Mwave

Device Driver Developer’s Guide

Version 1.0

NOTICE:

First Edition (July 1996)

This edition is prepared and maintained by IBM Microelectronics. For further information, see IBM Microelectronics' Homepage at http://www.chips.ibm.com.

Table of Contents

Chapter 1 -- Introduction to the Mwave Device Driver Developer's Guide

The Mwave Client/Server architecture provides a flexible method of communication among device drivers (clients) in the Mwave System. The Mwave Agent is the Operating System-independent interface that device drivers use to access Client/Server capabilities. Chapter 2 provides an overview of the Client/Server architecture and describes the messages used between clients, servers, and the Mwave Agent.

One implementation of the Mwave Client/Server architecture is the Mwave Notification Server. The Notification Server, in conjunction with the Mwave Agent, allows client device drivers to request notification of power management events and PCMCIA events such as card insertion and removal. Chapter 3 describes the Notification Server and the messages sent and received by the Server.

A recent implementation of the Mwave Client/Server architecture is the Mwave Contention Server. Prior to the integration of the Contention Server in the Mwave System, device drivers were not loaded until they were needed and were then unloaded as soon as the task had completed. This driver behavior resulted in performance problems in the following scenarios:

- A first task is loaded in the system and a second task needs to load but is constrained by resource limitations. There is no way for the first task to unload and free resources until the task has completed. As a result, the second task fails to load.
- Every task has to unload as soon as it completes to avoid the previous scenario. Continuous loading and unloading of tasks consumes host processor cycles and causes delays for needed tasks.

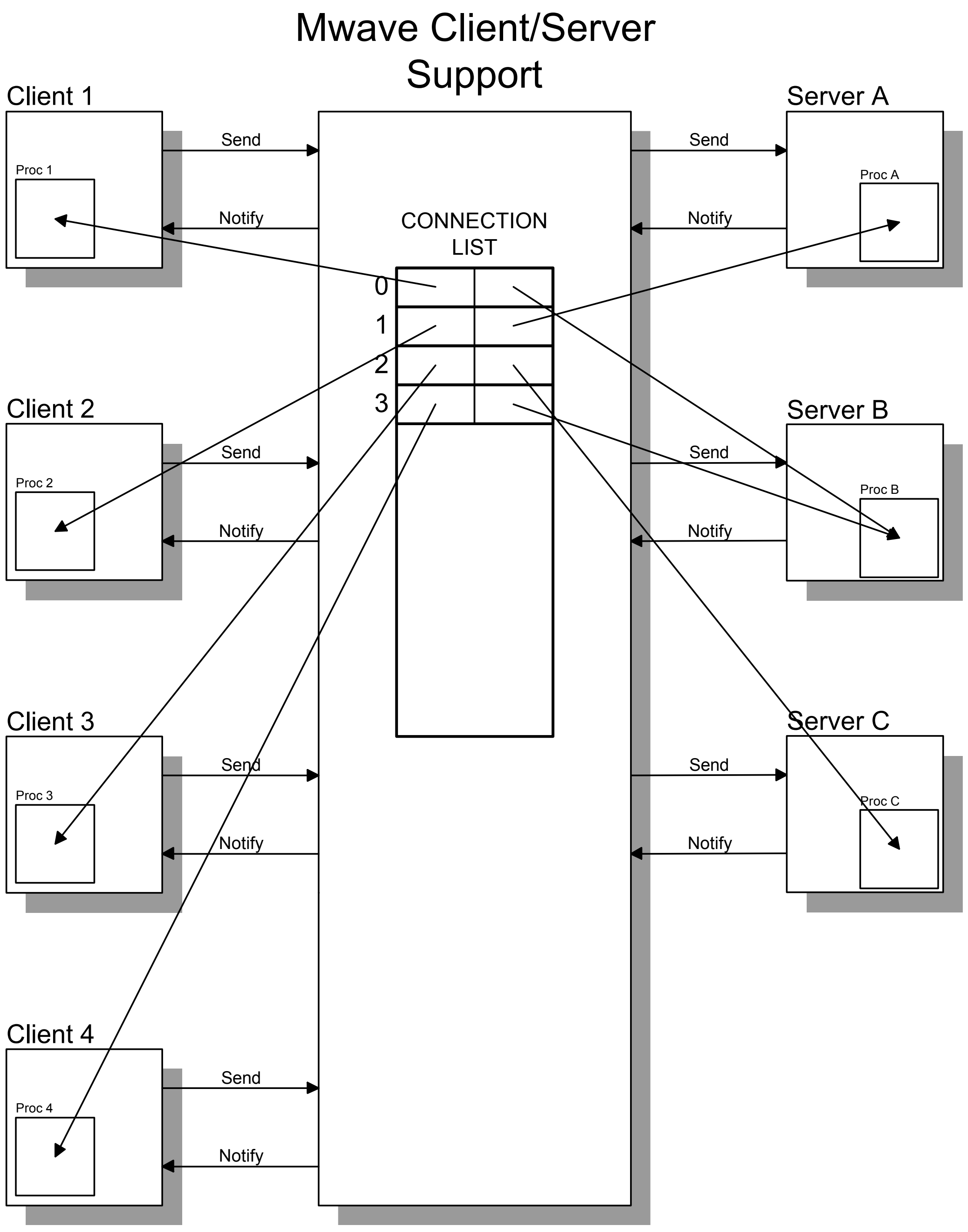
The Contention Server provides the framework in which Mwave device drivers contending for resources can cooperatively use Mwave resources. Each client device driver in the system has an assigned priority establishing a hierarchy in which higher priority drivers can request lower priority drivers to remove themselves from the system to free resources. This ensures that the critical tasks can load. Because drivers can receive requests to remove themselves from the system, this framework also allows lower priority device drivers to remain loaded until other drivers request that they unload.

For applications developed by independent software vendors to run compatibly in the Mwave System, application developers must use the Contention Server when developing device drivers. Chapter 4 describes the Mwave Contention Server.

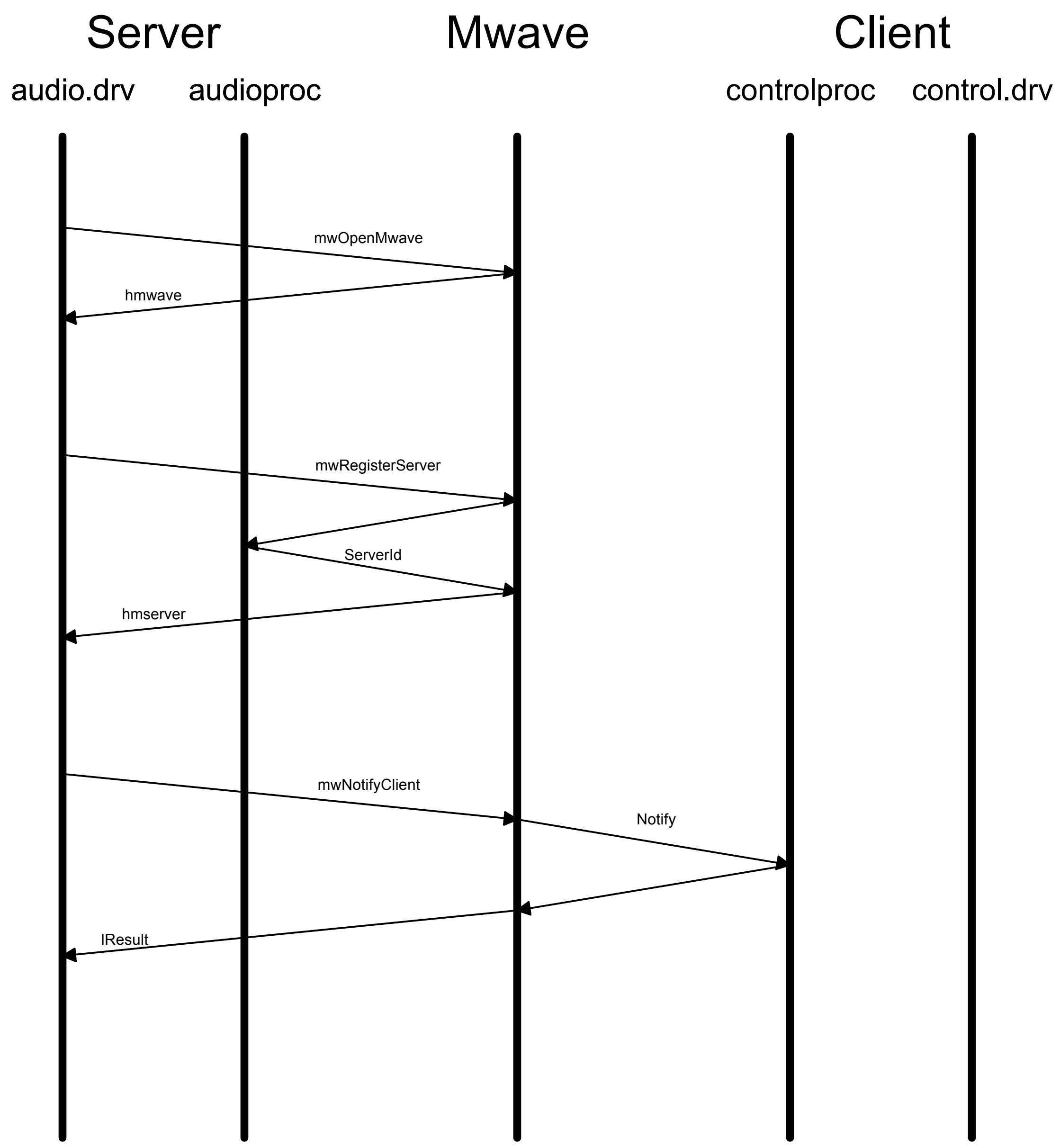
The API calls for the `ServerProc` and `ClientProc` are defined in Appendix A.

