha_gs_r.a**)

GSAPI Library (not thread-safe) (**libha_gs.a**)

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 Errors (err_gsapi)” on page 130.

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

Prerequisite Information

Chapter 1, “Understanding Group Services” on page 1.

Related Information

Subroutine: **ha_gs_subscribe**, **ha_gs_init**

ha_gs_vote Subroutine

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

GSAPI Thread-Safe Library (**libha_gs_r.a**)

GSAPI Library (not thread-safe) (**libha_gs.a**)

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 and/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 executing 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 Errors (err_gsapi)” on page 130.

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**, **HA_GS_VOTE_CONTINUE**, 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 executing a protocol.

Asynchronous Errors

None.

Files

ha_gs.h

Prerequisite Information

Chapter 1, “Understanding Group Services” on page 1.

Related Information

Subroutines: **ha_gs_join**, **ha_gs_change_state_value**, **ha_gs_send_message**, **ha_gs_change_attributes**, **ha_gs_goodbye**

Chapter 3. Group Services Files Reference

This chapter contains the reference material for the error numbers and header files in the Group Services Application Programming Interface (GSAPI). The files are listed alphabetically.

GSAPI Errors (err_gsapi)

Purpose

GSAPI Errors and Return Codes – Error numbers and return codes for the Group Services Application Programming Interface (GSAPI)

Description

The Group Services Application Programming Interface (GSAPI) provides error numbers and return codes that may be returned either synchronously (that is, immediately upon return from a subroutine when it is unsuccessful) or asynchronously 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.h** header file, as follows:

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 can be 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 can be returned either synchronously or asynchronously, depending on when the error was detected.

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 that does not support the new protocol request.

This error number is returned asynchronously.

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_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 would be returned in response to an attempt to join a system-defined group such as the host membership group or an adapter membership group.

This error 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 would be returned in response to an attempt to specify as a source group a system-defined group such as the host membership group or an adapter membership group.

This error 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_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 can be returned in response to the protocol requests resulting from calls to the following subroutines: **ha_gs_change_state_value**, **ha_gs_send_message**, and **ha_gs_leave**.

HA_GS_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_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 but 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.

Related Information

GSAPI subroutines: **ha_gs_change_state_value**, **ha_gs_dispatch**, **ha_gs_expel**, **ha_gs_init**, **ha_gs_join**, **ha_gs_leave**, **ha_gs_quit**, **ha_gs_send_message**, **ha_gs_subscribe**, **ha_gs_unsubscribe**, **ha_gs_vote**, **ha_gs_change_attributes**, **ha_gs_goodbye**

GSAPI header file: **ha_gs.h**

ha_gs.h File

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 **libha_gs.a** library. Any program that uses the GSAPI subroutines must include this file, which resides in the **/usr/include** directory.

The following listing shows the contents of the **ha_gs.h** file.

```
#ifndef _HA_GS_H_
#define _HA_GS_H_
/*****************************************************************/
/*                                                               */
/* CPRY PGM                                                      */
/*                                                               */
/*  Licensed Materials - Property of IBM                         */
/*                                                               */
/*  5765-529 PSSP                                                */
/*                                                               */
/*  (C) Copyright IBM Corp. 1996 All Rights Reserved.            */
/*                                                               */
/*  US Government Users Restricted Rights - Use, duplication or disclosure */
/*  restricted by GSA ADP Schedule Contract with IBM Corp.       */
/*                                                               */
/* CPRY                                                          */
/*****************************************************************/


#ifdef __cplusplus
extern "C" {
#endif

#define HA_GS_RELEASE 4

typedef enum
{
    HA_GS_OK,
    HA_GS_OK_SO_FAR = HA_GS_OK,
    HA_GS_NOT_OK,
    HA_GS_EXISTS,
    HA_GS_NO_INIT,
    HA_GS_NAME_TOO_LONG,
    HA_GS_NO_MEMORY,
    HA_GS_NOT_A_MEMBER,
    HA_GS_BAD_CLIENT_TOKEN,
    HA_GS_BAD_MEMBER_TOKEN,
    HA_GS_BAD_PARAMETER,
    HA_GS_UNKNOWN_GROUP,
    HA_GS_INVALID_GROUP,
    HA_GS_NO_SOURCE_GROUP_PROVIDER,
    HA_GS_BAD_GROUP_ATTRIBUTES,
    HA_GS_WRONG_OLD_STATE,
    HA_GS_DUPLICATE_INSTANCE_NUMBER,
    HA_GS_COLLIDE,
    HA_GS_SOCK_CREATE_FAILED,
    HA_GS_SOCK_INIT_FAILED,
    HA_GS_CONNECT_FAILED,
    HA_GS_VOTE_NOT_EXPECTED,
    HA_GS_NOT_SUPPORTED,
    HA_GS_INVALID_SOURCE_GROUP,
    HA_GS_UNKNOWN_PROVIDER,
```

```
    HA_GS_INVALID_DEACTIVATE_PHASE,
    HA_GS_PROVIDER_APPEARS_TWICE,
    HA_GS_BACKLEVEL_PROVIDERS
}ha_gs_rc_t;                    /* Return Codes */

typedef enum
{
    HA_GS_NO_BATCHING    = 0x0000,
    HA_GS_BATCH_JOINS    = 0x0001,
    HA_GS_BATCH_LEAVES   = 0x0002,
    HA_GS_BATCH_BOTH     = 0x0003,
    HA_GS_DEACTIVATE_ON_FAILURE = 0x0004
} ha_gs_batch_ctrl_t;           /* Controls Batching of Requests */

typedef enum
{
    HA_GS_1_PHASE        = 0x0001,
    HA_GS_N_PHASE        = 0x0002
} ha_gs_num_phases_t;           /* Protocol number of Phases selection */

typedef enum
{
    HA_GS_FIRST_MERGE_TYPE,
    HA_GS_DISSOLVE_MERGE = HA_GS_FIRST_MERGE_TYPE,
    HA_GS_LARGER_MERGE,
    HA_GS_SMALLER_MERGE,
    HA_GS_DONTCARE_MERGE,
    HA_GS_LAST_MERGE_TYPE = HA_GS_DONTCARE_MERGE
} ha_gs_merge_ctrl_t;           /* Controlling Merges */

typedef enum
{
    HA_GS_NULL_VOTE,
    HA_GS_VOTE_APPROVE,
    HA_GS_VOTE_CONTINUE,
    HA_GS_VOTE_REJECT
} ha_gs_vote_value_t;           /* Allowable Vote Responses */

typedef enum
{
    HA_GS_SOCKET_NO_SIGNAL,
    HA_GS_SOCKET_SIGNAL
} ha_gs_socket_ctrl_t;          /* Socket Control */

typedef enum
{
    HA_GS_NON_BLOCKING,
    HA_GS_BLOCKING
} ha_gs_dispatch_flag_t; /* Modify behavior of ha_gs_dispatch */

typedef enum
{
    HA_GS_RESPONSIVENESS_NOTIFICATION,
    HA_GS_QUERY_NOTIFICATION,
    HA_GS_DELAYED_ERROR_NOTIFICATION,
    HA_GS_N_PHASE_NOTIFICATION,
    HA_GS_APPROVED_NOTIFICATION,
    HA_GS_REJECTED_NOTIFICATION,
    HA_GS_ANNOUNCEMENT_NOTIFICATION,
    HA_GS_SUBSCRIPTION_NOTIFICATION,
    HA_GS_MERGE_NOTIFICATION
} ha_gs_notification_type_t;    /* Identify types of notifications */

typedef enum
{
    HA_GS_RESPONSIVENESS,
    HA_GS_JOIN,
    HA_GS_FAILURE_LEAVE,
    HA_GS_LEAVE,
    HA_GS_EXPEL,
    HA_GS_STATE_VALUE_CHANGE,
    HA_GS_PROVIDER_MESSAGE,
```

```
    HA_GS_CAST_OUT,
    HA_GS_SOURCE_STATE_REFLECTION,
    HA_GS_MERGE,
    HA_GS_SUBSCRIPTION,
    HA_GS_GROUP_ATTRIBUTE_CHANGE,
    MAX_REQUEST = HA_GS_GROUP_ATTRIBUTE_CHANGE
} ha_gs_request_t;             /* Type of request a notification was for */

typedef enum
{
    HA_GS_NO_RESPONSIVENESS,
    HA_GS_PING_RESPONSIVENESS,
    HA_GS_COUNTER_RESPONSIVENESS
} ha_gs_responsiveness_type_t;       /* Type of responsiveness checking */

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

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

typedef enum
{
    HA_GS_CALLBACK_NOT_OK,
    HA_GS_CALLBACK_OK
} ha_gs_callback_rc_t;         /* Callback Return Codes */

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

typedef enum
{
    HA_GS_QUERY_ALL,
```

```
    HA_GS_QUERY_GROUP
} ha_gs_query_type_t;

typedef enum
{
    HA_GS_SUBSCRIBE_STATE                = 0x01,
    HA_GS_SUBSCRIBE_DELTA_JOINS          = 0x02,
    HA_GS_SUBSCRIBE_DELTA_LEAVES         = 0x04,
    HA_GS_SUBSCRIBE_DELTAS_ONLY          = 0x06,
    HA_GS_SUBSCRIBE_MEMBERSHIP           = 0x08,
    HA_GS_SUBSCRIBE_ALL_MEMBERSHIP       = 0x0e,
    HA_GS_SUBSCRIBE_STATE_AND_MEMBERSHIP  = 0x0f
    } ha_gs_subscription_ctrl_t;
typedef enum
{
    HA_GS_SUBSCRIPTION_STATE         = 0x01,
    HA_GS_SUBSCRIPTION_DELTA_JOIN    = 0x02,
    HA_GS_SUBSCRIPTION_DELTA_LEAVE   = 0x04,
    HA_GS_SUBSCRIPTION_MEMBERSHIP    = 0x08,
    HA_GS_SUBSCRIPTION_SPECIAL_DATA  = 0x40,
    HA_GS_SUBSCRIPTION_DISSOLVED     = 0x80,
    HA_GS_SUBSCRIPTION_GS_HAS_DIED   = 0x100
} ha_gs_subscription_type_t;

typedef int ha_gs_token_t;
typedef int ha_gs_descriptor_t;
typedef unsigned short ha_gs_time_limit_t;

#define HA_GS_MAX_GROUP_NAME_LENGTH  32
typedef char *ha_gs_group_name_t;

/* Use this name to subscribe to processor membership. */
#define HA_GS_HOST_MEMBERSHIP_GROUP "HostMembership"

#define HA_GS_ENET_MEMBERSHIP_GROUP "enMembership"
#define HA_GS_CSS_MEMBERSHIP_GROUP "cssMembership"
#define HA_GS_CSSRAW_MEMBERSHIP_GROUP "cssRawMembership"
#define HA_GS_TOKENRING_MEMBERSHIP_GROUP "trMembership"
#define HA_GS_FDDI_MEMBERSHIP_GROUP "fddiMembership"
#define HA_GS_RS232_MEMBERSHIP_GROUP "rs232Membership"
#define HA_GS_TMSCSI_MEMBERSHIP_GROUP "tmscsiMembership"
#define HA_GS_SLIP_MEMBERSHIP_GROUP "slipMembership"
#define HA_GS_ATM_MEMBERSHIP_GROUP "atmMembership"

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

const short HA_GS_node_number = -1;
const short HA_GS_instance_number = -1;

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

typedef union
{
    struct
    {
        short      _gs_instance_number;
        short      _gs_node_number;
```

```
    } _gs_provider_info;
    int gs_provider_id;
} ha_gs_provider_t;                  /* Provider ID */


typedef struct
{
    int            gs_length;
    char    *gs_state;
} ha_gs_state_value_t;         /* State Vector */

typedef struct
{
    short          gs_version;
    ha_gs_state_value_t  gs_group_state_value;
} ha_gs_group_state_t;         /* encapsulation of state vector */

typedef struct
{
    int            gs_length;
    char    *gs_message;
} ha_gs_provider_message_t;    /* provider message */

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

typedef union
{
    struct {
      ha_gs_state_value_t      *_gs_info_state;
      ha_gs_provider_t   *_gs_info_providers;
    } _gs_group_info;
    ha_gs_group_name_t   gs_groups;
} ha_gs_group_info_t;

#define gs_group_info_state     _gs_group_info._gs_info_state
#define gs_group_info_providers _gs_group_info._gs_info_providers

typedef struct
{
    ha_gs_query_type_t  gs_query_type;
    ha_gs_rc_t          gs_query_return_code;
    int                 gs_number_of_groups;
    ha_gs_group_info_t  *gs_group_info;
} ha_gs_query_info_t;

typedef struct
{
    unsigned long  gs_count;
    ha_gs_provider_t     *gs_providers;
} ha_gs_membership_t;          /* Membership List */

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

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

typedef struct
{
```

```
    unsigned long  gs_count;
    ha_gs_leave_info_t   *gs_leave_codes;
} ha_gs_leave_array_t;

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

typedef enum {
    HA_GS_ADAPTER_DEATH_ARRAY           = 0x01,
    HA_GS_CURRENT_ADAPTER_ALIAS_ARRAY   = 0x02,
    HA_GS_CHANGING_ADAPTER_ALIAS_ARRAY  = 0x04
} ha_gs_subscription_special_type_t;

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

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

typedef struct ha_gs_special_block_t {
    unsigned int                  gs_special_flag;
    struct ha_gs_special_block_t  *gs_next_special_block;
    int                           gs_special_num_entries;
    int                           gs_special_length;
    void                          *gs_special;
} ha_gs_special_block_t;

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

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

typedef struct
{
    ha_gs_notification_type_t  gs_notification_type;
    unsigned int        gs_number_of_queries;
    ha_gs_query_info_t  *gs_query_info;
} ha_gs_query_notification_t;

typedef struct
{
    ha_gs_notification_type_t  gs_notification_type;
    ha_gs_token_t         gs_provider_token;
    ha_gs_request_t       gs_protocol_type;
    ha_gs_summary_code_t  gs_summary_code;
    ha_gs_time_limit_t    gs_time_limit;
```

```
    ha_gs_proposal_t           *gs_proposal;
} ha_gs_n_phase_notification_t;

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

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

typedef struct
{
    ha_gs_notification_type_t  gs_notification_type;
    ha_gs_token_t         gs_provider_token;
    ha_gs_summary_code_t  gs_summary_code;
    ha_gs_membership_t    *gs_announcement;
} ha_gs_announcement_notification_t;

typedef struct
{
    ha_gs_notification_type_t  gs_notification_type;
    ha_gs_token_t        gs_provider_token;
    ha_gs_request_t      gs_protocol_type;
    ha_gs_proposal_t          *gs_proposal;
    ha_gs_merge_ctrl_t   gs_merge_control;
    ha_gs_group_state_t  gs_alpha_group_state;
    ha_gs_group_state_t  gs_omega_group_state;
} ha_gs_merge_notification_t;

typedef struct
{
    ha_gs_notification_type_t  gs_notification_type;
    ha_gs_token_t         gs_subscriber_token;
    ha_gs_subscription_type_t  gs_subscription_type;
    ha_gs_state_value_t   *gs_state_value;
    ha_gs_membership_t    *gs_full_membership;
    ha_gs_membership_t    *gs_changing_membership;
    ha_gs_special_data_t  *gs_subscription_special_data;
} ha_gs_subscription_notification_t;

typedef void (ha_gs_subscription_cb_t)(const ha_gs_subscription_notification_t*);

typedef void (ha_gs_query_cb_t)(const ha_gs_query_notification_t*);

typedef ha_gs_callback_rc_t (ha_gs_responsiveness_cb_t)(const ha_gs_responsiveness_notification_t*);

typedef void (ha_gs_n_phase_cb_t)(const ha_gs_n_phase_notification_t*);

typedef void (ha_gs_approved_cb_t)(const ha_gs_approved_notification_t*);

typedef void (ha_gs_rejected_cb_t)(const ha_gs_rejected_notification_t*);

typedef void (ha_gs_announcement_cb_t)(const ha_gs_announcement_notification_t*);

typedef void (ha_gs_merge_cb_t)(const ha_gs_merge_notification_t*);

typedef     struct {
    ha_gs_group_attributes_t  *gs_group_attributes;
    short                     gs_provider_instance;
    char                      *gs_provider_local_name;
    ha_gs_n_phase_cb_t        *gs_n_phase_protocol_callback;
```

```
    ha_gs_approved_cb_t       *gs_protocol_approved_callback;
    ha_gs_rejected_cb_t       *gs_protocol_rejected_callback;
    ha_gs_announcement_cb_t   *gs_announcement_callback;
    ha_gs_merge_cb_t               *gs_merge_callback;
} ha_gs_join_request_t;

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

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

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

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

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

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

#define gs_join_request              _gs_protocol_info._gs_join_request
#define gs_state_change_request      _gs_protocol_info._gs_state_change_request
#define gs_message_request           _gs_protocol_info._gs_message_request
#define gs_leave_request             _gs_protocol_info._gs_leave_request
#define gs_expel_request             _gs_protocol_info._gs_expel_request
#define gs_subscribe_request         _gs_protocol_info._gs_subscribe_request
#define gs_attribute_change_request  _gs_protocol_info._gs_attribute_change_request

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

typedef struct
{
    ha_gs_notification_type_t  gs_notification_type;
    ha_gs_token_t          gs_request_token;
    ha_gs_request_t        gs_protocol_type;
    ha_gs_rc_t             gs_delayed_return_code;
```

```
    ha_gs_proposal_info_t  *gs_failing_request;
} ha_gs_delayed_error_notification_t;

typedef void (ha_gs_delayed_error_cb_t)(const ha_gs_delayed_error_notification_t*);

ha_gs_rc_t ha_gs_init(ha_gs_descriptor_t *,
                  const ha_gs_socket_ctrl_t,
                  const ha_gs_responsiveness_t *,
                  const char *,
                  ha_gs_responsiveness_cb_t*,
                  ha_gs_delayed_error_cb_t*,
                  ha_gs_query_cb_t*);

ha_gs_rc_t ha_gs_dispatch(const ha_gs_dispatch_flag_t);
ha_gs_rc_t ha_gs_join(ha_gs_token_t *,
                  const ha_gs_proposal_info_t *);

ha_gs_rc_t ha_gs_change_state_value(ha_gs_token_t,
                            const ha_gs_proposal_info_t *);

ha_gs_rc_t ha_gs_send_message(ha_gs_token_t,
                               const ha_gs_proposal_info_t *);

ha_gs_rc_t ha_gs_leave(ha_gs_token_t,
                   const ha_gs_proposal_info_t *);

ha_gs_rc_t ha_gs_expel(ha_gs_token_t,
                   const ha_gs_proposal_info_t *);

ha_gs_rc_t ha_gs_change_attributes(ha_gs_token_t,
                                  const ha_gs_proposal_info_t *);

ha_gs_rc_t ha_gs_goodbye(ha_gs_token_t);

ha_gs_rc_t ha_gs_vote(ha_gs_token_t,
                  ha_gs_vote_value_t,
                  const ha_gs_state_value_t *,
                  const ha_gs_provider_message_t *,
                  ha_gs_vote_value_t);

ha_gs_rc_t ha_gs_quit(void);

ha_gs_rc_t ha_gs_query_group_list(void);

ha_gs_rc_t ha_gs_query_group_info(const ha_gs_group_name_t);

ha_gs_rc_t ha_gs_subscribe(ha_gs_token_t *,
                           const ha_gs_proposal_info_t *);

ha_gs_rc_t ha_gs_unsubscribe(ha_gs_token_t);

void    ha_gs_copy_group_attributes(ha_gs_group_attributes_t *gAttrsTarg,
                                    ha_gs_group_attributes_t *gAttrsSrc);

#ifdef __cplusplus
}                                       /* end extern "C" */
#endif

#endif                                  /* _HA_GS_H_ */
```

Related Information

GSAPI subroutines: **ha_gs_change_state_value**, **ha_gs_dispatch**, **ha_gs_init**, **ha_gs_expel**, **ha_gs_join**, **ha_gs_leave**, **ha_gs_quit**, **ha_gs_send_message**, **ha_gs_subscribe**, **ha_gs_unsubscribe**, **ha_gs_vote**, **ha_gs_change_attributes**, **ha_gs_goodbye**

Chapter 4. Using the GSAPI: A Group Services Client Example

This chapter contains listings of the files required by a sample Group Services client (GS client) program called **sample_schg.c**. It also contains two sample deactivate scripts.

The **sample_schg.c** program, as a simple GS client, 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 on input
- Continues running until the program is killed (through the **kill** command or by otherwise sending it a signal).

For details on setting up the environment and starting the program on several nodes, see the comments in the program listing.

The program requires the following files:

- "The sample_schg.c Sample Program" on page 144 contains the **main()** function, all of the necessary callback functions, and some utility functions for the application.
- "The sample_utility.c Utility Functions" on page 162 contains utility functions that are used by the **sample_schg** program.
- "The sample_utility.h Header File" on page 215 contains declarations for the utility functions in the **sample_utility.c** program.
- "The sample_deactive_ksh.sh Deactivate Script" on page 218 is a deactivate script written as a Korn shell script.
- "The sample_deactive_c_prog.c Deactivate Script" on page 221 is a deactivate script written as a C program.

You can find the **sample_schg.c** program and its related files online in the PSSP product samples directory, **/usr/sbin/rsct/samples/hags**. The **hags** sample directory also contains files related to two other test programs, **sample_test.c**, which provides an interactive interface to the Group Services subsystem, and **Sample_Subscribe.C**, which provides a sample C++ language client that subscribes with Group Services for a user-specified group.