Chapter 2 -- Mwave Client/Server Support

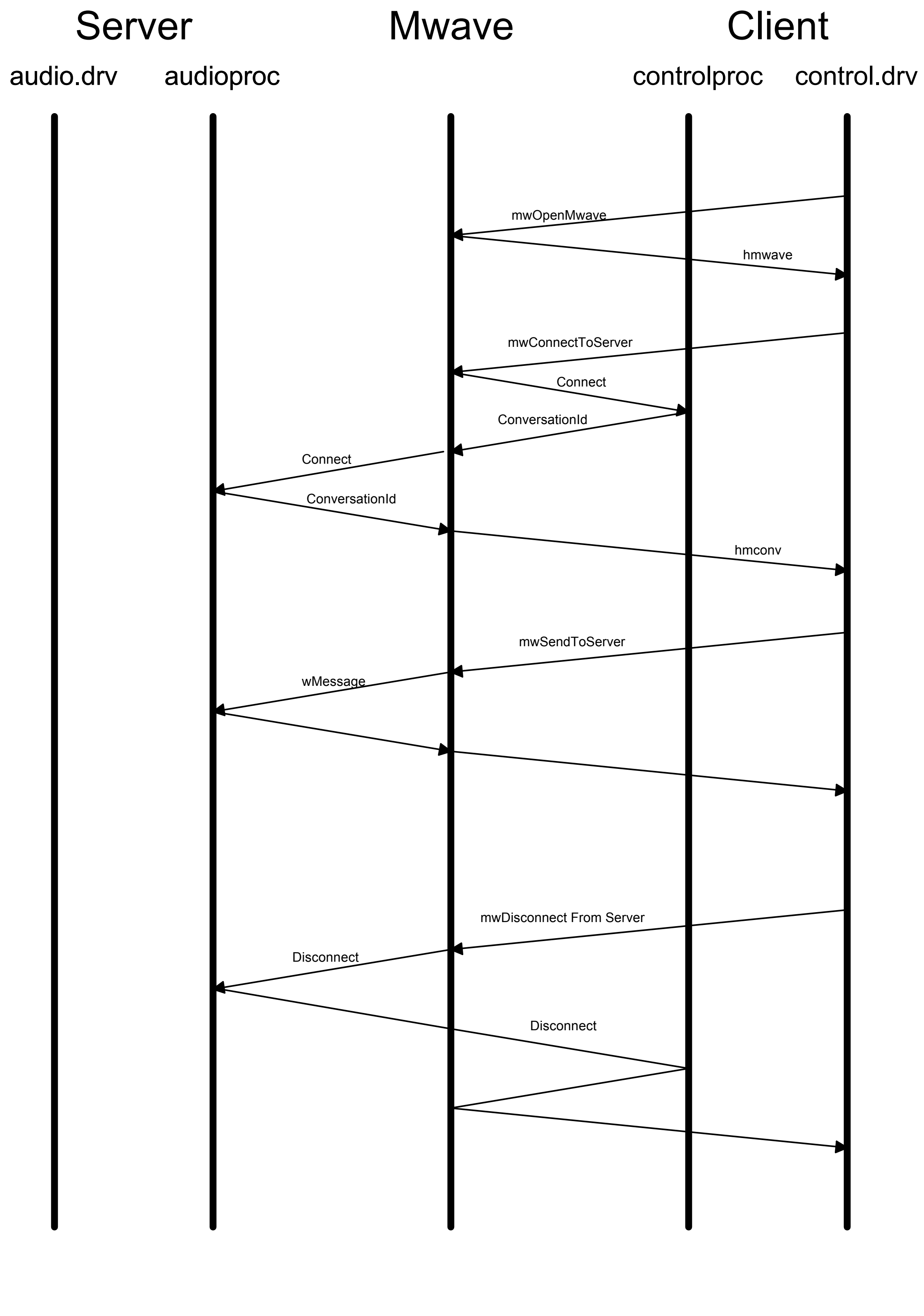
Mwave Agent/Client/Server Relationships



Server Registration Flows



Client Connect, Send, Disconnect Flows



API Description

This section describes the set of API calls for Mwave Client/Server support.

Mwave Agent Error Reporting

All API calls have identical error return protocol. Every API function returns some type that is 32 bits in length. If the Mwave Agent needs to return an error code, it is returned in the 32 bit type. Error return values can be distinguished from normal return values by the range of the value. Error return values, and conversely, no normal return values, fall within the range of 0xFF000000 to 0xFFFF0000. A macro in MWAGENT.H provides a simple test to determine if a returned value is an Mwave Agent error code. The macro is mwAGErrChk(rc), where rc is the value returned by the API function. The macro has a non-zero (TRUE) value if the returned value is an error code.

Error code values are listed below.

Mwave Agent Error Codes

Error Name	Error Value	Description
MWAGERR_INVALIDPARMS	0xFFFF0000	invalid parameters
MWAGERR_NAMEINUSE	0xFFFE0000	Object Name already in use
MWAGERR_SERVERNOTREGISTERED	0xFFFD0000	unknown server
MWAGERR_INVALIDHANDLE	0xFFFC0000	invalid handle
MWAGERR_UNSUPPORTEDMESSAGE	0xFFFB0000	unsupported message to client or server proc
MWAGERR_INVALIDPROC	0xFFFA0000	invalid procedure address
MWAGERR_SYSTEMERROR	0xFFF90000	operating system error
MWAGERR_NOMORERESOURCES	0xFFF80000	out of resources

API Calls for Mwave Client/Server Support

mwOpenMwave

HMWAVE	mwOpenMwave (lpszUserName, lParam1, lParam2)
LPSTR	lpszUserName
LPARAM	lParam1
LPARAM	lParam2

Opens an Mwave context. Allows a single call mwCloseMwave to clean up all resources allocated in this context.

Parameters

lpszUserName
Allows a text string name to be attached to the user of Mwave resources.

lParam1
Reserved for future use. Must be zero.

lParam2
Reserved for future use. Must be zero.

Return Value

The return value is the Mwave context handle if successful. A NULL context handle is returned if the function is unable to allocate storage for the Mwave context. Function returns error code: MWAGERR_INVALIDPARMS if lParam1 or lParam2 is not 0.

Note:

MWAGERCHK_INVALIDPARMS is necessary but not sufficient for this API. It does not check for null.

mwCloseMwave

LRESULT	mwCloseMwave (hmwave,reserved)
HMWAVE	hmwave
LPARAM	reserved

Close an Mwave context previously opened by mwOpenMwave. All resources allocated are freed.

Parameters

hmwave
Mwave context handle.

reserved
must be 0

Return Value

The return value is 0 if successful. Function returns error code MWAGERR_INVALIDHANDLE if the HMWAVE handle is invalid.

mwRegisterServer

HMSERVER	mwRegisterServer (hmwave, lpszServerName, serverprc, lParam. reserved)
HMWAVE	hmwave
LPSTR	lpszServerName
SRVPROC	serverprc
LPARAM	lParam
LPARAM	reserved

Registers a server with the Mwave Agent. Client establishes conversations with the server by issuing a mwConnectToServer specifying the name provided by the lpszServerName argument. This function sends the SRV_REGISTER message to the ServerProc (see Appendix A).

Parameters

hmwave
Mwave system handle that was returned by mwOpenMwave

lpszServerName
The name clients use to connect to the server. This name must be unique.

serverprc
Specifies the address of the procedure that is called when messages are sent to the server. (See Appendix A for the ServerProc function prototype on page 45.

lParam
User data that is passed as the lParam argument to ServerProc with the SRV_REGISTER message. This may be used by the ServerProc to create server instance specific data. The use of this parameter is entirely up to the implementer of the server.

reserved
Must be 0

Return Value

The return value is the handle of the registered server if successful. The function returns the error code MWAGERR_INVALIDHANDLE if the HMWAVE is invalid, or MWAGERR_NAMEINUSE if the lpszServerName is in use, or MWAGERR_INVALIDPARMS if reserved is non-zero.

mwUnregisterServer

LRESULT	mwUnregisterServer (hmserver, reserved)
HMSERVER	hmserver
LPARAM	reserved

Unregisters the specified server. For each conversation of this server, Mwave Agent first sends the SRV_DISCONNECT to the server followed by the CLI_DISCONNECT message to client. Then the SRV_UNREGISTER message is sent to the server.

Parameters

hmserver
    The handle of the server.

reserved
    Must be 0

Return Value

The return value is 0 if successful. Possible error return values are MWAGERR_INVALIDPARMS if the reserved value is non-zero or MWAGERR_INVALIDHANDLE if the HMSERVER handle is invalid.

mwConnectToServer

HMCONV	mwConnectToServer (hmwave, lpszServerName, lpszClientName, clientprc, lInitialData)
HMWAVE	hmwave
LPSTR	lpszServerName
LPSTR	lpszClientName,
CLTPROC	clientproc
LPARAM	lInitialData

Connects a client to the specified server. The server must have been previously registered with a mwRegisterServer call. mwConnectToServer() sends a CLI_CONNECT message to the specified ClientProc passing lInitialData, in dwConversationId, as the initial instance data and the lpszServerName in wSize and pBuffer. It also sends a SRV_CONNECT message to the ServerProc for the specified Server passing the lpszClientName in wSize and pBuffer. The HMCONV value is passed to both ClientProc and ServerProc in their respective lParam parameters.

Parameters

hmwave
Mwave system handle that was returned by mwOpenMwave.

lpszServerName
Used to specify the server to be connected to this client. The string is also passed to the ClientProc in wSize/pBuffer.

lpszClientName
Used to tell server name of connected client. Passed to the ServerProc in wSize/pBuffer.

clientprc
Specifies the address of the procedure that is called when messages are sent to the client. The ClientProc is called by the Mwave Agent during mwConnectToServer and mwDisconnectFromServer calls and also may be called by the connected server via mwNotifyClient calls. See the ClientProc function prototype in Appendix A on page 45.

lInitialData
User data that is passed as the dwConversationId argument to ClientProc with the CLI_CONNECT message. This is typically used as initial instance data by the ClientProc. Note that the ClientProc's dwConversationId is replaced, on all subsequent calls, with the value returned by the ClientProc after processing the CLI_CONNECT message.

Return Value

The return value is the conversation handle, if successful. The function returns the error code MWAGERR_INVALIDHANDLE if the HMWAVE is invalid, or MWAGERR_SERVERNOTREGISTERED if the server has not registered with the Mwave Agent.

mwDisconnectFromServer

LRESULT	mwDisconnectFromServer (hconversation, reserved)
HMCONV	hconversation
LPARAM	reserved

Disconnects the client from server. This call sends the SRV_DISCONNECT to the ServerProc followed by the CLI_DISCONNECT message to ClientProc.

Parameters

hconversation
Client/Server conversation handle returned from mwConnectToServer()

reserved
Must be 0

Return Value

The return value is 0 if successful. The function returns the error code MWAGERR_INVALIDHANDLE if the hconversation is invalid, or MWAGERR_INVALIDPARMS if reserved is non-zero.

mwSendToServer

LRESULT	mwSendToServer (hconversation, wMessage, lParam, wSize, pBuffer)
HMCONV	hconversation
USHORT	wMessage
LPARAM	lParam
USHORT	wSize
PVOID	pBuffer

Sends the message specified by wMessage and optionally a value, lParam, and a buffer length, wSize, and address, pBuffer, to the server.

Parameters

hconversation
The conversation handle returned by mwConnectToServer().

wMessage
This may be any message, greater than or equal to SRV_USERBASE, to which the server responds.

lParam
The meaning of this parameter is dependent on the wMessage value. This parameter may be used to pass any 32 bit (or shorter) data value to the server. It is not recommended that this be used as a pointer, however, since some addresses may be invalid in the server context on some operating systems.

wSize
Specifies the length in bytes of the buffer pointed to by pBuffer.

pBuffer
If wSize does not equal zero, pBuffer specifies a pointer to a buffer containing wSize bytes. If wSize=0, this parameter is ignored.

The meaning of this parameter is dependent on the wMessage value. This parameter may be used to pass a block of data to the server. The size of the block must be less than or equal to 65535 bytes. The data block should not contain pointers, however, since some addresses may be invalid in the server context on some operating systems.

Return Value

The return value is server/wMessage dependent. Refer to Server documentation for a description of return values. Mwave Agent returns the error code MWAGERR_INVALIDHANDLE if the hconversation is invalid, or MWAGERR_UNSUPPORTEDMESSAGE if the server does not support the wMessage value.

Comments

The values for lParam and pBuffer are entirely dependent on the ServerProc/wMessage combination. Refer to individual server documentation for descriptions of messages and their corresponding parameters. Note that wMessage values below SRV_USERBASE are reserved by the Mwave Agent and their corresponding parameters are described in Appendix A, ServerProc and ClientProc Protocol, on page 45.

mwNotifyClient

LRESULT	mwNotifyClient (hConversation, wMessage, lParam, wSize, pBuffer)
HCONV	hconversation
USHORT	wMessage
LPARAM	lParam
USHORT	wSize
PVOID	pBuffer

Sends the message specified by wMessage and optionally a value, lParam, and a buffer length, wSize, and address, pBuffer, to the Client.

Parameters

hconversation
The conversation handle passed to the server by way of the ServerProc in lParam with the SRV_CONNECT message.

wMessage
This may be any message, greater than or equal to CLI_USERBASE, to which the client responds.

lParam
The meaning of this parameter is dependent on the wMessage value. This parameter may be used to pass any 32 bit (or shorter) data value to the client. You should not use this parameter as a pointer, however, since some addresses may be invalid in the client context on some operating systems.

wSize
Specifies the length in bytes of the buffer pointed to by pBuffer.

pBuffer
If wSize does not equal zero, pBuffer specifies a pointer to a buffer containing wSize bytes. If wSize=0, this parameter is ignored.

The meaning of this parameter is dependent on the wMessage value. This parameter may be used to pass a block of data to the client. The size of the block must be less than or equal to 65535 bytes. The data block should not contain pointers, however, since some addresses may be invalid in the client context on some operating systems.

Return Value

The return value is client/wMessage dependent. Refer to Client documentation for description of return values. Mwave Agent returns the error code MWAGERR_INVALIDHANDLE if the hconversation is invalid, or MWAGERR_UNSUPPORTEDMESSAGE if the client does not support the wMessage value.

Comments

The values for lParam and pBuffer are entirely dependent on the ClientProc/wMessage combination. Refer to individual server documentation for descriptions of messages and their corresponding parameters. Note that wMessage values below CLI_USERBASE are reserved by the Mwave Agent and their corresponding parameters are described in Appendix A, ServerProc and ClientProc Protocol, on page 45.

Clients must be prepared to handle any notifications that a connected-to server are likely to generate.   Refer to server documentation relating to notifications and what, if any, events may cause a server to send notifications to the ClientProc.

Chapter 3 -- Mwave Notification Server

This chapter describes the messages and functions provided by the Mwave Notification Server. This server, used in conjunction with the Mwave Agent, (see Mwave Agent/Client/Server Relationships on page 3), provides Mwave Clients with notifications for events such as DSP INIT, DSP FAIL, and Power Management Events.

This Server is implemented for Windows, OS/2, WIN-OS2, and DOS. The bulk of the Server is OS independent and the Server could, therefore, be easily implemented for any operating systems supported by the Mwave Agent.

Notification Server Name

The Mwave Notification Server registers itself with the name MWNOTIFY.

Messages Supported by Notification Server

NS_REQNOTIFICATION

A client sends this message to request or update a previous request for notification services. A client can request notification of any combination of the following events:

Notification Events

EVENT TYPE	EVENT	DESCRIPTION
DSP INIT	DSP Initiation	Notify client when a DSP INIT has occurred.
DSP FAIL	DSP Failure	Notify client when one of the specific DSP failure events occur.
PCMCIA Events	PCMCIA card events	Notify client when an Mwave PCMCIA card has been added or removed from the system.
Power Management	Power Suspend Power Resume Power Critical Resume	Notify client when a power management event has occurred. Note: If the client processes any of the events, the client should process all of the power management events. Note: This event can only be generated on machines and operating systems providing Power Management capability.

If any of the requested events occur, mwNotifyClient() sends an NS_EVENTNOTIFICATION message to the client. The lParam parameter sent to the ClientProc contains bits indicating the cause of the event. The Event Filters table lists the masks that may be used to determine the exact events. While it is unlikely, it is possible for more than one event to be posted within a single notification. The ClientProc should check for all events that it processes. See ClientProc on page 48 for additional information.

The client may optionally specify a DSP handle with the request. If a DSP handle is included with the request, the client is only notified of the relevant events for that DSP. If no DSP handle is included, the client is notified of relevant events for any DSP. The pBuffer parameters pass the DSP handle to the Notification Server. The caller must set wSize to sizeof(HDSP) and the pBuffer parameter to the address of a valid HDSP variable containing the DSP handle. If a DSP handle is not being passed to the Notification Server then the caller merely sets wSize to 0. A client may specify a maximum of one DSP handle. If multiple requests are issued by a client, the last DSP handle value (including none) used. If a client wants to receive events for more than one DSP, the client must request notifications of the events composite for all DSPs and filter the notifications within its ClientProc().

Event Filters

NS_DSPINIT	DSP is initialized. This may be due to a dspInit() call to the Mwave Manager or the result of some internal exception processing of the Manager. All DSP data is lost. A DSP FAIL event is usually followed by a DSP INIT event.
NS_DSPFAIL	The DSP has failed in some way. This mask is the Boolean sum of the next set of individual failure filters. This event indicates that the DSP stopped processing but the DSP data may still be valid. Check individual failure events for likelihood of valid data.
NS_DSPMIPO	The DSP stopped processing due to a task overrunning its allocated cycle count. This indicates that the DSP was overloaded with too much real-time processing. This error should only occur in a development environment where Instruction Cycle Counter error trapping is disabled for debug purposes. This is a critical error which would cause unpredictable results if not detected and reported.
NS_DSPICC	The insturction cycle counter is a 16 bit counter which decrements each time a DSP instruction is executed. The insturction cycle counter interrupt occurs whenever the insturction cycle determines whether the current task has exceeded its allocated number of insturction cycles by checking the upper 16 bits of the task's instruction cycle allocation. If the upper 16 bits is not zero, then it is decremented and control is returned to the interrupted task. If the upper 16 bits is zero, then the task has exceeded its allocated instruction cycles. The Mwave Manager can identify the task which violated its cycle count by reading SYSDSPTR. This is a critial error which indicates a serious system error or a flaw in the task code.
NS_PCMCIA_CARDEVENT	This mask is the Boolean sum of the twc PCMCIA Event Filters. It can be used to set the value for the notification request and as a general filter for PCMCIA events.
NS_PCMCIA_CARDUNPLUGGED	The DSP is removed from the system and no longer available for use. The Mwave Manager returns DSP_NOTAVAILABLE on all calls related to this DSP after this event is processed by the Manager. The Notification Server generates an NS_PWRSUSPEND event for this DSP after the PCMCIA event is sent.
NS_PCMCIA_CARDREPLUGGED	The DSP is reinserted into the system. This event is sent and the Notification Server generates an NS_PWRCRITICALRESUME event for this DSP.

Event Filters (continued)

NS_PWRMGMT	This mask is the Boolean sum of the three Power Management Event Filters. It can be used to set the value for the notification request and as a general filter for Power Management events.
NS_PWRSUSPEND	This event is generated when the host system is suspending all activity in order to save power. The client should stop its DSP processing and save any pertinent state information. Clients are guaranteed that the Manager's services are available at the time of this notification but should assume that the DSP is halted and reset after return.
NS_PWRRESUME	This event is generated when the host system resumes processing after it had suspended operations. The client should reload all DSP tasks, reconnect all GPCs and ITCBs, and restore its DSP state in preparation for normal activity. Clients are guaranteed that the Manager's services are available at the time of this notification and the DSP has been initialized with MwaveOS.
NS_PWRCRITICALRESUME	This event is generated when the host system resumes processing after it has shutdown abnormally without having previously sent an NS_PWRSUSPEND notification. Consequently, no state information was saved during the last shutdown. The client should recover by reloading all DSP tasks, reconnecting all GPCs and ITCBs, and restoring its DSP state in preparation for normal activity. Clients are guaranteed that the Manager's services are available at the time of this notification and the DSP has been initialized with MwaveOS.

Parameters

dwServerId
Server Instance data returned fromSRV_REGISTERprocessing.

dwConversationId
Conversation Instance data returned fromSRV_CONNECTmessage

wMessage
NS_REQNOTIFICATION

lParam
Requested event filter using bits as defined in the above table. A zero value in a bit position resets a previous request for the corresponding event.

wSize
Should be 0 if no DSP handle is passed to the Notification Server, and should be set to sizeof(HDSP) if a DSP handle is passed.

pBuffer
Not valid if wSize = 0. pBuffer contains the address of a DSP handle if wSize = sizeof(HDSP).

Return Value

0

NS_EVENTNOTIFICATION

This message is sent by the Server to a client when an event occurs matching the client's requested event notification filter. See NS_REQNOTIFICATION on page 16 for additional information.

Parameters

dwConversationId
Conversation Instance data returned from CLI_CONNECT message

wMessage
NS_EVENTNOTIFICATION

lParam
Event filter using bits as defined in the table, Event Filters, on page 17.

wSize
Will be 0 if no DSP handle is passed to the Notification Server. Sets to sizeof(HDSP) if a DSP handle is passed.

pBuffer
Not valid if wSize = 0. pBuffer contains the address of a DSP handle if wSize = sizeof(HDSP).

Return Value

Returns any bits for events that are NOT processed by the client. Normally, a client would process the one (and only) event causing the notification and would, consequently, return 0.

Chapter 4 -- Mwave Contention Server

This chapter describes the messages used and functions provided by the Mwave Contention Server and includes:

- an overview of the Contention Server
- an in-depth explanation of how the Server manages contention among clients
- descriptions of the messages supported
- information about the development and testing environment for client writers

Contention Server Overview

The Contention Server, used in conjunction with the Mwave Agent (See Chapter 2 —Mwave Client/Server Support on page 3), provides its clients (drivers) with a mechanism to contend for limited Mwave resources. Resources include:

- data store
- instruction store
- MEIO connections to other hardware components
- DMA channels
- IPCs
- MIPS

The Contention Server is implemented for Windows 3.1, Windows 95, OS/2, and WIN-OS2. However, the bulk of the Server is OS-independent and could, therefore, be easily implemented for any operating systems supported by the Mwave Agent.