The sample_schg.c Sample Program

```
/*******************************************************************/
/*                                                                 */
/*CPRY                                                             */
/*                                                                 */
/* Licensed Materials - Property of IBM                            */
/*                                                                 */
/* 5765-529 PSSP                                                   */
/*                                                                 */
/* (C) Copyright IBM Corporation 1996 All Rights Reserved.         */
/*                                                                 */
/* US Government Users Restricted Rights -                         */
/* Use, duplication or disclosure restricted by                    */
/* GSA ADP Schedule Contract with IBM Corp.                        */
/*                                                                 */
/*CPRY                                                             */
/*******************************************************************/

static char *sccsid = "@(#)92   1.2
src/rsct/pgs/samples/sample_schg.c,
gssamples, rsct_rtro, rtrot1fx 6/27/96 16:39:39";
#if !defined(_HAGSD_COPYRIGHT_H)
#define _HAGSD_COPYRIGHT_H
static char copyright[] = "Licensed Materials - Property of IBM\n\
5765-529 (C) Copyright IBM Corp. 1996.  All Rights Reserved.\n\
US Government Users Restricted Rights - Use, duplication or \n\
disclosure restricted by GSA ADP Schedule Contract with IBM Corp.\n";
#endif

/*******************************************************************/
/*
 * Name:  sample_schg.c
 *
 * This module provides a non-interactive program that can be used as
 * an example for constructing an application that exploits the Group
 * Services interfaces provided by IBM PSSP.
 *
 * Components:
 *   sample_schg.c - is the bulk of the program, and contains the main()
 *      function, as well as all necessary callback functions, and some
 *      internal utility functions for the application.
 *
 *   sample_utility.c - provides the definitions for the utility functions
 *      used by the sample_test and sample_schg programs.
 *
 *   sample_utility.h - declarations for the utility functions contained
 *      in sample_utility.c
 *
 * This program is intended as a "simple" Group Services application.
 * When started, it will:
 *
 *   - read and validate the command-line arguments to set various
 *      options, such as group name and timing options.
 *   - attempt to initialize itself as a Group Services client.
 *   - once initialized, attempt to join the group as a provider.
 *   - once joined, each provider will use the timing control given
 *      and will propose a group state value change at each desired
 *      time interval.
 *   - this continues until the program is killed (via kill command,
 *      or otherwise sending it a signal.)
 *
 * The group name, protocol controls, and the various timing aspects
 * of the program are controllable via command-line arguments.
 *
 * This program will write out informational messages as it goes along
 * on stdout.  It is recommended to run this program in the background,
 * and to redirect stdout to a file, and monitor the program by
 * monitoring the file.
 */
/*******************************************************************/
```

```
/*********************************************************************/
/*
 * To build this program, use the Makefile in this directory:
 *
 *      make all (to build all sample programs)
 * or
 *      make sample_schg
 *
 * To execute this program, you need to have Group Services active on
 * your system.  Please refer to the manual if you have questions as to
 * how you may verify this.  Additionally, you need to execute as a
 * privileged (root) process.  Also, you should take care in the group
 * names you use, to avoid clashing with other groups.
 *
 * Prior to starting this program, you need to set the name of your SP
 * partition in the environment of the process running the program.
 * You should export the variable HA_SYSPAR_NAME to be the name of the
 * partition.  Contact your system administrator if you do not know how
 * to determine this.
 *
 * Once all this is done, you can then start up copies of this program
 * on as many nodes as desired.  All of those with the same group name
 * will attempt to join the same group.  For these, you should ensure
 * that any specified protocol control arguments match, to ensure that
 * the group attributes all match.  If this is not done, then not all
 * copies of the program will be allowed to join the group.
 */
/*********************************************************************/

/*********************************************************************/
/*
 * Command-line arguments:
 *
 *  -g <groupname> -- name of the group this process should join.
 *   Default is "SampleGroup01".
 *
 *  -t <time interval (in seconds)> -- number of seconds between attempts
 *      to submit a state value change proposal.
 *   Default is 10.
 *
 *  -P <n-phase pctg> -- percentage of the state value change proposals
 *      that should be n-phase.  Remainder will be 1-phase.
 *   Default is 0 (i.e., all state value change proposals will be 1-phase.)
 *
 *  -T <state time limit> -- voting time limit for n-phase state value
 *      change proposals.
 *   Default is 0.
 *
 *  -r -- randomize the time between making state value change proposals?
 *      If this is given, then the actual time interval between proposals
 *      will use a value inclusive of 1 second as the minimum, and (2 * the
 *      base time interval (specified by the '-t' flag)) as the maximum.
 *   Default is to not randomize.
 *
 *  -s <seed> -- random number seed.  If not specified, then the program
 *      will use the TOD clock to get a seed.  This allows you to set up
 *      "random" runs with recreatable series of proposal timing *for any
 *      one process*.  Note that if you are running multiple providers in
 *      a group, due to variations in message timing, there is no guarantee
 *      that the overall group timing and protocol execution is guaranteed
 *      to be recreated.
 *
 *  -j <join/failure num phases> -- number of phases (1-phase or n-phase)
 *      for join and failure leave protocols.  Specify '1' (the default) to
 *      specify 1-phase protocols, anything else for n-phase protocols.
 *   Default is 1-phase.
 *
 *  -p <provider instance number> -- provider instance number for this
 *      process.
 *   Default is 1.
 *
```

```
 *  -f <log print frequency> -- how many times per hour should a log status
 *      message be written, assuming debug level (see -d flag) is 1.
 *   Default is 2 (i.e., every 30 minutes write a status report message.)
 *
 *  -n <notify script> -- path name to "notify" script.  Used by the internal
 *      notify() routine to attempt to send information to interested users.
 *   Default is /usr/lpp/ssp/samples/hags/notify.
 *
 *  -d <debug level> -- amount of data to dump out.  The program will
 *      write out join and failure notifications, and using the '-f'
 *      flag, periodic status reports (e.g., number of protocols submitted
 *      and executed, etc.)  The debug settings:
 *      1 -- default, as above.
 *      2 -- as above, plus every protocol submission.
 *      3 -- as for 1 & 2, plus write out every notification received in
 *           full.
 *   Default is 1.
 */
/*********************************************************************/

/*********************************************************************/
/*
 * Include "standard" system files.  Note that pthread.h must be the
 * first file included, if it is to be included (see the standard AIX
 * documentation for more information about AIX thread support.)
 */
/*********************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <sys/select.h>
#include <sys/time.h>
#define _XOPEN_EXTENDED_SOURCE          /* AIX 4.1 */
#include <arpa/inet.h>
#undef _XOPEN_EXTENDED_SOURCE           /* AIX 4.1 */
#include <signal.h>
#include <errno.h>
#include <strings.h>
#include <memory.h>
#include <time.h>

/*********************************************************************/
/*
 * Include the Group Services declarations file.
 */
/*********************************************************************/

#include <ha_gs.h>

/*********************************************************************/
/*
 * Include the set of declarations for external utility functions for
 * this program.
 */
/*********************************************************************/

#include "sample_utility.h"

/*********************************************************************/
/*
 * The set of callback functions used by this program.
 */
void n_phase_cb(const ha_gs_n_phase_notification_t *note);
void approved_cb(const ha_gs_approved_notification_t *note);
void rejected_cb(const ha_gs_rejected_notification_t *note);
void my_delayed_error_cb(const ha_gs_delayed_error_notification_t *note);
void announce_cb(const ha_gs_announcement_notification_t *note);

void query_cb(const ha_gs_query_notification_t *note);
void delayed_error_cb(const ha_gs_delayed_error_notification_t *note);
ha_gs_callback_rc_t responsive_cb(const ha_gs_responsiveness_notification_t *note);
```

```
/*********************************************************************/
/*
 * Prepare a protocol proposal for submission.
 */
int     submit_state_change_proposal(void);

/*********************************************************************/
/*
 * Internal utility functions.
 */
void    notify( char *message );
int     get_node_number(void);
void    print_usage(void);

/*********************************************************************/
/*
 * Global variables.
 */

/*
 * Keep statistics.
 */
/*
 * Number of state value change proposals submitted, number that were
 * 1-phase/n-phase.
 */
int   nSubmitted, nSubmitted1Phase, nSubmittedNPhase;
/*
 * Number of our proposals that were returned due to collisions.
 */
int   nCollided, nSynchCollides;
/*
 * Number of state value change protocols that have executed in the
 * group and that we have participated in.
 */
int     nStateChangesSeen, nStateChanges1Phase, nStateChangesNPhase;

/*
 * Keep track of group membership, state value.
 */
int     state;                          /* Current group state value, from the */
                                        /*  last successfully-executed protocol. */
int     proposed_state;                 /* Our to-be-proposed state value. */
int     joined;                         /* Have we joined the group yet? */
struct  timeval proposal_time;          /* Keep track of time last proposal */
                                        /*  submitted. */

/*
 * These are the variables that hold values determined by the various
 * command-line flags.  Most are initialized here, but may be overridden
 * once we parse the command line.
 */
char   *theGroupName = "SampleGroup01"; /* -g flag: name of the group we should */
                                        /*  join. */

int     testInterval = 10;              /* -t flag: number of seconds between */
                                        /*  making state value change proposals. */
                                        /*  Used as time limit for select() call. */

int     providerInstanceNumber = 1;   /* -p flag: provider instance number for */
                                        /*  this process. */

int     debug = 1;                      /* -d flag: debug level.  Higher dumps */
                                        /* more info. */

int     freq = 2;                       /* -f flag: how many times per hour to */
                                        /*  print out status messages to log? */
int     printfreq;                      /* How often to print log msg? */
                                        /*  calculated as number of log prints/hour */

int     randomize = 0;                  /* -r flag: should the wait time between */
                                        /*  making state value change proposals */
```

```
                                        /*  be randomized or fixed? */

int     nphasestates = 0;               /* -P flag: percentag of time to make our */
                                        /*  state value change proposals n-phase. */
                                        /*  Remainder will be 1-phase. */

int     stateTimeLimit = 0;             /* -T flag: voting time limit for n-phase */
                                        /*  state value change protocols. */

ha_gs_num_phases_t jfphases = HA_GS_1_PHASE; /* -j flag: number of phases for */
                                             /*  join and failure protocols. */

/*
 * The "notify" script is a small shell script that can be modified
 * to mail important error/status messages to an interested user.
 * If you want to place this script elsewhere, you can dynamically
 * specify the path name via the '-n' command-line flag.
 *
 * The script is used by the notify() routine in this program.
 */
char   *theNotifyScript = "/usr/sbin/rsct/samples/hags/notify"
;

/*
 * The following are set during our join protocol.
 */
ha_gs_provider_t        ourProviderId;  /* Our provider ID, returned asynchronously */
                                        /*  when our join protocol executes. */

ha_gs_token_t           provider_token; /* Our provider token, returned */
                                        /*  synchronously by ha_gs_join(). */

/*
 * The arguments, program name, handy buffer.
 */
char   *args;
char   *progname;
char    buffer[256];

/*********************************************************************/
/*
 * Main() function.  Parse command-line arguments, initialize with Group
 * Services, join the group, then drop into the while() loop until we
 * get killed.  Use select() to wait for messages from Group Services,
 * and propose state value changes every so often.  Simple as that.
 */
int main(int argc, char ** argv)
{
    ha_gs_descriptor_t     socket_fd;
    ha_gs_responsiveness_t responsiveness = {HA_GS_PING_RESPONSIVENESS,
                                             3600,
                                             10,
                                             (char *)0,
                                             0};
    char               *script="/dev/null";
    ha_gs_rc_t          rc;
    ha_gs_proposal_info_t info;
    char                input;
    char               *function;
    time_t         overdue;
    char                theArgs[1024];

    int         i, found;
    int         phase;
    int           init_tries;
#define           MAX_TRIES   10
    int         argCtr;
    int         seedGiven;
    uint        randSeed;
    struct timeval  current_time;
    struct timezone tz;
    float       p;
```

```
/* select stuff */
int         select_rc;
int         highestDescriptor;
int         howMany;
fd_set      socketsForSelect; /* Maintain all registered sockets in mask. */
fd_set      socketSelectMask; /* Used for the actual select. */
struct      timeval nextJob;  /* Wait time for select. */

seedGiven = 0;

/*
 * Parse the command-line arguments.  Copy them to "args" to preserve
 * them so we can easily write them to the log.
 */
progname = argv[0];
args = theArgs;
args[0] = '\0';
for(argCtr=1;argCtr < argc;argCtr++) {
    if(!strcmp( argv[argCtr], "-g")) {
        theGroupName = argv[++argCtr];
        strcat(args, " -g ");
        strcat(args, theGroupName);
    } else if(!strcmp( argv[argCtr], "-n")) {
        theNotifyScript = argv[++argCtr];
        strcat(args, " -n ");
        strcat(args, theNotifyScript);
    } else if(!strcmp( argv[argCtr], "-t")) {
        testInterval = atoi( argv[++argCtr]);
        strcat(args, " -t ");
        strcat(args, argv[argCtr]);
    } else if (!strcmp(argv[argCtr], "-p")) {
        providerInstanceNumber = atoi( argv[++argCtr]);
        strcat(args, " -p ");
        strcat(args, argv[argCtr]);
    } else if (!strcmp(argv[argCtr], "-d")) {
        debug = atoi( argv[++argCtr]);
        strcat(args, " -d ");
        strcat(args, argv[argCtr]);
    } else if (!strcmp(argv[argCtr], "-j")) {
        phase = atoi( argv[++argCtr]);
        if (1 == phase) {
            jfphases = HA_GS_1_PHASE;
        } else {
            jfphases = HA_GS_N_PHASE;
        }
        strcat(args, " -j ");
        strcat(args, argv[argCtr]);
    } else if (!strcmp(argv[argCtr], "-f")) {
        freq = atoi( argv[++argCtr]);
        strcat(args, " -f ");
        strcat(args, argv[argCtr]);
    } else if (!strcmp(argv[argCtr], "-s")) {
        randSeed = atoi( argv[++argCtr]);
        seedGiven = 1;
        strcat(args, " -s ");
        strcat(args, argv[argCtr]);
    } else if (!strcmp(argv[argCtr], "-r")) {
        randomize = 1;
        strcat(args, " -r ");
    } else if (!strcmp(argv[argCtr], "-P")) {
        nphasestates = atoi( argv[++argCtr] );
        strcat(args, " -P ");
        strcat(args, argv[argCtr]);
    } else if (!strcmp(argv[argCtr], "-T")) {
        stateTimeLimit = atoi( argv[++argCtr] );
        strcat(args, " -T ");
        strcat(args, argv[argCtr]);
    } else {
        print_usage();
        exit(3);
    }
```

```
}

/*
 * Calculate time between logging via determining number of state changes per hour
 * divided by number of log entries to be made per hour.  Ensure no less than one.
 * Use this to then cut log entry every "printfreq" state changes.
 */
if (0 >= freq) {
    printfreq = 1;
} else {
    printfreq = ((3600 / testInterval) / freq);
    if (0 >= printfreq) {
        printfreq = 1;
    }
}

/*
 * If no random seed given (-s) then use the (TOD seconds * node_number).
 * In any case, print out the seed being used in case we want to use it
 * again, whatever it is.
 */
if (!seedGiven) {
  gettimeofday(&current_time, &tz);
    randSeed = current_time.tv_sec * get_node_number();
}
srandom(randSeed);
printf("Random seed is [%d]\n", randSeed);
fflush(stdout);

/*
 * This variable tells us how many seconds to wait between making each
 * state value change proposal.  Note that we may override this later,
 * if we are "randomizing" this time interval.
 */
nextJob.tv_sec = testInterval;
nextJob.tv_usec = 0;

/*
 * No sockets yet.
 */
highestDescriptor = 0;
FD_ZERO(&socketsForSelect);

/*
 * This outer while() loop is used in case we have to reconnect
 * to Group Services.
 *
 * In this loop, try to initialize (ha_gs_init()).  Once we
 * do that, set up our select() handling information (socket
 * file descriptor, etc.).
 *
 * Then, attempt to join the group.
 *
 * After that, fall into the inner loop, which is basically just a
 * select(), where we wait to either receive messages from Goup
 * Services, or time out when it is time to make our next state value
 * change proposal.
 */
while(1) {

  joined = 0;                    /* Not yet joined to the group. */
  state = 0;                     /* Start with a state value of 0. */
  proposal_time.tv_sec = 0;      /* No outstanding proposal. */

  /*
     * Attempt to initialize with Group Services.
     */
  for( init_tries = 1; MAX_TRIES > init_tries; init_tries++ ) {
       rc = ha_gs_init(&socket_fd,
                       HA_GS_SOCKET_NO_SIGNAL,
                       &responsiveness,
                       script,
```

```
                    responsive_cb,
                    my_delayed_error_cb,
                    query_cb);
      /*
       * Set up to use the select() system call.
       */
      if (HA_GS_OK == rc) {
          FD_SET(socket_fd, &socketsForSelect);
          if (socket_fd > highestDescriptor) {
              highestDescriptor = socket_fd;
          }
          break;
      } else if (HA_GS_EXISTS == rc) {
          printf("You have already initialized!  Why do it again?\n");
          break;
      } else {
          printf("Bad news - ha_gs_init() returned rc:[%s]\n", write_an_rc(rc));
          sleep(10);
      }
  }
if ( MAX_TRIES <= init_tries ) {
      sprintf(buffer, "ABORT \"Can't get started after %d tries!\n\"", init_tries );
      notify( buffer );
      exit(rc);
}
  fflush(stdout);

  /*
   * Prepare group attributes, join/failure phases set by
   * command-line 'j' argument.
   */
gattr[1] = malloc(sizeof(ha_gs_group_attributes_t));
gattr[1]->gs_version = 1;
gattr[1]->gs_sizeof_group_attributes = sizeof(ha_gs_group_attributes_t);
gattr[1]->gs_client_version = 1;
gattr[1]->gs_batch_control = HA_GS_BATCH_BOTH;
gattr[1]->gs_num_phases = jfphases;
gattr[1]->gs_source_reflection_num_phases = HA_GS_1_PHASE;
gattr[1]->gs_group_default_vote = HA_GS_VOTE_APPROVE;
gattr[1]->gs_merge_control = HA_GS_DISSOLVE_MERGE;
gattr[1]->gs_time_limit = 0;
gattr[1]->gs_source_reflection_time_limit = 0;
gattr[1]->gs_group_name = theGroupName;
gattr[1]->gs_source_group_name = NULL;
  instance_numbers[1] = providerInstanceNumber;

write_join_information(1);

info.gs_join_request.gs_group_attributes = gattr[1];
  info.gs_join_request.gs_provider_instance = providerInstanceNumber;
info.gs_join_request.gs_provider_local_name = "SampleProvider";
info.gs_join_request.gs_n_phase_protocol_callback = n_phase_cb;
info.gs_join_request.gs_protocol_approved_callback = approved_cb;
info.gs_join_request.gs_protocol_rejected_callback = rejected_cb;
info.gs_join_request.gs_announcement_callback = announce_cb;
info.gs_join_request.gs_merge_callback = NULL;

  /*
   * Attempt to join the group.
   */
rc = ha_gs_join(&gid[1], &info);

  /*
   * Save the provider token returned by ha_gs_join().
   */
provider_token = gid[1];

  printf("ha_gs_join returned rc:[%s]\n", write_an_rc(rc));

  rc = HA_GS_OK;

  /*
```

```
 * This inner while() loop will simply wait on select() for
 * notifications to arrive, and when ready, will attempt to
 * submit a state value change proposal.
 */
while (HA_GS_NOT_OK != rc) {

    /*
     * Determine the random interval until we should attempt to
     * make the our next state value change proposal.
     */
    if ( randomize ) {
        nextJob.tv_sec = randomn( 2 * testInterval );
    }

    /*
     * select().  See AIX documentation.
     *
     * The "nextJob" parameter is at timeval struct.  It is set to
     * the number of seconds until our next state value change
     * proposal should be made.  If no input appears on our socket
     * connection from Group Services, then we will wake up and
     * go to make the proposal.
     */
    memcpy(&socketSelectMask, &socketsForSelect, sizeof(socketsForSelect));
    select_rc = select(highestDescriptor + 1,
                       &socketSelectMask,
                       0,
                       0,
                       &nextJob);

    /*
     * If anything other than "select interrupted (EINTR)" just exit.
     */
    if (select_rc < 0) {
        if (errno == EINTR) {
            printf("Got EINTR during the select.\n");
            continue;
        } else {
            perror("Error on select");
            exit(errno);
        }
    } else if (0 < (howMany = NFDS(select_rc))) {
        /*
         * Input arrived on socket.  Should only ever be our one
         * socket.
         */
        if (1 < howMany) {
            printf("Input on more than our socket??  Have [%d]!\n",
                   howMany);
            exit(howMany);
        }
        /*
         * OK so far, call ha_gs_dispatch() and process the message(s).
         */
     rc = ha_gs_dispatch(HA_GS_NON_BLOCKING);
        if (HA_GS_OK != rc) {
            printf("Bad news, bad return code from dispatch[%s]!  Exiting.\n",
                   write_an_rc(rc));

            /*
             * Something is wrong.  Drop out of inner loop, and then
             * reinitialize and start over.
             */
            break;
        }
    } else {
        if (!joined) {
            /*
             * We have not yet seen the "approved" notification for our join
             * request, so we cannot yet handle any other proposals.  Just
             * wait some more.
             */
```

```
                    printf("Select timed out, not joined, nothing arrived in [%d] seconds.  Try again.\n",
                           nextJob.tv_sec);
                } else {
                    /*
                     * We are part of the group, so it is time to make a
                     * proposal.  Go do it.
                     */
                    submit_state_change_proposal();
                }
                fflush(stdout);
            }
        }                               /* end while(HA_GS_NOT_OK != rc) */

        /*
         * Close current connection to Group Services.
         */
        rc = ha_gs_quit();

        printf("ha_gs_quit returned rc:[%s]\n", write_an_rc(rc));

        if ( HA_GS_OK != rc ) {
            sprintf(buffer,
                    "ERROR \"ha_gs_quit returned rc:[%s]\n\"",
                    write_an_rc(rc));
            notify( buffer );
        }

        /*
         * Need to remove the "old" socket before establishing a new one.
         */
        FD_CLR(socket_fd, &socketsForSelect);
    }                                   /* end while(1) */
}                                       /* end main() */

/*********************************************************************/
/*
 * Our timer has popped, and is time to submit a state value change
 * proposal for our group.  Determine whether we should submit a 1-phase
 * or n-phase proposal, optionally display the current counts, and then
 * prepare the data to call ha_gs_change_state_value().
 *
 * The data submitted will be the 'current group state value + 1'.
 *
 * Note that if there are multiple providers in the group, then it is
 * possible for our proposal to be returned with a delayed error due to
 * a protocol collision (delayed error code of HA_GS_COLLIDE).  That is
 * fine, as that means another provider's proposal will have executed
 * to increment the state value.  We worry about that later, in the
 * delayed error callback (my_delayed_error_cb()).
 */
int submit_state_change_proposal(void)
{
    ha_gs_proposal_info_t info;
    ha_gs_num_phases_t    phases;
    ha_gs_token_t         group_token;

    int         rc;
    struct      timezone  tz;
    int         nphase;

    proposed_state = state + 1;         /* Proposed new state value. */

    /*
     * Use random number to determine how many phases.
     */
    if (randomn(100) > nphasestates) {
        phases = HA_GS_1_PHASE;
        nSubmitted1Phase++;
    } else {
        phases = HA_GS_N_PHASE;
        nSubmittedNPhase++;
    }
```

```
    gettimeofday(&proposal_time, &tz);

    /*
     * We do not normally display every proposal, but instead just "every
     * so often".  However, if 'debug' is set, then always display.
     */
    if((2 <= debug) ||
       (0 == (nStateChangesSeen % printfreq)))
        printf("state changes: submitted/seen(1-/n-phase) %d(%d/%d)/%d(%d/%d) collided %d.\n",
               nSubmitted,
               nSubmitted1Phase,
               nSubmittedNPhase,
               nStateChangesSeen,
               nStateChanges1Phase,
               nStateChangesNPhase,
               nCollided);
    if((2 <= debug) ||
       (0 == (nStateChangesSeen % printfreq)))
        printf(" submit %s state change to %d on %s",
               ((HA_GS_1_PHASE == phases) ? "1-phase" : "N-phase"),
               proposed_state,
               ctime( (time_t *) &proposal_time.tv_sec ) );

    /*
     * Prepare the proposal data for the call to ha_gs_change_state_value().
     */
    info.gs_state_change_request.gs_num_phases = phases;
    info.gs_state_change_request.gs_time_limit = stateTimeLimit;
    info.gs_state_change_request.gs_new_state.gs_length = sizeof(int);
    info.gs_state_change_request.gs_new_state.gs_state =
        (char *) & proposed_state;
    nSubmitted++;

    /*
     * Submit the request to the Group Services library code.
     */
    rc = ha_gs_change_state_value( provider_token, &info );

    /*
     * HA_GS_OK says that the proposal was accepted, but does NOT necessarily
     * indicate that it will be executed (it may be returned asynchronously
     * via a delayed error).  But, HA_GS_OK does indicate that there were no
     * syntactical problems with the proposal.
     *
     * At this point, HA_GS_COLLIDE indicates that another protocol is already
     * executing in the group.  It is rather rare to have timing such that we
     * get this, but, since we have providers distributed on many nodes, it is
     * possible.
     *
     * Any other return codes here indicate serious problems with the program...
     */
    if((3 <= debug) ||
       ((HA_GS_OK != rc) &&
        (HA_GS_COLLIDE != rc))) {
        sprintf(buffer,
                "ha_gs_change_state_value returned rc [%s]!\n\"",
                write_an_rc(rc));
        notify(buffer);
        printf("ha_gs_change_state_value returned rc:[%s]\n",
               write_an_rc(rc));
        if (HA_GS_COLLIDE == rc) {
            nCollided++;
            nSynchCollides++;
        }
    } else if (HA_GS_COLLIDE == rc) {
        nCollided++;
        nSynchCollides++;
    }
    return( rc );
}
```

```
/*********************************************************************/
/*
 * The callback functions used by this program.
 */

/*
 * This callback is executed when we need to "vote" on an n-phase protocol
 * proposal.  This may be a join, failure leave, or more commonly, a state
 * value change proposal.
 *
 * For simplicity, we always just vote APPROVE here.  Various experiments
 * here would be to add cases where CONTINUE or REJECT may be voted in
 * different cases.  Exercises left to the reader.
 *
 * Note that this would be the routine where we would put "actions" that
 * a "real" recovery program may want to execute.  It would perform those
 * actions here, and vote based upon the result of getting them done.
 * Additional actions may be taken in the approved and rejected callback
 * functions, but if we want the subsystem to perform a synchronized set
 * of steps, we would do it here.
 */
void n_phase_cb(const ha_gs_n_phase_notification_t *note)
{
    ha_gs_request_t pType;
    ha_gs_token_t   noteToken;
    ha_gs_rc_t      rCode;
    int             calledNotify = 0;

    /*
     * Grab the provider token, and the type of the protocol.
     */
    noteToken = note->gs_provider_token;
    pType = note->gs_protocol_type;

    /*
     * Process based on protocol type.  In all cases, we just want to vote
     * by calling ha_gs_vote(), and getting out.  Only interesting thing to
     * do is if we get any sort of error on the vote call.
     */
    switch(pType) {

        /*
         * Just approve any joins or failures.  Nothing else to bother doing
         * here.  We have more work to do when the approval notification for
         * a join arrives.  See the approved_cb().
         */
      case HA_GS_JOIN:
      case HA_GS_FAILURE_LEAVE:
        if (HA_GS_OK != (rCode = ha_gs_vote(noteToken,
                                            HA_GS_VOTE_APPROVE,
                                            NULL,
                                            NULL,
                                            HA_GS_NULL_VOTE))) {
            sprintf(buffer,
                    "Strange error code [%s] attempting to approve join/failure!\n\"",
                    write_an_rc(rCode));
            notify(buffer);
            write_the_notification(1, (void *)note, HA_GS_N_PHASE_NOTIFICATION);
            calledNotify = 1;
            printf("Strange error code from approve join/failure ha_gs_vote [%s]\n",
                   write_an_rc(rCode));
        }
        break;

        /*
         * Again, just vote approve.
         */
      case HA_GS_STATE_VALUE_CHANGE:
        if (HA_GS_OK != (rCode = ha_gs_vote(noteToken,
                                            HA_GS_VOTE_APPROVE,
                                            NULL,
                                            NULL,
```

```
                                        HA_GS_NULL_VOTE))) {
            sprintf(buffer,
                    "Strange error code [%s] attempting to approve state change!\n\"",
                    write_an_rc(rCode));
            notify(buffer);
            write_the_notification(1, (void *)note, HA_GS_N_PHASE_NOTIFICATION);
            calledNotify = 1;
            printf("Strange error code from approve state change ha_gs_vote [%s]\n",
                   write_an_rc(rCode));
        }
        if ((3 > debug) && (!calledNotify)) {
            calledNotify = 1;           /* Do not print these, unless error or high debug. */
        }
        break;

        /*
         * If this should happen, NOW we want to vote REJECT.  As currently
         * written, we will never propose any other protocols (provider-broadcast
         * messages, voluntary leaves) so just reject and get out.
         */
      default:
        notify("ERROR \"N-Phase Callback called but not for join/failure/state!  Reject!\n\"");
        if (HA_GS_OK != (rCode = ha_gs_vote(noteToken,
                                        HA_GS_VOTE_REJECT,
                                        NULL,
                                        NULL,
                                        HA_GS_NULL_VOTE))) {
            sprintf(buffer,
                    "Strange error code [%s] attempting to vote reject!\n\"",
                    write_an_rc(rCode));
            notify(buffer);
            write_the_notification(1, (void *)note, HA_GS_N_PHASE_NOTIFICATION);
            calledNotify = 1;
            printf("Strange error code from reject ha_gs_vote [%s]\n",
                   write_an_rc(rCode));
        }
    }

    if ( ! calledNotify ) {
        printf("\nNotifying....  %s\n", time_now() );
      write_the_notification(1, (void *)note, HA_GS_N_PHASE_NOTIFICATION);
    }

    fflush(stdout);
    return;
}

/*
 * Deal with the "protocol approved" notifications.  If this is for a
 * state value change protocol, then need to grab the newly-updated state
 * value, to use in proposing our next state value change protocol.
 *
 * If this is a join, may need to grab our provider ID for future reference,
 * as well as set our internal "joined" flag.
 *
 * If this is a failure, then one of the other providers in our group has
 * failed (or, the node on which it was running failed).  Nothing special
 * to do here.
 */
void approved_cb(const ha_gs_approved_notification_t *note)
{
    ha_gs_token_t       noteToken;
    ha_gs_num_phases_t  phases;
    int                 new_state;
    struct timeval      current_time;
    struct timezone     tz;
    ha_gs_provider_t    pProposer;
    int                 i;
    int                 numberCollided;

    /*
     * Grab the provider token, and the proposer.
```

```
 */
noteToken = note->gs_provider_token;
pProposer = note->gs_proposal->gs_proposed_by;

if (2 <= debug)
  printf("Approved Callback called\n");

/*
 * For the "unusual" notifications (not those related to state value
 * changes) write out the whole thing, unless debug is high.
 */
if ((HA_GS_STATE_VALUE_CHANGE != note->gs_protocol_type) || (3 <= debug) )
{
    printf("\nApproved callback called on %s", time_now() );
  write_the_notification(1, (void *)note, HA_GS_APPROVED_NOTIFICATION);
}

/*
 * Process based on protocol type.
 */
switch (note->gs_protocol_type) {

  case HA_GS_STATE_VALUE_CHANGE:
    /*
     * Ours or another providers state value change proposal has been
     * approved.  Grab the newly-update state value, and update our
     * various and sundry counters.
     */
    nStateChangesSeen++;
    phases = note->gs_proposal->gs_phase_info.gs_num_phases;
    if (HA_GS_1_PHASE == phases) {
        nStateChanges1Phase++;
    } else {
        nStateChangesNPhase++;
    }
    new_state = *((int *)note->gs_proposal->gs_current_state_value->gs_state);
    if ((2 <= debug) ||
        (0 == (nStateChangesSeen % printfreq)))
        printf("old state %d, new state %d;    %s",
               state, new_state, time_now() );
    if ( state + 1 != new_state ) {
        notify("ERROR \"bad state!\n\"");
    }
    numberCollided = nCollided;
    if ( ( pProposer.gs_provider_id != ourProviderId.gs_provider_id )
        && proposal_time.tv_sec ) {
        /*
         * We made a proposal, but ours wasn't run.  Therefore, ours
         * must have collided!
         */
        numberCollided++;
    }
    if((2 <= debug) ||
       (0 == (nStateChangesSeen % printfreq)))
        printf("state changes: submitted/seen(1-/n-phase) %d(%d/%d)/%d(%d/%d) collided %d.\n",
               nSubmitted,
               nSubmitted1Phase,
               nSubmittedNPhase,
               nStateChangesSeen,
               nStateChanges1Phase,
               nStateChangesNPhase,
               numberCollided);
    proposal_time.tv_sec = 0;
    state = new_state;
    break;

  case HA_GS_JOIN:
    /*
     * If this is a join, then the "proposer" id will always be us.
     *
     * If we are in the changing providers list, this is "our" join,
     * so grab the provider id from the "proposer" field and keep it.
```

```
         */
        if ( !joined ) {
            ourProviderId = pProposer;
            joined = 1;
            state = *((int *)note->gs_proposal->gs_current_state_value->gs_state);
            printf("Joining, state is %d\n", state);
        }
        break;

      case HA_GS_FAILURE_LEAVE:
        sprintf(buffer,
                "We have lost one or more providers due to failure.");
        notify(buffer);
        break;

      default:
        sprintf(buffer,
                "Received unexpected notification for protocol type %s",
                proto_type(note->gs_protocol_type));
        notify(buffer);
    }

    if (noteToken != gid[1])
    {
      printf("Provider token = %d my provider token is %d\n",
             note->gs_provider_token, gid[1]);
    }
    fflush(stdout);
    return;
}

/*
 * This should be quite rare, as we do not normally reject protocols, and
 * we set our group default vote to APPROVE, to approve protocols even if
 * a provider should fail during a voting protocol.
 *
 * However, we have to have this callback, and it IS possible that we will
 * vote to reject unrecognized proposals.
 *
 * Also, if you decide to add (in n_phase_cb()) the ability to reject the
 * occasional join or state value change, then this routine becomes more
 * interesting.
 */
void rejected_cb(const ha_gs_rejected_notification_t *note)
{
    ha_gs_token_t       noteToken;
    ha_gs_num_phases_t  phases;
    int                 new_state;
    struct timeval      current_time;
    struct timezone     tz;
    ha_gs_provider_t    pProposer;
    int                 i;
    int                 numberCollided;

    /*
     * Grab the provider token, and the proposer.
     */
    noteToken = note->gs_provider_token;
    pProposer = note->gs_proposal->gs_proposed_by;

    if (2 <= debug)
      printf("Rejected Callback called\n");

    voting_phase = 1;                   /* reset for next protocol. */

    write_the_notification(1, (void *)note, HA_GS_REJECTED_NOTIFICATION);

    if (noteToken != gid[1])
    {
      printf("Provider token = %d my provider token is %d\n",
             note->gs_provider_token, gid[1]);
    }
```

```
    fflush(stdout);
    return;
}

/*
 * Deal with a delayed error notification.  We normally expect these to only
 * occur for collisions with our state value change proposals.
 *
 * Note that it is also possible for a delayed error to arrive in response
 * to a join request, if different processes attempting to join the group
 * have different group attributes (e.g., they specify different join/failure
 * protocol controls via the '-j' command-line argument.)
 */
void my_delayed_error_cb(const ha_gs_delayed_error_notification_t *note)
{
    /*
     * The "normal" case, we need to adjust counts, and determine when our
     * next proposal should be made.
     */
    if ( (HA_GS_STATE_VALUE_CHANGE == note->gs_protocol_type)
        && (HA_GS_COLLIDE == note->gs_delayed_return_code) ) {

        /*
         * It would appear that someone else proposed a state value
         * change, and that ours lost the collision contest.
         */
        if ( 2 <= debug )
            printf("Delayed Error Callback; HA_GS_COLLIDE on HA_GS_STATE_VALUE_CHANGE\n");
        nCollided++;
        return;
    }

    notify("ERROR \"Delayed Error Callback called\n\"");

    write_the_delayed_error(note);

    fflush(stdout);
    return;
}

/*
 * Callback function executed when an announcement notification arrives
 * for our group from Group Services.
 */
void announce_cb(const ha_gs_announcement_notification_t *note)
{
    if ( HA_GS_GROUP_SERVICES_HAS_DIED_HORRIBLY == note->gs_summary_code ){
        notify( "CRASH \" - die a horrible death!\n\"" );
    } else {
      notify("ERROR \"Announce Callback One called\n\"");
    }

    write_the_notification(1, (void *)note, HA_GS_ANNOUNCEMENT_NOTIFICATION);
    fflush(stdout);
    return;
}

/*
 * Deal with responsiveness notification.  Quite simple, just say we are
 * ok.
 */
ha_gs_callback_rc_t responsive_cb(const ha_gs_responsiveness_notification_t *note) {

    ha_gs_callback_rc_t  _rc;
    const ha_gs_responsiveness_t *_response;
    ha_gs_time_limit_t      _time;

    handledResponsiveness = 1;

    _response = &(note->gs_responsiveness_information);
    _time = _response->gs_responsiveness_response_time_limit;
```

```
    if (2 <= debug) {
        write_the_notification(0, (void *)note, HA_GS_RESPONSIVENESS_NOTIFICATION);
    }

    fflush(stdout);
    return HA_GS_CALLBACK_OK;
}

/*
 * Query callback function for query response notifications.  Not
 * something we expect to see.
 */
void query_cb(const ha_gs_query_notification_t *note){
    write_the_time();
    printf("Query callback called\n");
    fflush(stdout); return;
}

/*********************************************************************/
/*
 * Used to write out exceptional conditions.  It will write the given
 * "message" to stdout, and it will also attempt to execute an external
 * shell script named by the variable "theNotifyScript", sending it the
 * given message.  That script may mail the info to an interested user,
 * log it, or do nothing.
 */
void notify( char *message )
{
    static char buffer[1024];
    static char msg[1024];
    printf( message );
    sprintf( msg, "\"%s\" \"%s\" \"state: %d settings: -g %s, -t %d, -p %d -d %d -j %d -f %d -n %s\" %s",
            progname, args, state,
            theGroupName, testInterval,
            providerInstanceNumber, debug,
            jfphases, freq,
            theNotifyScript,
            message );
    printf("\nNotifying....  %s\n", time_now() );
    sprintf( buffer, "%s %s", theNotifyScript, msg );
    printf( buffer );
    fflush( stdout );
    system( buffer );
}

/*********************************************************************/
/*
 * Internal-usage utility functions.
 */
/*
 * Print help, such as it is.
 */
void print_usage(void)
{
    fprintf(stderr,
            "Usage: schg [ -g <groupname> -t <time interval (in seconds)> -d <debug level>\n");
    fprintf(stderr,
            "              -p <provider instance number> -n <notify script> \n");
    fprintf(stderr,
            "              -j <join/failure num phases> -f <log print frequency>\n");
    fprintf(stderr,
            "              -r -s <seed> -P <n-phase pctg> -T <state time limit>]\n");
    fprintf(stderr,
            "\ndefaults:   -g SampleGroup01 -t 10 -p 1 -d 1 -n /u/tedkirby/bin/notify -j 1 -f 2\n");
    fflush(stderr);
}

/*
 * Function to obtain the number of the node upon which schg is executing.
 * The number is obtained by executing the 'node_number' program.
 */
#define NODE_NUMBER_PROG "/usr/lpp/ssp/install/bin/node_number"
```

```
int     get_node_number(void)
{
    FILE    *fp;
    char    inbuf[16];
    int     status;
    int     node;

    fp = popen(NODE_NUMBER_PROG, "r");
    if (NULL == fp) {
        /* oh well, not here.  return something. */
        return(1);
    }
    if (NULL == fgets(inbuf, sizeof(inbuf), fp)) {
        /* oh well, not here.  return something. */
        return(2);
    }
    fclose(fp);

    /* clean up terminated process created by popen() */

    wait(&status);

    node = atoi(inbuf);

    if (0 == node) {
        /* if on CWS, return real high node number */
        return(2049);
    }

    return(node);
}
```