Contention Server Name

The Mwave Contention Server registers itself with the name MWCONTEND.

Understanding Contention Management

The Contention Server is implemented to provide clients (drivers) with a mechanism to contend for limited system resources. This server does not enforce contention management activities such as loading and unloading tasks; that function is the handled by the Mwave Manager. Instead, when clients (drivers) register with the Contention Server, it keeps a list of the client names, the priority at which they register, and their state (active or stopped). When a client attempts to use Mwave resources and resource constraints prevent it from successfully loading, the client notifies the Server. The Server then requests that all clients, whose priority is lower than or equal to that of the client whose load failed, unload to free resources for the higher priority client.

The Contention Server only operates as a notification mechanism and has no enforcement capability. Therefore, the Contention Server uses a buddy system to maintain a reasonable balance of concurrent function in the Mwave system. Each client is a buddy who agrees to operate within the established policy constraints and to allow optimum concurrency in the system.

The priorities of each client are established based on a projection of the customer's personal preferences. Different markets call for different sets of priority relationships. For example, in the small office/home office (SOHO) market and in general, telephony functions are higher priority than audio functions. In the home market, however, the priorities might be a little more balanced to give priority to games and audio functions.
Client developers should not consider their client the most important in the system. Instead, they should consider that the customer can best be served by all the clients working cooperatively with the Contention Server to provide the greatest concurrency and most seamless service to the end user.

The Contention Server is accessible to clients when they register with the Mwave Agent. The following diagram illustrates how clients communicate with the Contention Server via the Mwave Agent.

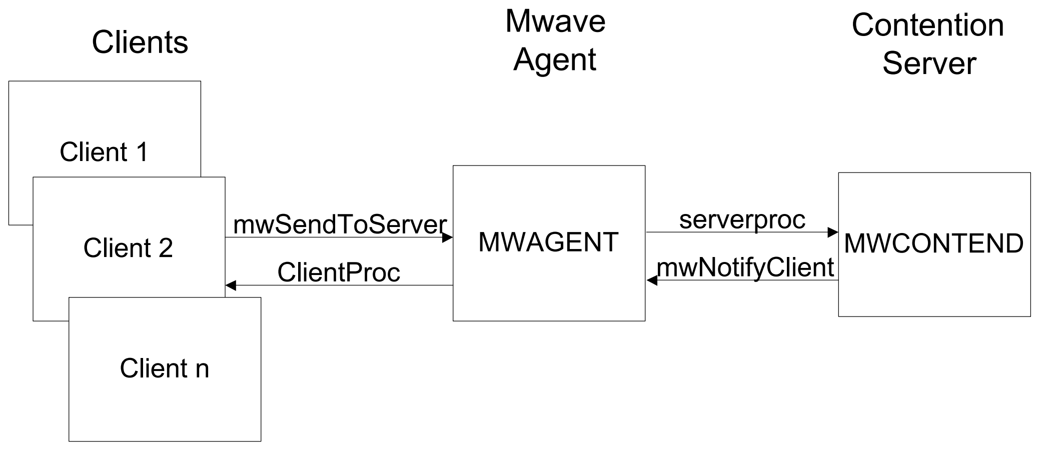


Figure 4- 1. Mwave Client-Server Environment

As shown in Figure 4- 1. Mwave Client-Server Environment , the clients use the mwSendToServer()call to communicate with the Contention Server via the Mwave Agent. The Mwave Agent calls the Contention Server's serverproc to pass the information contained in the mwSendToServer() Conversely, the Contention Server uses the mwNotifyClient()call to send information to the Agent, which calls the client's ClientProc to pass that information on to the client.

For more information about how clients and servers communicate via the Mwave Agent, see Mwave Client/Server Support on page 3.

Understanding Relative Priority Values

The Contention Server uses the priority of a client (relative to the priority of other clients in the system) to determine which clients should load and in what order. The priority is always passed in the lParam of the messages sent between the client and the Contention Server.

Priorities are defined in the MWAVE.INI file. However, client writers should NOT get priorities directly from the MWAVE.INI or hardcode the priorities in the client code. Instead, client writers should include the MWHELPER.DLL or MWHLPOS2.DLL, which includes the mwHelpGetLOSPriorities() function. (See the description of the mwHelpGetLOSPriorities function on page 44 for more information about the function.)

Because the priorities may change in different implementations of the Mwave product, client writers should write clients to work at any priority level. Clients can register with the Contention Server multiple times at different priorities to provide granular levels of service. For example, the Mwave Synth driver registers its 32-voice driver at a low level of priority because it consumes the greatest amount of resources. Mwave Synth also registers its 24, 16, and 8-voice drivers at higher levels of priority. When an application requests the Synth driver by an MCI call, the Mwave Synth can attempt to load its 32-voice driver at a median priority. If that attempt fails, the Mwave Synth can keep trying until one of its higher priority drivers successfully loads. As resources become available, the higher priority Synth drivers get an opportunity to load and increase the level of service and the quality of sound.

Typical Scenarios

This section includes scenarios that illustrate how the Contention Server and clients communicate in a dynamic system environment. These examples show the role of the Server and how it manages contention among several clients.

Scenario 1: Simple Transaction Without Contention

Figure 4- 2. Simple Transaction Without Contention, illustrates a simple transaction that occurs when clients request resources, obtain them without contention, and later release the resources. Figure 4- 2 also shows how the client uses the mwConnectToServer() call to register with the Contention Server. All subsequent communications use the mwSendToServer() call to communicate with the Contention Server. The messages (such as mwCfgServiceRequest) and the priority are passed as parameters.

Note: For the sake of brevity, subsequent illustrations omit the mwConnectToServer() and mwDisconnectFromServer() calls. Also, the mwSendToServer() call is assumed; only the message and priority that are passed are shown in the format (messagename, priority).

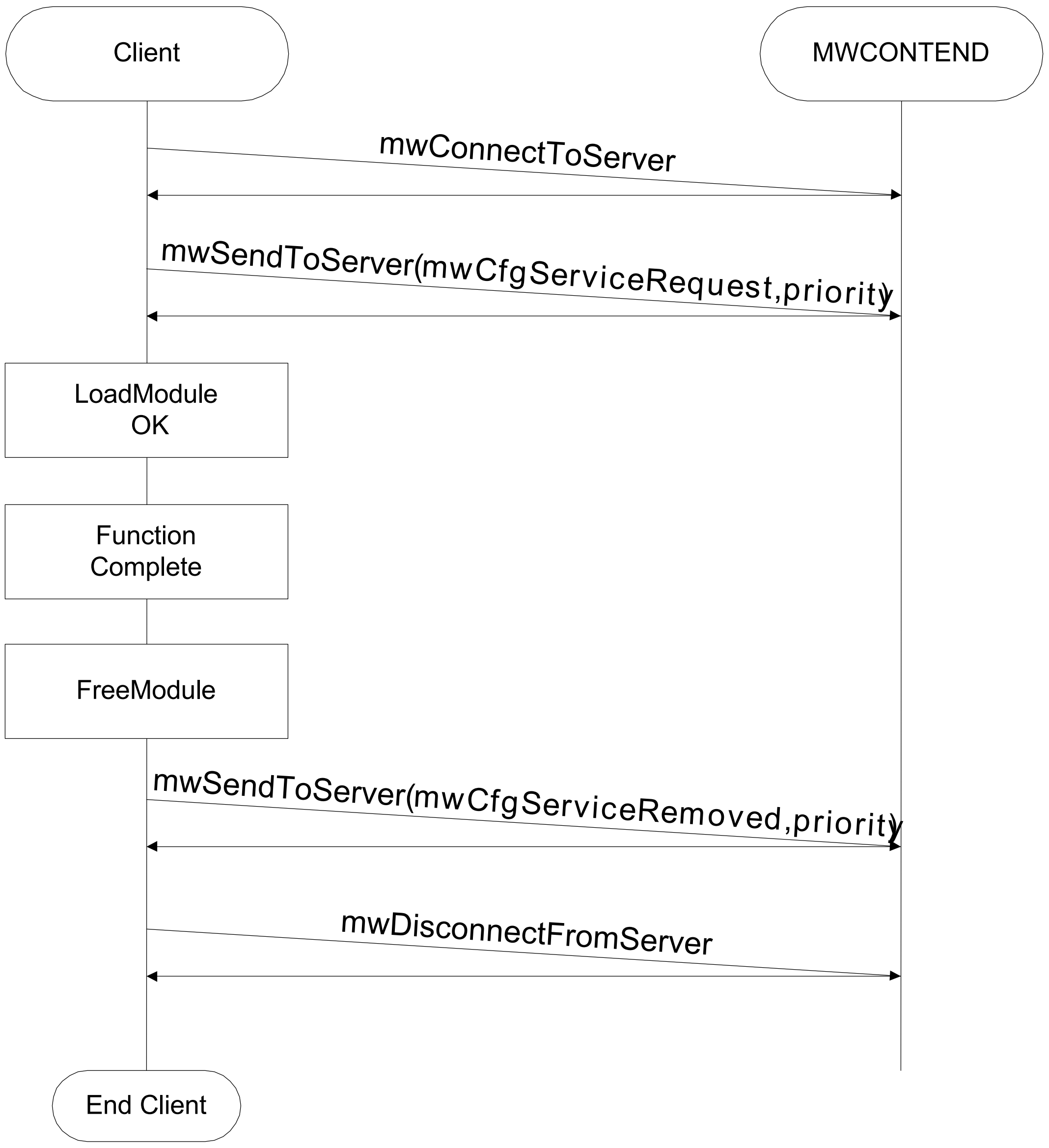


Figure 4- 2. Simple Transaction Without Contention

Scenario 2: Contention Between Two Clients

When two clients that must contend for resources attempt to connect, the scenario is more involved. In the example shown in Figure 4- 3. Contention Between Two Clients, the high priority client loads first. Then the low priority client attempts to load and is unable to do so; that client sends a `mwCfgServiceFailure`message to the Server. The Server then checks the list for any active clients whose priority is lower than that of the client who failed to load. Finding none, the Server marks the low priority client as stopped, which indicates that the Server will notify the client when resources are available. When the high priority client releases the resources, the Server removes it from the list, checks the list for clients that are stopped, and sends a `mwCfgReinstateService`message to the low priority client, who then loads successfully. The low priority client then sends a return code (RC=1) to the Server to indicate that it successfully loaded. The Server then marks the client as active.

Note: In Figure 4- 3, the `mwSendToServer()`calls are not shown. Only the message and priority that are passed are shown in the format (`messagename,priority`).

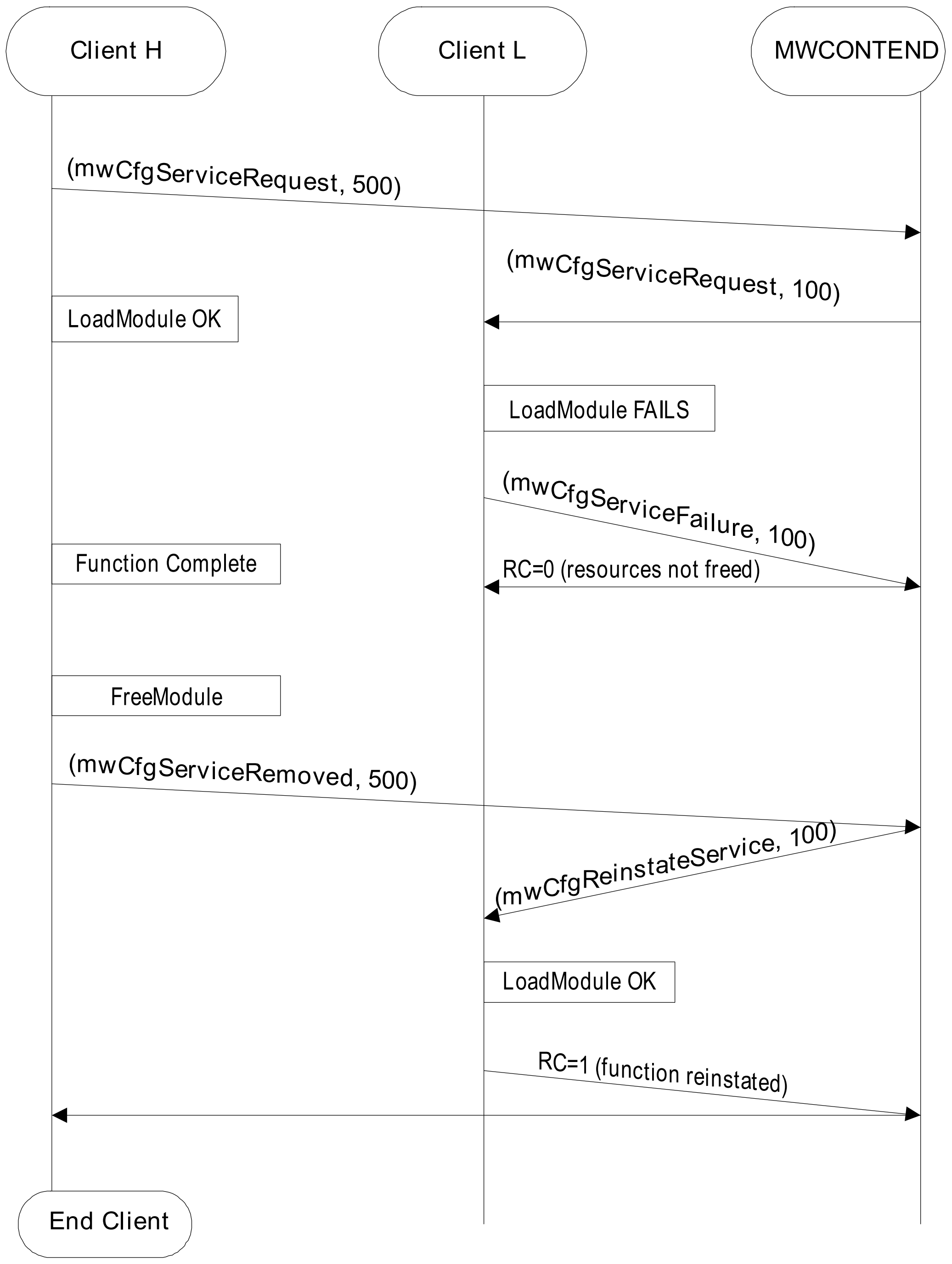

Client H
Client L
MWCONTEND
(mwCfgServiceRequest, 500)
(mwCfgServiceRequest, 100)
LoadModule OK
LoadModule FAILS
(mwCfgServiceFailure, 100)
Function Complete
RC=0 (resources not freed)
FreeModule
(mwCfgServiceRemoved, 500)
(mwCfgReinstateService, 100)
LoadModule OK
RC=1 (function reinstated)
End Client

Figure 4-3. Contention Between Two Clients

Scenario 3: Contention Among Three Clients

A more complicated scenario involves three clients contending for resources. As illustrated in Figure 4- 4. Contention Among Three Clients, the high priority client loads first; the Server marks it active in the list. The low priority client then loads successfully and is marked active in the list. When the medium priority client attempts to load and fails, it sends a mwCfgServiceFailure message to the Server. The Server checks the list and sends a mwCfgReleaseService message to the low priority client. The low priority client unloads and sends ReleaseServiceCanReinstate return code to indicate to the Server that the client released and should be notified when resources are available. The Server returns a return code of 1 to the medium priority client, who tries again, loads successfully and returns a mwCfgServiceAvailable message. After marking the medium priority client active, the Server sends a mwCfgReinstateService message to the low priority client, who does not load successfully and returns a zero return code.

Note: In Figure 4- 4, the mwSendToServer() calls are not shown. Only the message and priority that are passed are shown in the format (messagename, priority)

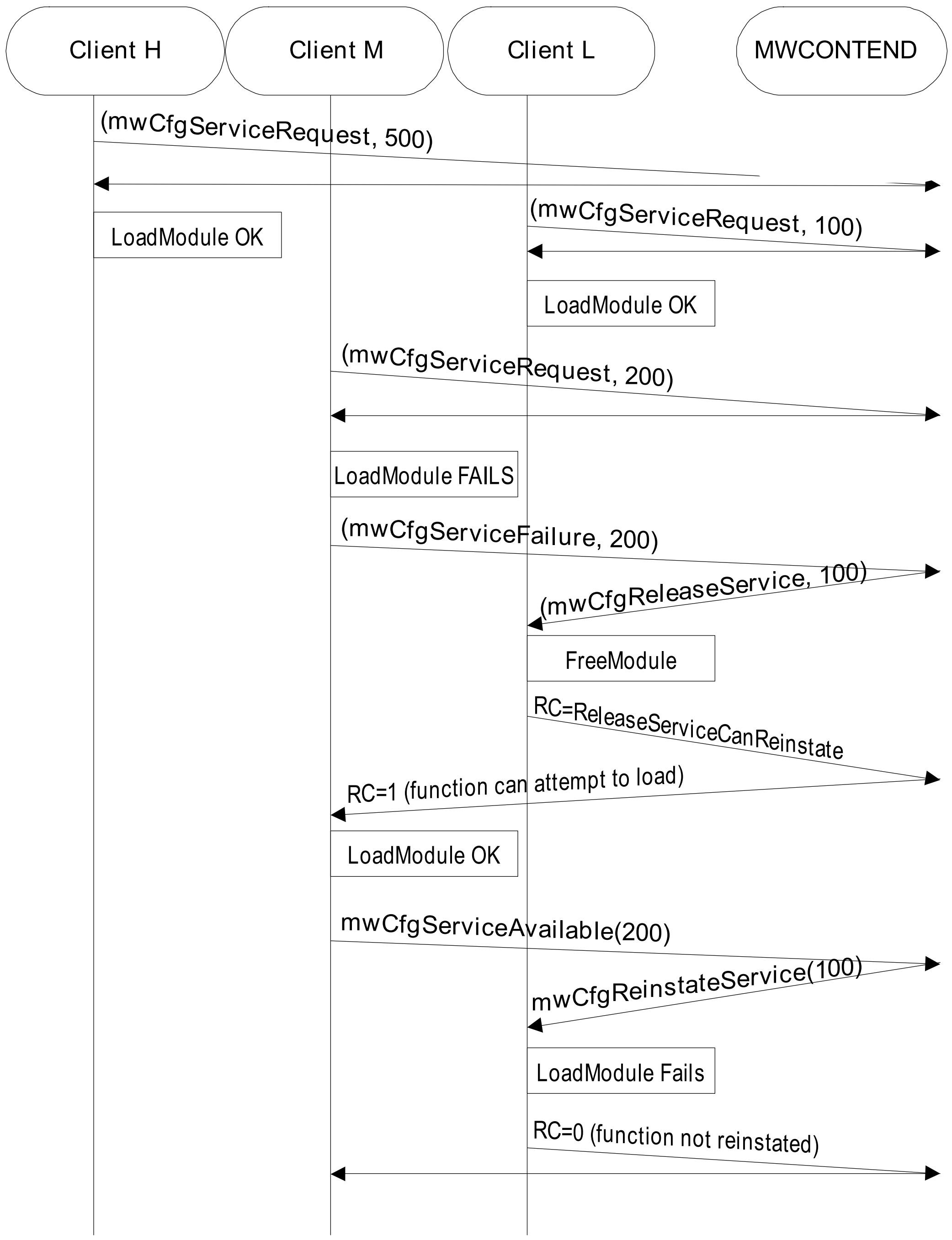

Client H
Client M
Client L
MWCONTEND
(mwCfgServiceRequest, 500)
LoadModule OK
(mwCfgServiceRequest, 100)
LoadModule OK
(mwCfgServiceRequest, 200)
LoadModule FAILS
(mwCfgServiceFailure, 200)
(mwCfgReleaseService, 100)
FreeModule
RC=ReleaseServiceCanReinstate
RC=1 (function can attempt to load)
LoadModule OK
mwCfgServiceAvailable(200)
mwCfgReinstateService(100)
LoadModule Fails
RC=0 (function not reinstated)

Figure 4-4. Contention Among Three Clients

Scenario 4: Client Removed Rather Than Reinstated

This scenario illustrates a complex example in which the Server's request to release causes the low priority client to terminate and exit the system rather than wait to be reinstated.

As shown in Figure 4- 5. Client Removed Rather Than Reinstated, the following events occur:

1. A medium priority client sends a mwCfgServiceRequest message to the Server and successfully loads.
2. Next, a high priority client sends a mwCfgServiceRequest message to the Server, attempts to load, and fails. This client sends a mwCfgServiceFailure message to the Server, which starts a contention cycle.
3. The Server sends a mwCfgReleaseService message to the medium priority client, which frees the module and sends a ReleaseServiceCanReinstate return code to indicate that the Server should notify this client when resources are available.
4. To indicate that a lower priority task has been removed and resources freed, the Server sends a return code of 1 to the a mwCfgServiceFailure message sent by the high priority client.
5. The high priority client then attempts to load.
6. While the high priority client is loading, the low priority client sends a mwCfgServiceRequest message to the Server and loads.
7. When the high priority service finishes loading, it sends a mwCfgServiceAvailable message to the Server.
8. The Server then sends a mwCfgReinstateService message to the medium priority client, which attempts to load, and fails.
9. The Server then requests the low priority task to unload.
10. When the low priority client unloads, it also terminates and sends a ReleaseServiceTerminated return code to the Server to indicate that the client has left the system and should be taken out of the list.
11. The Server then sends a mwCfgReinstateService message to the medium priority client, which successfully loads and returns a mwCfgServiceAvailable message.