The sample_utility.c Utility Functions

```
/*********************************************************************/
/*                                                                   */
/*CPRY                                                               */
/*                                                                   */
/* Licensed Materials - Property of IBM                              */
/*                                                                   */
/* 5765-529 PSSP                                                     */
/*                                                                   */
/* (C) Copyright IBM Corporation 1996 All Rights Reserved.           */
/*                                                                   */
/* US Government Users Restricted Rights -                           */
/* Use, duplication or disclosure restricted by                      */
/* GSA ADP Schedule Contract with IBM Corp.                          */
/*                                                                   */
/*CPRY                                                               */
/*********************************************************************/

static char *sccsid = "@(#)95   1.19
src/rsct/pgs/samples/sample_utility.c, gssamples, rsct_rtro, rtrot1fx 4/26/98 16:16:50";

#if !defined(_HAGSD_COPYRIGHT_H)
#define _HAGSD_COPYRIGHT_H
static char copyright[] = "Licensed Materials - Property of IBM\n\
5765-529 (C) Copyright IBM Corp. 1996.  All Rights Reserved.\n\
US Government Users Restricted Rights - Use, duplication or \n\
disclosure restricted by GSA ADP Schedule Contract with IBM Corp.\n";
#endif

/*********************************************************************/
/*
 * Name:  sample_utility.c
 *
 * This program provides the set of utility functions used by the
 * "sample_test" and "sample_schg" programs.
 *
 * Components:
 *   sample_utility.c - this module, provides the definitions for the
 *      utility functions used by the sample_test and sample_schg
 *      programs.
 *
 *   sample_callbacks.c - provides the definitions for the callback
 *      functions used for the groups created by the sample_test
 *      program.
 *
 *   sample_test.c - contains the main() function for sample_test, which
 *      supports interaction with the user, and most calls to the Group
 *      Services interfaces.
 *
 *   sample_schg.c - contains the main() function for sample_schg, which
 *      contains the calls to Group Services interfaces, and also the
 *      callback functions used by sample_schg.
 *
 *   sample_callbacks.h - declarations for the callback functions
 *      contained in sample_callbacks.c.
 *
 *   sample_utility.h - declarations for the utility functions contained
 *      in sample_utility.c
 *
 * The information here assumes that you are familiar with the information
 * presented in the IBM PSSP Group Services Programming Guide and Reference
 * manual.
 *
 * This program is provided for illustrative purposes only, and is not
 * intended to be an authoritative description of the "best" methods to
 * use when writing a Group Services application.  It is intended to
 * demonstrate the various interfaces in a relatively verbose manner,
 * and to allow you to relatively easily manipulate groups and their
 * members.
 *
```

```
 * To this end, various aspects of this program (in particular its
 * handling of screen input and output) are neither robust nor
 * foolproof.  Therefore, you should take care when giving input to
 * this program.
 */
/*********************************************************************/

/*********************************************************************/
/*
 * Please refer to sample_test.c for information about building and
 * using the sample_test program.  Refer to sample_schg.c for such
 * information for the sample_schg program.
 *
 * Note that since this file is shared by sample_test and sample_schg,
 * all data and functions defined here are used only by sample_test.
 * Those sections NOT included via "#ifdef _SAMPLE_TEST/#endif" are
 * not used by sample_schg.
 */
/*********************************************************************/

/*********************************************************************/
/*
 * Utility functions.
 *
 * Ths utility functions handle pedestrian tasks such as formatting
 * and displaying the information contained in various notifications,
 * as well as querying the user for input data when creating protocol
 * proposals or asking for votes.
 *
 * These functions are used by the various callback functions in
 * sample_callbacks.c, as well as by the main() function contained in
 * sample_test.c.  The file sample_callbacks.h contains the prototypes
 * for the utility functions.
 */
/*********************************************************************/

/*********************************************************************/
/*
 * Include "standard" system files.  Note that pthread.h must be the
 * first file included, if it is to be included (see the standard AIX
 * documentation for more information about AIX thread support.)
 */
/*********************************************************************/

#ifdef _THREAD_SAFE                     /* begin _THREAD_SAFE */
#include <pthread.h>
#endif                                  /* end _THREAD_SAFE */

#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <strings.h>
#include <memory.h>
#include <time.h>
#include <arpa/inet.h>

/*********************************************************************/
/*
 * Include the Group Services declarations file.
 */
/*********************************************************************/

#include <ha_gs.h>

/*********************************************************************/
/*
 * Include the set of declarations for utility functions for this program.
 */
/*********************************************************************/

#include "sample_utility.h"
```

```
#ifdef  _SAMPLE_TEST                    /* _SAMPLE_TEST */
/*********************************************************************/
/*
 * Include the set of declarations for callbacks for this program.
 */
/*********************************************************************/

#include "sample_callbacks.h"
#endif                                  /* ifdef _SAMPLE_TEST */

/*********************************************************************/
/*
 * These are "global" variables shared among the modules within the
 * sample_test program.
 */

int voting_phase;                       /* Which voting phase for the */
                                        /* current protocol? */
int handledResponsiveness = 0;          /* Received responsiveness notification? */

#ifdef  _SAMPLE_TEST                    /* _SAMPLE_TEST */

int verbose = 0;                        /* Verbose prompts? */

/*
 * Set up variables to hold the data pertaining to the groups which are
 * allowed for joining and subscribing.  We initializing the token arrays
 * to -1 to show that no token is yet in that slot (we assume that -1
 * will never represent a valid token, which is valid in AIX, as all tokens
 * will be in the range of 0-2048.)
 */

const int  num_groups = NUM_GROUPS;
const int  num_groups_for_subscribe = NUM_GROUPS_FOR_SUBSCRIBE;
const int  predef_groups_for_subscribe = NUM_GROUPS + 8;

#endif                                  /* ifdef _SAMPLE_TEST */

/* ha_gs_tokens_t for each group */
ha_gs_token_t  gid[] = {-1,-1,-1,-1,-1,-1,-1, -1};

/* provider id's for each group */
ha_gs_provider_t provId[NUM_GROUPS];

 /* ha_gs_tokens_t for subscribing to each group */
ha_gs_token_t  sid[] = {-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,\
                          -1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1};

/* subscription controls */
ha_gs_subscription_ctrl_t subCtrl[] = {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,\
                                       0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};

char   *subNames[NUM_GROUPS_FOR_SUBSCRIBE];

#ifdef  _SAMPLE_TEST                    /* _SAMPLE_TEST */

/*
 * The set of callback functions that are used for subscribing.  The
 * initial 8 entries are for subscribing to the pre-defined and user-
 * defined groups supported by sample_test for joining, the remaining
 * are used for subscriptions to other groups.
 */
ha_gs_subscription_cb_t *subCallbacks[] = {
    subscribe_cb0,
    subscribe_cb1,
    subscribe_cb2,
    subscribe_cb3,
    subscribe_cb4,
    subscribe_cb5,
    subscribe_cb6,
    subscribe_cb7,
    subscribe_cb_HostMbr,
```

```
    subscribe_cb_EnetMbr,
    subscribe_cb_CssMbr,
    subscribe_cb_TrMbr,
    subscribe_cb_FddiMbr,
    subscribe_cb_RS232Mbr,
    subscribe_cb_TmScsiMbr,
    subscribe_cb_SlipMbr
    };

int     pickIdx = 99;
int     listIdx = 999;
ha_gs_subscription_cb_t        *pickCallback = subscribe_cb_Pick;

#endif                                  /* ifdef _SAMPLE_TEST */

/* flag for successful join of group */
int     in_group[] = {0,0,0,0,0,0,0,0};

 /* how many successes? */
int     in_group_count = 0;

/* how many subscriptions? */
int     subscribed_to_count = 0;

/* are we dealing with responsiveness interactively? */
int     interactiveInit = 0;
int     interactiveResponse = 0;

char *starz="**********==========**********==========\n";
char *asterisks="**************************************************************************\n";

/*********************************************************************/
/*
 * These variables provide the pre-defined names of groups, and portions
 * of the group attributes for these groups.
 */

#ifdef  _SAMPLE_TEST                    /* _SAMPLE_TEST */

/*
 * The names of the "pre-defined" groups for joining.
 */
char   *group_names[] = {
    "theSourceGroup",
    "OnePhaseJoin",
    "theTargetGroup",
    "theLonelyGroup",
    "ifFirstIWin",
    "SourceOrNot",
    "ChainGang",
    "",
    "",
    "",
    "",
    "",
    "",
    "",
    "",
    "",
    ""
};

/*
 * The names of the source-groups to be used by the respectives
 * groups listed in group_names[].
 */
char   *source_group_names[] = {
    "",
    "",
    "theSourceGroup",
    "theMissingGroup",
    "",
    "theSourceGroup",
```

```
    "theTargetGroup",
    "",
    ""
};

/*
 * The provider local names used for the join requests to the
 * respective groups listed in group_names[].
 */
char   *prov_local_names[] = {
    "SourceJoin",
    "SinglePhase",
    "Existing",
    "Missing",
    "AmIFirstOrALoser",
    "MaybeSourceIt",
    "DaisyChain",
    "NeverBeMe",
    ""
};

#endif                                  /* ifdef _SAMPLE_TEST */

/*
 * The base provider instance numbers used for the join requests
 * to the respective groups listed in group_names[].  The actual
 * instance number is built by adding the program's index number
 * to this base value.
 */
int     instance_numbers[] = {100, 200, 300, 400, 500, 600, 700, 800};

/*
 * Array to hold the group attributes for the pre-defined groups.
 */
ha_gs_group_attributes_t *gattr[] = {0, 0, 0, 0, 0, 0, 0, 0};

/*********************************************************************/
/*
 * We will prepend all submitted state values and provider-broadcast
 * messages with a tag constructed from the program index, to ensure
 * that we don't truncate due to finding a null value there, and to
 * indicate which provider submitted the data.
 *
 * See sample_test.c for full details of how this "tag" is prepared,
 * these variables are used by the utility routines to construct the
 * values.
 */

int     sample_index = 1;               /* what is my index, for numbers, flags, etc? */

#ifdef  _SAMPLE_TEST                    /* _SAMPLE_TEST */

typedef struct {
    int st_pbm_index;
    char st_pbm_data[2500];
} st_pbm_struct;

char   *sample_prefix;                  /* prepend to msgs, states, etc. */
int     sample_prefix_len = 0;

char   *sample_pp = "<";
char   *sample_ee = "<";


/*********************************************************************/

/*
 * These arrays provide a rotating set of provider-broadcast messages,
 * proposed group state values, and updated default vote values, all of
 * which may be submitted along with vote values whenever the user needs
 * to vote on an n-phase protocol.
 *
```

```
 * The entry is chosen by taking the phase number for this protocol
 * modulo with the variable "NUMBER_MSG_ENTRIES", and using the result
 * as an index into these arrays.  An entry of '{0,0}' indicates that
 * no value will be submitted with this vote.
 */

ha_gs_provider_message_t provider_msg_array[] = {
{ 35, "I am Born!  Or, is that, I am Borg?"},
{ 49, "Things do seem to be looking up now, do they not?"},
{ 36, "What a long, strange trip it's been!"},
{ 0, 0},
{ 252, "Fourscore and seven years ago, our fourfeathers brought into existence "\
       "that which cannot ever be compared to anything else, that which is to be, "\
       "umm, well, something.  Anyway, trust us.  It was quite a good plan they "\
       "had.  Plastics.  That's the ticket."},
{ 0, 0},
{ 124, "We come in peace.  All we want is a peace of what you got, and a peace "\
       "of what he's got, and a peace of what they've got..."},
{ 20, "Forest for the Cup!"}
};

ha_gs_state_value_t         state_value_array[] = {
{ 4, "1111"},
{ 0, 0},
{ 8, "22223333"},
{ 120, "hhhhhhhhppppppppppppLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLL"\
       "LLLLLLLLxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx"},
{ 0, 0},
{ 7, "7654321"},
{ 0, 0},
{ 13, "Forza azzuri"}
};

ha_gs_vote_value_t  default_vote_array[] = {
HA_GS_NULL_VOTE,
HA_GS_NULL_VOTE,
HA_GS_VOTE_APPROVE,
HA_GS_VOTE_REJECT,
HA_GS_NULL_VOTE,
HA_GS_NULL_VOTE,
HA_GS_NULL_VOTE,
HA_GS_VOTE_APPROVE
};

/********************************************************************/
/*
 * Display the help information.
 */
char *instruct1="\nEnter: 'i'(ha_gs_init) '1'(1 phase join) 'j'(n phase join)\n";
char *instruct2=" 'x'(good target join, if 'j' was done) 'm'(bad target join)\n";
char *instruct3=" 'a'(mismatch attributes, if done by both even and odd indexed sample_tests)\n";
char *instruct4=" 't'(state value change) 'p'(provider broadcast message)\n";
char *instruct5=" 's'(subscribe to a group) 'u'(unsubscribe from a group)\n";
char *instruct6=" 'l'(leave a group) 'e'(expel one or more providers from a group)\n";
char *instruct7=" 'g'(even index: source 'theSourceGroup', odd index: no source-group)\n";
char *instruct8=" '3'(good target join, if 'j' and 'x' were done)\n";
char *instruct9=" 'b'(build a group by defining its attributes)\n";
char *instruct10=" 'n'(modify a group's attributes)\n";
char *instruct11=" 'y'(say goodbye (\"exit immediately from\") a group)\n";
char *instruct12=" 'z'(toggle HA_GS_DEACTIVATE_ON_FAILURE)\n";
char *instructL=" 'd'(ha_gs_dispatch) 'h' (print these instructions) 'q'(quit)\n";

void    write_instructions(int _verbose)
{
    printf(instruct1);
    printf(instruct2);
    printf(instruct3);
    printf(instruct4);
    printf(instruct5);
    printf(instruct6);
    printf(instruct7);
    printf(instruct8);
```

```
    printf(instruct9);
    printf(instruct10);
    printf(instruct11);
    printf(instructL);
    if (!_verbose) {
        printf("\nTo automatically get these instructions at each prompt, use\n");
        printf(" 'verbose' mode via '-v' on the command line.\n\n");
    }
    printf("To interactively specify initialization parameters, specify '-i' on\n");
    printf(" the command line.\n\n");
    fflush(stdout);
    return;
}

#endif                                  /* ifdef _SAMPLE_TEST */

/*********************************************************************/
/*
 * Calculate the current TOD, compare times, generate random numbers.
 */
char   *time_now( )
{
    struct timeval  current_time;
    struct timezone tz;

    gettimeofday(&current_time, &tz);

    return( ctime( (time_t *) &current_time.tv_sec ) );
}

time_t  time_since( struct timeval *event )
{
    time_t overdue;
    struct timeval  current_time;
    struct timezone tz;

    gettimeofday(&current_time, &tz);

    overdue = (time_t) ( current_time.tv_sec - event->tv_sec );

    return ( overdue );
}

void    write_the_time( )
{
    struct timeval  current_time;
    struct timezone tz;
    char            tod[32];            /* extract part of string. */
    char           *cTod;

    gettimeofday(&current_time, &tz);

                                        /* Grab month/day/time only. */
    cTod = ctime((time_t *) &current_time.tv_sec);
    memcpy(tod, cTod+4, 16);
    tod[15] = '\0';

    printf("\n[TOD(%s)]", tod);

    return;
}

float   randomp( )
{
    const unsigned long RANDMAX = 2147483648; /* 2**31 */
                                              /* random() returns 0 to RANDMAX-1 */

    return( ((float) random()) / ((float) RANDMAX) );
}

long    randomn( unsigned long max )
{
```

```
    const unsigned long RANDMAX = 2147483648; /* 2**31 */
                                              /* random() returns 0 to RANDMAX-1 */
    long rand;

    if ( 1 >= max ) return 0;

    if ( RANDMAX < max ) {
        rand = random();
        return 2*rand + rand%2;
    }

    while( (rand = random()) > max * ( RANDMAX/max ) );

    return( rand % max );
}

/*********************************************************************/
/*
 * These functions display various Group Services data types.  These are
 * used to display data from protocol submissions and notifications.
 */

/*
 * Display the type of protocol represented by a notification.
 */
char  *proto_type(int req)
{
    switch(req) {
      case HA_GS_RESPONSIVENESS:
        return("HA_GS_RESPONSIVENESS");
      case HA_GS_JOIN:
        return("HA_GS_JOIN");
      case HA_GS_FAILURE_LEAVE:
        return("HA_GS_FAILURE_LEAVE");
      case HA_GS_LEAVE:
        return("HA_GS_LEAVE");
      case HA_GS_EXPEL:
        return("HA_GS_EXPEL");
      case HA_GS_STATE_VALUE_CHANGE:
        return("HA_GS_STATE_VALUE_CHANGE");
      case HA_GS_PROVIDER_MESSAGE:
        return("HA_GS_PROVIDER_MESSAGE");
      case HA_GS_CAST_OUT:
        return("HA_GS_CAST_OUT");
      case HA_GS_SOURCE_STATE_REFLECTION:
        return("HA_GS_SOURCE_STATE_REFLECTION");
      case HA_GS_MERGE:
        return("HA_GS_MERGE");
      case HA_GS_SUBSCRIPTION:
        return("HA_GS_SUBSCRIPTION");
      case HA_GS_GROUP_ATTRIBUTE_CHANGE:
        return("HA_GS_GROUP_ATTRIBUTE_CHANGE");
      default:
        printf("?? Unknown protocol request[%d] ??", req);
        return("Unknown");
    }
}

/*
 * Display the summary code contained in a notification.
 */
char  *sum_code(int sum)
{
    switch(sum) {
      case HA_GS_EXPLICIT_APPROVE:
        return("HA_GS_EXPLICIT_APPROVE");
      case HA_GS_EXPLICIT_REJECT:
        return("HA_GS_EXPLICIT_REJECT");
      case HA_GS_DEFAULT_APPROVE:
        return("HA_GS_DEFAULT_APPROVE");
      case HA_GS_DEFAULT_REJECT:
        return("HA_GS_DEFAULT_REJECT");
```

```
      case HA_GS_TIME_LIMIT_EXCEEDED:
        return("HA_GS_TIME_LIMIT_EXCEEDED");
      case HA_GS_PROVIDER_FAILED:
        return("HA_GS_PROVIDER_FAILED");
      case HA_GS_RESPONSIVENESS_NO_RESPONSE:
        return("HA_GS_RESPONSIVENESS_NO_RESPONSE");
      case HA_GS_RESPONSIVENESS_RESPONSE:
        return("HA_GS_RESPONSIVENESS_RESPONSE");
      case HA_GS_GROUP_DISSOLVED:
        return("HA_GS_GROUP_DISSOLVED");
      case HA_GS_GROUP_SERVICES_HAS_DIED_HORRIBLY:
        return("HA_GS_GROUP_SERVICES_HAS_DIED_HORRIBLY");
      case HA_GS_DEACTIVATE_UNSUCCESSFUL:
        return("HA_GS_DEACTIVATE_UNSUCCESSFUL");
      case HA_GS_DEACTIVATE_TIME_LIMIT_EXCEEDED:
        return("HA_GS_DEACTIVATE_TIME_LIMIT_EXCEEDED");
      default:
        printf("?? Unknown summary code[%d] ??", sum);
        return("Unknown");
    }
}

/*
 * Display the number of phases for a given protocol.
 */
char    *phase_print(int phase)
{
    switch(phase) {
      case HA_GS_1_PHASE:
        return("1_PHASE");
      case HA_GS_N_PHASE:
        return("N_PHASE");
      default:
        printf("?? Unknown phase specification[%d] ??", phase);
        return("Unknown");
    }
}

/*
 * Display the batching control group attribute.
 */
char    *batch_print(int batch)
{    static char msg[100];
    if(batch & HA_GS_DEACTIVATE_ON_FAILURE) {
      strcpy(msg, "HA_GS_DEACTIVATE_ON_FAILURE|");
    } else {
      strcpy(msg,"");
    }

    switch( (batch &  (HA_GS_DEACTIVATE_ON_FAILURE))) {
      case HA_GS_NO_BATCHING:
            strcat(msg, "HA_GS_NO_BATCHING");
            return msg;
      case HA_GS_BATCH_JOINS:
                  strcat(msg, "HA_GS_BATCH_JOINS");
            return msg;
      case HA_GS_BATCH_LEAVES:
            strcat(msg,"HA_GS_BATCH_LEAVES");
            return msg;

      case HA_GS_BATCH_BOTH:
            strcat(msg, "HA_GS_BATCH_BOTH");
            return msg;
      default:
        printf("?? Unknown batch control[%d] ??", batch);
        return("Unknown");
    }
}

/*
 * Display the merge control group attribute.
```

```
 */
char    *merge_print(int merge)
{
    switch(merge) {
      case HA_GS_DISSOLVE_MERGE:
        return("HA_GS_DISSOLVE_MERGE");
      case HA_GS_LARGER_MERGE:
        return("HA_GS_LARGER_MERGE");
      case HA_GS_SMALLER_MERGE:
        return("HA_GS_SMALLER_MERGE");
      case HA_GS_DONTCARE_MERGE:
        return("HA_GS_DONTCARE_MERGE");
      default:
        printf("?? Unknown merge control[%d] ??", merge);
        return("Unknown");
    }
}

/*
 * Display the type of responsiveness specified.
 */
char    *responsive_print(int resp)
{
    switch(resp) {
      case HA_GS_NO_RESPONSIVENESS:
        return("HA_GS_NO_RESPONSIVENESS");
      case HA_GS_PING_RESPONSIVENESS:
        return("HA_GS_PING_RESPONSIVENESS");
      case HA_GS_COUNTER_RESPONSIVENESS:
        return("HA_GS_COUNTER_RESPONSIVENESS");
      default:
        printf("?? Unknown responsiveness type [%d] ??", resp);
        return("Unknown");
    }
}

/*
 * Display the contents of the whats changed field from a notification.
 */
char    *what_changed(int what)
{
    switch(what) {
      case HA_GS_NO_CHANGE:
        return("HA_GS_NO_CHANGE");
      case HA_GS_PROPOSED_MEMBERSHIP:
        return("HA_GS_PROPOSED_MEMBERSHIP");
      case HA_GS_ONGOING_MEMBERSHIP:
        return("HA_GS_ONGOING_MEMBERSHIP");
      case HA_GS_PROPOSED_STATE_VALUE:
        return("HA_GS_PROPOSED_STATE_VALUE");
      case HA_GS_ONGOING_STATE_VALUE:
        return("HA_GS_ONGOING_STATE_VALUE");
      case HA_GS_UPDATED_PROVIDER_MESSAGE:
        return("HA_GS_UPDATED_PROVIDER_MESSAGE");
      case HA_GS_UPDATED_MEMBERSHIP:
        return("HA_GS_UPDATED_MEMBERSHIP");
      case HA_GS_REJECTED_MEMBERSHIP:
        return("HA_GS_REJECTED_MEMBERSHIP");
      case HA_GS_UPDATED_STATE_VALUE:
        return("HA_GS_UPDATED_STATE_VALUE");
      case HA_GS_REFLECTED_SOURCE_STATE_VALUE:
        return("HA_GS_REFLECTED_SOURCE_STATE_VALUE");
      case HA_GS_EXPEL_INFORMATION:
        return("HA_GS_EXPEL_INFORMATION");
      case HA_GS_PROPOSED_GROUP_ATTRIBUTES:
        return("HA_GS_PROPOSED_GROUP_ATTRIBUTES");
      case HA_GS_ONGOING_GROUP_ATTRIBUTES:
        return("HA_GS_ONGOING_GROUP_ATTRIBUTES");
      case HA_GS_UPDATED_GROUP_ATTRIBUTES:
        return("HA_GS_UPDATED_GROUP_ATTRIBUTES");
      case HA_GS_REJECTED_GROUP_ATTRIBUTES:
        return("HA_GS_REJECTED_GROUP_ATTRIBUTES");
```

```
      default:
        printf("?? Unknown what's changed control[%d] ??", what);
        return("Unknown");
    }
}

/*
 * Display the subscription control flag.
 */
char    *write_sub_ctrl(ha_gs_subscription_ctrl_t subCtrl)
{
    switch(subCtrl) {
      case 0:
        return("No subscriptions!");
      case HA_GS_SUBSCRIBE_STATE:
        return("HA_GS_SUBSCRIBE_STATE");
      case HA_GS_SUBSCRIBE_DELTA_JOINS:
        return("HA_GS_SUBSCRIBE_DELTA_JOINS");
      case HA_GS_SUBSCRIBE_DELTA_LEAVES:
        return("HA_GS_SUBSCRIBE_DELTA_LEAVES");
      case HA_GS_SUBSCRIBE_DELTAS_ONLY:
        return("HA_GS_SUBSCRIBE_DELTAS_ONLY");
      case HA_GS_SUBSCRIBE_MEMBERSHIP:
        return("HA_GS_SUBSCRIBE_MEMBERSHIP");
      case HA_GS_SUBSCRIBE_ALL_MEMBERSHIP:
        return("HA_GS_SUBSCRIBE_ALL_MEMBERSHIP");
      case HA_GS_SUBSCRIBE_STATE_AND_MEMBERSHIP:
        return("HA_GS_SUBSCRIBE_STATE_AND_MEMBERSHIP");
      default:
        printf("?? Unknown subscription control[%d] ??", subCtrl);
        return("Unknown");
    }
}

/*
 * Display the type of a specified subscription request.
 */
void    write_sub_type(ha_gs_subscription_type_t subType)
{
    int foo;
    foo = 0;

    printf("Subscription data[");
    if (HA_GS_SUBSCRIPTION_STATE & subType) {
        printf("HA_GS_SUBSCRIPTION_STATE");
        foo++;
    }
    if (HA_GS_SUBSCRIPTION_DELTA_JOIN & subType) {
        if (foo) {
            printf("\n                  ");
            foo = 0;
        }
        printf("HA_GS_SUBSCRIPTION_DELTA_JOIN");
        foo++;
    }
    if (HA_GS_SUBSCRIPTION_DELTA_LEAVE & subType) {
        if (foo) {
            printf("\n                  ");
            foo = 0;
        }
        printf("HA_GS_SUBSCRIPTION_DELTA_LEAVE");
        foo++;
    }
    if (HA_GS_SUBSCRIPTION_MEMBERSHIP & subType) {
        if (foo) {
            printf("\n                  ");
            foo = 0;
        }
        printf("HA_GS_SUBSCRIPTION_MEMBERSHIP");
        foo++;
    }
    if (HA_GS_SUBSCRIPTION_DISSOLVED & subType) {
```

```
        if (foo) {
            printf("\n                  ");
            foo = 0;
        }
        printf("HA_GS_SUBSCRIPTION_DISSOLVED");
        foo++;
    }
    if (HA_GS_SUBSCRIPTION_GS_HAS_DIED & subType) {
        if (foo) {
            printf("\n                  ");
            foo = 0;
        }
        printf("HA_GS_SUBSCRIPTION_GS_HAS_DIED");
    }
    printf("]\n");
    fflush(stdout);
}

/*
 * Display the "name" of a Group Services return code.
 */
char    *write_an_rc(ha_gs_rc_t pRC)
{
    switch(pRC) {
      case HA_GS_OK:
        return("HA_GS_OK");
      case HA_GS_NOT_OK:
        return("HA_GS_NOT_OK");
      case HA_GS_EXISTS:
        return("HA_GS_EXISTS");
      case HA_GS_NO_INIT:
        return("HA_GS_NO_INIT");
      case HA_GS_NAME_TOO_LONG:
        return("HA_GS_NAME_TOO_LONG");
      case HA_GS_NO_MEMORY:
        return("HA_GS_NO_MEMORY");
      case HA_GS_NOT_A_MEMBER:
        return("HA_GS_NOT_A_MEMBER");
      case HA_GS_BAD_CLIENT_TOKEN:
        return("HA_GS_BAD_CLIENT_TOKEN");
      case HA_GS_BAD_MEMBER_TOKEN:
        return("HA_GS_BAD_MEMBER_TOKEN");
      case HA_GS_BAD_PARAMETER:
        return("HA_GS_BAD_PARAMETER");
      case HA_GS_UNKNOWN_GROUP:
        return("HA_GS_UNKNOWN_GROUP");
      case HA_GS_INVALID_GROUP:
        return("HA_GS_INVALID_GROUP");
      case HA_GS_NO_SOURCE_GROUP_PROVIDER:
        return("HA_GS_NO_SOURCE_GROUP_PROVIDER");
      case HA_GS_BAD_GROUP_ATTRIBUTES:
        return("HA_GS_BAD_GROUP_ATTRIBUTES");
      case HA_GS_WRONG_OLD_STATE:
        return("HA_GS_WRONG_OLD_STATE");
      case HA_GS_DUPLICATE_INSTANCE_NUMBER:
        return("HA_GS_DUPLICATE_INSTANCE_NUMBER");
      case HA_GS_COLLIDE:
        return("HA_GS_COLLIDE");
      case HA_GS_SOCK_CREATE_FAILED:
        return("HA_GS_SOCK_CREATE_FAILED");
      case HA_GS_SOCK_INIT_FAILED:
        return("HA_GS_SOCK_INIT_FAILED");
      case HA_GS_CONNECT_FAILED:
        return("HA_GS_CONNECT_FAILED");
      case HA_GS_VOTE_NOT_EXPECTED:
        return("HA_GS_VOTE_NOT_EXPECTED");
      case HA_GS_NOT_SUPPORTED:
        return("HA_GS_NOT_SUPPORTED");
      case HA_GS_INVALID_SOURCE_GROUP:
        return("HA_GS_INVALID_SOURCE_GROUP");
      case HA_GS_UNKNOWN_PROVIDER:
        return("HA_GS_UNKNOWN_PROVIDER");
```

```
      case HA_GS_INVALID_DEACTIVATE_PHASE:
        return("HA_GS_INVALID_DEACTIVATE_PHASE");
      case HA_GS_PROVIDER_APPEARS_TWICE:
        return("HA_GS_PROVIDER_APPEARS_TWICE");
      case HA_GS_BACKLEVEL_PROVIDERS:
        return("HA_GS_BACKLEVEL_PROVIDERS");
      default:
        printf("?? Unknown return code[%d] ??", pRC);
        return("Unknown");
    }
}

/*
 * Display the value of the given vote value.
 */
char    *write_a_vote(ha_gs_vote_value_t vote)
{
    switch(vote) {
      case HA_GS_NULL_VOTE:
        return("HA_GS_NULL_VOTE");
      case HA_GS_VOTE_APPROVE:
        return("HA_GS_VOTE_APPROVE");
      case HA_GS_VOTE_CONTINUE:
        return("HA_GS_VOTE_CONTINUE");
      case HA_GS_VOTE_REJECT:
        return("HA_GS_VOTE_REJECT");
      default:
        printf("?? Unknown vote value[%d] ??", vote);
        return("Unknown");
    }
}

/*
 * Display the leave reason as given in a notification.
 */
char   *write_leave_reason(ha_gs_leave_reasons_t why)
{
    switch(why) {
      case HA_GS_VOLUNTARY_LEAVE:
        return("HA_GS_VOLUNTARY_LEAVE");
        break;
      case HA_GS_PROVIDER_FAILURE:
        return("HA_GS_PROVIDER_FAILURE");
        break;
      case HA_GS_HOST_FAILURE:
        return("HA_GS_HOST_FAILURE");
        break;
      case HA_GS_PROVIDER_EXPELLED:
        return("HA_GS_PROVIDER_EXPELLED");
        break;
      case HA_GS_SOURCE_PROVIDER_LEAVE:
        return("HA_GS_SOURCE_PROVIDER_LEAVE");
        break;      case HA_GS_PROVIDER_SAID_GOODBYE:
        return("HA_GS_PROVIDER_SAID_GOODBYE");
        break;
      default:
        printf("Unknown leave code[%d]!", why);
        break;
    }

    return("Unknown");
}

/*
 * Display the group attributes.
 */
void    write_the_attributes(ha_gs_group_attributes_t *gAttrs)
{
    printf("Group attributes: \n");
    printf(" version[%d] size[%d] client version[%d]\n",
           gAttrs->gs_version,
           gAttrs->gs_sizeof_group_attributes,
```

```
           gAttrs->gs_client_version);
    printf(" Batching[%s] Join/Fail phases[%s] Reflection phases[%s]\n",
           batch_print(gAttrs->gs_batch_control),
           phase_print(gAttrs->gs_num_phases),
           phase_print(gAttrs->gs_source_reflection_num_phases));
    printf(" Default vote[%s] Merge[%s]\n",
           write_a_vote(gAttrs->gs_group_default_vote),
           merge_print(gAttrs->gs_merge_control));
    printf(" Time limits: join/fail[%d] reflection[%d]\n",
           gAttrs->gs_time_limit,
           gAttrs->gs_source_reflection_time_limit);
    printf(" Group name[%s]", gAttrs->gs_group_name);
    if (gAttrs->gs_source_group_name) {
        printf(" Source group name[%s]", gAttrs->gs_source_group_name);
    }
    printf("\n");
    fflush(stdout);
    return;
}

#ifdef  _SAMPLE_TEST                    /* _SAMPLE_TEST */
/*
 * Copy a set of group attributes to save them.
 */
void    copy_the_attributes(int groupIdx,
                            ha_gs_group_attributes_t *gAttrsTarg,
                            ha_gs_group_attributes_t *gAttrsSrc)
{

    ha_gs_copy_group_attributes(gAttrsTarg, gAttrsSrc);
    gattr[groupIdx] = gAttrsTarg;

    /*
     * The names in the new attributes better match what we have cached!
     */
    if (0 != strcmp(gAttrsTarg->gs_group_name, group_names[groupIdx])) {
        printf("Error: New attributes have group name [%s], old have [%s]!\n",
               gAttrsSrc->gs_group_name,
               group_names[groupIdx]);
    }
    if (0 != strcmp(gAttrsTarg->gs_source_group_name, source_group_names[groupIdx])) {
        printf("Error: New attributes have source group name [%s], old have [%s]!\n",
               gAttrsSrc->gs_source_group_name,
               source_group_names[groupIdx]);
    }

    return;
}
#endif                                  /* ifdef _SAMPLE_TEST */

/*
 * Display the information to be submitted for a join (via ha_gs_join()),
 * any of the various group join commands.
 */
void    write_join_information(int gIndex)
{
    ha_gs_group_attributes_t *gAttrs;
    int inst_num;

    printf(starz);
    if (0 == (gAttrs = gattr[gIndex])) {
        printf("No group attributes established for index [%d]!  Leaving.\n",
               gIndex);
        printf(starz);
        return;
    }

    printf("Attempting JOIN for group[%s]:\n",
           gAttrs->gs_group_name);
    inst_num = instance_numbers[gIndex];
#ifdef  _SAMPLE_TEST                    /* _SAMPLE_TEST */
    printf(" My instance #[%d]  local name[%s]\n",
```

```
           inst_num,
           prov_local_names[gIndex]);
#else                                   /* else ifdef _SAMPLE_TEST */
    printf(" My instance #[%d]\n",
           inst_num);
#endif                                   /* ifdef _SAMPLE_TEST */
    write_the_attributes(gAttrs);

    printf(starz);
    fflush(stdout);
    return;
}

/*
 * Display a vote value.
 */
void    write_the_vote(ha_gs_vote_value_t vote)
{
    printf("vote value[%s]", write_a_vote(vote));

    return;
}

/*********************************************************************/
/*
 * Display the information contained in various notifications.
 */

/*
 * Display an arbitrary group state value or provider message.  For
 * those non-predefined groups, we do not know the format of the group's
 * state value, or any provider messages.  Also, certain other routines
 * may not know which group.  At these times, we use this routine to
 * display these values in a (somewhat) intelligent format.
 *
 * We first display as characters, representing non-printable characters
 * as periods ('.').  We then do either or both of the following:
 * - if the length is equivalent to one or more integers, display each
 *    word as its integral value.
 * - display the whole value as a raw hex stream.
 */
void    write_arbitrary_value(int   which,
                              void *value)
{
    char *_val, *_val2;
    int   _len, _len2, _cnt, _isize, _stpbmint;
    char *_buffer;
    char *_bufPtr;

    ha_gs_state_value_t      *_state;
    ha_gs_provider_message_t *_pbm;

    if (ARBITRARY_STATE_VALUE == which) {
        _state = (ha_gs_state_value_t *)value;
        _len = _state->gs_length;
        _val = _state->gs_state;
        printf("state value[");
    } else {
        _pbm = (ha_gs_provider_message_t *)value;
        _len = _pbm->gs_length;
        _val = _pbm->gs_message;
        printf("provider message[");
    }

    if (0 >= _len) {

        /* Nothing here. */

        printf("]\n");
        fflush(stdout);
        return;
    }
```

```
_buffer = (char *)malloc((_len * 5) + 1); /* Buffer in which to build string. */

/*
 * Display assuming it is actually printable.
 */
for (_len2 = _len, _val2 = _val, _bufPtr = _buffer;
     0 < _len2;
     ) {

    for (_cnt = 0;
         ((4 > _cnt) && (0 < _len2));
         _cnt++, _len2--) {

        if (isprint(*_val2)) {
            sprintf(_bufPtr++, "%c", *_val2++);
        } else {
            *_bufPtr++ = '.';
            _val2++;
        }
    }
}

*_bufPtr++ = '\0';

printf("[%s]\n",_buffer);
fflush(stdout);

/*
 * If integral number of words, display as series of ints.
 */
_isize = sizeof(_stpbmint);
if (0 == (_len % _isize)) {
    printf("[");
    if (_isize == _len) {
        /* assume an int */
        memcpy(&_stpbmint, _val, sizeof(_stpbmint));
        printf("%d", _stpbmint);
    } else {
        /* assume series of ints */
        for (_len2 = 0, _val2 = _val;
             _len2 < _len;
             _len2 += 4, _val2 += 4) {
            memcpy(&_stpbmint, _val2, sizeof(_stpbmint));
            printf("%d", _stpbmint);
            if (_len2 != (_len - 4)) {
                printf(" ");
            }
        }
    }
    printf("]\n");
    fflush(stdout);
}

/*
 * Now, convert to "printable" hex.
 */
for (_len2 = _len, _val2 = _val, _bufPtr = _buffer;
     0 < _len2;
     ) {

    sprintf(_bufPtr, "0x");
    _bufPtr += 2;

    for (_cnt = 0;
         ((4 > _cnt) && (0 < _len2));
         _cnt++, _len2--) {

        sprintf(_bufPtr, "%02x", *_val2++);
        _bufPtr += 2;
    }
```

```
        *_bufPtr++ = ' ';
    }

    *_bufPtr++ = '\0';

    printf("[%s]]\n",_buffer);
    fflush(stdout);

    free(_buffer);

    return;
}

/*
 * Display a notification received from Group Services.
 */
void    write_the_notification(int groupIdx,
                               void * addr,
                               ha_gs_notification_type_t whatItBe)
{
    unsigned int        i, j, mask, whatchanged, which_msg;
    int                 stpbmint, full_proposal, grabbedIt, iHaveLeft;
    int                 expelledMyself, iBeenExpelled;
    char               *ctr;

    ha_gs_proposal_t       *proposal;
    ha_gs_responsiveness_t *response;

    ha_gs_request_t         pType;
    ha_gs_provider_t        pProposer;
    ha_gs_token_t           pToken;

    ha_gs_membership_t     *membership;
    ha_gs_provider_t       *provider;

    ha_gs_summary_code_t    summary_code;

    ha_gs_state_value_t    *state;
    ha_gs_provider_message_t *pbm;

    ha_gs_n_phase_notification_t  *note;
    ha_gs_approved_notification_t *appr;
    ha_gs_rejected_notification_t *rej;
    ha_gs_announcement_notification_t *announce;
    ha_gs_responsiveness_notification_t *resp;

    ha_gs_leave_array_t *leave_array;
    ha_gs_leave_info_t  *leave_info;
    ha_gs_expel_info_t  *expel_info;

    note = (ha_gs_n_phase_notification_t *)addr;
    appr = (ha_gs_approved_notification_t *)addr;
    rej = (ha_gs_rejected_notification_t *)addr;
    announce = (ha_gs_announcement_notification_t *)addr;
    resp = (ha_gs_responsiveness_notification_t *)addr;

    printf("%.21s  %.24s  %.21s\n", asterisks, time_now(), asterisks);

    full_proposal = 1;
    grabbedIt = 0;
    iHaveLeft = 0;
    iBeenExpelled = 0;
    expelledMyself = 0;

    switch(whatItBe) {
      case HA_GS_N_PHASE_NOTIFICATION:
        pType = note->gs_protocol_type;
        pToken = note->gs_provider_token;
        printf("Type[%s] Token[%d] TimeLimit[%d]\n",
               proto_type(pType),
               pToken,
               note->gs_time_limit);
```

```
  proposal = note->gs_proposal;
  pProposer = proposal->gs_proposed_by;
  summary_code = note->gs_summary_code;
  membership = proposal->gs_current_providers;
  break;
case HA_GS_APPROVED_NOTIFICATION:
  pType = appr->gs_protocol_type;
  pToken = appr->gs_provider_token;
  printf("Type[%s] Token[%d]\n",
         proto_type(pType),
         pToken);

  proposal = appr->gs_proposal;
  pProposer = proposal->gs_proposed_by;
  summary_code = appr->gs_summary_code;
  membership = proposal->gs_current_providers;
  break;
case HA_GS_REJECTED_NOTIFICATION:
  pType = rej->gs_protocol_type;
  pToken = rej->gs_provider_token;
  printf("Type[%s] Token[%d]\n",
         proto_type(pType),
         pToken);

  proposal = rej->gs_proposal;
  pProposer = proposal->gs_proposed_by;
  summary_code = rej->gs_summary_code;
  membership = proposal->gs_current_providers;
  break;
case HA_GS_ANNOUNCEMENT_NOTIFICATION:
  printf("Type[HA_GS_ANNOUNCEMENT_NOTIFICATION] Token[%d]\n",
         announce->gs_provider_token);

  summary_code = announce->gs_summary_code;
  membership = announce->gs_announcement;
  full_proposal = 0;
  break;

case HA_GS_RESPONSIVENESS_NOTIFICATION:

  response = &(resp->gs_responsiveness_information);

  printf("Type[HA_GS_RESPONSIVENESS_NOTIFICATION] Type[%s]\n",
         responsive_print(response->gs_responsiveness_type));

  printf("Interval[%d seconds] Response time limit[%d seconds]\n",
         response->gs_responsiveness_interval,
         response->gs_responsiveness_response_time_limit);

  if (HA_GS_COUNTER_RESPONSIVENESS == response->gs_responsiveness_type) {
      printf(" Counter location[%x] Counter length[%d] Current value[\n",
             response->gs_counter_location,
             response->gs_counter_length);
      ctr = (char *)response->gs_counter_location;
      for (i = 0;
           i < response->gs_counter_length;
           i++, ctr++) {
          printf("%1.1x",
                 (void *)ctr);
      }
      printf("]\n");
  }

  printf(asterisks);
  fflush(stdout);
  return;

default:
  printf("Unknown protocol notification type: %d\n", whatItBe);
  fflush(stdout);
  return;
```

```
    }
    fflush(stdout);

    printf("Summary_Code[");
    mask = HA_GS_MIN_SUMMARY_CODE;
    for(i=1, j=0; mask <= HA_GS_MAX_SUMMARY_CODE; i++,mask <<= 1) {
      if (summary_code & mask) {
            if (2 == j) {
                printf("\n             ");
                j = 0;
            } else if (1 == j) {
                printf(" ");
            }
            printf("%s",sum_code(mask));
            j++;
       }
      else continue;
    }
    printf("]\n");
    fflush(stdout);

    if (full_proposal) {
#ifndef _VERBOSE_PROVIDER_OUTPUT
        printf("NumPhases[%s] ThisPhase[%d] Proposer[%d/%d]\n",
               phase_print(proposal->gs_phase_info.gs_num_phases),
               proposal->gs_phase_info.gs_phase_number,
               proposal->gs_proposed_by.gs_instance_number,
               proposal->gs_proposed_by.gs_node_number);
#else
        /* do not print out the int value. */
        printf("NumPhases[%s] ThisPhase[%d] Proposer[%d/%d(%d)]\n",
               phase_print(proposal->gs_phase_info.gs_num_phases),
               proposal->gs_phase_info.gs_phase_number,
               proposal->gs_proposed_by.gs_instance_number,
               proposal->gs_proposed_by.gs_node_number,
               proposal->gs_proposed_by.gs_provider_id);
#endif  /* if ! _VERBOSE_PROVIDER_OUTPUT */
        printf("WhatsChanged[");
        if (proposal->gs_whats_changed == HA_GS_NO_CHANGE) {
            printf("%s", what_changed(HA_GS_NO_CHANGE));
        } else {
            j = 0;
            for (i=0;i<31;i++) {
                if (proposal->gs_whats_changed & (1<<i)) {
                    if (2 == j) {
                        printf("\n             ");
                        j = 0;
                    } else if (1 == j) {
                        printf(" ");
                    }
                    printf("%s", what_changed(1<<i));
                    j++;
                }
            }
        }
        printf("]\n");
        fflush(stdout);
    }

    if (NULL != membership) {
        if (full_proposal) {
            printf("CurrentProviders[");
        } else {
            printf("AffectedProviders[");
        }
        if (membership->gs_count == 0) {
            printf("No providers in list!");
        } else {
            printf("count[%d] Members[", membership->gs_count);
            provider = membership->gs_providers;
            for (i = 0; i < membership->gs_count; i++) {
#ifndef _VERBOSE_PROVIDER_OUTPUT
```

```
                printf("%d/%d ",
                       provider->gs_instance_number,
                       provider->gs_node_number);
#else
                /* do not print out the int value. */
                printf("%d/%d(%d) ",
                       provider->gs_instance_number,
                       provider->gs_node_number,
                       provider->gs_provider_id);
#endif  /* if ! _VERBOSE_PROVIDER_OUTPUT */
                provider++;
            }
            printf("]");
        }
        printf("]\n");
        fflush(stdout);
    }

    if (!full_proposal) {
        printf(asterisks);
        fflush(stdout);
        return;
    }

    whatchanged = proposal->gs_whats_changed;

    if ((HA_GS_PROPOSED_MEMBERSHIP & whatchanged) ||
        (HA_GS_ONGOING_MEMBERSHIP & whatchanged) ||
        (HA_GS_UPDATED_MEMBERSHIP & whatchanged) ||
        (HA_GS_REJECTED_MEMBERSHIP & whatchanged)) {
        membership = proposal->gs_changing_providers;
        printf("ChangingProviders[");
        if (membership->gs_count == 0) {
            printf("No changing providers!");
        } else {
            printf("count[%d] Members[", membership->gs_count);
            provider = membership->gs_providers;
            for (i = 0; i < membership->gs_count; i++) {

                /* If this is a join, then the "proposer" id will always be us. */
                /* If we are in the changing providers list, this is "our" join, */
                /* so grab the provider id. */
                if ((HA_GS_JOIN == pType) &&
                    (HA_GS_APPROVED_NOTIFICATION == whatItBe) &&
                    (provider->gs_provider_id == pProposer.gs_provider_id)) {

                    provId[groupIdx] = pProposer;
                    grabbedIt = 1;
                }

                /* If this is a "voluntary" leave, then maybe it is us that is */
                /* leaving.  In this case, the proposer should be us, and the changing */
                /* provider list better also be us.  Note that if we are the provider */
                /* leaving, then this will be the only notification we will get! */
                if (HA_GS_LEAVE == pType) {
                    if (provId[groupIdx].gs_provider_id == pProposer.gs_provider_id) {
                        iHaveLeft = 1;
                    }
                    if (provider->gs_provider_id != pProposer.gs_provider_id) {
#ifndef _VERBOSE_PROVIDER_OUTPUT
                        printf("\nError: proposer is [%d/%d] but changer is:",
                               pProposer.gs_instance_number,
                               pProposer.gs_node_number);
#else
                        /* do not print out the int value. */
                        printf("\nError: proposer is [%d/%d(%d)] but changer is:",
                               pProposer.gs_instance_number,
                               pProposer.gs_node_number,
                               pProposer.gs_provider_id);
#endif  /* if ! _VERBOSE_PROVIDER_OUTPUT */
                    }
                }
```

```
                /* If *WE* have just been the target of an expel, and that expel */
                /* was approved, bad news!  We have been tossed from the group! */
                if ((HA_GS_EXPEL == pType) &&
                    (provId[groupIdx].gs_provider_id == provider->gs_provider_id)) {
                    iBeenExpelled = 1;
                    if (provider->gs_provider_id == pProposer.gs_provider_id) {
                        expelledMyself = 1;
                    }
                }
#ifndef _VERBOSE_PROVIDER_OUTPUT
                printf("%d/%d ",
                       provider->gs_instance_number,
                       provider->gs_node_number);
#else
                /* do not print out the int value. */
                printf("%d/%d(%d) ",
                       provider->gs_instance_number,
                       provider->gs_node_number,
                       provider->gs_provider_id);
#endif  /* if ! _VERBOSE_PROVIDER_OUTPUT */
                provider++;
            }
            printf("]");
        }
        printf("]\n");

        if (grabbedIt) {
#ifndef _VERBOSE_PROVIDER_OUTPUT
            printf(">>>> Grabbed my provider id [%d/%d]\n",
                   provId[groupIdx].gs_instance_number,
                   provId[groupIdx].gs_node_number);
#else
            /* do not print out the int value. */
            printf(">>>> Grabbed my provider id [%d/%d(%d)]\n",
                   provId[groupIdx].gs_instance_number,
                   provId[groupIdx].gs_node_number,
                   provId[groupIdx].gs_provider_id);
#endif  /* if ! _VERBOSE_PROVIDER_OUTPUT */
        }
        if (iHaveLeft) {
#ifndef _VERBOSE_PROVIDER_OUTPUT
            printf(">>>> I have left the group, provider id was [%d/%d]\n",
                   provId[groupIdx].gs_instance_number,
                   provId[groupIdx].gs_node_number);
#else
            /* do not print out the int value. */
            printf(">>>> I have left the group, provider id was [%d/%d(%d)]\n",
                   provId[groupIdx].gs_instance_number,
                   provId[groupIdx].gs_node_number,
                   provId[groupIdx].gs_provider_id);
#endif  /* if ! _VERBOSE_PROVIDER_OUTPUT */
            provId[groupIdx].gs_provider_id = -1;
            in_group[groupIdx] = 0;
            in_group_count--;
        }
        if (iBeenExpelled) {
            if (HA_GS_APPROVED_NOTIFICATION == whatItBe) {
                if (expelledMyself) {
#ifndef _VERBOSE_PROVIDER_OUTPUT
                    printf(">>>> I've expelled myself [%d/%d] from the group!  Ouch!\n",
                           provId[groupIdx].gs_instance_number,
                           provId[groupIdx].gs_node_number);
#else
                    /* do not print out the int value. */
                    printf(">>>> I've expelled myself [%d/%d(%d)] from the group!  Ouch!\n",
                           provId[groupIdx].gs_instance_number,
                           provId[groupIdx].gs_node_number,
                           provId[groupIdx].gs_provider_id);
#endif  /* if ! _VERBOSE_PROVIDER_OUTPUT */
                } else {
#ifndef _VERBOSE_PROVIDER_OUTPUT
```

```
                    printf(">>>> They've expelled me [%d/%d] from the group!  The rats!\n",
                           provId[groupIdx].gs_instance_number,
                           provId[groupIdx].gs_node_number);
#else
                    /* do not print out the int value. */
                    printf(">>>> They've expelled me [%d/%d(%d)] from the group!  The rats!\n",
                           provId[groupIdx].gs_instance_number,
                           provId[groupIdx].gs_node_number,
                           provId[groupIdx].gs_provider_id);
#endif  /* if ! _VERBOSE_PROVIDER_OUTPUT */
                }
                provId[groupIdx].gs_provider_id = -1;
                in_group[groupIdx] = 0;
                in_group_count--;
            } else if (HA_GS_REJECTED_NOTIFICATION == whatItBe) {
                if (expelledMyself) {
#ifndef _VERBOSE_PROVIDER_OUTPUT
                    printf("I failed to expel myself [%d/%d].  Strange, but true!\n",
                           provId[groupIdx].gs_instance_number,
                           provId[groupIdx].gs_node_number);
#else
                    /* do not print out the int value. */
                    printf("I failed to expel myself [%d/%d(%d)].  Strange, but true!\n",
                           provId[groupIdx].gs_instance_number,
                           provId[groupIdx].gs_node_number,
                           provId[groupIdx].gs_provider_id);
#endif  /* if ! _VERBOSE_PROVIDER_OUTPUT */
                } else {
#ifndef _VERBOSE_PROVIDER_OUTPUT
                    printf("They tried to expel me [%d/%d] but failed.  HA!\n",
                           provId[groupIdx].gs_instance_number,
                           provId[groupIdx].gs_node_number);
#else
                    /* do not print out the int value. */
                    printf("They tried to expel me [%d/%d(%d)] but failed.  HA!\n",
                           provId[groupIdx].gs_instance_number,
                           provId[groupIdx].gs_node_number,
                           provId[groupIdx].gs_provider_id);
#endif  /* if ! _VERBOSE_PROVIDER_OUTPUT */
                }
            }
        }
    }

    /* An expel protocol only carries the leave info on the approval notification. */

    if ((HA_GS_FAILURE_LEAVE == note->gs_protocol_type) ||
        (HA_GS_LEAVE == note->gs_protocol_type) ||
        ((HA_GS_EXPEL == note->gs_protocol_type) &&
         (HA_GS_APPROVED_NOTIFICATION == whatItBe)) ||
        (HA_GS_CAST_OUT == note->gs_protocol_type)) {
        printf("LeaveInfo[");
        if (NULL == (leave_array = proposal->gs_leave_info)) {
            printf("No leave info?!?");
        } else {
            leave_info = leave_array->gs_leave_codes;
            for (i = 0; i < leave_array->gs_count; i++) {
                printf("[%d: reason[%s] code[%d]]\n",
                       i,
                       write_leave_reason(leave_info->gs_voluntary_or_failure),
                       leave_info->gs_voluntary_leave_code);
                leave_info++;
            }
        }
        printf("]\n");
    }

    /*
     * For an expel, we would have already written the list of targeted
     * providers earlier (they are in the "changing providers" field).  Here,
     * display the deactivate script information - phase and flag.
     */
```

```
    if (HA_GS_EXPEL_INFORMATION & whatchanged) {
        if (HA_GS_EXPEL == note->gs_protocol_type) {
            printf("ExpelInfo[");
            if (NULL == (expel_info = proposal->gs_expel_info)) {
                printf("No expel info?!?");
            } else {
                printf("Deactivate script execution phase [%d]\n flag:[%s]",
                       expel_info->gs_deactivate_phase,
                       expel_info->gs_expel_flag);
            }
            printf("]\n");
        } else {
            printf("Have expel information but not an expel protocol notification?!?\n");
        }
    }

    /*
     * We make the assumption that we only join the pre-defined set of groups,
     * which will always enforce a set format to their state values:
     *  int
     *  tag
     *  rest of the data.
     * Therefore, we simply display the state value using this format.  If
     * you modify this program to join arbitrary groups, this display method
     * may not properly work.
     *
     * Please refer to the routine "write_arbitrary_value()" where we attempt
     * (weakly) to deal with displaying arbitrary group state values.
     */

    state = proposal->gs_current_state_value;
    printf("CurrentState[");
    if (0 >= state->gs_length) {
        printf("No current state!?!");
    } else {
        printf("Length[%d] Value[", state->gs_length);
        memcpy(&stpbmint, &state->gs_state[0], sizeof(stpbmint));
        printf("%d", stpbmint);
        for (i=sizeof(stpbmint); i<state->gs_length; i++) {
            printf("%c",  state->gs_state[i]);
        }
        printf("]");
    }
    printf("]\n");

    if ((HA_GS_PROPOSED_STATE_VALUE & whatchanged) ||
        (HA_GS_ONGOING_STATE_VALUE & whatchanged) ||
        (HA_GS_UPDATED_STATE_VALUE & whatchanged)) {
        if ((state = proposal->gs_proposed_state_value) != 0) {
            printf("ProposedState[");
            if (state->gs_length == 0) {
                printf("No proposed state");
            } else {
                printf("Length[%d] Value[", state->gs_length);
                memcpy(&stpbmint, &state->gs_state[0], sizeof(stpbmint));
                printf("%d", stpbmint);
                for (i=sizeof(stpbmint); i<state->gs_length; i++) {
                    printf("%c",  state->gs_state[i]);
                }
                printf("]");
            }
            printf("]\n");
        }
    }

    if (HA_GS_UPDATED_PROVIDER_MESSAGE & whatchanged) {
        if ((pbm = proposal->gs_provider_message) != 0) {
            printf("ProviderMessage[");
            if (pbm->gs_length == 0) {
                printf("No message");
            } else {
                printf("Length[%d] Value[", pbm->gs_length);
```

```
                memcpy(&stpbmint, &pbm->gs_message[0], sizeof(stpbmint));
                printf("%d", stpbmint);
                for (i=sizeof(stpbmint); i<pbm->gs_length; i++) {
                    printf("%c", pbm->gs_message[i]);
                }
                printf("]");
            }
            printf("]\n");
        }
    }

    if (HA_GS_REFLECTED_SOURCE_STATE_VALUE & whatchanged) {
        if ((state = proposal->gs_source_state_value) != 0) {
            printf("SourceState[");
            if (state->gs_length == 0) {
                printf("No source state");
            } else {
                printf("Length[%d] Value[", state->gs_length);
                memcpy(&stpbmint, &state->gs_state[0], sizeof(stpbmint));
                printf("%d", stpbmint);
                for (i=sizeof(stpbmint); i<state->gs_length; i++) {
                    printf("%c",  state->gs_state[i]);
                }
                printf("]");
            }
            printf("]\n");
        }
    }

    /* Do we have a new set of group attributes? */
    if ((HA_GS_UPDATED_GROUP_ATTRIBUTES & whatchanged) ||
        (HA_GS_PROPOSED_GROUP_ATTRIBUTES & whatchanged) ||
        (HA_GS_ONGOING_GROUP_ATTRIBUTES & whatchanged) ||
        (HA_GS_REJECTED_GROUP_ATTRIBUTES & whatchanged)) {

        if ((HA_GS_PROPOSED_GROUP_ATTRIBUTES & whatchanged) ||
            (HA_GS_ONGOING_GROUP_ATTRIBUTES & whatchanged)) {
            printf("New group attributes proposed for the group!\n");
            write_the_attributes(proposal->gs_new_group_attributes);
        } else if (HA_GS_UPDATED_GROUP_ATTRIBUTES & whatchanged) {
            printf("New group attributes established for the group!\n");
            write_the_attributes(proposal->gs_new_group_attributes);
#ifdef  _SAMPLE_TEST                    /* _SAMPLE_TEST */
            free(gattr[groupIdx]);
            gattr[groupIdx] = malloc(sizeof(ha_gs_group_attributes_t));
            copy_the_attributes(groupIdx,
                                gattr[groupIdx],
                                proposal->gs_new_group_attributes);
#endif                                  /* ifdef _SAMPLE_TEST */
        } else {
            printf("New group attributes rejected for the group!\n");
            write_the_attributes(proposal->gs_new_group_attributes);
        }
    }

    printf(asterisks);
    fflush(stdout);

    /* End: Write out the data received. */
    return;
}

/*
 * Display the protocol proposal information contained in a notification.
 */
void    write_the_proposal(ha_gs_request_t eReq,
                           ha_gs_token_t   eTok,
                           ha_gs_proposal_info_t *prop)
{
    int i, grpIdx;
    ha_gs_provider_t *_expellee;
```

```
    switch(eReq) {
      case HA_GS_JOIN:
        printf("Join proposal: Instance[%d]  Local name[%s]\n",
               prop->gs_join_request.gs_provider_instance,
               prop->gs_join_request.gs_provider_local_name);
        write_the_attributes(prop->gs_join_request.gs_group_attributes);
        break;
      case HA_GS_STATE_VALUE_CHANGE:
        printf("State change proposal: Number phases[%s]  Time limit[%d]\n",
               phase_print(prop->gs_state_change_request.gs_num_phases),
               prop->gs_state_change_request.gs_time_limit);
        printf("Proposed state: length[%d]\n",
               prop->gs_state_change_request.gs_new_state.gs_length);
        write_arbitrary_value(ARBITRARY_STATE_VALUE,
                              (void *)&(prop->gs_state_change_request.gs_new_state));
        break;
      case HA_GS_PROVIDER_MESSAGE:
        printf("Provider message proposal: Number phases[%s]  Time limit[%d]\n",
               phase_print(prop->gs_message_request.gs_num_phases),
               prop->gs_message_request.gs_time_limit);
        printf("Provider message: length[%d]\n",
               prop->gs_message_request.gs_message.gs_length);
        write_arbitrary_value(ARBITRARY_PROV_MESSAGE,
                              (void *)&(prop->gs_message_request.gs_message));
        break;
      case HA_GS_LEAVE:
        printf("Voluntary leave proposal: Number phases[%s]  Time limit[%d]\n",
               phase_print(prop->gs_leave_request.gs_num_phases),
               prop->gs_leave_request.gs_time_limit);
        printf("Leave code: value[%d]\n",
               prop->gs_leave_request.gs_leave_code);
        break;
      case HA_GS_EXPEL:
        printf("Expel proposal: Number phases[%s]  Time limit[%d]\n",
               phase_print(prop->gs_leave_request.gs_num_phases),
               prop->gs_expel_request.gs_time_limit);
        printf("Deactivate phase[%d] Number of providers[%d]\n",
               prop->gs_expel_request.gs_deactivate_phase,
               prop->gs_expel_request.gs_expel_list.gs_count);
        printf("Targeted providers[");
        for (i = prop->gs_expel_request.gs_expel_list.gs_count,
                  _expellee = prop->gs_expel_request.gs_expel_list.gs_providers;
             0 < i;
             i--, _expellee++) {
#ifndef _VERBOSE_PROVIDER_OUTPUT
            printf("%d/%d ",
                   _expellee->gs_instance_number,
                   _expellee->gs_node_number);
#else
            /* do not print out the int value. */
            printf("%d/%d(%d) ",
                   _expellee->gs_instance_number,
                   _expellee->gs_node_number,
                   _expellee->gs_provider_id);
#endif  /* if ! _VERBOSE_PROVIDER_OUTPUT */
            _expellee++;
        }
        printf("]\n");
        if (NULL != prop->gs_expel_request.gs_deactivate_flag) {
            printf("Deactivate flag[%s]\n",
                   prop->gs_expel_request.gs_deactivate_flag);
        } else {
            printf("No deactivate flag given.\n");
        }
        fflush(stdout);
        break;      case HA_GS_GROUP_ATTRIBUTE_CHANGE:
        printf("Change attributes proposal: Number of phases[%s]  Time limit[%d]\n",
               phase_print(prop->gs_attribute_change_request.gs_num_phases),
               prop->gs_attribute_change_request.gs_time_limit);
        write_the_attributes(prop->gs_attribute_change_request.gs_group_attributes);
        fflush(stdout);
        break;
```

```
      case HA_GS_MERGE:
      case HA_GS_FAILURE_LEAVE:
      case HA_GS_CAST_OUT:
      case HA_GS_SOURCE_STATE_REFLECTION:
        printf("Merge/failure/cast out/reflection protocols should never be here!\n");
        break;
      case HA_GS_SUBSCRIPTION:
        printf("subscription request:\nGroup[%s] Subscription desired[%s]\n",
               prop->gs_subscribe_request.gs_subscription_group,
               write_sub_ctrl(prop->gs_subscribe_request.gs_subscription_control));
#ifdef  _SAMPLE_TEST                    /* _SAMPLE_TEST */
        if (-1 != (i = get_sub_index(eTok))) {
            sid[i] = -1;
            subCtrl[i] = 0;
            free(subNames[i]);
            subNames[i] = 0;
            subscribed_to_count--;
            break;
        } else {
            printf("Received delayed error on unkown subscription token [%d]\n",
                   eTok);
        }
#endif                                  /* ifdef _SAMPLE_TEST */
        break;

      default:
        printf("Bad news, bucko.  I don't understand the given request[%d]!\n",
              eReq);
    }

    fflush(stdout);
    return;
}

/*
 * Display the information contained in a delayed error notification.
 */
void    write_the_delayed_error(const ha_gs_delayed_error_notification_t *note)
{
    ha_gs_token_t   eToken;
    ha_gs_request_t eRequest;
    ha_gs_rc_t      eRC;

    eToken = note->gs_request_token;
    eRequest = note->gs_protocol_type;
    eRC = note->gs_delayed_return_code;

    printf(starz);
    printf("Very sorry to report that your proposal request is erroneous.\n");
    printf(" Token[%d]  Request[%s]  Return code[%s]\n",
           eToken,
           proto_type(eRequest),
           write_an_rc(eRC));
    write_the_proposal(eRequest, eToken, note->gs_failing_request);
    printf(starz);
    fflush(stdout);
    return;
}

#ifdef  _SAMPLE_TEST                    /* _SAMPLE_TEST */

/*
 * If it is given, display the "special" data from a subscription
 * notification.
 */

void    write_the_special_flag(unsigned int     flag)
{
    if (HA_GS_ADAPTER_DEATH_ARRAY & flag) {
        printf(" HA_GS_ADAPTER_DEATH_ARRAY");
    }
    if (HA_GS_CURRENT_ADAPTER_ALIAS_ARRAY & flag) {
```

```
        printf(" HA_GS_CURRENT_ADAPTER_ALIAS_ARRAY");
    }
    if (HA_GS_CHANGING_ADAPTER_ALIAS_ARRAY & flag) {
        printf(" HA_GS_CHANGING_ADAPTER_ALIAS_ARRAY");
    }

    return;
}

/*
 * We have a set of adapter alias (IP) addresses.  Print each out.
 */
void    write_special_alias_array(const ha_gs_special_block_t *block)
{
    unsigned int *IPaddr;
    int           count, length, increment, i, j, k;
    char         *IPprint;

    count = block->gs_special_num_entries;
    length = block->gs_special_length;
    increment = length / 4;

    /*
     * Write the addresses in the order given.
     */

    IPaddr = (unsigned int *)block->gs_special;
    for (i = 0, j = 0; i < count; i++, j += k) {
        IPprint = inet_ntoa(*(struct in_addr *)IPaddr);
        k = strlen(IPprint) + 1;
        if (80 < (j + k)) {
            printf("\n");
            j = 0;
        }
        printf("%s ", IPprint);
        IPaddr += increment;
    }
    if (0 != j) printf("\n");
    fflush(stdout);
    return;
}

/*
 * We have a set of "death reasons" for each specified adapter.
 */
void    write_special_death_array(const ha_gs_special_block_t *block)
{
    ha_gs_adapter_death_t *why_died;
    int           count, length, i, j;

    count = block->gs_special_num_entries;
    length = block->gs_special_length;

    why_died = (ha_gs_adapter_death_t *)block->gs_special;
    for (i = 0, j = 0; i < count; i++, why_died++) {
        if (HA_GS_ADAPTER_REMOVED == *why_died) {
            printf("%d:Removed! ", i);
            j += 2;
        } else if (HA_GS_ADAPTER_DEAD == *why_died) {
            printf("%d:Died ", i);
            j += 1;
        } else {
            printf("%d:Unknown? ", i);
            j += 2;
        }
        if (8 <= j) {
            printf("\n");
            j = 0;
        }
    }
    if (0 != j) printf("\n");
    fflush(stdout);
```

```
    return;
}

/*
 * Control writing out the special data.  Find the individual blocks.
 */
void    write_the_special_data(const ha_gs_special_data_t *special)
{
    ha_gs_special_block_t *special_block;
    int block_count;

    printf("\nThe subscription has 'special' data.  How special!\n");
    printf("Special data contains [%d] block%s  Flags:\n",
           special->gs_length,
           ((1 == special->gs_length) ? "." : "s."));
    write_the_special_flag(special->gs_flag);
    printf("\nDisplaying data:\n");

    special_block = (ha_gs_special_block_t *)special->gs_special_data;
    block_count = 1;

    while (NULL != special_block) {
        printf("Special block %d:  Flag:",
               block_count);
        write_the_special_flag(special_block->gs_special_flag);
        printf("\n Number entries [%d]  Entry length [%d]  Data pointer [0x%x]\n",
               special_block->gs_special_num_entries,
               special_block->gs_special_length,
               special_block->gs_special);
        fflush(stdout);
        if (HA_GS_ADAPTER_DEATH_ARRAY & special_block->gs_special_flag) {
            write_special_death_array(special_block);
        } else if (HA_GS_CURRENT_ADAPTER_ALIAS_ARRAY & special_block->gs_special_flag) {
            write_special_alias_array(special_block);
        } else if (HA_GS_CHANGING_ADAPTER_ALIAS_ARRAY & special_block->gs_special_flag) {
            write_special_alias_array(special_block);
        } else {
            printf("Unknown special data block type [%d] !!!\n",
                   special_block->gs_special_flag);
        }

        fflush(stdout);
        special_block = special_block->gs_next_special_block;
        block_count++;
    }

    return;
}
/*
 * Display the information contained in a subscription notification.
 */
void    write_the_subscription(const ha_gs_subscription_notification_t *note)
{
    int i, len, isize;
    int stpbmint, grpIdx, predefSub;

    ha_gs_membership_t     *membership;
    ha_gs_provider_t       *provider;
    ha_gs_state_value_t    *state;

    printf("************************************************************************\n");

    printf("Type[%s] Token[%d] GroupName[",
           proto_type(HA_GS_SUBSCRIPTION),
           note->gs_subscriber_token);

    predefSub = 0;                      /* one of sample_test's predefined groups? */

    if (-1 != (grpIdx = get_sub_index(note->gs_subscriber_token))) {
        printf("%s",
               subNames[grpIdx]);
        for (i = 0;
```

```
             ((i < num_groups) && (!predefSub));
             i++) {
            if (!strcmp(group_names[i], subNames[grpIdx])) {
                predefSub = 1;
            }
        }
    } else {
        printf("Unknown!!");
        grpIdx = -2;
    }
    printf("]\n");

    write_sub_type(note->gs_subscription_type);

    if (HA_GS_SUBSCRIPTION_MEMBERSHIP & note->gs_subscription_type) {
        membership = note->gs_full_membership;
        printf("CurrentProviders[");
        if (membership->gs_count == 0) {
            printf("No providers in list!");
        } else {
            printf("count[%d] Members[", membership->gs_count);
            provider = membership->gs_providers;
            for (i = 0; i < membership->gs_count; i++) {
#ifndef _VERBOSE_PROVIDER_OUTPUT
                printf("%d/%d ",
                       provider->gs_instance_number,
                       provider->gs_node_number);
#else
                /* do not print out the int value. */
                printf("%d/%d(%d) ",
                       provider->gs_instance_number,
                       provider->gs_node_number,
                       provider->gs_provider_id);
#endif  /* if ! _VERBOSE_PROVIDER_OUTPUT */
                provider++;
            }
            printf("]");
        }
        printf("]\n");
    }

    if (HA_GS_SUBSCRIPTION_DELTA_JOIN & note->gs_subscription_type) {
        membership = note->gs_changing_membership;
        printf("JoiningProviders[");
        if (membership->gs_count == 0) {
            printf("No changing providers!");
        } else {
            printf("count[%d] Members[", membership->gs_count);
            provider = membership->gs_providers;
            for (i = 0; i < membership->gs_count; i++) {
#ifndef _VERBOSE_PROVIDER_OUTPUT
                printf("%d/%d ",
                       provider->gs_instance_number,
                       provider->gs_node_number);
#else
                /* do not print out the int value. */
                printf("%d/%d(%d) ",
                       provider->gs_instance_number,
                       provider->gs_node_number,
                       provider->gs_provider_id);
#endif  /* if ! _VERBOSE_PROVIDER_OUTPUT */
                provider++;
            }
            printf("]");
        }
        printf("]\n");
    }
    if (HA_GS_SUBSCRIPTION_DELTA_LEAVE & note->gs_subscription_type) {
        membership = note->gs_changing_membership;
        printf("LeavingProviders[");
        if (membership->gs_count == 0) {
            printf("No changing providers!");
```

```
        } else {
            printf("count[%d] Members[", membership->gs_count);
            provider = membership->gs_providers;
            for (i = 0; i < membership->gs_count; i++) {
#ifndef _VERBOSE_PROVIDER_OUTPUT
                printf("%d/%d ",
                       provider->gs_instance_number,
                       provider->gs_node_number);
#else
                /* do not print out the int value. */
                printf("%d/%d(%d) ",
                       provider->gs_instance_number,
                       provider->gs_node_number,
                       provider->gs_provider_id);
#endif  /* if ! _VERBOSE_PROVIDER_OUTPUT */
                provider++;
            }
            printf("]");
        }
        printf("]\n");
    }

    if (HA_GS_SUBSCRIPTION_STATE & note->gs_subscription_type) {
        state = note->gs_state_value;
        printf("CurrentState[");
        if (0 >= state->gs_length) {
            printf("No current state!?!");
        } else {
            len = state->gs_length;
            printf("Length[%d] Value[", len);

            if (predefSub) {
                /*
                 * For the "pre-defined" set of groups, we know that
                 * the format of the state values is predictable.
                 */
                memcpy(&stpbmint, &state->gs_state[0], sizeof(stpbmint));
                printf("%d", stpbmint);
                for (i=sizeof(stpbmint); i<state->gs_length; i++) {
                    printf("%c",  state->gs_state[i]);
                }
            } else {
                write_arbitrary_value(ARBITRARY_STATE_VALUE, (void *)state);
            }
            printf("]");
        }
        printf("]\n");
    }

    if (HA_GS_SUBSCRIPTION_SPECIAL_DATA & note->gs_subscription_type) {
    write_the_special_data(note->gs_subscription_special_data);
    }

    if (HA_GS_SUBSCRIPTION_DISSOLVED & note->gs_subscription_type) {
        printf("\n Subscription has been DISSOLVED.\n");
        if (HA_GS_SUBSCRIPTION_GS_HAS_DIED & note->gs_subscription_type) {
            printf("   Group Services has gone away, taking all groups with it...\n\n");
        } else {
            printf("   Group has gone away...\n\n");
        }
        if (-2 != grpIdx) {
            sid[grpIdx] = -1;
            subCtrl[grpIdx] = 0;
            free(subNames[grpIdx]);
            subNames[grpIdx] = 0;
            subscribed_to_count--;
        }
    }

    printf("********************************************************************\n");
    fflush(stdout);
```

```
    return;
}

/*********************************************************************/
/*
 * The user wants to subscribe to a group.  Find a free slot in our
 * subscriber token array.
 */
int     find_free_sub_slot()
{
    int i, slot;

    for (i = 0;
         i < num_groups_for_subscribe;
         i++) {
        if (-1 == sid[i]) {
            slot = i;
            break;
        }
    }

    return(slot);
}

/*
 * A subscription notification has arrived.  Find the given token in
 * our subscriber token array.
 */
int     get_sub_index(ha_gs_token_t token)
{
    int  i, found;

    found = 0;
    for (i = 0;
         i < num_groups_for_subscribe;
         i++) {
        if (token == sid[i]) {
            found = 1;
            break;
        }
    }
    if (!found) {
        printf("Received subscriber token [%d], but cannot find it in subscription array!\n",
               token);
        i = -1;
    }

    return(i);
}

/*********************************************************************/
/*
 * These routines are used when the user is interactively constructing
 * a protocol proposal to request the various bits of information
 * needed.
 */

/*
 * Determine to which group this proposal request will be directed.
 */
ha_gs_token_t       get_group_token(int *gAddr)
{
    ha_gs_token_t   _gTok;
    int _goodGroup, _insultLevel;
    char _input;
    int _numput;

    _gTok = -1;
    _insultLevel = 0;

    if (1 == in_group_count) {
        for (_goodGroup = 0;
```

```
             num_groups > _goodGroup;
             _goodGroup++) {
            if (in_group[_goodGroup]) {
                _gTok = gid[_goodGroup];
                *gAddr = _goodGroup;
                break;
            }
        }
        printf("I would ask you to which group is this proposal to be targeted, but since\n");
        printf(" you are only in the group named [%s] you are going to use that one!\n",
               group_names[_goodGroup]);
        fflush(stdout);
        return(_gTok);
    }

    while (_gTok == -1) {
        printf("You need to specify to which group this proposal is targeted.\n");
        printf(" There are [%d] groups to choose from, choose one by its index number:\n",
               in_group_count);
        fflush(stdout);
        for (_goodGroup = 0;
             num_groups > _goodGroup;
             _goodGroup++) {
            if (in_group[_goodGroup]) {
                printf("  Index[%d]: Group name[%s]\n",
                       _goodGroup,
                       group_names[_goodGroup]);
            }
        }
        printf("Your choice, please?");
        fflush(stdout);
        scanf("%d%c", &_numput, &_input);
        if ((num_groups >= (unsigned int)_numput) && in_group[_numput]) {
            _gTok = gid[_numput];
            *gAddr = _numput;
            printf("You have chosen group [%s].  Good luck.\n",
                   group_names[_numput]);
        } else {
            switch(_insultLevel) {
              case 0:
                printf("I'm sorry, there is no valid group at index [%d]!  Try again!\n",
                       _numput);
                break;
              case 1:
                printf("I'm sorry, there is no valid group at index [%d]!  Concentrate!\n",
                       _numput);
                break;
              case 2:
                printf("Look, [%d] is NOT a valid index.  Read the instructions...\n",
                       _numput);
                break;
              case 3:
                printf("I don't know about you, but I have better things to do than to\n");
                printf("sit in this silly loop with you giving me invalid index values\n");
                printf("such as [%d]!!  Please try one more time.\n",
                       _numput);
                break;
              default:
                printf("I am very sorry.  5 chances to pick %d valid index number.  Buut\n",
                       in_group_count);
                printf("nnoooo.  Well, time to give it up.  I'm outta here.\n");
                return(_gTok);
            }
            fflush(stdout);
            _insultLevel++;
        }
    }

    return(_gTok);
}

/*
```

```
 * Ask for the number of phases (1 or n) for this proposal.
 */
ha_gs_num_phases_t      get_number_of_phases()
{
    int _phases;
    ha_gs_num_phases_t _numPhases;

    char    _input, _newline;
    char    _inline[80];
    char    _intest;

    _phases = -1;

    while(-1 == _phases) {
        printf("Number of phases('1' or 'N'): "); fflush(stdout);
        gets(_inline);
        _intest = _inline[0];
        _phases = 1;
        switch(_intest) {
          case '1':
            _numPhases = HA_GS_1_PHASE;
            break;
          case 'n':
          case 'N':
            _numPhases = HA_GS_N_PHASE;
            break;
          default:
            printf("Very sorry, but [%c] is not acceptable for number of phases.\n",
                   _intest);
            _phases = -1;
        }
    }

    return(_numPhases);
}

/*
 * Constructing our own group attributes, determine the batch control
 * attribute.
 */
ha_gs_batch_ctrl_t   get_batch_control()
{
    int  _numput, _good;
    char _input, _newline;

    printf("Please specify the group's batch control attribute.\n");
    _good = 0;
    while (!_good) {
        printf("Enter: 0 (no batching) 1 (joins only) 2 (failures only) 3 (joins and failures)\n");
      printf("       (add 4 for deact-on-failure, i.e., 5=deact-on-failure & joins)\n");
        fflush(stdout);
        scanf("%d%c", &_numput, &_newline);
        switch(_numput) {
          case 0:
            return(HA_GS_NO_BATCHING);
          case 1:
            return(HA_GS_BATCH_JOINS);
          case 2:
            return(HA_GS_BATCH_LEAVES);
          case 3:
            return(HA_GS_BATCH_BOTH);
        case 4:
          return HA_GS_DEACTIVATE_ON_FAILURE;
        case 5:
          return HA_GS_DEACTIVATE_ON_FAILURE|HA_GS_BATCH_JOINS;
        case 6:
          return HA_GS_DEACTIVATE_ON_FAILURE|HA_GS_BATCH_LEAVES;
        case 7:
          return HA_GS_DEACTIVATE_ON_FAILURE|HA_GS_BATCH_BOTH;
          default:
            printf("Invalid batch control entry [%d].  Try again.\n", _numput);
        }
```

```
    }

    return(HA_GS_NO_BATCHING);
}

/*
 * Constructing our own group attributes, determine the base default
 * group vote attribute.
 */
ha_gs_vote_value_t       get_default_vote()
{
    int  _numput, _good;
    char _input, _newline;

    printf("Please specify the group's default vote attribute.\n");
    _good = 0;
    while (!_good) {
        printf("Enter: 0 (APPROVE) 1 (REJECT)\n");
        fflush(stdout);
        scanf("%d%c", &_numput, &_newline);
        switch(_numput) {
          case 0:
            return(HA_GS_VOTE_APPROVE);
          case 1:
            return(HA_GS_VOTE_REJECT);
          default:
            printf("Invalid default vote entry [%d].  Try again.\n", _numput);
        }
    }

    return(HA_GS_VOTE_APPROVE);
}

/*
 * Ask for the time limit to be applied to a protocol proposal.
 */
ha_gs_time_limit_t      get_time_limit(char *_banner)
{
    int  _numput, _gIndex, _good;
    char _input, _newline;

    ha_gs_time_limit_t  _time;

    printf("Please specify time limit for %s.\n", _banner);
    printf(" In seconds, range 0 (no time limit) to 65,535: ");
    fflush(stdout);
    scanf("%d%c", &_numput, &_newline);
    _time = _numput;

    return(_time);
}

/*
 * Ask for the group name, for defining our own group, or subscribing to
 * an arbitrary group.
 */
char    *get_group_name(char *_banner)
{
    int  _numput, _gIndex, _good;
    char _input, _newline;

    char *_pName;
    char  _nameWanted[80];

    printf("Specify %s.\n", _banner);
    fflush(stdout);
    scanf("%s", _nameWanted);
    _pName = malloc(strlen(_nameWanted) + 1);
    strcpy(_pName, _nameWanted);

    return(_pName);
}
```

```
/*
 * Ask for the socket control setting if the user is interactively
 * specifying initialization parameters.
 */
ha_gs_socket_ctrl_t      build_socket_control()
{
    int  _numput, _good;
    char _input, _newline;

    printf("Please specify the client's socket control attribute.\n");
    printf("Enter: 0 (SIGNAL) anything else (NO SIGNAL): ");
    fflush(stdout);
    scanf("%d%c", &_numput, &_newline);
    switch(_numput) {
      case 0:
        printf("Using HA_GS_SOCKET_SIGNAL!  Brave move.\n");
        return(HA_GS_SOCKET_SIGNAL);
      default:
        printf("Using HA_GS_SOCKET_NO_SIGNAL!  Good idea.\n");
        return(HA_GS_SOCKET_NO_SIGNAL);
    }

    return(HA_GS_SOCKET_NO_SIGNAL);
}

/*
 * Ask for the name of the deactivate script if the user is interactively
 * specifying initialization parameters.
 */
char    *build_deact_script(char *default_deact)
{
    int  _numput, _gIndex, _good;
    char _input, _newline;

    char *_pName;
    char  _nameWanted[80];

    printf("The default deactivate script is[%s].\n  Keep it (0 [no] 1 [yes])? ",
           default_deact);
    scanf("%d%c", &_numput, &_newline);
    if (0 != _numput) {
        printf("Keeping default deactivatate script [%s].\n",
               default_deact);
        fflush(stdout);
        return(default_deact);
    }

    printf("Specify new deactivate script path name:\n");
    fflush(stdout);
    scanf("%s", _nameWanted);
    _pName = malloc(strlen(_nameWanted) + 1);
    strcpy(_pName, _nameWanted);

    return(_pName);
}

/*********************************************************************/
/*
 * These routines handle creating the responsiveness parameters if the
 * user is interactively specifying initialization parameters.
 */

/*
 * Display the responsiveness settings to be given on ha_gs_init().
 */
void    display_responsiveness(ha_gs_responsiveness_t *resp)
{
    int i;
    char *ctr;

    printf("Responsiveness parameters specified:  Type[%s]\n",
```

```
            responsive_print(resp->gs_responsiveness_type));

    if (HA_GS_NO_RESPONSIVENESS == resp->gs_responsiveness_type) {
        fflush(stdout);
        return;
    }

    printf("Interval[%d seconds] Response time limit[%d seconds]\n",
           resp->gs_responsiveness_interval,
           resp->gs_responsiveness_response_time_limit);

    if (HA_GS_COUNTER_RESPONSIVENESS == resp->gs_responsiveness_type) {
        printf(" Counter location[%x] Counter length[%d] Current value[\n",
               resp->gs_counter_location,
               resp->gs_counter_length);
        ctr = (char *)resp->gs_counter_location;
        for (i = 0;
             i < resp->gs_counter_length;
             i++, ctr++) {
            printf("%1.1x",
                   (void *)ctr);
        }
        printf("]\n");
    }
    fflush(stdout);

    return;
}

/*
 * Ask the user for the desired responsiveness settings for submission
 * on the ha_gs_init() call.
 */
ha_gs_responsiveness_t   *build_responsiveness(ha_gs_responsiveness_t *def_resp)
{
    ha_gs_responsiveness_t      *_response;
    ha_gs_responsiveness_type_t        _type;
    unsigned int          _interval;
    ha_gs_time_limit_t          _time_limit;
    void                 *_location;
    unsigned int          _length;

    int  i,j,ping;
    int  _numput, _gIndex, _good;
    char _input, _newline;

    char *_pName;
    char  _nameWanted[80];
    char  _inline[80];
    char  _intest;

    if (!interactiveInit) {
        display_responsiveness(def_resp);
        return(def_resp);
    }

    _response = (ha_gs_responsiveness_t *)malloc(sizeof(ha_gs_responsiveness_t));
    memset(_response, '\0', sizeof(ha_gs_responsiveness_t));

    printf("What kind of responsiveness?\n 1 [Ping]  2 [Counter]  3 [None]: ");
    scanf("%d%c", &_numput, &_newline);
    if (2 == _numput) {
        printf("Using HA_GS_COUNTER_RESPONSIVENESS (currently not supported by HAGS)!\n");
        fflush(stdout);
        _type = HA_GS_COUNTER_RESPONSIVENESS;
        ping = 0;
    } else if (1 == _numput) {
        printf("Using HA_GS_PING_RESPONSIVENESS.  Good idea.\n");
        fflush(stdout);
        _type = HA_GS_PING_RESPONSIVENESS;
        ping = 1;
    } else {
```

```
        printf("No responsiveness checks?  You a coward, or what?\n");
        fflush(stdout);
        _type = HA_GS_NO_RESPONSIVENESS;
        ping = 2;
    }

    if (2 > ping) {

        printf("Specify the interval (in seconds) between responsiveness checks: ");
        fflush(stdout);
        scanf("%d%c", &_numput, &_newline);
        _interval = _numput;

        printf("Specify time limit (in seconds) for response to responsiveness check: ");
        fflush(stdout);
        scanf("%d%c", &_numput, &_newline);
        _time_limit = _numput;

        if (1 == ping) {
            _location = 0;
            _length = 0;
        } else {
            printf("Specify counter location for responsiveness check: ");
            fflush(stdout);
            scanf("%x%c", &_location, &_newline);

            printf("Specify counter length for responsiveness check: ");
            fflush(stdout);
            scanf("%d%c", &_numput, &_newline);
            _length = _numput;
        }

        _response->gs_responsiveness_type = _type;
        _response->gs_responsiveness_interval = _interval;
        _response->gs_responsiveness_response_time_limit = _time_limit;
        _response->gs_counter_location = _location;
        _response->gs_counter_length = _length;
    }

    display_responsiveness(_response);

    if (HA_GS_NO_RESPONSIVENESS == _type) {
        return(_response);
    }

    printf("\nDo you want to respond manually or automatically to all responsiveness\n");
    printf(" notifications?  If auto, then we will always return OK.  If manual, you\n");
    printf(" will be able to specify OK or NOT_OK at notification time.\n");
    printf(" Specify 'A' [auto] or 'M' [manual] (default is auto): ");
    gets(_inline);
    _intest = _inline[0];
    switch(_intest) {
      case 'm':
      case 'M':
        printf("Manual responsiveness responses!\n");
        interactiveResponse = 1;
        break;
      default:
        printf("Automatic responsiveness responses!\n");
        interactiveResponse = 0;
    }

    return(_response);
}

/*
 * A responsiveness notification has arrived.  Ask the user if we
 * should return a "good" (HA_GS_CALLBACK_OK) or "bad" (HA_GS_CALLBACK_NOT_OK)
 * return code.
 */
ha_gs_callback_rc_t       get_response(ha_gs_time_limit_t _time)
{
```

```
    int  _numput;
    char _input, _newline;

    ha_gs_callback_rc_t  _rc;

    printf("Manual responsiveness mode!!!  You have [%d] seconds to respond!\a\a\a\n",
           _time);
    printf("\nSpecify 0 [to say NOT OK] or anything else [to say OK]:\n");
    fflush(stdout);
    scanf("%d%c", &_numput, &_newline);
    switch(_numput) {
      case 0:
        _rc = HA_GS_CALLBACK_NOT_OK;
        break;
      default:
        _rc = HA_GS_CALLBACK_OK;
    }

    return(_rc);
}

/*********************************************************************/
/*
 * This routine handles dealing with the user to construct the group
 * attributes to be submitted:
 * - on the ha_gs_join() for "user-defined" groups (via the 'b' command).
* - to change the attributes via ha_gs_change_attributes() (via 'n' command).
 */
ha_gs_group_attributes_t        *build_group_attributes(int _offset, int _changing)
{
    int  _numput, _gIndex, _good;
    char _input, _newline;

    ha_gs_num_phases_t  _phases;
    ha_gs_time_limit_t  _time;

    ha_gs_group_attributes_t *_attributes;

    _good = 1;
    _attributes = malloc(sizeof(ha_gs_group_attributes_t));

    _attributes->gs_version = HA_GS_RELEASE;
    _attributes->gs_sizeof_group_attributes = sizeof(ha_gs_group_attributes_t);

    if (_changing) {
        printf("You desire to change the group's attributes.  Brave!\n");
    } else {
        printf("Awesome!  You're going to build a group from scratch!  Good move!\n");
    }

    printf("Please define your group's attributes in response to each prompt.\n");
    printf(" 'Client version' (any integer): ");
    fflush(stdout);
    scanf("%d%c", &_numput, &_newline);
    _attributes->gs_client_version = _numput;

    _attributes->gs_batch_control = get_batch_control();

    printf("Please specify the number of phases for joins and failure protocols:\n");
    _attributes->gs_num_phases = get_number_of_phases();

    if (HA_GS_N_PHASE == _attributes->gs_num_phases) {
        _attributes->gs_time_limit = get_time_limit("join/failure protocol time limit");
    } else {
        _attributes->gs_time_limit = 0;
    }

    _attributes->gs_group_default_vote = get_default_vote();

    printf("Please specify the merge control setting:\n");
    printf("\t1:HA_GS_DISSOLVE_MERGE\n");
    printf("\t2:HA_GS_LARGER_MERGE\n");
```

```
printf("\t3:HA_GS_SMALLER_MERGE\n");
printf("\t4:HA_GS_DONTCARE_MERGE\n");
printf(" setting desired: ");
fflush(stdout);
scanf("%d%c", &_numput, &_newline);
switch(_numput) {
  case 1:
    _attributes->gs_merge_control = HA_GS_DISSOLVE_MERGE;
    break;
  case 2:
    _attributes->gs_merge_control = HA_GS_LARGER_MERGE;
    break;
  case 3:
    _attributes->gs_merge_control = HA_GS_SMALLER_MERGE;
    break;
  case 4:
    _attributes->gs_merge_control = HA_GS_DONTCARE_MERGE;
    break;
  default:
    _attributes->gs_merge_control = HA_GS_DISSOLVE_MERGE;
}

if (_changing) {
    _attributes->gs_group_name = NULL;
    _attributes->gs_source_group_name = NULL;

    printf("Please specify the number of phases for source-reflection protocols:\n");
    _attributes->gs_source_reflection_num_phases = get_number_of_phases();

    if (HA_GS_N_PHASE == _attributes->gs_source_reflection_num_phases) {
        _attributes->gs_source_reflection_time_limit = get_time_limit("source-reflection time limit");
    } else {
        _attributes->gs_source_reflection_time_limit = 0;
    }

    return(_attributes);
}

_attributes->gs_group_name = get_group_name("the group's name");
group_names[_offset] = malloc(strlen(_attributes->gs_group_name) + 1);
strcpy(group_names[_offset], _attributes->gs_group_name);

printf("Do you want to specify a source-group name (0[no] 1[yes])? ");
fflush(stdout);
scanf("%d%c", &_numput, &_newline);
if (_numput) {
    _attributes->gs_source_group_name = get_group_name("the source-group's name");
    source_group_names[_offset] = malloc(strlen(_attributes->gs_source_group_name) + 1);
    strcpy(source_group_names[_offset], _attributes->gs_source_group_name);

    printf("Please specify the number of phases for source-reflection protocols:\n");
    _attributes->gs_source_reflection_num_phases = get_number_of_phases();

    if (HA_GS_N_PHASE == _attributes->gs_source_reflection_num_phases) {
        _attributes->gs_source_reflection_time_limit = get_time_limit("source-reflection time limit");
    } else {
        _attributes->gs_source_reflection_time_limit = 0;
    }

} else {
    _attributes->gs_source_group_name = NULL;
    _attributes->gs_source_reflection_num_phases = HA_GS_1_PHASE;
    _attributes->gs_source_reflection_time_limit = 0;
}

printf("Please specify the provider instance number\n");
printf(" (if negative, use as offset and add to [%d]): ",
       instance_numbers[_offset]);
fflush(stdout);
scanf("%d%c", &_numput, &_newline);
if (0 > _numput) {
    instance_numbers[_offset] = (-1 * _numput) + instance_numbers[_offset];
```

```
    } else {
        instance_numbers[_offset] = _numput;
    }

    prov_local_names[_offset] = get_group_name("the provider's local name");

    return(_attributes);
}

/*********************************************************************/
/*
 * Request from the user a vote value, in response to an n-phase
 * notification arriving for a group from Group Services.
 */
ha_gs_vote_value_t get_a_vote(int phase)
{
    char  input, error;
    ha_gs_vote_value_t vote;
    char *instruct="nter a vote value: C [Continue]  A [Approve]  R [Reject]\n";
    char *instruct2=" or, to force 'nested' proposals: T [State change] P [PBM]\n";
    char *function;

    ha_gs_proposal_info_t info;
    ha_gs_token_t         group_token;
    ha_gs_rc_t            rc;

    vote = HA_GS_NULL_VOTE;

    while (HA_GS_NULL_VOTE == vote) {
        printf(starz);
        printf(starz);
        printf("%c%s", 'E', instruct);
        printf("%s", instruct2);
        fflush(stdout);
        while (scanf("%c", &input)) {
            switch(input) {
              case 'C':
              case 'c':
                vote = HA_GS_VOTE_CONTINUE;
                break;
              case 'A':
              case 'a':
                vote = HA_GS_VOTE_APPROVE;
                break;
              case 'R':
              case 'r':
                vote = HA_GS_VOTE_REJECT;
                break;
              case 'T':
              case 't':
                error = 't';
                vote = HA_GS_NULL_VOTE;
                break;
              case 'P':
              case 'p':
                error = 'p';
                vote = HA_GS_NULL_VOTE;
                break;
              case '\n':
                continue;
              default:
                printf("I do not understand your given vote value - %c - try again and please\n");
                printf("%c%s", 'e', instruct);
                printf("%s", instruct2);
                fflush(stdout);
                continue;
            }
            break;
        }
        if (HA_GS_NULL_VOTE != vote) {
            printf("You have chosen well, grasshopper. This protocol will ");
            if (HA_GS_VOTE_CONTINUE == vote) {
```

```
                printf("continue yet one more phase!\n");
            } else if (HA_GS_VOTE_APPROVE == vote) {
                printf("be approved here and now!\n");
            } else {
                printf("be rejected utterly!\n");
            }
        } else {
            printf("We're in a bad mood aren't we?  Hehehe.\n");
            function="could not create";
            if ('t' == error) {
                if (build_state_change(&group_token, &info)) {
                    function="ha_gs_change_state_value";
                    rc = ha_gs_change_state_value(group_token, &info);
                }
            } else {
                if (build_pbm(&group_token, &info)) {
                    function="ha_gs_send_message";
                    rc = ha_gs_send_message(group_token, &info);
                }
            }
            printf("Your nested %s proposal has been submitted, rc = %s\n",
                   function,
                   write_an_rc(rc));
        }
        printf(starz);
        fflush(stdout);
    }

    return(vote);
}

/*
 * Construct a group state value (for submission via a call to
 * ha_gs_change_state_value()), in response to a 't' command.
 */
int     build_state_change(ha_gs_token_t *gToken, ha_gs_proposal_info_t *proposal)
{
    int  _numput, _gIndex;
    char _input, _newline;

    ha_gs_num_phases_t  _phases;
    ha_gs_time_limit_t  _time;

    st_pbm_struct *_state;
    char   *_state2;
    char    _inline[80];
    char    _intest;
    int     _sLen;

    _phases = -1;
    _time = 0;

    printf(starz);
    printf("\nSo, you want to change the group's state value?\n");
    if (0 == in_group_count) {
        printf("Unfortunately, since you haven't yet joined any groups, that wouldn't be\n");
        printf(" a very good idea, would it, bucko?  Try again later, after you convince\n");
        printf(" some group to allow you to join.  Otherwise, leave me alone.\n");
        printf(starz);
        fflush(stdout);
        return(0);
    }

    if (-1 == (*gToken = get_group_token(&_gIndex))) {
        return(0);
    }

    printf("\nNow that you've chosen group [%s], we need the protocol information:\n",
           group_names[_gIndex]);

    _phases = get_number_of_phases();
```

```
    if (HA_GS_N_PHASE == _phases) {
        _time = get_time_limit("for the state value change protocol");
    } else {
        _time = 0;
    }

    printf("Please enter your proposed state value, anything up to 256 bytes:\n");
    fflush(stdout);
    _state = malloc(sizeof(st_pbm_struct));

    /* insert prefix.  */
    _state2 = &_state->st_pbm_data[sample_prefix_len];
    _state->st_pbm_index = sample_index;
    memcpy(&_state->st_pbm_data[0], sample_prefix, sample_prefix_len);

    gets(_state2);
    _sLen = sizeof(_state->st_pbm_index) + sample_prefix_len;
    if ((_sLen += strlen(_state2)) > 256) {
        printf("Your given state value is %d bytes long, I'm truncating it to 256 bytes.\n",
               _sLen);
        fflush(stdout);
        _sLen = 256;
    }
    printf("You have chosen [%s] with time limit of [%d] and a proposed state of:\n%d%s\n",
           phase_print(_phases),
           _time,
           _state->st_pbm_index,
           _state->st_pbm_data);

    printf("Thank you.  Will now submit your proposal.\n");

    proposal->gs_state_change_request.gs_num_phases = _phases;
    proposal->gs_state_change_request.gs_time_limit = _time;
    proposal->gs_state_change_request.gs_new_state.gs_length = _sLen;
    proposal->gs_state_change_request.gs_new_state.gs_state = (char *)_state;

    printf(starz);
    fflush(stdout);
    return(1);
}

/*
 * Construct a provider-broadcast message (for submission via a call to
 * ha_gs_send_message()), in response to a 'p' command.
 */
int     build_pbm(ha_gs_token_t *gToken, ha_gs_proposal_info_t *proposal)
{
    int  _numput, _gIndex;
    char _input, _newline;

    ha_gs_num_phases_t  _phases;
    ha_gs_time_limit_t  _time;

    st_pbm_struct *_pbm;
    char   *_pbm2;
    char    _inline[80];
    char    _intest;
    int     _pLen;

    _phases = -1;
    _time = 0;

    printf(starz);
    printf("\nSo, you want to ship a provider message?\n");
    if (0 == in_group_count) {
        printf("Unfortunately, since you haven't yet joined any groups, that wouldn't be\n");
        printf(" a very good idea, would it, bucko?  Try again later, after you convince\n");
        printf(" some group to allow you to join.  Otherwise, leave me alone.\n");
        printf(starz);
        fflush(stdout);
        return(0);
    }
```

```
    if (-1 == (*gToken = get_group_token(&_gIndex))) {
        return(0);
    }

    printf("\nNow that you've chosen group [%s], we need the protocol information:\n",
           group_names[_gIndex]);

    _phases = get_number_of_phases();

    if (HA_GS_N_PHASE == _phases) {
        _time = get_time_limit("for the provider message protocol");
    } else {
        _time = 0;
    }

    printf("Please enter your provider message, anything up to 2048 bytes:\n");
    fflush(stdout);
    _pbm = malloc(sizeof(st_pbm_struct));

    /* insert prefix.  */
    _pbm2 = &_pbm->st_pbm_data[sample_prefix_len];
    _pbm->st_pbm_index = sample_index;
    memcpy(&_pbm->st_pbm_data[0], sample_prefix, sample_prefix_len);

    gets(_pbm2);
    _pLen = sizeof(_pbm->st_pbm_index) + sample_prefix_len;
    if ((_pLen += strlen(_pbm2)) > 2048) {
        printf("Your given message is %d bytes long, I'm truncating it to 2048 bytes.\n",
               _pLen);
        fflush(stdout);
        _pLen = 2048;
    }
    printf("You have chosen [%s] with time limit of [%d] and a provider_message of:\n%d%s\n",
           phase_print(_phases),
           _time,
           _pbm->st_pbm_index,
           _pbm->st_pbm_data);

    printf("Thank you.  Will now submit your proposal.\n");

    proposal->gs_message_request.gs_num_phases = _phases;
    proposal->gs_message_request.gs_time_limit = _time;
    proposal->gs_message_request.gs_message.gs_length = _pLen;
    proposal->gs_message_request.gs_message.gs_message = (char *)_pbm;

    printf(starz);
    fflush(stdout);
    return(1);
}

/*
 * Construct the "voluntary leave" (ha_gs_leave()) request, in response
 * to the 'l' command.
 */
int     build_leave_request(ha_gs_token_t *gToken, ha_gs_proposal_info_t *proposal)
{
    int  _numput, _gIndex;
    char _input, _newline;

    ha_gs_num_phases_t  _phases;
    ha_gs_time_limit_t  _time;

    unsigned int        _leaveCode;

    char    _inline[80];
    char    _intest;

    _phases = -1;
    _time = 0;

    printf(starz);
```

```
    printf("So you want to leave a group?  You leave me, almost, speechless.\n");
    if (0 == in_group_count) {
        printf("Unfortunately, since you haven't yet joined any groups, that wouldn't be\n");
        printf(" a very good idea, would it, bucko?  Try again later, after you convince\n");
        printf(" some group to allow you to join.  Otherwise, leave me alone.\n");
        printf(starz);
        fflush(stdout);
        return(0);
    } else {
        printf("\nQuitter. When your group needs you, you want to run?\n");
    }

    if (-1 == (*gToken = get_group_token(&_gIndex))) {
        return(0);
    }

    printf("\nNow that you've chosen group [%s], we need the protocol information:\n",
           group_names[_gIndex]);

    _phases = get_number_of_phases();

    if (HA_GS_N_PHASE == _phases) {
        _time = get_time_limit("for the leave protocol");
    } else {
        _time = 0;
    }

    printf("Please enter your voluntary leave code (any integer value).\n");
    fflush(stdout);
    scanf("%d%c", &_numput, &_newline);
    _leaveCode = _numput;

    printf("You have chosen [%s] with time limit of [%d] and a leave code of [%d].\n",
           phase_print(_phases),
           _time,
           _leaveCode);

    printf("Thank you.  Will now submit your proposal.\n");

    proposal->gs_leave_request.gs_num_phases = _phases;
    proposal->gs_leave_request.gs_time_limit = _time;
    proposal->gs_leave_request.gs_leave_code = _leaveCode;

    printf(starz);
    fflush(stdout);
    return(1);
}

/*
 * Construct the "expel" (ha_gs_expel()) request, in response to the 'e' command.
 */
int     build_expel_request(ha_gs_token_t *gToken, ha_gs_proposal_info_t *proposal)
{
    int  _numput, _numput2, _gIndex;
    char _input, _newline;

    ha_gs_num_phases_t  _phases;
    ha_gs_time_limit_t  _time;

    int         _numExpelled, _cnt;
    ha_gs_provider_t    *_targets, *_victim;
    short       _instance;
    short       _node;
    int         _deactPhase;
    int         _deactFlagLen;
    char       *_deactFlag;

    char    _inline[300];
    char    _intest;

    _phases = -1;
    _time = 0;
```

```
printf(starz);
printf("So you want to expel a provider?  A heartless, but necessary, act.\n");
if (0 == in_group_count) {
    printf("Unfortunately, since you haven't yet joined any groups, that wouldn't be\n");
    printf(" a very good idea, would it, bucko?  Try again later, after you convince\n");
    printf(" some group to allow you to join.  Otherwise, just too bad.\n");
    printf(starz);
    fflush(stdout);
    return(0);
} else {
    printf("\nBrutal.  Nuke someone!\n");
}

if (-1 == (*gToken = get_group_token(&_gIndex))) {
    return(0);
}

printf("\nNow that you've chosen group [%s], we need the protocol information:\n",
       group_names[_gIndex]);

_phases = get_number_of_phases();

if (HA_GS_N_PHASE == _phases) {
    _time = get_time_limit("for the expel protocol");
} else {
    _time = 0;
}

printf("How many providers do you want to expel?\n");
fflush(stdout);
scanf("%d%c", &_numput, &_newline);
_numExpelled = _numput;
while (0 >= _numExpelled) {
    printf("An expel protocol with no one (count==%d) to expel is kind of useless!\n",
           _numExpelled);
    printf("How many providers to expel?\n");
    fflush(stdout);
    scanf("%d%c", &_numput, &_newline);
    _numExpelled = _numput;
}
_targets = _victim = (ha_gs_provider_t *)malloc(sizeof(ha_gs_provider_t) * _numExpelled);
for (_cnt = 0;
     _cnt < _numExpelled;
     _cnt++) {
    printf("Enter 'instance_number/node_number' of provider number %d: ", _cnt+1);
    fflush(stdout);
    scanf("%d/%d%c", &_numput, &_numput2, &_newline);
    _victim->gs_instance_number = _numput;
    _victim->gs_node_number = _numput2;
    _victim++;
}

printf("In which phase should the deactivate script be executed?\n");
printf(" If 0 (zero) is given, the deactivate script will not be executed.\n");
if (HA_GS_1_PHASE == _phases) {
    printf(" Since you specified a 1-phase expel, the deactivate script phase\n");
    printf(" must be 0 (zero) or 1 (one).\n");
}
while (1) {
    printf("Deactivate script execution phase: ");
    fflush(stdout);
    scanf("%d%c", &_numput, &_newline);
    _deactPhase = _numput;
    if (0 > _deactPhase) {
        printf("Invalid value [%d] given!  Must be greater than or equal to zero!\n",
               _deactPhase);
    } else if ((HA_GS_1_PHASE == _phases) &&
               (1 < _deactPhase)) {
        printf("Invalid value [%d] given for 1-phase expel.  Must be 0 or 1.\n",
               _deactPhase);
    } else {
```

```
            break;
        }
        fflush(stdout);
    }

    _deactFlag = NULL;
    _deactFlagLen = 0;
    if (0 != _deactPhase) {
        printf("You specified a non-zero deactivate script execution phase (%d).\n");
        printf(" Do you want to specify a deactivate script flag (0 [no] -- 1 [yes]? ");
        fflush(stdout);
        scanf("%d%c", &_numput, &_newline);
        if ((_numput) && ('n' != _numput)) {
            printf(" Enter flag (maximum 256 bytes): ");
            gets(_inline);
            _deactFlagLen = strlen(_inline);
            _deactFlag = malloc(_deactFlagLen + 1);
            strcpy(_deactFlag, _inline);
        }
    }

    printf("You have chosen [%s] with time limit of [%d] and a deactivation phase of [%d].\n",
           phase_print(_phases),
           _time,
           _deactPhase);
    if (NULL != _deactFlag) {
        printf(" Deactivation script flag: %s\n",
               _deactFlag);
    }
    printf("Providers being expelled: count [%d] victim%s[",
           _numExpelled,
           ((1 != _numExpelled) ? "s " : " "));
    _victim = _targets;
    for (_cnt = 0; _cnt < _numExpelled; _cnt++) {
#ifndef _VERBOSE_PROVIDER_OUTPUT
        printf("%d/%d ",
               _victim->gs_instance_number,
               _victim->gs_node_number);
#else
        /* do not print out the int value. */
        printf("%d/%d(%d) ",
               _victim->gs_instance_number,
               _victim->gs_node_number,
               _victim->gs_provider_id);
#endif  /* if ! _VERBOSE_PROVIDER_OUTPUT */
        _victim++;
    }
    printf("]\n");

    printf("Thank you.  Will now submit your proposal.\n");

    proposal->gs_expel_request.gs_num_phases = _phases;
    proposal->gs_expel_request.gs_time_limit = _time;
    proposal->gs_expel_request.gs_expel_list.gs_count = _numExpelled;
    proposal->gs_expel_request.gs_expel_list.gs_providers = _targets;
    proposal->gs_expel_request.gs_deactivate_phase = _deactPhase;
    proposal->gs_expel_request.gs_deactivate_flag = _deactFlag;

    printf(starz);
    fflush(stdout);
    return(1);
}

/*
 * Construct a new set of group attributes (for submission via a call to
 * ha_gs_change_attributes()), in response to a 'n' command.
 */
int     build_attributes_change(ha_gs_token_t *gToken, ha_gs_proposal_info_t *proposal)
{
    int  _numput, _gIndex;
    char _input, _newline;
```

```
    ha_gs_num_phases_t  _phases;
    ha_gs_time_limit_t  _time;

    ha_gs_group_attributes_t *_attributes;

    _phases = -1;
    _time = 0;

    printf(starz);
    printf("\nSo, you want to change the group's attributes?\n");
    if (0 == in_group_count) {
        printf("Unfortunately, since you haven't yet joined any groups, that wouldn't be\n");
        printf(" a very good idea, would it, bucko?  Try again later, after you convince\n");
        printf(" some group to allow you to join.  Otherwise, leave me alone.\n");
        printf(starz);
        fflush(stdout);
        return(0);
    }

    if (-1 == (*gToken = get_group_token(&_gIndex))) {
        return(0);
    }

    printf("\nNow that you've chosen group [%s], we need the protocol information:\n",
           group_names[_gIndex]);

    _phases = get_number_of_phases();

    if (HA_GS_N_PHASE == _phases) {
        _time = get_time_limit("for the attribute change protocol");
    } else {
        _time = 0;
    }

    _attributes = build_group_attributes(0, 1);

    printf("You have chosen [%s] with time limit of [%d] and attributes:\n",
           phase_print(_phases),
           _time);
    write_the_attributes(_attributes);

    printf("Thank you.  Will now submit your proposal.\n");

    proposal->gs_attribute_change_request.gs_num_phases = _phases;
    proposal->gs_attribute_change_request.gs_time_limit = _time;
    proposal->gs_attribute_change_request.gs_group_attributes = _attributes;

    printf(starz);
    fflush(stdout);
    return(1);
}

/*
 * Determine which group we want to separate ourself from, prior to
 * calling ha_gs_goodbye(), in response to a 'y' command.
 */
int     build_goodbye_request(ha_gs_token_t *gToken)
{
    int  _numput, _gIndex;
    char _input, _newline;

    printf(starz);
    printf("\nSo, you want to say goodbye to a group?\n");
    if (0 == in_group_count) {
        printf("Unfortunately, since you haven't yet joined any groups, that wouldn't be\n");
        printf(" a very good idea, would it, bucko?  Try again later, after you convince\n");
        printf(" some group to allow you to join.  Otherwise, leave me alone.\n");
        printf(starz);
        fflush(stdout);
        return(0);
    }
```

```
    if (-1 == (*gToken = get_group_token(&_gIndex))) {
        return(0);
    }
    return(1);
}

/*
 * We received OK from an ha_gs_goodbye() request, thus, we are out
 * of the group.  Glory days are here again!  Need to clean up the
 * various structures that we keep for each group.
 *
 * The work here is that the gToken is the "provider_id" for the
 * group we just exited, but, we need a reverse lookup into the
 * group table, which is not indexed by the provider_id.  So, walk
 * the table until we find the specified provider_id.
 */
void    told_group_goodbye(ha_gs_token_t gToken)
{
    ha_gs_token_t   _gTok;
    int _goodGroup, _insultLevel;
    int _numput;

    for (_goodGroup = 0;
         num_groups > _goodGroup;
         _goodGroup++) {
        if ((in_group[_goodGroup]) && (gToken == gid[_goodGroup])) {
            break;
        }
    }

    printf("You have successfully told group \"%s\" goodbye and are no longer a member in it.\n",
           group_names[_goodGroup]);
    fflush(stdout);
    provId[_goodGroup].gs_provider_id = -1;
    in_group[_goodGroup] = 0;
    in_group_count--;

    return;
}
/*********************************************************************/
/*
 * This routine is used to determine if we will be submitting a changed
 * state value and/or provider-broadcast message when we submit a vote.
 *
 * If values are given, then set those to be used.  If values are not
 * given, then get them from the proper arrays:
 *      provider_msg_array[] -- for provider-broadcast messages.
 *      state_value_array[] -- for group state values.
 */
void    create_state_and_pbm(int        index,
                             ha_gs_state_value_t **_sValue,
                             ha_gs_provider_message_t **_pValue,
                             ha_gs_state_value_t *_srcState,
                             ha_gs_provider_message_t *_srcPBM)
{
    ha_gs_state_value_t      *_s2, *_s3;
    ha_gs_provider_message_t *_p2, *_p3;

    char *_cPtr;

    if (-1 == index) {
        _s2 = _srcState;                /* values given.  use these. */
        _p2 = _srcPBM;
    } else {
        _s2 = &state_value_array[index]; /* pull from array. */
        _p2 = &provider_msg_array[index];
    }

    _s3 = malloc(sizeof(ha_gs_state_value_t));
    if (0 == _s2->gs_length) {
        _s3->gs_length = 0;
        _s3->gs_state = 0;
```

```
    } else {
        _s3->gs_length = _s2->gs_length + sample_prefix_len + sizeof(sample_index);
        _s3->gs_state = _cPtr = malloc(_s3->gs_length);
        memcpy(_s3->gs_state, &sample_index, sizeof(sample_index));
        memcpy(_s3->gs_state + sizeof(sample_index), sample_prefix, sample_prefix_len);
        _cPtr += sample_prefix_len + sizeof(sample_index);
        memcpy(_cPtr, _s2->gs_state, _s2->gs_length);
    }
    *_sValue = _s3;

    _p3 = malloc(sizeof(ha_gs_provider_message_t));
    if (0 == _p2->gs_length) {
        _p3->gs_length = 0;
        _p3->gs_message = 0;
    } else {
        _p3->gs_length = _p2->gs_length + sample_prefix_len + sizeof(sample_index);
        _p3->gs_message = _cPtr = malloc(_p3->gs_length);
        memcpy(_p3->gs_message, &sample_index, sizeof(sample_index));
        memcpy(_p3->gs_message + sizeof(sample_index), sample_prefix, sample_prefix_len);
        _cPtr += sample_prefix_len + sizeof(sample_index);
        memcpy(_cPtr, _p2->gs_message, _p2->gs_length);
    }
    *_pValue = _p3;

    return;
}

/*********************************************************************/
/*
 * User wants to subscribe to a group.  Determine the group name and
 * also the subscription control setting.
 */
ha_gs_rc_t      subscribe_to_a_group()
{
    int i, j, grpIdx, idx;
    char *pName;

    ha_gs_proposal_info_t info;

    ha_gs_subscription_ctrl_t subCtl;
    ha_gs_rc_t                subRC;
    ha_gs_token_t             subToken;

    int   wantToPick;
    char  nameWanted[80];

    grpIdx = -1;

    if (NUM_GROUPS_FOR_SUBSCRIBE <= subscribed_to_count) {
        printf("\nYou cannot subscribe to any more groups!\n");
        return(HA_GS_NOT_OK);
    }

    while (-1 == grpIdx) {
        printf("Groups available for subscription.  Specify index of group:\n");
        printf("%5s   %-16s%10s   %-16s\n", "index", "name", "index", "name");
        for (i = 0;
             predef_groups_for_subscribe > i;
             i += 2) {
            printf("%5d   %-16s",
                   i,
                   group_names[i]);
            if (predef_groups_for_subscribe > (i+1)) {
                printf("%10d   %-16s\n",
                       i + 1,
                       group_names[i+1]);
            } else {
                printf("\n");
            }
        }
        printf("%5d   %s\n",
               pickIdx,
```

```
            "Specify a group name");
    printf("%5d   %s\n",
            listIdx,
            "List current subscriptions");

    printf("Desired index: ");
    fflush(stdout);

    scanf("%d", &j);

    if (j == listIdx) {
        if (0 == subscribed_to_count) {
            printf("No current subscriptions!\n");
        } else {
            printf("Current subscriptions:\n%5s   %20s     %s\n",
                   "index",
                   "group name",
                   "current subscriptions");
            for (i = 0;
                 num_groups_for_subscribe > i;
                 i++) {
                if (-1 != sid[i]) {
                    printf("%5d   %20s     %s\n",
                           i,
                           subNames[i],
                           write_sub_ctrl(subCtrl[i]));
                }
            }
        }
        fflush(stdout);
    } else if ((j != pickIdx) &&
               ((0 > j) || (j >= num_groups_for_subscribe))) {
        printf("Um, sorry about this, but the index [%d] is invalid.  Try again.\n",
               j);
    } else {
        grpIdx = j;
    }
}

wantToPick = (grpIdx == pickIdx);

if (wantToPick) {
    pName = get_group_name("group name for subscription");
    printf("You are subscribing to group [%s]!  Specify subscription type:\n",
           pName);
} else {
    printf("You are subscribing to group [%s]!  Specify subscription type:\n",
           group_names[grpIdx]);
}
subCtl = (ha_gs_subscription_ctrl_t) 0;

while (0 == subCtl) {
    printf("Specify a value between %d and %d.  You may specify one of the specific\n",
           HA_GS_SUBSCRIBE_STATE, HA_GS_SUBSCRIBE_STATE_AND_MEMBERSHIP);
    printf("following flag values, or combine them.\n");
    printf("Value:   Subscription control:\n");
    printf("%5d   %s\n",
           HA_GS_SUBSCRIBE_STATE,
           write_sub_ctrl(HA_GS_SUBSCRIBE_STATE));
    printf("%5d   %s\n",
           HA_GS_SUBSCRIBE_DELTA_JOINS,
           write_sub_ctrl(HA_GS_SUBSCRIBE_DELTA_JOINS));
    printf("%5d   %s\n",
           HA_GS_SUBSCRIBE_DELTA_LEAVES,
           write_sub_ctrl(HA_GS_SUBSCRIBE_DELTA_LEAVES));
    printf("%5d   %s\n",
           HA_GS_SUBSCRIBE_DELTAS_ONLY,
           write_sub_ctrl(HA_GS_SUBSCRIBE_DELTAS_ONLY));
    printf("%5d   %s\n",
           HA_GS_SUBSCRIBE_MEMBERSHIP,
           write_sub_ctrl(HA_GS_SUBSCRIBE_MEMBERSHIP));
    printf("%5d   %s\n",
```

```
               HA_GS_SUBSCRIBE_ALL_MEMBERSHIP,
               write_sub_ctrl(HA_GS_SUBSCRIBE_ALL_MEMBERSHIP));
        printf("%5d   %s\n",
               HA_GS_SUBSCRIBE_STATE_AND_MEMBERSHIP,
               write_sub_ctrl(HA_GS_SUBSCRIBE_STATE_AND_MEMBERSHIP));
        printf("\nYour choice: ");
        fflush(stdout);

        scanf("%d", &j);

        if ((HA_GS_SUBSCRIBE_STATE > j) || (j > HA_GS_SUBSCRIBE_STATE_AND_MEMBERSHIP)) {
            printf("Um, sorry about this, but the index [%d] is out of range.  Try again.\n",
                   j);
        } else {
            subCtl = (ha_gs_subscription_ctrl_t)j;
        }
    }

    if (wantToPick) {
        info.gs_subscribe_request.gs_subscription_control = subCtl;
        info.gs_subscribe_request.gs_subscription_group = pName;
        info.gs_subscribe_request.gs_subscription_callback = pickCallback;
    } else {
        info.gs_subscribe_request.gs_subscription_control = subCtl;
        info.gs_subscribe_request.gs_subscription_group = group_names[grpIdx];
        info.gs_subscribe_request.gs_subscription_callback = subCallbacks[grpIdx];
    }

    subRC = ha_gs_subscribe(&subToken,
                            &info);

    if (HA_GS_OK == subRC) {
        idx = find_free_sub_slot();
        printf("Subscription request accepted.  Token == %d\n", subToken);
        sid[idx] = subToken;
        subCtrl[idx] = subCtl;

        if (wantToPick) {
            subNames[idx] = malloc(strlen(pName) + 1);
            strcpy(subNames[idx], pName);
            free(pName);
        } else {
            subNames[idx] = malloc(strlen(group_names[grpIdx]) + 1);
            strcpy(subNames[idx], group_names[grpIdx]);
        }

        subscribed_to_count++;
        if (NUM_GROUPS_FOR_SUBSCRIBE <= subscribed_to_count) {
            printf("\nYou cannot subscribe to any more groups!\n");
        }
    } else {
        printf("Problem with subscription: %s\n",
               write_an_rc(subRC));
    }

    printf(starz);
    fflush(stdout);
    return(subRC);
}

/*
 * User wants to unsubscribe from a group.  Determine which group.
 */
extern  ha_gs_rc_t      unsubscribe_from_a_group()
{
    int i, j, grpIdx;

    ha_gs_subscription_ctrl_t subCtl;
    ha_gs_rc_t                subRC;
    ha_gs_token_t             subToken;

    grpIdx = -1;
```

```
    if (0 == subscribed_to_count) {
        printf("\nYou have no current subscriptions.  Cannot unsubscribe!\n\n");
        printf(starz);
        fflush(stdout);
        return(HA_GS_OK);
    }

    while (-1 == grpIdx) {
        printf("You are currently subscribed to the following.  Specify an index:\n");
        printf("%5s %20s Current subscriptions\n", "index", "name");
        for (i = 0;
             num_groups_for_subscribe > i;
             i++) {
            if (-1 != sid[i]) {
                printf("%5d %20s %s\n",
                       i,
                       subNames[i],
                       write_sub_ctrl(subCtrl[i]));
            }
        }

        printf("Desired index: ");
        fflush(stdout);

        scanf("%d", &j);

        if ((0 > j) || (j >= num_groups_for_subscribe) || (-1 == sid[j])) {
            printf("Um, sorry about this, but the index [%d] is invalid.  Try again.\n",
                   j);
        } else {
            grpIdx = j;
        }
    }

    printf("You wish to unsubscribe from group [%s]!\n",
           subNames[grpIdx]);

    subRC = ha_gs_unsubscribe(sid[grpIdx]);

    if (HA_GS_OK == subRC) {
        printf("Unsubscription request accepted.  Token[%d], group[%s]\n",
               sid[grpIdx],
               subNames[grpIdx]);
        sid[grpIdx] = -1;
        subCtrl[grpIdx] = 0;
        free(subNames[grpIdx]);
        subNames[grpIdx] = 0;
        subscribed_to_count--;
    } else {
        printf("Problem with unsubscription: %s\n",
               write_an_rc(subRC));
    }

    printf(starz);
    fflush(stdout);
    return(subRC);
}

/*********************************************************************/
/*
 * Initialize the data fields for setting up subscriptions to the
 * "System Membership" groups.
 */
void init_membership_subscriptions()
{
    /* Set up names for subscribing to system membership groups. */

    char *_temp;

    /* Only host membership for now. */
```

```
    _temp = malloc(strlen(HA_GS_HOST_MEMBERSHIP_GROUP) + 1);
    strcpy(_temp, HA_GS_HOST_MEMBERSHIP_GROUP);
    group_names[num_groups] = _temp;

    _temp = malloc(strlen(HA_GS_ENET_MEMBERSHIP_GROUP) + 1);
    strcpy(_temp, HA_GS_ENET_MEMBERSHIP_GROUP);
    group_names[num_groups + 1] = _temp;

    _temp = malloc(strlen(HA_GS_CSS_MEMBERSHIP_GROUP) + 1);
    strcpy(_temp, HA_GS_CSS_MEMBERSHIP_GROUP);
    group_names[num_groups + 2] = _temp;

    _temp = malloc(strlen(HA_GS_TOKENRING_MEMBERSHIP_GROUP) + 1);
    strcpy(_temp, HA_GS_TOKENRING_MEMBERSHIP_GROUP);
    group_names[num_groups + 3] = _temp;

    _temp = malloc(strlen(HA_GS_FDDI_MEMBERSHIP_GROUP) + 1);
    strcpy(_temp, HA_GS_FDDI_MEMBERSHIP_GROUP);
    group_names[num_groups + 4] = _temp;

    _temp = malloc(strlen(HA_GS_RS232_MEMBERSHIP_GROUP) + 1);
    strcpy(_temp, HA_GS_RS232_MEMBERSHIP_GROUP);
    group_names[num_groups + 5] = _temp;

    _temp = malloc(strlen(HA_GS_TMSCSI_MEMBERSHIP_GROUP) + 1);
    strcpy(_temp, HA_GS_TMSCSI_MEMBERSHIP_GROUP);
    group_names[num_groups + 6] = _temp;

    _temp = malloc(strlen(HA_GS_SLIP_MEMBERSHIP_GROUP) + 1);
    strcpy(_temp, HA_GS_SLIP_MEMBERSHIP_GROUP);
    group_names[num_groups + 7] = _temp;

    return;
}

#endif                                  /* ifdef _SAMPLE_TEST */
```