Note: In Figure 4- 5, the mwSendToServer() calls are not shown. Only the message and priority that are passed are shown in the format (messagename, priority)

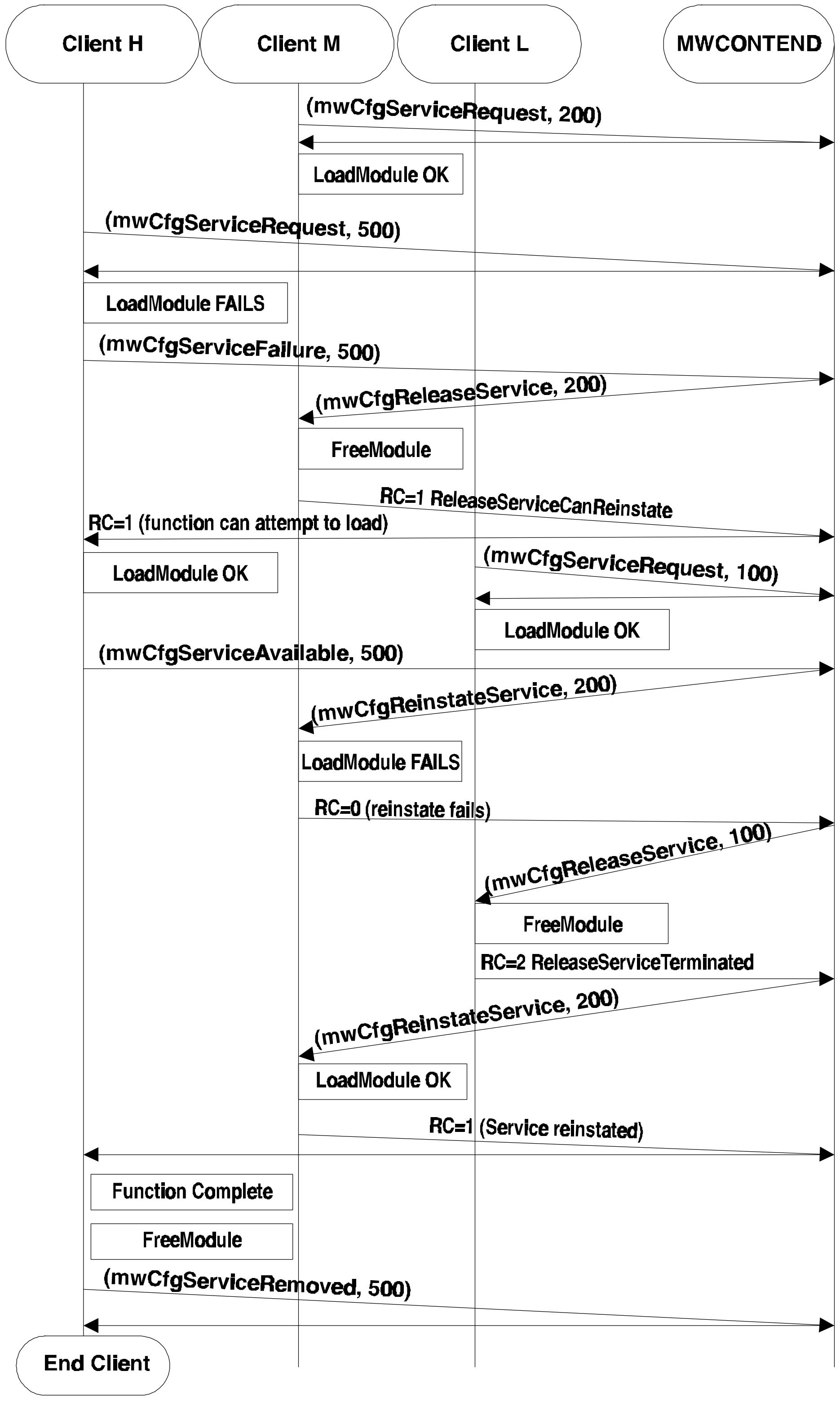

Client H
Client M
Client L
MWCONTEND
(mwCfgServiceRequest, 200)
LoadModule OK
(mwCfgServiceRequest, 500)
LoadModule FAILS
(mwCfgServiceFailure, 500)
(mwCfgReleaseService, 200)
FreeModule
RC=1 ReleaseServiceCanReinstate
RC=1 (function can attempt to load)
LoadModule OK
(mwCfgServiceRequest, 100)
LoadModule OK
(mwCfgServiceAvailable, 500)
(mwCfgReinstateService, 200)
LoadModule FAILS
RC=0 (reinstate fails)
(mwCfgReleaseService, 100)
FreeModule
RC=2 ReleaseServiceTerminated
(mwCfgReinstateService, 200)
LoadModule OK
RC=1 (Service reinstated)
Function Complete
FreeModule
(mwCfgServiceRemoved, 500)
End Client

Figure 4-5. Client Removed Rather Than Reinstated

Development Environment for Contention Server Clients

Diagnostics

A diagnostics program (MWCONFGR.EXE) is included to help client writers debug their client implementations. When invoked, the diagnostic program uses the client name MWCONTEND to register with Mwave Agent. The Contention Server recognizes MWCONTEND as a special client. Using the MwNotifyClient()call, the Contention Server forwards a record of every action it performs to the MWCONTEND client. Developers can use the log kept by MWCONFGR as an audit trail when debugging their client implementations.

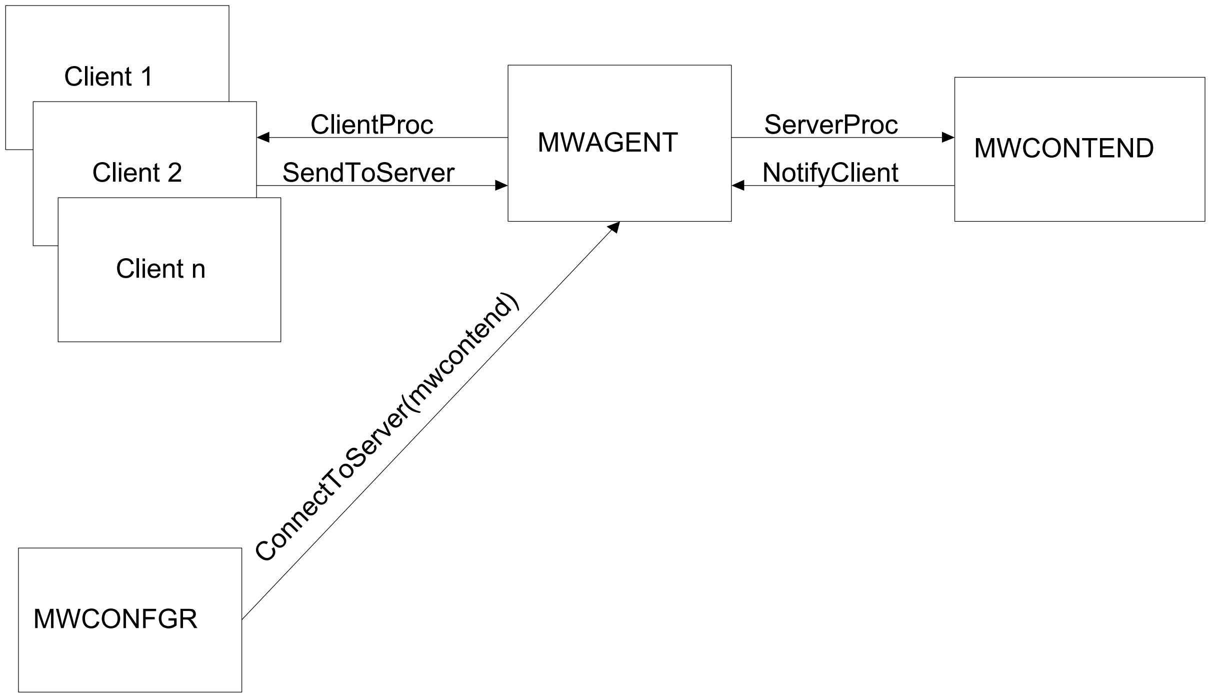
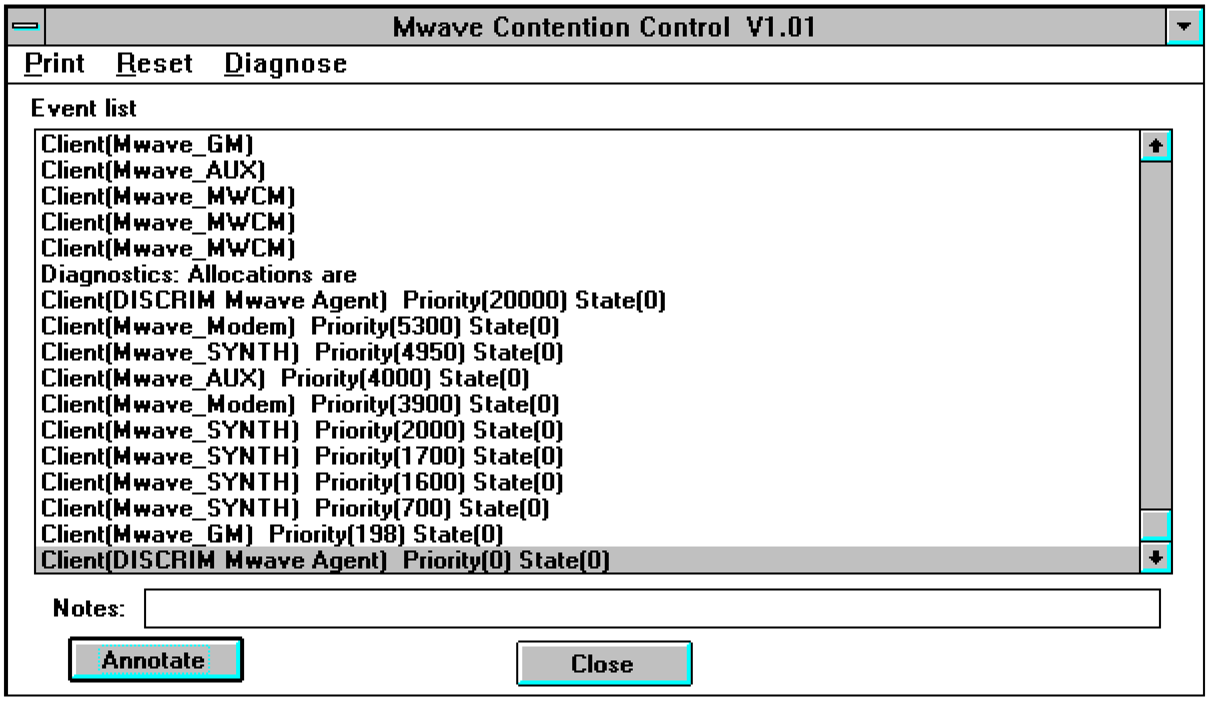