The sample_utility.h Header File

```
#ifndef _SAMPLE_UTILITY_H
#define _SAMPLE_UTILITY_H
/*******************************************************************/
/*                                                                 */
/*CPRY                                                             */
/*                                                                 */
/* Licensed Materials - Property of IBM                            */
/*                                                                 */
/* 5765-529 PSSP                                                   */
/*                                                                 */
/* (C) Copyright IBM Corporation 1996 All Rights Reserved.         */
/*                                                                 */
/* US Government Users Restricted Rights -                         */
/* Use, duplication or disclosure restricted by                    */
/* GSA ADP Schedule Contract with IBM Corp.                        */
/*                                                                 */
/*CPRY                                                             */
/*******************************************************************/

static char *sample_utility_sccsid = "@(#)96   1.7
src/rsct/pgs/samples/sample_utility.h,
gssamples, rsct_rtro, rtrot1fx 4/26/98 16:17:34";

/*******************************************************************/
/*
 * Name:  sample_utility.h
 *
 * This program provides the declarations for the set of callback
 * functions used by the "sample_test" program.
 *
 * Components:
 *   sample_test.c - contains the main() function, supports interaction
 *      with the user, and most calls to the Group Services interfaces.
 *
 *   sample_utility.c - provides the definitions for the
 *      utility functions used by the sample_test program.
 *
 *   sample_callbacks.c - provides the definitions for the callback
 *      functions used for the groups created by this program.
 *
 *   sample_callbacks.h - declarations for the callback functions
 *      contained in sample_callbacks.c.
 *
 *   sample_utility.h - declarations for the utility functions contained
 *      in sample_utility.c
 *
 * The information here assumes that you are familiar with the information
 * presented in the IBM PSSP Group Services Programming Guide and Reference
 * manual.
 *
 * This program is provided for illustrative purposes only, and is not
 * intended to be an authoritative description of the "best" methods to
 * use when writing a Group Services application.  It is intended to
 * demonstrate the various interfaces in a relatively verbose manner,
 * and to allow you to relatively easily manipulate groups and their
 * members.
 *
 * To this end, various aspects of this program (in particular its
 * handling of screen input and output) are neither robust nor
 * foolproof.  Therefore, you should take care when giving input to
 * this program.
 */
/*******************************************************************/

/*******************************************************************/
/*
 * Include the Group Services declarations file.
 */
/*******************************************************************/
```