Figure 4-6. MWCONFGR Registers With the Contention Server via MWAGENT

Using the Diagnostics Tool (MWCONFGR.EXE)

The diagnostics tool (MWCONFGR.EXE) enables you to view a list of the clients registered with the Contention Server, the states of each of those clients, and the transactions. The Mwave Contention Control main window is shown in Figure 4- 7.



```
Client(Mwave_GM)
Client(Mwave_AUX)
Client(Mwave_MWCM)
Client(Mwave_MWCM)
Client(Mwave_MWCM)
Diagnostics: Allocations are
Client(DISCRIM Mwave Agent)  Priority(20000) State(0)
Client(Mwave_Modem)  Priority(5300) State(0)
Client(Mwave_SYNTH)  Priority(4950) State(0)
Client(Mwave_AUX)  Priority(4000) State(0)
Client(Mwave_Modem)  Priority(3900) State(0)
Client(Mwave_SYNTH)  Priority(2000) State(0)
Client(Mwave_SYNTH)  Priority(1700) State(0)
Client(Mwave_SYNTH)  Priority(1600) State(0)
Client(Mwave_SYNTH)  Priority(700) State(0)
Client(Mwave_GM)  Priority(198) State(0)
Client(DISCRIM Mwave Agent)  Priority(0) State(0)
```

Figure 4- 7. Mwave Contention Control Main Window

As shown in Figure 4- 7, you can select the following menu bar items:

Print	Writes the log to a printable text file.
Reset	Clears the log.
Diagnose	Displays current client assignments and resource allocations.

To add notes to the log, type text in the Notes entry field and click the Annotate button.   To close the diagnostics tool, click Close.

Test Application

A test application (MWTEST.EXE) is included to enable developers to simulate a dynamic environment for testing purposes.   Multiple instances of MWTEST can be invoked to simulate multiple clients attempting to load at various priorities.   All instances of MWTEST use one client, MWCFGCLI.DLL. This client includes the ClientProc through which messages are returned from the Contention Server.

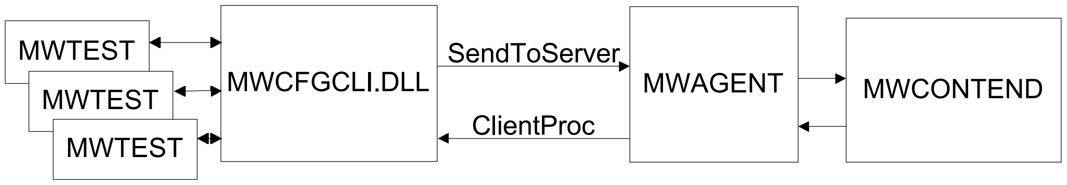
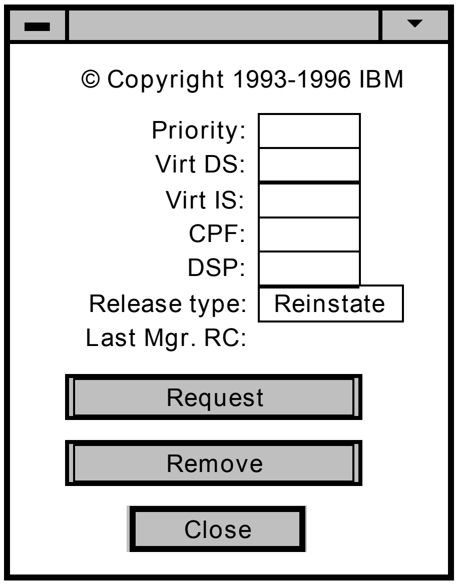


Figure 4- 8. Test Environment Using MWTEST

Figure 4- 8. Test Environment Using MWTEST, shows a sample client that uses MWCFGCLI.DLL to communicate with the Contention Server (MWCONTND) via the Mwave Agent (MWAGENT). MWTEST.C, the source for the test application, is also included as sample code for developer reference. This test application is written in C and compiled using Borland C.

Using the Test Application (MWTEST.EXE)

The test application (MWTEST.EXE) enables you to simulate a client that registers with the Contention Server and attempts to load in the system. The MWTEST main window is shown in Figure 4- 9.



© Copyright 1993-1996 IBM

Priority:
Virt DS:
Virt IS:
CPF:
DSP:
Release type: Reinstate
Last Mgr. RC:

Figure 4- 9. MWTEST Main Window

As shown in Figure 4- 9, you can set the following parameters on the MWTEST main window:

Priority	Sets the priority of the test client.
Virt DS	Sets the virtual DSP data store to be used by the test client.
Virt IS	Sets the virtual DSP instruction store to be used by the client.
CPF	Cycles per frame is used to compute client consumed MIPS.
Release Type	Return code formwCfgReleaseService(Reinstate, Ignore, Terminate).

Click the Request button to attempt to load the test client. Click Remove to unload the test client. Click Close to close the test client.

Using the Test Application (MWTEST) with the Diagnostics Tool (MWCONFGR)

The test application (MWTEST.EXE) used with the diagnostics tool (MWCONFGR.EXE) assists you in understanding the transactions that occur between the clients and the Contention Server. They are also used for testing purposes during client development. The log produced by MWCONFGR is particularly helpful when doing problem determination.
To better understand how the clients and the Contention Server interact, perform the following test scenario.

Note: This scenario assumes that you have an Mwave Reference Adapter installed in your personal computer and that the corresponding system software and test applications are installed and running under Windows 3.1, Windows 95, or Win-OS/2.

1. Open the diagnostics tool (MWCONFGR.EXE). Press Diagnose to view the list of clients and the current resource allocations.
2. Open the first instance of MWTEST.EXE as MWTEST1. On the Run menu, type:
   C:\fully_qualified_path\MWTEST.EXE MWTEST1 1000 20000 20000
3. On the Mwave Contention Control window, click Diagnose again to view the updated list of clients and resource assignments. Notice that MWTEST1 is in the list.
4. Open a second instance of MWTEST.EXE as MWTEST2. On the Run menu, type:
   C:\fully_qualified_path\MWTEST.EXE MWTEST2 800 20000 20000
5. Open a third instance as MWTEST3. On the Run menu, type:
   C:\fully_qualified_path\MWTEST.EXE MWTEST3 500 20000 20000
6. On the MWTEST2 window, click Request to attempt to load the client. On the Mwave Contention Control window, click Diagnose to view the updated list of allocations. Whether or not MWTEST2 loads is dependend on what else is running in your Mwave system. In many cases, MWTEST2 loads successfully and the following entry is written to the log: Client(MWTEST2) Priority(800) State(0)
7. On the MWTEST3 window, click Request to attempt to load the client. On the Mwave Contention Control window, click Diagnose to view the updated list of allocations. In most cases, MWTEST3 is not able to load and the following entry is written to the log: Client(MWTEST3) Priority(500) State(1)
8. On the MWTEST1 window, click Request to attempt to load the client. On the Mwave Contention Control window, click Diagnose to view the updated list of allocations. In most cases, MWTEST2 releases resources and goes to a stopped state. MWTEST1 loads successfully, and MWTEST3, still unable to load, is in a stopped state.

   The log entries are:

   ```
   Client(MWTEST1) Priority(1000) State(0)
   Client(MWTEST2) Priority(800) State(1)
   Client(MWTEST3) Priority(500) State(1)
   ```
9. Continue to attempt to load and unload the three test clients and notice the Contention Server actions that are documented in the diagnostics tool log.

Messages Supported by Contention Server

The Contention Server supports the following messages:

mwCfgServiceRequest
mwCfgServiceRemoved
mwCfgServiceAvailable
mwCfgServiceFailure
mwCfgServiceStopped

The client sends these messages to notify the Contention Server when the client requests services, removes services, successfully loads services, and fails to load services. These messages are sent through the mwSendToServer()call, which is supported by the Mwave Agent. For more information, see Mwave Client/Server Support on page 3.

The following code sample illustrates how the MWTEST client implements these messages:

```
ClientProc(..)
  {/switch(message)
        {
        ...
        case mwCfgReleaseService:
             FreeAllModules:
             break;
        case mwCfgReinstateService:
             .  .  .
             if (service is restored)
                  return true.
             else return false.
        }
  }

  ...
  hMwave = mwOpenMwave("MWContend User" ,0,0);
  hConv  = mwConnectToServer(hMwave,"mwContend","MWContend
                     User",ClientProc,0);

  mwSendToServer(hConv,mwCfgServiceRequest,Priority,0,0);

  ModuleLoadedOK = LoadDriverModules();
  if (!ModulesLoadedOK)
     {mwSendToServer(hConv,mwCfgServiceFailed,Priority,0,0);
     ModuleLoadedOK = LoadDriverModules();
     mwSendToServer(hConv,
       (ModulesLoadedOK ? mwCfgServiceAvailable :
mwCfgServiceStopped),
       Priority,0,0);
     }

  ...  (when all done)
  mwSendToServer(hConv,mwCfgServiceRemoved,Priority,0,0);
```

For more information about the MWTEST implementation, refer to the source files included with this document.

mwCfgServiceRequest

This message is sent to the Contention Server by a client to register a function at a given priority. When the Contention Server receives the mwCfgServiceRequest message, it adds the client's name and priority to the list, and marks the client's status as active—unless it receives a mwCfgServiceFailure message.

mwSendToServer Parameters

dwConversationId
: Conversation Instance data returned from SRV_CONNECT message

wMessage
: mwCfgServiceRequest

lParam
: Requested priority obtained using the mwHelpGetLOSPriorities function on page 44.

wSize
: Either 0 or sizeof(HDSP).

pBuffer
: If wSize=0, pBuffer=0. If wSize=sizeof(HDSP), pBuffer=address of an HDSP.

Return Value

The return code is 1.

mwCfgServiceRemoved

A client sends this message to the Contention Server when the client has removed a function from the system.

mwSendToServer Parameters

dwConversationId
: Conversation Instance data returned from SRV_CONNECT message

wMessage
: mwCfgServiceRemoved

lParam
: Requested priority as obtained using the mwHelpGetLOSPriorities function on page 44.

wSize
: Either 0 or sizeof(HDSP).

pBuffer
: If wSize=0, pBuffer=0. If wSize=sizeof(HDSP), pBuffer=address of an HDSP.

Return Value

The return code is 1.

mwCfgServiceFailure

This message is sent to the Contention Server by a client when the client fails to load.

mwSendToServer Parameters

dwConversationId
Conversation Instance data returned from SRV_CONNECT message

wMessage
mwCfgServiceFailure

lParam
Requested priority as obtained using the mwHelpGetLOSPriorities function on page 44.

wSize
Either 0 or sizeof(HDSP).

pBuffer
If wSize=0, pBuffer=0. If wSize=sizeof(HDSP), pBuffer=address of an HDSP.