sample_utility.h

```
#include <ha_gs.h>

extern  char   *time_now( );
extern  time_t  time_since( struct timeval *event );
extern  float   randomp( );
extern  long    randomn( unsigned long max );

#define ARBITRARY_STATE_VALUE   1
#define ARBITRARY_PROV_MESSAGE  2
extern  void    write_arbitrary_value(int which, void *value);

extern  char   *write_leave_reason(ha_gs_leave_reasons_t why);
extern  void    write_join_information(int gIndex);
extern  void    write_the_notification(int groupIdx,
                                       void * addr,
                                       ha_gs_notification_type_t whatItBe);
extern  void    write_the_proposal(ha_gs_request_t eReq,
                                   ha_gs_token_t   eTok,
                                   ha_gs_proposal_info_t *prop);
extern  void    write_the_attributes(ha_gs_group_attributes_t *gAttrs);

extern  ha_gs_token_t   gid[];
extern  ha_gs_group_attributes_t       *gattr[];
extern  int             voting_phase;
extern  int             handledResponsiveness;
extern  int             instance_numbers[];

#define NUM_GROUPS 8
#define NUM_GROUPS_FOR_SUBSCRIBE 32

#ifdef  _SAMPLE_TEST                    /* _SAMPLE_TEST */

extern  void    write_instructions(int _verbose);
extern  void    init_membership_subscriptions();
extern  ha_gs_rc_t      subscribe_to_a_group();
extern  ha_gs_rc_t      unsubscribe_from_a_group();
extern  ha_gs_token_t   get_group_token(int *gAddr);
extern  ha_gs_group_attributes_t *build_group_attributes(int _offset, int _changing);
extern  ha_gs_socket_ctrl_t      build_socket_control();
extern  char    *build_deact_script(char *default_deact);
extern  ha_gs_responsiveness_t   *build_responsiveness(ha_gs_responsiveness_t *def_resp);
extern  ha_gs_callback_rc_t       get_response(ha_gs_time_limit_t _time);
extern  int     build_leave_request(ha_gs_token_t *gToken, ha_gs_proposal_info_t *proposal);
extern  int     build_expel_request(ha_gs_token_t *gToken, ha_gs_proposal_info_t *proposal);
extern  int     build_state_change(ha_gs_token_t *gToken, ha_gs_proposal_info_t *proposal);
extern  int     build_attributes_change(ha_gs_token_t *gToken, ha_gs_proposal_info_t *proposal);
extern  int     build_goodbye_request(ha_gs_token_t *gToken);
extern  void    told_group_goodbye(ha_gs_token_t gToken);
extern  int     build_pbm(ha_gs_token_t *gToken, ha_gs_proposal_info_t *proposal);

extern  void    write_the_vote(ha_gs_vote_value_t vote);
extern  void    copy_the_attributes(int groupIdx,
                                    ha_gs_group_attributes_t *gAttrsTarg,
                                    ha_gs_group_attributes_t *gAttrsSrc)

extern  ha_gs_vote_value_t get_a_vote(int phase);
extern  void    create_state_and_pbm(int index,
                                     ha_gs_state_value_t **_sValue,
                                     ha_gs_provider_message_t **_pValue,
                                     ha_gs_state_value_t *_srcState,
                                     ha_gs_provider_message_t *_srcPBM);

extern  char           *group_names[];
extern  char           *source_group_names[];
extern  char           *subNames[];
extern  char           *prov_local_names[];
extern  int             instance_numbers2[];
extern  ha_gs_vote_value_t              default_vote_array[];
extern  int             in_group[];
extern  int             in_group_count;
extern  const   int     num_groups;
```

```
extern  int             sample_index;
extern  char           *sample_prefix;
extern  int             sample_prefix_len;

extern  char           *sample_pp;
extern  char           *sample_ee;

extern  char           *starz;
extern  int             verbose;
extern  int             interactiveInit;
extern  int             interactiveResponse;

#define NUMBER_MSG_ENTRIES 8

#endif                                  /* ifdef _SAMPLE_TEST */

#endif                                  /* _SAMPLE_UTILITY_H */
```