Return Value

A non-zero return code indicates that a lower priority task has been removed; the driver can attempt LoadModule() again. A return code of 0 indicates that the driver should not attempt to load.

mwCfgServiceAvailable

This message is sent to the Contention Server by the client when the client loads successfully after a contention cycle.

mwSendToServer Parameters

dwConversationId
Conversation Instance data returned from SRV_CONNECT message

wMessage
mwCfgServiceAvailable

lParam
Requested priority as obtained using the mwHelpGetLOSPriorities function on page 44.

wSize
Either 0 or sizeof(HDSP).

pBuffer
If wSize=0, pBuffer=0. If wSize=sizeof(HDSP), pBuffer=address of an HDSP.

Return Value

The return code is 1.

mwCfgServiceStopped

This message is sent to the Contention Server by the client when a function still fails to load after a contention cycle.

mwSendToServer Parameters

dwConversationId
: Conversation Instance data returned from SRV_CONNECT message

wMessage
: mwCfgServiceStopped

lParam
: Requested priority as obtained using the mwHelpGetLOSPriorities function on page 44.

wSize
: Either 0 or sizeof(HDSP).

pBuffer
: If wSize=0, pBuffer=0. If wSize=sizeof(HDSP), pBuffer=address of an HDSP.

Return Value

The return code is 1.

Messages That Must Be Supported by A Client

The following messages must be supported by a client:

> mwCfgReleaseService
> mwCfgReinstateService

These messages are sent by the Contention Server to request a client to release services or notify a client to reinstate services. These messages are sent via the `mwNotifyClient()`call supported by the Mwave Agent. For more information, see Mwave Client/Server Support on page 3.

Additionally, client writers should use the mwHelpGetLOSPriorities function to obtain priorities for their clients.

mwCfgReleaseService

This message is sent by the Contention Server to a client during a contention cycle to request that the client release resources. When the Contention Server sends a mwCfgReleaseService message to a client at a given priority, the Contention Server expects the client to release all resources less than or equal to that priority.

mwNotifyClient Parameters

dwConversationId
: Conversation Instance data returned from SRV_CONNECT message

wMessage
: mwCfgReleaseService

lParam
: Client's priority as passed in the mwCfgServiceRequest message.

wSize
: Either 0 or sizeof(HDSP).

pBuffer
: If wSize=0, pBuffer=0. If wSize=sizeof(HDSP), pBuffer=address of an HDSP.

Return Value

Return value of 0 (ReleaseServiceIgnored) indicates that the driver ignored the mwCfgReleaseService message.

NOTE: Clients should not use the return value of 0 except in extreme circumstances. Misuse of this return code seriously disrupts Mwave system performance.

A return value of 1 (ReleaseServiceCanReinstate) indicates that the driver released the resources and can be reinstated when resources become available.

A return value of 2 (ReleaseServiceTerminated) indicates that the driver released the resources and removed itself from the system.

mwCfgReinstateService

Contention Server send this message to the client to allow the client to again attempt to load.

mwNotifyClient Parameters

dwConversationId
Conversation Instance data returned from SRV_CONNECT message

wMessage
mwCfgReinstateService

lParam
Client's priority as passed in the mwCfgServiceRequest message.

wSize
Either 0 or sizeof(HDSP).

pBuffer
If wSize=0, pBuffer=0. If wSize=sizeof(HDSP), pBuffer=address of an HDSP.

Return Value

Return value of 0 indicates that the driver could not reinstate in the system. A return value of 1 indicates that the driver was reinstated.

mwHelpGetLOSPriorities

The client uses this function to obtain the correct priority for use when communicating with the Contention Server. This function is provided in the MWHELPER.DLL (Windows) or the MWHLPOS2.DLL (OS/2).

Parameters

ServiceName
Pointer to a string that contains the name of the client.

LoadPriority
Pointer to an integer that represents the priority the client uses when it loads into the system.

RunPriority
Pointer to an integer that represents the priority the client uses when it runs in the system.

Return Value

A return code of 0 is returned when the name of the client is not found.

A return code of 1 is used when the Contention Server sets the load priority.

A return code of 2 is used when the Contention Server sets the run and load priorities.

Appendix A — ServerProc and ClientProc Protocol

ServerProc

LRESULT APIENTRY ServerProc (dwServerId, dwConversationId, wMessage, lParam, wSize, pBuffer)

DWORD	dwServerId
DWORD	dwConversationId
USHORT	wMessage
LPARAM	lParam
USHORT	wSize
PVOID	pBuffer

Parameters

dwServerId
Server instance specific data. This is the value that was returned when the server processed the SRV_REGISTER message. This value is NULL when wMessage=SRV_REGISTER.

dwConversationId
Conversation specific data. This is the value that was returned when the server processed the SRV_CONNECT message. This value is NULL when wMessage=SRV_REGISTER, SRV_UNREGISTER, or SRV_CONNECT.

wMessage
Identifies the message that the server must process.

Message	Description	Return Value
SRV_REGISTER	Sent to ServerProc when Server makes an mwRegisterServer() call. The ServerProc should create any instance data it needs to support this instance of the server. The instance (or reference to it) is returned and is passed on subsequent calls to ServerProc as dwServerId. lParam—contains the lParam passed in the mwRegisterServer call. This may be used by the ServerProc in the creation of its server instance data. WSize/pBuffer— contains the name of the Server being registered (lpszServerName).	Server instance data to be passed as dwServerId on all subsequent calls to this server instance.
SRV_UNREGISTER	Sent to ServerProc when the Server makes an mwUnregisterServer() call or as a result of a fatal error in the Mwave Agent. ServerProc deallocates any resources allocated on the SRV_REGISTER message.	0
SRV_CONNECT	Sent when a client executes an mwConnectToServer call. The ServerProc should create any instance data it needs to support this client connection. The instance (or its reference) is returned and passed on subsequent calls to ServerProc as dwConversationId. lParam—contains the conversation handle HMCONV created by the agent. This handle is required by the server to send, by way of mwNotifyClient(), notifications to the corresponding client. wSize/pBuffer— contains the unique Client name lpszClientName.	Conversation instance data is passed as dwConversationId on all subsequent calls to this instance of the client/server conversation.

SRV_DISCONNECT	Sent when a client issues mwDisconnectFromServer call or when a fatal error causing the connection to break occurs. ServerProc should deallocate any resources allocated while processing the SRV_CONNECT message.	0

lParam

The meaning of this parameter is dependent on the wMessage value. This parameter may be used to pass any 32 bit (or shorter) data value to the server. This should not be used as a pointer, however, since some addresses may be invalid in the server context on some operating systems.

wSize

Specifies the length in bytes of the buffer pointed to by pBuffer.

pBuffer

If wSize not equal zero, specifies a pointer to a buffer containing wSize bytes. If wSize=0 this parameter is ignored.

The meaning of this parameter is dependent on the wMessage value. This parameter may be used to pass a block of data to the server. The size of the block must be less than or equal to 65535 bytes. The data block should not contain pointers, however, since some addresses may be invalid in the server context on some operating systems.

Return Value

The return value is server/wMessage dependent. Refer to Server documentation for a description of return values. mwDefServerProc() returns the error code MWAGERR_UNSUPPORTEDMESSAGE if the server does not support the wMessage value.

See wMessage table on page 45 for values returned on standard Mwave Agent messages.

Comments

The values for lParam and pBuffer are dependent on the ServerProc/wMessage combination. Refer to individual server documentation for descriptions of messages and their corresponding parameters. Note that wMessage values below SRV_USERBASE are reserved by the Mwave Agent.

Server Proc defined message values must start at SRV_USERBASE.

ServerProc is shown here for prototype purposes. The actual name of ServerProc is at the discretion of the user. The ServerProc name is, in general, not exported. The Mwave Agent uses the registered name (by way of mwRegisterServer()) to associate a connection to a specific ServerProc.

Unsupported messages must be passed to

```
LRESULT mwDefServerProc(dwServerId,
                        dwClientId,
                        wMessage,
                        lParam,
                        wSize,
                        pBuffer);
```

ClientProc

```
LRESULT    ClientProc (dwConversationId, wMessage, lParam, wSize, pBuffer)
DWORD      dwConversationId
USHORT     wMessage
LPARAM     lParam
USHORT     wSize
PVOID      pBuffer
```

Parameters

dwConversationId

Conversation specific data. This is the value that was returned when the ClientProc processed the CLI_CONNECT message. It is set to lParam from mwConnectToServer call when wMessage=CLI_CONNECT.

wMessage

Identifies the message that the client must process.

Message	Description	Returned value
CLI_CONNECT	Sent when the client is connected with mwConnectToServer call. The ClientProc creates any instance data it needs to support this server connection. The instance (or reference to it) is returned and is passed on subsequent calls to ClientProc as dwConversationID. dwConversationId —set to value of lParam in the mwConnectToServer call. This is used as a means of passing initial conversation instance data from the client to the ClientProc. lParam — contains HMCONV for the established client/server connection. wSize/pBuffer—contains the unique Server name lpszServerName.	Conversation instance data passed back in dwConversationId on all subsequent calls on this client/server conversation.
CLI_DISCONNECT	Sent when a client executes a mwDisconnectFromServer call or when a fatal error causing the connection to break occurs.	0

lParam

The meaning of this parameter is dependent on the wMessage value and may be used to pass any 32 bit (or shorter) data value to the client. This parameter should not be used as a pointer, however, since some addresses may be invalid in the client context on some operating systems.

wSize

Specifies the length in bytes of the buffer pointed to by pBuffer.

pBuffer

If wSize not equal zero, specifies a pointer to a buffer containing wSize bytes. If wSize=0 this parameter is ignored.

The meaning of this parameter is dependent on the wMessage value and may be used to pass a block of data to the client. The size of the block must be less than or equal to 65535 bytes. The data block should not contain pointers, however, since some addresses may be invalid in the client context on some operating systems.

Return Value

The return value is client/wMessage dependent. Refer to Client documentation for description of return values. mwDefClientProc() returns the error code MWAGERR_UNSUPPORTEDMESSAGE if the client does not support the wMessage value.

Comments

The values for lParam and pBuffer are dependent on the ClientProc/wMessage combination. Refer to individual server documentation for descriptions of messages and their corresponding parameters. Note that wMessage values below CLI_USERBASE are reserved by the Mwave Agent and their corresponding parameters are described in the wMessage table above.

Clients must be prepared to handle any notifications that a connected-to server is likely to generate. Refer to server documentation relating to notifications and what, if any, events may cause a server to send notifications to the ClientProc.

Unsupported messages must be passed to

```
LRESULT mwDefClientProc(dwConversationId,
                        wMessage,
                        lParam,
                        wSize,
                     pBuffer);
```