The sample_deactive_ksh.sh Deactivate Script

```
#!/bin/ksh
# IBM_PROLOG_BEGIN_TAG
# This is an automatically generated prolog.
#
#
# Licensed Materials - Property of IBM
#
# (C) COPYRIGHT International Business Machines Corp. 1997,1998 # All Rights Reserved
#
# US Government Users Restricted Rights - Use, duplication or
# disclosure restricted by GSA ADP Schedule Contract with IBM Corp.
#
# IBM_PROLOG_END_TAG
################################################################################
#                                                                              #
#CPRY                                                                          #
#    Licensed Materials - Property of IBM                                      #
#                                                                              #
#    5765-529 PSSP                                                             #
#                                                                              #
#    (C) Copyright IBM Corporation 1997  All Rights Reserved.                  #
#                                                                              #
#    US Government Users Restricted Rights - Use, duplication or               #
#    disclosure restricted by GSA ADP Schedule Contract with IBM Corp.         #
#CPRY                                                                          #
#                                                                              #
################################################################################

# @(#)62   1.3
src/rsct/pgs/samples/sample_deactive_ksh.sh,
gssamples, rsct_rtro 7/28/98 15:43:22

#/*******************************************************************/
#/*
# * Name:  sample_deactive_ksh
# *
# * This module provides a simple sample deactivation program that can
# * be executed as part of an "expel" protocol against a provider,
# * or an "failure" protocol if the deactivate-on-failure is enabled.
# * It will expect the input parameters described as part of the
# * Expel Protocol and the Deactivate-On-Failure Handling description
# * in the IBM PSSP Group Services Programming Guide and Reference manual.
# */
#/*******************************************************************/
#
#/*******************************************************************/
#/*
# * Expected input conditions:
# *  - effective uid and gid set to that of the client process connected
# *     to the Group Services daemon, at the time it issued ha_gs_init().
# *  - current directory matches that of the client process connected
# *     to the Group Services daemon, at the time it issued ha_gs_init().
# *  - path and other environment variables set to those used by the
# *     Group Services daemon.
# *
# * Expected input parameters:
# *  targetPid -- client process pid containing the provider targeted by the expel.
# *  timeLimit -- number of seconds within which this program must complete.
# *  groupName -- name of the group to which the targeted provider is joined.
# *  expelFlag -- flag specified by the provider initiating the expel.
# *               or "providerdied" as the deactivate-on-failure handling.
# *  failedProviders -- comma(,)-delimited list of the failed providers'
# *         local instance numbers. This parameter will be presented
# *         only for deactivate-on-failure handling.
# *
# *  Special conditions:
# *   targetPid == 0 -- the provider's process id already failed during the
# *                     expel protocol, or the deactivate-on-failure handling
# *                     is initiated.  Take any other necessary actions.
```

```
# *
# *   expelFlag == NULL -- no flag was specified.
# *              == "providerdied" -- the deactivate-on-failure handling
# *                      is initiated.
# *
# * Output (exit code):
# *  0  -- means this program is "successful".  It is up to this program to
# *         define "successful".
# *  !0 -- means this program is "unsuccessful".  It is up to this program to
# *         define "unsuccessful".
# */
#/*********************************************************************/
#
#/*********************************************************************/
#/*
# * In the case of the expel protocol:
# * It is assumed that part of the normal action of a deactivate "script"
# * will be to "kill" the targeted provider's process, since if we are
# * running an expel we assume that the process is hung, or otherwise
# * misbehaving.
# *
# * For the deactivate-on-failure:
# * It will wait  until (timeLimit + 5) to exit. Exit with zero.
# *
# * However, this is optional.  This program should perform whatever
# * actions make sense for the group and its providers.
# */
#/*********************************************************************/
#
#/*********************************************************************/
#/*
# * For this sample program, we will use the *expel flag* as a key to
# * our behavior.
# *
# * Flag values:
# *  NULL  -- (no flag) No action, simply exit with 0 exit code.
# *  "kill" -- kill given process id (unless it is zero), exit with the
# *              return code from the kill command.
# *  "kill N" -- kill given process id (unless it is zero), exit with the
# *               value given by the integer N.
# *  "killwait" -- kill given process id (unless it is zero), but wait
# *                 to exit until (timeLimit + 5).  Always exit with zero.
# *  "wait" -- do not try to kill given process id, wait until
# *             (timeLimit + 5) to exit.  Always exit with zero.
# *  "wait N" -- do not try to kill given process id, wait until
# *               (timeLimit + 5) to exit.  N should be an integer, use it
# *               as our exit code.
# *  "N" -- N should be an integer, exit immediately with N as our exit
# *          code.
# *
# *  "providerdied" -- wait until (timeLimit + 5) to exit. Exit with zero.
# *                This indicates deactivate-on-failure.
# *
# */
#/*********************************************************************/

extraWait=5                     # Used to wait "extra" amount of time.

targetPid=$1                    # First three paramaters always there.
timeLimit=$2
groupName=$3
if ((4 <= $#))
then
  expelFlag=$4
else
  expelFlag=""
fi

# Check expel flag, and see what to do.

if [[ -z ${expelFlag} ““
then
```

```
  exit 0                        # Null flag, just exit with zero exit code.
fi

if [[ $expelFlag = "killwait" “” ]]  # Try to kill target process, then wait.
then
  if ((0 != targetPid))
  then
    kill -9 $targetPid          # Kill the target process.
  fi
  ((sleepTime=timeLimit + extraWait))# Wait as long as specified.
  sleep $sleepTime

  exit 0
fi

# Kill the process?  If so, we may have a specific exit code that we
# should use, rather than the kill return code.

if [[ $expelFlag = "kill" “” ]]
then
  killRc=0
  if ((0 != targetPid))         # Have a pid?
  then
    kill -9 $targetPid          # Try to kill it.
    killRc=$?                   # Save return code.
  fi
  if [[ -z $5 “” ]]                # Was a flag given for exit code?
  then
    exit $killRc                # No, exit with kill return code.
  fi

  if ((0 == killRc))            # Good result from kill, use given code.
  then
    exit $5
  else
    exit $killRc                # Reflect results from kill.
  fi
fi

# No killing, but we need to wait for some amount of time before
# exiting.  If given, use specified value for exit code.

if [[ $expelFlag = "wait" “” ]]
then
  if [[ -z $5 “” ]]                  # Was a flag given for exit code?
  then
    ((sleepTime=timeLimit + extraWait))# Wait as long as specified.
    sleep $sleepTime
    exit 0                      # No special flag.
  else
    ((sleepTime=timeLimit + extraWait))# Wait as long as specified.
    sleep $sleepTime
    exit $5
  fi
fi

# In the case of the deactivate-on-failure:
# No killing, but we need to wait for some amount of time before
# exiting.  If given, use specified value for exit code.

if [[ $expelFlag = "providerdied" “” ]]
then
  failedProviders=$5     # holds the list of failed providers
  ((sleepTime=timeLimit + extraWait))# Wait as long as specified.
  sleep $sleepTime
  exit 0                        # No special flag.
fi

# Nothing else, simply exit with the given value.

exit $expelFlag
```

The sample_deactive_c_prog.c Deactivate Script

```
/* IBM_PROLOG_BEGIN_TAG                                                   */
/* This is an automatically generated prolog.                             */
/*                                                                        */
/*                                                                        */
/*                                                                        */
/* Licensed Materials - Property of IBM                                   */
/*                                                                        */
/* (C) COPYRIGHT International Business Machines Corp. 1996,1998          */
/* All Rights Reserved                                                    */
/*                                                                        */
/* US Government Users Restricted Rights - Use, duplication or            */
/* disclosure restricted by GSA ADP Schedule Contract with IBM Corp.      */
/*                                                                        */
/* IBM_PROLOG_END_TAG                                                     */
/**************************************************************************/
/*                                                                       */
/*CPRY                                                                   */
/*                                                                       */
/* Licensed Materials - Property of IBM                                  */
/*                                                                       */
/* 5765-529 PSSP                                                         */
/*                                                                       */
/* (C) Copyright IBM Corporation 1996 All Rights Reserved.               */
/*                                                                       */
/* US Government Users Restricted Rights -                               */
/* Use, duplication or disclosure restricted by                          */
/* GSA ADP Schedule Contract with IBM Corp.                              */
/*                                                                       */
/*CPRY                                                                   */
/**************************************************************************/

static char *sccsid = "@(#)70   1.4
src/rsct/pgs/samples/sample_deactive_c_prog.c,
gssamples, rsct_rtro 7/28/98 15:43:15";

#if !defined(_HAGSD_COPYRIGHT_H)
#define _HAGSD_COPYRIGHT_H
static char copyright[] = "Licensed Materials - Property of IBM\n\
5765-529 (C) Copyright IBM Corp. 1996.  All Rights Reserved.\n\
US Government Users Restricted Rights - Use, duplication or \n\
disclosure restricted by GSA ADP Schedule Contract with IBM Corp.\n";
#endif

/**************************************************************************/
/*
 * Name:  sample_deactive_c_prog.c
 *
 * This module provides a simple sample deactivation program that can
 * be executed as part of an "expel" protocol against a provider,
 * or an "failure" protocol if the deactivate-on-failure is enabled.
 * It will expect the input parameters described as part of the
 * Expel Protocol and the Deactivate-On-Failure Handling description
 * in the IBM PSSP Group Services Programming Guide and Reference manual.
 */
/**************************************************************************/

/**************************************************************************/
/*
 * Expected input conditions:
 *  - effective uid and gid set to that of the client process connected
 *     to the Group Services daemon, at the time it issued ha_gs_init().
 *  - current directory matches that of the client process connected
 *     to the Group Services daemon, at the time it issued ha_gs_init().
 *  - path and other environment variables set to those used by the
 *     Group Services daemon.
 *
 * Expected input parameters:
 *  target_pid -- client process pid containing the provider targeted by the expel.
 *  time_limit -- number of seconds within which this program must complete.
```

```
 *  group_name -- name of the group to which the targeted provider is joined.
 *  expel_flag -- flag specified by the provider initiating the expel,
 *             or "providerdied" as the deactivate-on-failure handling.
 *  failedProviders -- comma(,)-delimited list of the failed providers'
 *             local instance numbers. This parameter will be presented
 *             only for deactivate-on-failure handling.
 *
 *  Special conditions:
 *   target_pid == 0 -- the provider's process id already failed during the
 *                      expel protocol, or the deactivate-on-failure handling
 *                is initiated.  Take any other necessary actions.
 *
 *   expel_flag == NULL -- no flag was specified.
 *              == "providerdied" -- the deactivate-on-failure handling
 *             is initiated.
 *
 * Output (exit code):
 *  0  -- means this program is "successful".  It is up to this program to
 *         define "successful".
 *  !0 -- means this program is "unsuccessful".  It is up to this program to
 *         define "unsuccessful".
 */
/*********************************************************************/

/*********************************************************************/
/*
 * In the case of the expel protocol:
 * It is assumed that part of the normal action of a deactivate "script"
 * will be to "kill" the targeted provider's process, since if we are
 * running an expel we assume that the process is hung, or otherwise
 * misbehaving.
 *
 * For the deactivate-on-failure:
 * It will wait  until (timeLimit + 5) to exit. Exit with zero.
 *
 * However, this is optional.  This program should perform whatever
 * actions make sense for the group and its providers.
 */
/*********************************************************************/

/*********************************************************************/
/*
 * For this sample program, we will use the *expel flag* as a key to
 * our behavior.
 *
 * Flag values:
 *  NULL  -- (no flag) No action, simply exit with 0 exit code.
 *  "kill" -- kill given process id (unless it is zero), exit with the
 *             return code from the kill command.
 *  "kill N" -- kill given process id (unless it is zero), exit with the
 *               value given by the integer N.
 *  "killwait" -- kill given process id (unless it is zero), but wait
 *                 to exit until (timeLimit + 5).  Always exit with zero.
 *  "wait" -- do not try to kill given process id, wait until
 *             (timeLimit + 5) to eixt.  Always exit with zero.
 *  "wait N" -- do not try to kill given process id, wait until
 *               (timeLimit + 5) to exit.  N should be an integer, use it
 *               as our exit code.
 *  "N" -- N should be an integer, exit immediately with N as our exit
 *          code.
 *  "providerdied" -- wait until (timeLimit + 5) to exit. Exit with zero.
 *                     This indicates deactivate-on-failure.
 *
 */
/*********************************************************************/

#include <sys/types.h>
#include <unistd.h>
#include <errno.h>
#include <signal.h>

#include "ha_gs.h"
```

```
#define KILL             "kill"
#define KILLWAIT         "killwait"
#define WAIT             "wait"
#define PROVIDERDIED     "providerdied"
#define EXTRA_WAIT       5

int     main(int    argc,
             void **argv)
{
    char  *sPid, *sTime, *groupName, *expelFlag;
    char  *failedProviders = NULL;

    pid_t  targetPid;
    int    timeLimit;

    int    killRC;
    int    exitCode, givenExit, haveExit;

    haveExit = 0;

    sPid = argv[1];
    sTime = argv[2];
    groupName = argv[3];
    if (5 <= argc) {
        expelFlag = argv[4];
        if (6 <= argc) {
            haveExit = 1;
            givenExit = atoi(argv[5]);
        }
    } else {
        expelFlag = NULL;
    }

    targetPid = atoi(sPid);
    timeLimit = atoi(sTime);

    /* check expel flag, and see what to do. */

    if (NULL == expelFlag) {

        exit(0);                        /* No flag given, just exit now. */

    }

    /*
     * See if we should kill the target process, then wait for some
     * amount of time before exiting.
     */

    if (0 == strncmp(KILLWAIT, expelFlag, strlen(KILLWAIT))) {

        if (0 != targetPid) {

            kill(targetPid, 9);         /* Have a pid.  Kill the process. */

        }

        sleep(timeLimit + EXTRA_WAIT);

        exit(0);                        /* Ignore kill return code. */

    }

    /*
     * Kill the process?  If so, we may have a specific exit code that we
     * should use, rather than the kill return code.
     */

    if (0 == strncmp(KILL, expelFlag, strlen(KILL))) {

        if (haveExit) {
```

```
            /* Use given code for exit value unless error. */
            exitCode = givenExit;
        } else {
            exitCode = 0;
        }

        if (0 != targetPid) {           /* Kill the process. */

            if ((-1 == kill(targetPid, 9)) && (!haveExit)) {
                exitCode = errno;       /* Error, grab the error return code. */
            }
        }

        exit(exitCode);                 /* Exit with whatever code worked out. */
    }

    /*
     * No killing, but we need to wait for some amount of time before
     * exiting.  If given, use specified value for exit code.
     */

    if (0 == strncmp(WAIT, expelFlag, strlen(WAIT))) {

        if (haveExit) {
            exitCode = givenExit;       /* Use given code for exit value. */
        } else {
            exitCode = 0;
        }

        sleep(timeLimit + EXTRA_WAIT);

        exit(exitCode);
    }


    /*
     * In the case of the deactivate-on-failure:
     * No killing, but we need to wait for some amount of time before
     * exiting.  If given, use specified value for exit code.
     */

    if (0 == strncmp(PROVIDERDIED, expelFlag, strlen(PROVIDERDIED))) {

failedProviders=argv[5]; /* holds the list of failed providers */

        exitCode = 0;

        sleep(timeLimit + EXTRA_WAIT);

        exit(exitCode);
    }



    /*
     * Nothing else, simply exit with the given value.
     */

    exitCode = atoi(expelFlag);

    exit(exitCode);
}
```

Appendix A. Subscription Special Data

This appendix provides information about subscription special data.

Formats

The subscription special data format is very flexible, due to the wide variety of data that can be 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/or membership fields. The following subscription special data items have been added:

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

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 of the 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

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

gs_flag

contains an OR'ing together of 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 points 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

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

gs_next_special_block

points 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

is 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

is 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

points 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 are used to describe the contents of the **ha_gs_special_block_t** that may be included.

HA_GS_ADAPTER_DEATH_ARRAY

is set to indicate 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

is set to indicate 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

is set to indicate 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 vs. deconfiguration, the **ha_gs_special_block_t**, if included, will contain an array of **ha_gs_adapter_death_t** entries, each containing flags specifying 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 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 the to which the events relate. However, this support is not sufficient in certain cases, where a single adapter may be represented multiple times due to IP alias addresses being used.

To allow subscribers to differentiate among the different aliases, if a subscription notification includes one or more adapters with multiple aliases, then a **ha_gs_special_block_t** will be included that has a 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_CURRENT_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 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.

Prerequisite System Conditions for Receiving Subscription Special Data

Once a node has been migrated to PSSP 3.1 or to HACMP/ES at the HACMP 4.3 level for both Group Services and Topology Services, then this support will be activated on that node, since 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, and all should work exactly as they expect today.

Bibliography

This bibliography helps you find product documentation related to the RS/6000 SP hardware and software products.

You can find most of the IBM product information for RS/6000 SP products on the World Wide Web. Formats for both viewing and downloading are available.

PSSP documentation is shipped with the PSSP product in a variety of formats and can be installed on your system. The man pages for public code that PSSP includes are also available online.

You can order hard copies of the product documentation from IBM. This bibliography lists the titles that are available and their order numbers.

Finally, this bibliography contains a list of non-IBM publications that discuss parallel computing and other topics related to the RS/6000 SP.

Finding Documentation on the World Wide Web

Most of the RS/6000 SP hardware and software books are available from the IBM RS/6000 web site at **http://www.rs6000.ibm.com**. You can view a book or download a Portable Document Format (PDF) version of it. At the time this manual was published, the full path to the "RS/6000 SP Product Documentation Library" page was **http://www.rs6000.ibm.com/resource/aix_resource/sp_books**. However, the structure of the RS/6000 web site can change over time.

Accessing PSSP Documentation Online

On the same medium as the PSSP product code, IBM ships PSSP man pages, HTML files, and PDF files. In order to use these publications, you must first install the **ssp.docs** file set.

To view the PSSP HTML publications, you need access to an HTML document browser such as Netscape. The HTML files and an index that links to them are installed in the **/usr/lpp/ssp/html** directory. Once installed, you can also view the HTML files from the RS/6000 SP Resource Center.

If you have installed the SP Resource Center on your SP system, you can access it by entering the **/usr/lpp/ssp/bin/resource_center** command. If you have the SP Resource Center on CD-ROM, see the **readme.txt** file for information about how to run it.

/usr/lpp/ssp/bin/resource_center

To view the PSSP PDF publications, you need access to the Adobe Acrobat Reader 3.0.1. The Acrobat Reader is shipped with the AIX Version 4.3 Bonus Pack and is also freely available for downloading from the Adobe web site at URL **http://www.adobe.com**.

Manual Pages for Public Code

The following manual pages for public code are available in this product:

SUP	/usr/lpp/ssp/man/man1/sup.1
NTP	/usr/lpp/ssp/man/man8/xntpd.8
	/usr/lpp/ssp/man/man8/xntpdc.8
Perl (Version 4.036)	/usr/lpp/ssp/perl/man/perl.man

/usr/lpp/ssp/perl/man/h2ph.man

/usr/lpp/ssp/perl/man/s2p.man

/usr/lpp/ssp/perl/man/a2p.man

Perl (Version 5.003) Man pages are in the /usr/lpp/ssp/perl5/man/man1 directory

Manual pages and other documentation for **Tcl**, **TclX**, **Tk**, and **expect** can be found in the compressed **tar** files located in the **/usr/lpp/ssp/public** directory.

RS/6000 SP Planning Publications

This section lists the IBM product documentation for planning for the IBM RS/6000 SP hardware and software.

IBM RS/6000 SP:

- *Planning, Volume 1, Hardware and Physical Environment*, GA22-7280
- *Planning, Volume 2, Control Workstation and Software Environment*, GA22-7281

RS/6000 SP Hardware Publications

This section lists the IBM product documentation for the IBM RS/6000 SP hardware.

IBM RS/6000 SP:

- *Planning, Volume 1, Hardware and Physical Environment*, GA22-7280
- *Planning, Volume 2, Control Workstation and Software Environment*, GA22-7281
- *Maintenance Information, Volume 1, Installation and Relocation*, GA22-7375
- *Maintenance Information, Volume 2, Maintenance Analysis Procedures*, GA22-7376
- *Maintenance Information, Volume 3, Locations and Service Procedures*, GA22-7377
- *Maintenance Information, Volume 4, Parts Catalog*, GA22-7378

RS/6000 SP Switch Router Publications

The RS/6000 SP Switch Router is based on the Ascend GRF switched IP router product from Ascend Communications, Inc.. You can order the SP Switch Router as the IBM 9077.

The following publications are shipped with the SP Switch Router. You can also order these publications from IBM using the order numbers shown.

- *Ascend GRF Getting Started*, GA22-7368
- *Ascend GRF Configuration Guide*, GA22-7366
- *Ascend GRF Reference Guide*, GA22-7367
- *IBM SP Switch Router Adapter Guide*, GA22-7310.

RS/6000 SP Software Publications

This section lists the IBM product documentation for software products related to the IBM RS/6000 SP. These products include:

- IBM Parallel System Support Programs for AIX (PSSP)
- IBM LoadLeveler for AIX (LoadLeveler)
- IBM Parallel Environment for AIX (Parallel Environment)

- IBM General Parallel File System for AIX (GPFS)
- IBM Engineering and Scientific Subroutine Library (ESSL) for AIX
- IBM Parallel ESSL for AIX
- IBM High Availability Cluster Multi-Processing for AIX (HACMP)
- IBM Client Input Output/Sockets (CLIO/S)
- IBM Network Tape Access and Control System for AIX (NetTAPE)

PSSP Publications

IBM RS/6000 SP:

- *Planning, Volume 2, Control Workstation and Software Environment*, GA22-7281

PSSP:

- *Installation and Migration Guide*, GA22-7347
- *Administration Guide*, SA22-7348
- *Managing Shared Disks*, SA22-7349
- *Performance Monitoring Guide and Reference*, SA22-7353
- *Diagnosis Guide*, GA22-7350
- *Command and Technical Reference*, SA22-7351
- *Messages Reference*, GA22-7352

RS/6000 Cluster Technology (RSCT):

- *Event Management Programming Guide and Reference*, SA22-7354
- *Group Services Programming Guide and Reference*, SA22-7355

As an alternative to ordering the individual books, you can use SBOF-8587 to order the PSSP software library.

LoadLeveler Publications

LoadLeveler:

- *Using and Administering*, SA22-7311
- *Diagnosis and Messages Guide*, GA22-7277

GPFS Publications

GPFS:

- *Installation and Administration Guide*, SA22-7278

Parallel Environment Publications

Parallel Environment:

- *Installation Guide*, GC28-1981
- *Hitchhiker's Guide*, GC23-3895
- *Operation and Use, Volume 1*, SC28-1979
- *Operation and Use, Volume 2*, SC28-1980
- *MPI Programming and Subroutine Reference*, GC23-3894
- *MPL Programming and Subroutine Reference*, GC23-3893
- *Messages*, GC28-1982

As an alternative to ordering the individual books, you can use SBOF-8588 to order the PE library.

Parallel ESSL and ESSL Publications

- *ESSL Products: General Information*, GC23-0529
- *Parallel ESSL: Guide and Reference*, SA22-7273
- *ESSL: Guide and Reference*, SA22-7272

HACMP Publications

HACMP:

- *Concepts and Facilities*, SC23-1938
- *Planning Guide*, SC23-1939
- *Installation Guide*, SC23-1940
- *Administration Guide*, SC23-1941
- *Troubleshooting Guide*, SC23-1942
- *Programming Locking Applications*, SC23-1943
- *Programming Client Applications*, SC23-1944
- *Master Index and Glossary*, SC23-1945
- *HANFS for AIX Installation and Administration Guide*, SC23-1946
- *Enhanced Scalability Installation and Administration Guide*, SC23-1972

CLIO/S Publications

CLIO/S:

- *General Information*, GC23-3879
- *User's Guide and Reference*, GC28-1676

NetTAPE Publications

NetTAPE:

- *General Information*, GC23-3990
- *User's Guide and Reference*, available from your IBM representative

AIX and Related Product Publications

For the latest information on AIX and related products, including RS/6000 hardware products, see *AIX and Related Products Documentation Overview*, SC23-2456. You can order a hard copy of the book from IBM. You can also view it online from the "AIX Online Publications and Books" page of the RS/6000 web site, at URL **http://www.rs6000.ibm.com/resource/aix_resource/Pubs**.

Red Books

IBM's International Technical Support Organization (ITSO) has published a number of redbooks related to the RS/6000 SP. For a current list, see the ITSO website, at URL **http://www.redbooks.ibm.com**.

Non-IBM Publications

Here are some non-IBM publications that you may find helpful.

- Almasi, G., Gottlieb, A., *Highly Parallel Computing*, Benjamin-Cummings Publishing Company, Inc., 1989.
- Foster, I., *Designing and Building Parallel Programs*, Addison-Wesley, 1995.
- Gropp, W., Lusk, E., Skjellum, A., *Using MPI*, The MIT Press, 1994.
- Message Passing Interface Forum, *MPI: A Message-Passing Interface Standard, Version 1.1*, University of Tennessee, Knoxville, Tennessee, June 6, 1995.
- Message Passing Interface Forum, *MPI-2: Extensions to the Message-Passing Interface, Version 2.0*, University of Tennessee, Knoxville, Tennessee, July 18, 1997.
- Ousterhout, John K., *Tcl and the Tk Toolkit*, Addison-Wesley, Reading, MA, 1994, ISBN 0-201-63337-X.
- Pfister, Gregory, F., *In Search of Clusters*, Prentice Hall, 1998.

Glossary of Terms and Abbreviations

This glossary includes terms and definitions from:

- The *IBM Dictionary of Computing*, New York: McGraw-Hill, 1994.
- The *American National Standard Dictionary for Information Systems*, ANSI X3.172-1990, copyright 1990 by the American National Standards Institute (ANSI). Copies can be purchased from the American National Standards Institute, 1430 Broadway, New York, New York 10018. Definitions are identified by the symbol (A) after the definition.
- The *ANSI/EIA Standard - 440A: Fiber Optic Terminology* copyright 1989 by the Electronics Industries Association (EIA). Copies can be purchased from the Electronic Industries Association, 2001 Pennsylvania Avenue N.W., Washington, D.C. 20006. Definitions are identified by the symbol (E) after the definition.
- The *Information Technology Vocabulary* developed by Subcommittee 1, Joint Technical Committee 1, of the International Organization for Standardization and the International Electrotechnical Commission (ISO/IEC JTC1/SC1). Definitions of published parts of this vocabulary are identified by the symbol (I) after the definition; definitions taken from draft international standards, committee drafts, and working papers being developed by ISO/IEC JTC1/SC1 are identified by the symbol (T) after the definition, indicating that final agreement has not yet been reached among the participating National Bodies of SC1.

The following cross-references are used in this glossary:

> **Contrast with.** This refers to a term that has an opposed or substantively different meaning.
> **See.** This refers the reader to multiple-word terms in which this term appears.
> **See also.** This refers the reader to terms that have a related, but not synonymous, meaning.
> **Synonym for.** This indicates that the term has the same meaning as a preferred term, which is defined in the glossary.

This section contains some of the terms that are commonly used in the SP publications.

IBM is grateful to the American National Standards Institute (ANSI) for permission to reprint its definitions from the American National Standard *Vocabulary for Information Processing* (Copyright 1970 by American National Standards Institute, Incorporated), which was prepared by Subcommittee X3K5 on Terminology and Glossary of the American National Standards Committee X3. ANSI definitions are preceded by an asterisk (*).

Other definitions in this glossary are taken from *IBM Vocabulary for Data Processing, Telecommunications, and Office Systems* (SC20-1699) and *IBM DATABASE 2 Application Programming Guide for TSO Users* (SC26-4081).

A

adapter. An adapter is a mechanism for attaching parts. For example, an adapter could be a part that electrically or physically connects a device to a computer or to another device. In the SP system, network connectivity is supplied by various adapters, some optional, that can provide connection to I/O devices, networks of workstations, and mainframe networks. Ethernet, FDDI, token-ring, HiPPI, SCSI, FCS, and ATM are examples of adapters that can be used as part of an SP system.

address. A character or group of characters that identifies a register, a device, a particular part of storage, or some other data source or destination.

AFS. A distributed file system that provides authentication services as part of its file system creation.

AIX. Abbreviation for Advanced Interactive Executive, IBM's licensed version of the UNIX operating system. AIX is particularly suited to support technical computing applications, including high function graphics and floating point computations.

AIX Version 3. The IBM AIX Version 3 for RS/6000 (also known as AIX/6000). It is based on UNIX System V, conforms with POSIX IEEE Standard 1003.1, is compatible with Berkeley Software Distribution 4.3 (4.3 BSD), and is designed to meet Trusted Computing Base level C2 security.

Amd. Berkeley Software Distribution automount daemon.

API. Application Programming Interface. A set of programming functions and routines that provide access between the Application layer of the OSI seven-layer model and applications that want to use the network. It is a software interface.

application. The use to which a data processing system is put; for example, a payroll application, an airline reservation application.

application data. The data that is produced using an application program.

ARP. Address Resolution Protocol.

ATM. Asynchronous Transfer Mode. (See *TURBOWAYS 100 ATM Adapter*.)

Authentication. The process of validating the identity of a user or server.

Authorization. The process of obtaining permission to perform specific actions.

B

batch processing. * (1) The processing of data or the accomplishment of jobs accumulated in advance in such a manner that each accumulation thus formed is processed or accomplished in the same run. * (2) The processing of data accumulating over a period of time. * (3) Loosely, the execution of computer programs serially. (4) Computer programs executed in the background.

BMCA. Block Multiplexer Channel Adapter. The block multiplexer channel connection allows the RS/6000 to communicate directly with a host System/370 or System/390; the host operating system views the system unit as a control unit.

BOS. The AIX Base Operating System.

C

call home function. The ability of a system to call the IBM support center and open a PMR to have a repair scheduled.

CDE. Common Desktop Environment. A graphical user interface for UNIX.

charge feature. An optional feature for either software or hardware for which there is a charge.

CLI. Command Line Interface.

client. * (1) A function that requests services from a server and makes them available to the user. * (2) A term used in an environment to identify a machine that uses the resources of the network.

Client Input/Output Sockets (CLIO/S). A software package that enables high-speed data and tape access between SP systems, AIX systems, and ES/9000 mainframes.

CLIO/S. Client Input/Output Sockets.

CMI. Centralized Management Interface provides a series of SMIT menus and dialogues used for defining and querying the SP system configuration.

connectionless. A communication process that takes place without first establishing a connection.

connectionless network. A network in which the sending logical node must have the address of the receiving logical node before information interchange can begin. The packet is routed through nodes in the network based on the destination address in the packet. The originating source does not receive an acknowledgment that the packet was received at the destination.

control workstation. A single point of control allowing the administrator or operator to monitor and manage the SP system using the IBM AIX Parallel System Support Programs.

css. Communication subsystem.

D

daemon. A process, not associated with a particular user, that performs system-wide functions such as administration and control of networks, execution of time-dependent activities, line printer spooling and so forth.

DASD. Direct Access Storage Device. Storage for input/output data.

DCE. Distributed Computing Environment.

DFS. distributed file system. A subset of the IBM Distributed Computing Environment.

DNS. Domain Name Service. A hierarchical name service which maps high level machine names to IP addresses.

E

Error Notification Object. An object in the SDR that is matched with an error log entry. When an error log entry occurs that matches the Notification Object, a user-specified action is taken.

ESCON. Enterprise Systems Connection. The ESCON channel connection allows the RS/6000 to communicate directly with a host System/390; the host operating system views the system unit as a control unit.

Ethernet. (1) Ethernet is the standard hardware for TCP/IP local area networks in the UNIX marketplace. It is a 10-megabit per second baseband type LAN that allows multiple stations to access the transmission

medium at will without prior coordination, avoids contention by using carrier sense and deference, and resolves contention by collision detection (CSMA/CD). (2) A passive coaxial cable whose interconnections contain devices or components, or both, that are all active. It uses CSMA/CD technology to provide a best-effort delivery system.

Ethernet network. A baseband LAN with a bus topology in which messages are broadcast on a coaxial cabling using the carrier sense multiple access/collision detection (CSMA/CD) transmission method.

event. In Event Management, the notification that an expression evaluated to true. This evaluation occurs each time an instance of a resource variable is observed.

expect. Programmed dialogue with interactive programs.

expression. In Event Management, the relational expression between a resource variable and other elements (such as constants or the previous value of an instance of the variable) that, when true, generates an event. An example of an expression is `X < 10` where X represents the resource variable `IBM.PSSP.aixos.PagSp.%totalfree` (the percentage of total free paging space). When the expression is true, that is, when the total free paging space is observed to be less than 10%, the Event Management subsystem generates an event to notify the appropriate application.

F

failover. Also called fallover, the sequence of events when a primary or server machine fails and a secondary or backup machine assumes the primary workload. This is a disruptive failure with a short recovery time.

fall back. Also called fall back, the sequence of events when a primary or server machine takes back control of its workload from a secondary or backup machine.

FDDI. Fiber Distributed Data Interface.

Fiber Distributed Data Interface (FDDI). An American National Standards Institute (ANSI) standard for 100-megabit-per-second LAN using optical fiber cables. An FDDI local area network (LAN) can be up to 100 km (62 miles) and can include up to 500 system units. There can be up to 2 km (1.24 miles) between system units and/or concentrators.

file. * A set of related records treated as a unit, for example, in stock control, a file could consist of a set of invoices.

file name. A CMS file identifier in the form of 'filename filetype filemode' (like: TEXT DATA A).

file server. A centrally located computer that acts as a storehouse of data and applications for numerous users of a local area network.

File Transfer Protocol (FTP). The Internet protocol (and program) used to transfer files between hosts. It is an application layer protocol in TCP/IP that uses TELNET and TCP protocols to transfer bulk-data files between machines or hosts.

foreign host. Any host on the network other than the local host.

FTP. File transfer protocol.

G

gateway. An intelligent electronic device interconnecting dissimilar networks and providing protocol conversion for network compatibility. A gateway provides transparent access to dissimilar networks for nodes on either network. It operates at the session presentation and application layers.

H

HACMP. AIX High Availability Cluster Multi-Processing.

HACMP/ES. Enhanced Scalability feature of the IBM High Availability Cluster Multi-Processing for AIX Licensed Program.

HACWS. High Availability Control Workstation function, based on HACMP, provides for a backup control workstation for the SP system.

help key. In the SP graphical interface, the key that gives you access to the SP graphical interface help facility.

High Availability Cluster Multi-Processing/6000. An IBM facility to cluster nodes or components to provide high availability by eliminating single points of failure.

HiPPI. High Performance Parallel Interface. RS/6000 units can attach to a HiPPI network as defined by the ANSI specifications. The HiPPI channel supports burst rates of 100 Mbps over dual simplex cables; connections can be up to 25 km in length as defined by the standard and can be extended using third-party HiPPI switches and fiber optic extenders.

home directory. The directory associated with an individual user.

host. A computer connected to a network, and providing an access method to that network. A host provides end-user services.

HSD. The data striping device for the IBM Virtual Shared Disk. The device driver lets application programs stripe data across physical disks in multiple IBM Virtual Shared Disks, thus reducing I/O bottlenecks and hot spots.

I

instance vector. Obsolete term for resource identifier.

Intermediate Switch Board. Switches mounted in the Sp Switch expansion frame.

Internet. A specific inter-network consisting of large national backbone networks such as APARANET, MILNET, and NSFnet, and a myriad of regional and campus networks all over the world. The network uses the TCP/IP protocol suite.

Internet Protocol (IP). (1) A protocol that routes data through a network or interconnected networks. IP acts as an interface between the higher logical layers and the physical network. This protocol, however, does not provide error recovery, flow control, or guarantee the reliability of the physical network. IP is a connectionless protocol. (2) A protocol used to route data from its source to it destination in an Internet environment.

IP address. A 32-bit address assigned to devices or hosts in an IP internet that maps to a physical address. The IP address is composed of a network and host portion.

ISB. Intermediate Switch Board.

K

Kerberos. A service for authenticating users in a network environment.

kernel. The core portion of the UNIX operating system which controls the resources of the CPU and allocates them to the users. The kernel is memory-resident, is said to run in “kernel mode” and is protected from user tampering by the hardware.

L

LAN. (1) Acronym for Local Area Network, a data network located on the user's premises in which serial transmission is used for direct data communication among data stations. (2) Physical network technology that transfers data a high speed over short distances. (3) A network in which a set of devices is connected to another for communication and that can be connected to a larger network.

local host. The computer to which a user's terminal is directly connected.

log database. A persistent storage location for the logged information.

log event. The recording of an event.

log event type. A particular kind of log event that has a hierarchy associated with it.

logging. The writing of information to persistent storage for subsequent analysis by humans or programs.

M

mask. To use a pattern of characters to control retention or elimination of portions of another pattern of characters.

menu. A display of a list of available functions for selection by the user.

Motif. The graphical user interface for OSF, incorporating the X Window System. Also called OSF/Motif.

MTBF. Mean time between failure. This is a measure of reliability.

MTTR. Mean time to repair. This is a measure of serviceability.

mutability. The ability of certain group attributes to be dynamically changed by the group's providers, using the **ha_gs_change_attributes** asynchronous interface.

N

naive application. An application with no knowledge of a server that fails over to another server. Client to server retry methods are used to reconnect.

network. An interconnected group of nodes, lines, and terminals. A network provides the ability to transmit data to and receive data from other systems and users.

NFS. Network File System. NFS allows different systems (UNIX or non-UNIX), different architectures, or vendors connected to the same network, to access remote files in a LAN environment as though they were local files.

NIM. Network Installation Management is provided with AIX to install AIX on the nodes.

NIM client. An AIX system installed and managed by a NIM master. NIM supports three types of clients:

- Standalone
- Diskless
- Dataless

NIM master. An AIX system that can install one or more NIM clients. An AIX system must be defined as a NIM master before defining any NIM clients on that system. A NIM master managers the configuration database containing the information for the NIM clients.

NIM object. A representation of information about the NIM environment. NIM stores this information as objects in the NIM database. The types of objects are:

- Network
- Machine
- Resource

NIS. Network Information System.

node. In a network, the point where one or more functional units interconnect transmission lines. A computer location defined in a network. The SP system can house several different types of nodes for both serial and parallel processing. These node types can include thin nodes, wide nodes, 604 high nodes, as well as other types of nodes both internal and external to the SP frame.

Node Switch Board. Switches mounted on frames that contain nodes.

NSB. Node Switch Board.

NTP. Network Time Protocol.

O

ODM. Object Data Manager. In AIX, a hierarchical object-oriented database for configuration data.

P

parallel environment. A system environment where message passing or SP resource manager services are used by the application.

Parallel Environment. A licensed IBM program used for message passing applications on the SP or RS/6000 platforms.

parallel processing. A multiprocessor architecture which allows processes to be allocated to tightly coupled multiple processors in a cooperative processing environment, allowing concurrent execution of tasks.

parameter. * (1) A variable that is given a constant value for a specified application and that may denote the application. * (2) An item in a menu for which the operator specifies a value or for which the system provides a value when the menu is interpreted. * (3) A name in a procedure that is used to refer to an argument that is passed to the procedure. * (4) A particular piece of information that a system or application program needs to process a request.

partition. See system partition.

Perl. Practical Extraction and Report Language.

perspective. The primary window for each SP Perspectives application, so called because it provides a unique view of an SP system.

pipe. A UNIX utility allowing the output of one command to be the input of another. Represented by the | symbol. It is also referred to as filtering output.

PMR. Problem Management Report.

POE. Formerly Parallel Operating Environment, now Parallel Environment for AIX.

port. (1) An end point for communication between devices, generally referring to physical connection. (2) A 16-bit number identifying a particular TCP or UDP resource within a given TCP/IP node.

predicate. Obsolete term for expression.

Primary node or machine. (1) A device that runs a workload and has a standby device ready to assume the primary workload if that primary node fails or is taken out of service. (2) A node on the SP Switch that initializes, provides diagnosis and recovery services, and performs other operations to the switch network. (3) In IBM Virtual Shared Disk function, the node at which the logical volume is actually local and that acts as server node to I/O requests from other nodes.

Problem Management Report. The number in the IBM support mechanism that represents a service incident with a customer.

process. * (1) A unique, finite course of events defined by its purpose or by its effect, achieved under defined conditions. * (2) Any operation or combination of operations on data. * (3) A function being performed or waiting to be performed. * (4) A program in operation. For example, a daemon is a system process that is always running on the system.

protocol. A set of semantic and syntactic rules that defines the behavior of functional units in achieving communication.

R

RAID. Redundant array of independent disks.

rearm expression. In Event Management, an expression used to generate an event that alternates with an original event expression in the following way: the event expression is used until it is true, then the rearm expression is used until it is true, then the event expression is used, and so on. The rearm expression is commonly the inverse of the event expression (for example, a resource variable is on or off). It can also be used with the event expression to define an upper and lower boundary for a condition of interest.

rearm predicate. Obsolete term for rearm expression

remote host. *See foreign host.*

resource. In Event Management, an entity in the system that provides a set of services. Examples of resources include hardware entities such as processors, disk drives, memory, and adapters, and software entities such as database applications, processes, and file systems. Each resource in the system has one or more attributes that define the state of the resource.

resource identifier. In Event Management, a set of elements, where each element is a name/value pair of the form `name=value`, whose values uniquely identify the copy of the resource (and by extension, the copy of the resource variable) in the system.

resource monitor. A program that supplies information about resources in the system. It can be a command, a daemon, or part of an application or subsystem that manages any type of system resource.

resource variable. In Event Management, the representation of an attribute of a resource. An example of a resource variable is `IBM.AIX.PagSp.%totalfree`, which represents the percentage of total free paging space. `IBM.AIX.PagSp` specifies the resource name and `%totalfree` specifies the resource attribute.

RISC. Reduced Instruction Set Computing (RISC), the technology for today's high performance personal computers and workstations, was invented in 1975. Uses a small simplified set of frequently used instructions for rapid execution.

rlogin (remote LOGIN). A service offered by Berkeley UNIX systems that allows authorized users of one machine to connect to other UNIX systems across a network and interact as if their terminals were connected directly. The rlogin software passes information about the user's environment (for example, terminal type) to the remote machine.

RPC. Acronym for Remote Procedure Call, a facility that a client uses to have a server execute a procedure call. This facility is composed of a library of procedures plus an XDR.

RSH. A variant of RLOGIN command that invokes a command interpreter on a remote UNIX machine and passes the command line arguments to the command interpreter, skipping the LOGIN step completely. See also *rlogin*.

S

SCSI. Small Computer System Interface.

server. (1) A function that provides services for users. A machine may run client and server processes at the same time. (2) A machine that provides resources to the network. It provides a network service, such as disk storage and file transfer, or a program that uses such a service. (3) A device, program, or code module on a network dedicated to providing a specific service to a network. (4) On a LAN, a data station that provides facilities to other data stations. Examples are file server, print server, and mail server.

shell. The shell is the primary user interface for the UNIX operating system. It serves as command language interpreter, programming language, and allows foreground and background processing. There are three different implementations of the shell concept: Bourne, C and Korn.

Small Computer System Interface (SCSI). An input and output bus that provides a standard interface for the attachment of various direct access storage devices (DASD) and tape drives to the RS/6000.

Small Computer Systems Interface Adapter (SCSI Adapter). An adapter that supports the attachment of various direct-access storage devices (DASD) and tape drives to the RS/6000.

SMIT. The System Management Interface Toolkit is a set of menu driven utilities for AIX that provides functions such as transaction login, shell script creation, automatic updates of object database, and so forth.

SNMP. Simple Network Management Protocol. (1) An IP network management protocol that is used to monitor attached networks and routers. (2) A TCP/IP-based protocol for exchanging network management information and outlining the structure for communications among network devices.

socket. (1) An abstraction used by Berkeley UNIX that allows an application to access TCP/IP protocol functions. (2) An IP address and port number pairing. (3) In TCP/IP, the Internet address of the host computer

on which the application runs, and the port number it uses. A TCP/IP application is identified by its socket.

standby node or machine. A device that waits for a failure of a primary node in order to assume the identity of the primary node. The standby machine then runs the primary's workload until the primary is back in service.

subnet. Shortened form of subnetwork.

subnet mask. A bit template that identifies to the TCP/IP protocol code the bits of the host address that are to be used for routing for specific subnetworks.

subnetwork. Any group of nodes that have a set of common characteristics, such as the same network ID.

subsystem. A software component that is not usually associated with a user command. It is usually a daemon process. A subsystem will perform work or provide services on behalf of a user request or operating system request.

SUP. Software Update Protocol.

Sysctl. Secure System Command Execution Tool. An authenticated client/server system for running commands remotely and in parallel.

syslog. A BSD logging system used to collect and manage other subsystem's logging data.

System Administrator. The user who is responsible for setting up, modifying, and maintaining the SP system.

system partition. A group of nonoverlapping nodes on a switch chip boundary that act as a logical SP system.

T

tar. Tape ARchive, is a standard UNIX data archive utility for storing data on tape media.

Tcl. Tool Command Language.

TclX. Tool Command Language Extended.

TCP. Acronym for Transmission Control Protocol, a stream communication protocol that includes error recovery and flow control.

TCP/IP. Acronym for Transmission Control Protocol/Internet Protocol, a suite of protocols designed to allow communication between networks regardless of the technologies implemented in each network. TCP provides a reliable host-to-host protocol between hosts in packet-switched communications networks and in interconnected systems of such networks. It assumes that the underlying protocol is the Internet Protocol.

Telnet. Terminal Emulation Protocol, a TCP/IP application protocol that allows interactive access to foreign hosts.

Tk. Tcl-based Tool Kit for X Windows.

TMPCP. Tape Management Program Control Point.

token-ring. (1) Network technology that controls media access by passing a token (special packet or frame) between media-attached machines. (2) A network with a ring topology that passes tokens from one attaching device (node) to another. (3) The IBM Token-Ring LAN connection allows the RS/6000 system unit to participate in a LAN adhering to the IEEE 802.5 Token-Passing Ring standard or the ECMA standard 89 for Token-Ring, baseband LANs.

transaction. An exchange between the user and the system. Each activity the system performs for the user is considered a transaction.

transceiver (transmitter-receiver). A physical device that connects a host interface to a local area network, such as Ethernet. Ethernet transceivers contain electronics that apply signals to the cable and sense collisions.

transfer. To send data from one place and to receive the data at another place. Synonymous with move.

transmission. * The sending of data from one place for reception elsewhere.

TURBOWAYS 100 ATM Adapter. An IBM high-performance, high-function intelligent adapter that provides dedicated 100 Mbps ATM (asynchronous transfer mode) connection for high-performance servers and workstations.

U

UDP. User Datagram Protocol.

UNIX operating system. An operating system developed by Bell Laboratories that features multiprogramming in a multiuser environment. The UNIX operating system was originally developed for use on minicomputers, but has been adapted for mainframes and microcomputers. **Note:** The AIX operating system is IBM's implementation of the UNIX operating system.

user. Anyone who requires the services of a computing system.

User Datagram Protocol (UDP). (1) In TCP/IP, a packet-level protocol built directly on the Internet

Protocol layer. UDP is used for application-to-application programs between TCP/IP host systems. (2) A transport protocol in the Internet suite of protocols that provides unreliable, connectionless datagram service. (3) The Internet Protocol that enables an application programmer on one machine or process to send a datagram to an application program on another machine or process.

user ID. A nonnegative integer, contained in an object of type *uid_t*, that is used to uniquely identify a system user.

V

Virtual Shared Disk, IBM. The function that allows application programs executing at different nodes of a system partition to access a raw logical volume as if it were local at each of the nodes. In actuality, the logical volume is local at only one of the nodes (the server node).

W

workstation. * (1) A configuration of input/output equipment at which an operator works. * (2) A terminal or microcomputer, usually one that is connected to a mainframe or to a network, at which a user can perform applications.

X

X Window System. A graphical user interface product.

Index

E

F

G

H

I

J

L

Q

R

S

T

U

V

Communicating Your Comments to IBM

RS/6000 Cluster Technology
Group Services Programming
Guide and Reference

Publication No. SA22-7355-00

If you especially like or dislike anything about this book, please use one of the methods listed below to send your comments to IBM. Whichever method you choose, make sure you send your name, address, and telephone number if you would like a reply.

Feel free to comment on specific errors or omissions, accuracy, organization, subject matter, or completeness of this book. However, the comments you send should pertain to only the information in this manual and the way in which the information is presented. To request additional publications, or to ask questions or make comments about the functions of IBM products or systems, you should talk to your IBM representative or to your IBM authorized remarketer.

When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute your comments in any way it believes appropriate without incurring any obligation to you.

If you are mailing a readers' comment form (RCF) from a country other than the United States, you can give the RCF to the local IBM branch office or IBM representative for postage-paid mailing.

- If you prefer to send comments by mail, use the RCF at the back of this book.
- If you prefer to send comments by FAX, use this number:

  FAX: (International Access Code)+1+914+432-9405
- If you prefer to send comments electronically, use this network ID:
  - IBMLink (United States customers only): S390VM(MHVRCFS)
  - IBM Mail Exchange: USIB6TC9 at IBMMAIL
  - Internet: mhvrcfs@vnet.ibm.com

Make sure to include the following in your note:

- Title and publication number of this book
- Page number or topic to which your comment applies.

Readers' Comments — We'd Like to Hear from You

RS/6000 Cluster Technology
Group Services Programming
Guide and Reference

Publication No. SA22-7355-00

You may use this form to communicate your comments about this publication, its organization, or subject matter, with the understanding that IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you. Your comments will be sent to the author's department for whatever review and action, if any, are deemed appropriate.

Note: Copies of IBM publications are not stocked at the location to which this form is addressed. Please direct any requests for copies of publications, or for assistance in using your IBM system, to your IBM representative or to the IBM branch office serving your locality.

Today's date: _____

What is your occupation?

Newsletter number of latest Technical Newsletter (if any) concerning this publication:

How did you use this publication?

[ ]	As an introduction	[ ]	As a text (student)
[ ]	As a reference manual	[ ]	As a text (instructor)
[ ]	For another purpose (explain)		

_____

_____

Is there anything you especially like or dislike about the organization, presentation, or writing in this manual? Helpful comments include general usefulness of the book; possible additions, deletions, and clarifications; specific errors and omissions.

Page Number: Comment:

Name	Address
Company or Organization	
Phone No.	

Readers' Comments — We'd Like to Hear from You
SA22-7355-00

Cut or Fold
Along Line

Fold and Tape | **Please do not staple** | Fold and Tape

NO POSTAGE
NECESSARY
IF MAILED IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST-CLASS MAIL PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

IBM Corporation
Department 55JA, Mail Station P384
522 SOUTH ROAD
POUGHKEEPSIE NY 12601-5400

Fold and Tape | **Please do not staple** | Fold and Tape

Cut or Fold
Along Line

Program Number: 5765–D51 (PSSP); 5765–D28 (HACMP)

Printed in the United States of America
on recycled paper containing 10%
recovered post-consumer fiber.

SA22-7355-00